aprel Documentation

Release 1.0.0

Erdem Biyik, Aditi Talati, Dorsa Sadigh

Oct 28, 2022

GENERAL

APReL is a unified Python3 library for active preference-based reward learning methods. It offers a modular framework for experimenting with and implementing preference-based reward learning techniques; which include active querying, multimodal learning, and batch generation methods.

Not sure what preference-based learning means? Read on or check our talk at AI-HRI 2021. You can also take a look at our video which contains a simple example.

CHAPTER

ONE

OVERVIEW

APReL is a unified Python3 library for active preference-based reward learning methods. It offers a modular framework for experimenting with and implementing preference-based reward learning techniques; which include active querying, multimodal learning, and batch generation methods.

Not sure what preference-based learning means? Read on or check our video for a simple example.

1.1 Introduction

As robots enter our daily lives, we want them to act in ways that are aligned with our preferences, and goals. Learning a reward function that captures human preferences about how a robot should operate is a fundamental robot learning problem that is the core of the algorithms presented with **APReL**.

There are a number of different information modalities that can be avenues for humans to convey their preferences to a robot. These include demonstrations, physical corrections, observations, language instructions and narrations, ratings, comparisons and rankings, each of which has its own advantages and drawbacks. Learning human preferences using comparisons and rankings is well-studied outside of robotics, and the paradigm of learning human preferences based on comparisons and rankings shows promise in robotics applications as well.

However, preference-based learning poses another important challenge: each comparison or ranking gives a very small amount of information. For example, a pairwise comparison between a trajectory of a car that speeds up at an intersection with another trajectory that slows down gives at most one bit of information. Hence, it becomes critical to optimize for what the user should compare or rank. To this end, researchers have developed several active learning techniques to improve data-efficiency of preference-based learning by maximizing the information acquired from each query to the user. **APReL** enables these techniques to be applied on any simulation environment that is compatible with the standard OpenAI Gym structure.

In essence, **APReL** provides a modular framework for the solutions of the following three problems:

- How do we learn from user preferences after optionally initializing with other feedback types, e.g., demonstrations?
- How do we actively generate preference/ranking queries that are optimized to be informative for the learning model?
- How do we actively generate *batches* of queries to alleviate the computational burden of active query generation?

1.2 Structure of APReL

Let's now briefly look at **APReL**'s modules to see how it deals with solving these problems. These modules include: query types, user models, belief distributions, query optimizers and different acquisition functions. An overview of **APReL**'s general workflow is shown below. We next briefly go over each of the modules.

1.2.1 Basics

APReL implements *Environment* and *Trajectory* classes. An **APReL** environment requires an OpenAI Gym environment and a features function that maps a given sequence of state-action pairs to a vector of trajectory features. *Trajectory* instances then keep trajectories of the *Environment* along with their features.

1.2.2 Query Types

Researchers developed and used several comparison and ranking query types. Among those, **APReL** readily implements preference queries, weak comparison queries, and full ranking queries. More importantly, the module for query types is customizable, allowing researchers to implement other query types and information sources. As an example, demonstrations are already included in **APReL**.

1.2.3 User Models

Preference-based reward learning techniques rely on a human response model, e.g. the softmax model, which gives the probabilities for possible responses conditioned on the query and the reward function. **APReL** allows to adopt any parametric human model and specify which parameters will be fixed or learned.

1.2.4 Belief Distributions

After receiving feedback from the human (Human in the figure above), Bayesian learning is performed based on an assumed human model (Human-hat in the figure) by a belief distribution module. **APReL** implements the sampling-based posterior distribution model that has been widely employed by the researchers. However, its modular structure also allows to implement other belief distributions, e.g., Gaussian processes.

1.2.5 Query Optimizers

After updating the belief distribution with the user feedback, a query optimizer completes the active learning loop by optimizing an acquisition function to find the best query to the human. **APReL** implements the widely-used "optimize-over-a-trajectory-set" idea for this optimization, and allows the acquisition functions that we discussed earlier. Besides, the optimizer module also implements the batch optimization methods that output a batch of queries using different techniques. All of these three components (optimizer, acquisition functions, batch generator) can be extended to other techniques.

1.2.6 Assessing

After (or during) learning, it is often desired to assess the quality of the learned reward function or user model. The final module does this by comparing the learned model with the information from the human.

1.3 Citations

Please cite APReL if you use this library in your publications:

```
@article{biyik2021aprel,
  title={APReL: A Library for Active Preference-based Reward Learning Algorithms},
  author={B{\i}y{\i}k, Erdem and Talati, Aditi and Sadigh, Dorsa},
  journal={arXiv preprint arXiv:2108.07259},
  year={2021}
}
```

CHAPTER

TWO

INSTALLATION

APReL runs on Python 3.

2.1 Install from Source

1. **APReL** uses ffmpeg for trajectory visualizations. Install it with the following command on Linux:

```
apt install ffmpeg
```

If you are using a Mac, you can use Homebrew to install it:

```
brew install ffmpeg
```

2. Clone the aprel repository

```
git clone https://github.com/Stanford-ILIAD/APReL.git
cd APReL
```

3. Install the base requirements with

```
pip3 install -r requirements.txt
```

4. (Optional) If you want to build the docs locally, you will also need some additional packages, which can be installed with:

```
pip3 install -r docs/requirements.txt
```

5. Install **APReL** from the source by running:

```
pip3 install -e .
```

6. Test **APReL**'s runner file by running

```
python examples/python simple.py
```

You should be able to see the MountainCarContinuous-v0 environment rendering multiple times. After it renders (and saves) 10 trajectories, it is going to query you for your preferences. See the next section for more information about this runner file.

CHAPTER

THREE

EXAMPLE

Let's now go over a simple example of how to use **APReL**. This example is based on the examples/simple.py file.

We first import **APReL** and the other necessary libraries. Gym library is needed for inputting an environment.

```
import aprel
import numpy as np
import gym
```

In this example, we will be using the MountainCarContinuous-v0 environment. Let's create an environment object and set the random seeds for reproducibility:

```
env_name = 'MountainCarContinuous-v0'
gym_env = gym.make(env_name)
np.random.seed(0)
env.seed(0)
```

The original goal in MountainCarContinuous-v0 is to move the car such that it reaches the yellow flag.

In preference-based reward learning, a *trajectory features function* must accompany the environment. In **APReL**, this is handled with a user-provided function which takes a list of state-action pairs (of a trajectory) and outputs the array of features. For the MountainCarContinuous-v0 where states consist of position and velocity values, let's use the minimum position, maximum position and the average speed as our features. **Note**: As in Biyik et al. (2019), our feature function below normalizes the features by subtracting the mean and dividing by the standard deviation. These mean and standard deviation values come from randomly generated trajectories, which we pre-computed offline. While this is not a necessary step, it may sometimes make the learning faster.

```
def feature_func(traj):
    """Returns the features of the given MountainCar trajectory, i.e. \Phi(traj).

    Args:
        traj: List of state-action tuples, e.g. [(state0, action0), (state1, action1), ..
↪.]

    Returns:
        features: a numpy vector corresponding the features of the trajectory
    """
    states = np.array([pair[0] for pair in traj])
    actions = np.array([pair[1] for pair in traj[:-1]])
    min_pos, max_pos = states[:,0].min(), states[:,0].max()
    mean_speed = np.abs(states[:,1]).mean()
```

(continues on next page)

(continued from previous page)

```
    mean_vec = [-0.703, -0.344, 0.007]
    std_vec = [0.075, 0.074, 0.003]
    return (np.array([min_pos, max_pos, mean_speed]) - mean_vec) / std_vec
```

We are now ready to wrap the environment into an **APReL** environment along with the feature function:

```
env = aprel.Environment(gym_env, feature_func)
```

APReL comes with a query optimizer that works over a predefined set of trajectories. For this, let's create a trajectory set that consists of 10 randomly generated trajectories:

```
trajectory_set = aprel.generate_trajectories_randomly(env, num_trajectories=10,
                                                      max_episode_length=300,
                                                      file_name=env_name, seed=0)
features_dim = len(trajectory_set[0].features)
```

Let's now define the optimizer which will optimize the queries by considering trajectories from the trajectory set we have just created:

```
query_optimizer = aprel.QueryOptimizerDiscreteTrajectorySet(trajectory_set)
```

APReL allows both simulated and real humans. In this example, we will assume a real human is going to respond to the queries. Next, we define this such that there will be a 0.5 seconds delay time after each trajectory visualization during querying.

```
true_user = aprel.HumanUser(delay=0.5)
```

We will learn a reward function that is linear in trajectory features by assuming a softmax human response model. Let's initiate this model with a random vector of weights. Here, we are using a random vector for weights, because we will already be learning them. So the values we pass here are not important. But we still need to pass them so that the model knows the feature dimensionality. If we wanted to set the other parameters of the softmax model, e.g., rationality coefficient, we would also do that here.

```
params = {'weights': aprel.util_funs.get_random_normalized_vector(features_dim)}
user_model = aprel.SoftmaxUser(params)
```

After defining our user model, we now create a belief distribution over the parameters we want to learn. We will be learning only the *weights*, so let's use the same dictionary of parameters. If we wanted to learn the other parameters of the softmax model, we would pass them here.

```
belief = aprel.SamplingBasedBelief(user_model, [], params)
print('Estimated user parameters: ' + str(belief.mean))
```

Running the above code should print an estimate for the weights. Since we have not provided any data yet, this estimate is not meaningful. We need to query the user to elicit their preferences. For this, we will first start a dummy query. The query optimizer will then optimize a query of the same kind. For example, let's create a dummy preference query (*do you prefer trajectory A or B?* kind of query) with the first two trajectories in the trajectory set:

```
query = aprel.PreferenceQuery(trajectory_set[:2])
```

Now, every time we call the query optimizer with this query, it is going to give us an optimized *preference query*. If we created a, say, weak comparison query, then the optimized queries would also be weak comparison queries. In the next for-loop, we repeatedly do three things: (i) optimize a query, (ii) ask the user for a response to the optimized query, (iii) update the belief distribution with the response.

```
for query_no in range(10):
    queries, objective_values = query_optimizer.optimize('mutual_information', belief,␣
↪query)
    # queries and objective_values are lists even when we do not request a batch of␣
↪queries.
    print('Objective Value: ' + str(objective_values[0]))

    responses = true_user.respond(queries[0])
    belief.update(aprel.Preference(queries[0], responses[0]))
    print('Estimated user parameters: ' + str(belief.mean))
```

Running this code will ask you to respond 10 preference queries that are optimized with respect to the mutual information acquisition function. Below is the first query that is asked to the user:

We select 0 for this query. In other words, we say we prefer the first trajectory. Because it gets closer to solving the task by moving closer to the yellow flag, even though it cannot make it. Continuing in this fashion, we responded the following 9 queries with: [0, 0, 0, 1, 0, 1, 0, 0, 0]. At the end, we see this output:

```
Estimated user parameters: {'weights': array([-0.28493522,  0.72942661,  0.62189126])}
```

Remember our features function: minimum position, maximum position and average speed. The second coefficient being ~0.73 means that we want the maximum position to be high. And it is indeed the case, because we tried to make the car go as further as possible. But how about the other two features? Well, in this case, all features were correlated: In this environment, you have to go back to move further, so we indeed want the minimum position to be low. Similarly, to go further, we need high speeds. Although this is not a part of **APReL**, we trained a reinforcement learning agent using Soft-Actor Critic with this learned reward function (we used this implementation). This is what we got:

Only after 10 queries, we were able to learn a reward function that solves the game! Note that the agent also makes sure to go as back as possible because of the way we designed the features.

Interested in learning other options and features of **APReL**? Take a look at a more advanced example at: examples/advanced.py!

CHAPTER

FOUR

INSTALLATION VIDEO

You can also check our talk at AI-HRI 2021 for a presentation about APReL.

CHAPTER

FIVE

APREL PACKAGE

5.1 Subpackages

5.1.1 aprel.assessing package

aprel.assessing.metrics module

Functions that are useful for assessing the accuracy of the given learning agent.

aprel.assessing.metrics.**cosine_similarity**(*belief:* aprel.learning.belief_models.LinearRewardBelief, *true_user:* aprel.learning.user_models.User) → float

This function tests how well the belief models the true user, when the reward model is linear. It performs this test by returning the cosine similarity between the true and predicted reward weights.

Parameters

- **belief** (LinearRewardBelief) – the learning agent's belief about the user
- **true_user** (User) – a User which has given true weights

Returns the cosine similarity of the predicted weights and the true weights

Return type float

5.1.2 aprel.basics package

aprel.basics.environment module

Environment-related modules.

class aprel.basics.environment.**Environment**(*env: gym.core.Env*, *feature_func: Callable*)

Bases: object

This is a wrapper around an OpenAI Gym environment, so that we can store the features function along with the environment itself.

Parameters

- **env** (*gym.Env*) – An OpenAi Gym environment.
- **features** (*Callable*) – Given a *Trajectory*, this function must return a numpy.array of features.

env

The wrapped environment.

Type gym.Env

features
Features function.

Type Callable

action_space
Inherits from *env*.

observation_space
Inherits from *env*.

reset
Inherits from *env*.

Type Callable

step
Inherits from *env*.

Type Callable

render
Inherits from *env*, if it exists; None otherwise.

Type Callable

render_exists
True if *render* exists.

Type bool

close
Inherits from *env*, if it exists; None otherwise.

Type Callable

close_exists
True if *close* exists.

Type bool

aprel.basics.trajectory module

Modules that are related to environment trajectories.

class aprel.basics.trajectory.**Trajectory**(*env:* aprel.basics.environment.Environment, *trajectory:* *List[Tuple[numpy.array, numpy.array]]*, *clip_path:* *Optional[str] = None*)

Bases: object

A class for keeping trajectories that consist of a sequence of state-action pairs, the features and a clip path that keeps a video visualization of the trajectory.

This class supports indexing, such that t^th index returns the state-action pair at time step t. However, indices cannot be assigned, i.e., a specific state-action pair cannot be changed, because that would enable infeasible trajectories.

Parameters

- **env** (Environment) – The environment object that generated this trajectory.

- **trajectory** (*List[Tuple[numpy.array, numpy.array]]*) – The sequence of state-action pairs.
- **clip_path** (*str*) – The path to the video clip that keeps the visualization of the trajectory.

trajectory
The sequence of state-action pairs.

Type List[Tuple[numpy.array, numpy.array]]

features
Features of the trajectory.

Type numpy.array

clip_path
The path to the video clip that keeps the visualization of the trajectory.

Type str

property length: int
The length of the trajectory, i.e., the number of time steps in the trajectory.

visualize()
Visualizes the trajectory with a video if the clip exists. Otherwise, prints the trajectory information.

Note FPS is fixed at 25 for video visualizations.

class aprel.basics.trajectory.**TrajectorySet**(*trajectories: List[*aprel.basics.trajectory.Trajectory*]*)
Bases: object

A class for keeping a set of trajectories, i.e. *Trajectory* objects.

This class supports indexing, such that t^th index returns the t^th trajectory in the set. Similarly, t^th trajectory in the set can be replaced with a new trajectory using indexing. Only for reading trajectories with indexing, list indices are also allowed.

Parameters **trajectories** (*List[Trajectory]*) – The list of trajectories to be stored in the set.

trajectories
The list of trajectories in the set.

Type List[*Trajectory*]

features_matrix
n x d array of features where each row consists of the d features of the corresponding trajectory.

Type numpy.array

append(*new_trajectory:* aprel.basics.trajectory.Trajectory)
Appends a new trajectory to the set.

property size: int
The number of trajectories in the set.

5.1.3 aprel.learning package

aprel.learning.belief_models module

This file contains Belief classes, which store and update the belief distributions about the user whose reward function is being learned.

TODO GaussianBelief class will be implemented so that the library will include the following work: E. Biyik, N. Huynh, M. J. Kochenderger, D. Sadigh; "Active Preference-Based Gaussian Process Regression for Reward Learning", RSS'20.

class aprel.learning.belief_models.**Belief**

Bases: `object`

An abstract class for Belief distributions.

update(*data: Union[*aprel.learning.data_types.QueryWithResponse, *List[*aprel.learning.data_types.QueryWithResponse*]]*, ***kwargs*)

Updates the belief distribution with a given feedback or a list of feedbacks.

class aprel.learning.belief_models.**LinearRewardBelief**

Bases: *aprel.learning.belief_models.Belief*

An abstract class for Belief distributions for the problems where reward function is assumed to be a linear function of the features.

property mean: Dict

Returns the mean parameters with respect to the belief distribution.

class aprel.learning.belief_models.**SamplingBasedBelief**(*user_model: aprel.learning.user_models.User, dataset: List[aprel.learning.data_types.QueryWithResponse], initial_point: Dict, logprior: Callable = <function uniform_logprior>, num_samples: int = 100, **kwargs*)

Bases: *aprel.learning.belief_models.LinearRewardBelief*

A class for sampling based belief distributions.

In this model, the entire dataset of user feedback is stored and used for calculating the true posterior value for any given set of parameters. A set of parameter samples are then sampled from this true posterior using Metropolis-Hastings algorithm.

Parameters

- **logprior** (*Callable*) – The logarithm of the prior distribution over the user parameters.
- **user_model** (User) – The user response model that will be assumed by this belief distribution.
- **dataset** (*List[*QueryWithResponse*]*) – A list of user feeedbacks.
- **initial_point** (*Dict*) – An initial set of user parameters for Metropolis-Hastings to start.
- **logprior** – The logarithm of the prior distribution over the user parameters. Defaults to a uniform distribution over the hyperball.
- **num_samples** (*int*) – The number of parameter samples that will be sampled using Metropolis-Hastings.
- ****kwargs** – Hyperparameters for Metropolis-Hastings, which include:

- *burnin* (int): The number of initial samples that will be discarded to remove the correlation with the initial parameter set.
- *thin* (int): Once in every *thin* sample will be kept to reduce the autocorrelation between the samples.
- *proposal_distribution* (Callable): The proposal distribution for the steps in Metropolis-Hastings.

user_model
The user response model that is assumed by the belief distribution.

Type *User*

dataset
A list of user feeedbacks.

Type List[*QueryWithResponse*]

num_samples
The number of parameter samples that will be sampled using Metropolis-Hastings.

Type int

sampling_params
Hyperparameters for Metropolis-Hastings, which include:

- *burnin* (int): The number of initial samples that will be discarded to remove the correlation with the initial parameter set.
- *thin* (int): Once in every *thin* sample will be kept to reduce the autocorrelation between the samples.
- *proposal_distribution* (Callable): The proposal distribution for the steps in Metropolis-Hastings.

Type Dict

create_samples(*initial_point: Dict*) → Tuple[List[Dict], List[float]]
Samples num_samples many user parameters from the posterior using Metropolis-Hastings.

Parameters **initial_point** (*Dict*) – initial point to start the chain for Metropolis-Hastings.

Returns

- List[Dict]: dictionaries where each dictionary is a sample of user parameters.
- List[float]: float values where each entry is the log-probability of the corresponding sample.

Return type 2-tuple

property mean: Dict
Returns the mean of the belief distribution by taking the mean over the samples generated by Metropolis-Hastings.

update(*data: Union[*aprel.learning.data_types.QueryWithResponse, *List[*aprel.learning.data_types.QueryWithResponse*]]*, *initial_point: Optional[Dict] = None*)
Updates the belief distribution based on the new feedback (query-response pairs), by adding these to the current dataset and then re-sampling with Metropolis-Hastings. :param data: one or more QueryWithResponse, which

contains multiple trajectory options and the index of the one the user selected as most optimal

Parameters **initial_point** (*Dict*) – the initial point to start Metropolis-Hastings from, will be set to the mean from the previous distribution if None

aprel.learning.data_types module

Modules for queries and user responses.

TODO OrdinalQuery classes will be implemented so that the library will include ordinal data, which was used for reward learning in: K. Li, M. Tucker, E. Biyik, E. Novoseller, J. W. Burdick, Y. Sui, D. Sadigh, Y. Yue, A. D. Ames; "ROIAL: Region of Interest Active Learning for Characterizing Exoskeleton Gait Preference Landscapes", ICRA'21.

class aprel.learning.data_types.**Demonstration**(*trajectory:* aprel.basics.trajectory.Trajectory, *query: Optional[*aprel.learning.data_types.DemonstrationQuery*] = None*)

Bases: *aprel.learning.data_types.QueryWithResponse*

The trajectory generated by the DemonstrationQuery, along with the DemonstrationQuery that prompted the user with the initial state.

For preference-based reward learning initialized with demonstrations, this class should be used (without actually querying the user). First, the demonstration should be collected as a *Trajectory* object. Then, a *Demonstration* instance should be created with this trajectory without specifying the query parameter, in which case it is automatically assigned as the initial state of the trajectory.

Parameters

- **trajectory** (Trajectory) – The demonstrated trajectory.
- **query** (DemonstrationQuery) – The query that led to the trajectory, i.e., the initial state of the trajectory.

trajectory
The demonstrated trajectory.

Type *Trajectory*

features
The features of the demonstrated trajectory.

Type numpy.array

Raises **AssertionError** – if the initial state of the trajectory does not match with the query.

class aprel.learning.data_types.**DemonstrationQuery**(*initial_state: numpy.array*)

Bases: *aprel.learning.data_types.Query*

A demonstration query is one where the initial state is given to the user, and they are asked to control the robot.

Although not practical for optimization, this class is defined for coherence with other query types.

Parameters **initial_state** (*numpy.array*) – The initial state of the environment.

initial_state
The initial state of the environment.

Type numpy.array

class aprel.learning.data_types.**FullRanking**(*query:* aprel.learning.data_types.FullRankingQuery, *response: List[int]*)

Bases: *aprel.learning.data_types.QueryWithResponse*

A Full Ranking feedback.

Contains the *FullRankingQuery* the user responded to and the response.

Parameters

- **query** (FullRankingQuery) – The query for which the feedback was given.
- **response** (*numpy.array*) – The response of the user to the query, indices from the most preferred to the least.

response
The response of the user to the query, indices from the most preferred to the least.

Type numpy.array

Raises **AssertionError** – if the response is not in the response set of the query.

class aprel.learning.data_types.**FullRankingQuery**(*slate: Union[*aprel.basics.trajectory.TrajectorySet, *List[*aprel.basics.trajectory.Trajectory*]]*)

Bases: *aprel.learning.data_types.Query*

A full ranking query is one where the user is presented with multiple trajectories and asked for a ranking from their most preferred trajectory to the least.

Parameters **slate** (*TrajectorySet or List[Trajectory]*) – The set of trajectories that will be presented to the user.

K
The number of trajectories in the query.

Type int

response_set
The set of possible responses to the query, which is all *K*-combinations of the trajectory indices in the slate.

Type numpy.array

Raises **AssertionError** – if slate has less than 2 trajectories.

property slate: aprel.basics.trajectory.TrajectorySet
Returns a *TrajectorySet* of the trajectories in the query.

visualize(*delay: float = 0.0*) → List[int]
Visualizes the query and interactively asks for a response.

Parameters **delay** (*float*) – The waiting time between each trajectory visualization in seconds.

Returns The response of the user, as a list from the most preferred to the least.

Return type List[int]

class aprel.learning.data_types.**Preference**(*query:* aprel.learning.data_types.PreferenceQuery, *response: int*)

Bases: *aprel.learning.data_types.QueryWithResponse*

A Preference feedback.

Contains the *PreferenceQuery* the user responded to and the response.

Parameters

- **query** (PreferenceQuery) – The query for which the feedback was given.
- **response** (*int*) – The response of the user to the query.

response
The response of the user to the query.

Type int

Raises **AssertionError** – if the response is not in the response set of the query.

class aprel.learning.data_types.**PreferenceQuery**(*slate: Union[*aprel.basics.trajectory.TrajectorySet, *List[*aprel.basics.trajectory.Trajectory*]]*)

Bases: *aprel.learning.data_types.Query*

A preference query is one where the user is presented with multiple trajectories and asked for their favorite among them.

Parameters **slate** (*TrajectorySet or List[Trajectory]*) – The set of trajectories that will be presented to the user.

K

The number of trajectories in the query.

Type int

response_set

The set of possible responses to the query.

Type numpy.array

Raises **AssertionError** – if slate has less than 2 trajectories.

property slate: aprel.basics.trajectory.TrajectorySet

Returns a *TrajectorySet* of the trajectories in the query.

visualize(*delay: float = 0.0*) → int

Visualizes the query and interactively asks for a response.

Parameters **delay** (*float*) – The waiting time between each trajectory visualization in seconds.

Returns The response of the user.

Return type int

class aprel.learning.data_types.**Query**

Bases: object

An abstract parent class that is useful for typing.

A query is a question to the user.

copy()

Returns a deep copy of the query.

visualize(*delay: float = 0.0*)

Visualizes the query, i.e., asks it to the user.

Parameters **delay** (*float*) – The waiting time between each trajectory visualization in seconds.

class aprel.learning.data_types.**QueryWithResponse**(*query:* aprel.learning.data_types.Query)

Bases: object

An abstract parent class that is useful for typing.

An instance of this class holds both the query and the user's response to that query.

Parameters **query** (*Query*) – The query.

query

The query.

Type *Query*

class aprel.learning.data_types.**WeakComparison**(*query:* aprel.learning.data_types.WeakComparisonQuery, *response: int*)

Bases: *aprel.learning.data_types.QueryWithResponse*

A Weak Comparison feedback.

Contains the *WeakComparisonQuery* the user responded to and the response.

Parameters

- **query** (WeakComparisonQuery) – The query for which the feedback was given.
- **response** (*int*) – The response of the user to the query.

response
The response of the user to the query.

Type int

Raises **AssertionError** – if the response is not in the response set of the query.

class aprel.learning.data_types.**WeakComparisonQuery**(*slate:* *Union[*aprel.basics.trajectory.TrajectorySet, *List[*aprel.basics.trajectory.Trajectory*]]*)

Bases: *aprel.learning.data_types.Query*

A weak comparison query is one where the user is presented with two trajectories and asked for their favorite among them, but also given an option to say 'they are about equal'.

Parameters **slate** (TrajectorySet *or List[*Trajectory*]*) – The set of trajectories that will be presented to the user.

K
The number of trajectories in the query. It is always equal to 2 and kept for consistency with *PreferenceQuery* and *FullRankingQuery*.

Type int

response_set
The set of possible responses to the query, which is always equal to [-1, 0, 1] where -1 represents the *About Equal* option.

Type numpy.array

Raises **AssertionError** – if slate does not have exactly 2 trajectories.

property slate: aprel.basics.trajectory.TrajectorySet
Returns a *TrajectorySet* of the trajectories in the query.

visualize(*delay: float = 0.0*) → int
Visualizes the query and interactively asks for a response.

Parameters **delay** (*float*) – The waiting time between each trajectory visualization in seconds.

Returns The response of the user.

Return type int

aprel.learning.data_types.**isinteger**(*input: str*) → bool
Returns whether input is an integer.

Note This function returns False if input is a string of a float, e.g., '3.0'.

TODO Should this go to utils?

Parameters **input** (*str*) – The string to be checked for being an integer.

Returns True if the input is an integer, False otherwise.

Return type bool

Raises **AssertionError** – if the input is not a string.

aprel.learning.user_models module

Modules for user response models, including human users.

class aprel.learning.user_models.**HumanUser**(*delay: float = 0.0*)
Bases: *aprel.learning.user_models.User*

Human user class whose response model is unknown. This class is useful for interactive runs, where a real human responds to the queries rather than simulated user models.

Parameters **delay** (*float*) – The waiting time between each trajectory visualization during querying in seconds.

delay
The waiting time between each trajectory visualization during querying in seconds.

Type float

respond(*queries: Union[*aprel.learning.data_types.Query*, List[*aprel.learning.data_types.Query*]]*) → List
Interactively asks for the user's responses to the given queries.

Parameters **queries** (*Query or List[Query]*) – A query or a list of queries for which the user's response(s) is/are requested.

Returns

A list of user responses where each response corresponds to the query in the queries.

Note The return type is always a list, even if the input is a single query.

Return type List

class aprel.learning.user_models.**SoftmaxUser**(*params_dict: Dict*)
Bases: *aprel.learning.user_models.User*

Softmax user class whose response model follows the softmax choice rule, i.e., when presented with multiple trajectories, this user chooses each trajectory with a probability that is proportional to the expontential of the reward of that trajectory.

Parameters **params_dict** (*Dict*) – the parameters of the softmax user model, which are: - *weights* (numpy.array): the weights of the linear reward function. - *beta* (float): rationality coefficient for comparisons and rankings. - *beta_D* (float): rationality coefficient for demonstrations. - *delta* (float): the perceivable difference parameter for weak comparison queries.

Raises **AssertionError** – if a *weights* parameter is not provided in the params_dict.

loglikelihood(*data:* aprel.learning.data_types.QueryWithResponse) → float
Overwrites the parent's method. See *User* for more information.

Note The loglikelihood value is the logarithm of the *unnormalized* likelihood if the input is a demonstration. Otherwise, it is the exact loglikelihood.

response_logprobabilities(*query:* aprel.learning.data_types.Query) → numpy.array
Overwrites the parent's method. See *User* for more information.

reward(*trajectories: Union[*aprel.basics.trajectory.Trajectory, aprel.basics.trajectory.TrajectorySet*]*) → Union[float, numpy.array]
Returns the reward of a trajectory or a set of trajectories conditioned on the user.

Parameters **trajectories** (*Trajectory or TrajectorySet*) – The trajectories for which the reward will be calculated.

Returns the reward value of the `trajectories` conditioned on the user.

Return type numpy.array or float

class aprel.learning.user_models.**User**(*params_dict: Optional[Dict] = None*)
Bases: `object`

An abstract class to model the user of which the reward function is being learned.

Parameters **params_dict** (*Dict*) – parameters of the user model.

copy()

likelihood(*data:* aprel.learning.data_types.QueryWithResponse) → float
Returns the likelihood of the given user feedback under the user.

Parameters **data** (QueryWithResponse) – The data (which keeps a query and a response) for which the likelihood is going to be calculated.

Returns The likelihood of `data` under the user.

Return type float

likelihood_dataset(*dataset: List[*aprel.learning.data_types.QueryWithResponse*]*) → float
Returns the likelihood of the given feedback dataset under the user.

Parameters **dataset** (*List[QueryWithResponse]*) – The dataset (which keeps a list of feedbacks) for which the likelihood is going to be calculated.

Returns The likelihood of `dataset` under the user.

Return type float

loglikelihood(*data:* aprel.learning.data_types.QueryWithResponse) → float
Returns the loglikelihood of the given user feedback under the user.

Parameters **data** (QueryWithResponse) – The data (which keeps a query and a response) for which the loglikelihood is going to be calculated.

Returns The loglikelihood of `data` under the user.

Return type float

loglikelihood_dataset(*dataset: List[*aprel.learning.data_types.QueryWithResponse*]*) → float
Returns the loglikelihood of the given feedback dataset under the user.

Parameters **dataset** (*List[QueryWithResponse]*) – The dataset (which keeps a list of feedbacks) for which the loglikelihood is going to be calculated.

Returns The loglikelihood of `dataset` under the user.

Return type float

property params
Returns the parameters of the user.

respond(*queries: Union[*aprel.learning.data_types.Query, *List[*aprel.learning.data_types.Query*]]*) → List
Simulates the user's responses to the given queries.

Parameters **queries** (*Query or List[Query]*) – A query or a list of queries for which the user's response(s) is/are requested.

Returns

A list of user responses where each response corresponds to the query in the queries.

Note The return type is always a list, even if the input is a single query.

Return type List

response_logprobabilities(*query:* aprel.learning.data_types.Query) → numpy.array
Returns the log probability for each response in the response set for the query under the user.

Parameters **query** (*Query*) – The query for which the log-probabilites are being calculated.

Returns

An array, where each entry is the log-probability of the corresponding response in the query's response set.

Return type numpy.array

response_probabilities(*query:* aprel.learning.data_types.Query) → numpy.array
Returns the probability for each response in the response set for the query under the user.

Parameters **query** (*Query*) – The query for which the probabilites are being calculated.

Returns

An array, where each entry is the probability of the corresponding response in the query's response set.

Return type numpy.array

5.1.4 aprel.querying package

aprel.querying.acquisition_functions module

This module contains a set of acquisition functions that determine the value of a given query, which is useful for acitive query optimization.

aprel.querying.acquisition_functions.**disagreement**(*weights: numpy.array*, *logprobs: List[float]*, ***kwargs*) → float
This function returns the disagreement value between two sets of reward weights (weights's). This is useful as an acquisition function when a trajectory planner is available and when the desired query contains only two trajectories. The pair of weights with the highest disagreement is found and then the best trajectories according to them forms the optimized query.

This is implemented based on the following paper:

- Learning an Urban Air Mobility Encounter Model from Expert Preferences

Parameters

- **weights** (*numpy.array*) – 2 x d array where each row is a set of reward weights. The disagreement between these two weights will be calculated.

- **logprobs** (*List[float]*) – log probabilities of the given reward weights under the belief.
- ****kwargs** – acquisition function hyperparameters:
  - *lambda* **(float) The tradeoff parameter. The higher lambda, the more important the** disagreement between the weights is. The lower lambda, the more important their log probabilities. Defaults to 0.01.

Returns the disagreement value (always nonnegative)

Return type float

Raises **AssertionError** – if `weights` and `logprobs` have mismatching number of elements.

aprel.querying.acquisition_functions.**mutual_information**(*belief:* aprel.learning.belief_models.Belief, *query:* aprel.learning.data_types.Query, ***kwargs*) → float

This function returns the mutual information between the given belief distribution and the query. Maximum mutual information is often desired for data-efficient learning.

This is implemented based on the following paper:

- Asking Easy Questions: A User-Friendly Approach to Active Reward Learning

Parameters

- **belief** (Belief) – the current belief distribution over the reward function
- **query** (Query) – a query to ask the user
- ****kwargs** – none used currently

Returns the mutual information value (always nonnegative)

Return type float

aprel.querying.acquisition_functions.**random**()

This function does nothing, but is added so that *aprel.querying.query_optimizer* can use it as a check.

aprel.querying.acquisition_functions.**regret**(*weights: numpy.array*, *logprobs: List[float]*, *planned_trajectories:* *List[*aprel.basics.trajectory.Trajectory*]*, ***kwargs*) → float

This function returns the regret value between two sets of reward weights (weights's). This is useful as an acquisition function when a trajectory planner is available and when the desired query contains only two trajectories. The pair of weights with the highest regret is found and then the best trajectories according to them forms the optimized query.

This is implemented based on the following paper:

- Active Preference Learning using Maximum Regret

TODO This acquisition function requires all rewards to be positive, but there is no check for that.

Parameters

- **weights** (*numpy.array*) – 2 x d array where each row is a set of reward weights. The regret between these two weights will be calculated.
- **logprobs** (*List[float]*) – log probabilities of the given reward weights under the belief.

- **planned_trajectories** (*List[Trajectory]*) – the optimal trajectories under the given reward weights.
- ****kwargs** – none used currently

Returns the regret value

Return type float

Raises **AssertionError** – if `weights`, `logprobs` and `planned_trajectories` have mismatching number of elements.

aprel.querying.acquisition_functions.**thompson**()
This function does nothing, but is added so that *aprel.querying.query_optimizer* can use it as a check.

aprel.querying.acquisition_functions.**volume_removal**(*belief:* aprel.learning.belief_models.Belief, *query:* aprel.learning.data_types.Query, ***kwargs*) → float
This function returns the expected volume removal from the *unnormalized* belief distribution. Maximum volume removal is often desired for data-efficient learning.

This is implemented based on the following two papers:

- Active Preference-Based Learning of Reward Functions
- The Green Choice: Learning and Influencing Human Decisions on Shared Roads

Note As Bıyık et al. (2019) pointed out, volume removal has trivial global maximizers when query maximizes the uncertainty for the user, e.g., when all trajectories in the slate of a Preference-Query is identical. Hence, the optimizations with volume removal are often ill-posed.

Parameters

- **belief** (Belief) – the current belief distribution over the reward function
- **query** (Query) – a query to ask the user
- ****kwargs** – none used currently

Returns the expected volume removal value (always nonnegative)

Return type float

aprel.querying.query_optimizer module

This file contains classes which have functions to optimize the queries to ask the human.

class aprel.querying.query_optimizer.**QueryOptimizer**
Bases: `object`

An abstract class for query optimizer frameworks.

acquisition_functions
keeps name-function pairs for the acquisition functions. If new acquisition functions are implemented, they should be added to this dictionary.

Type Dict

class aprel.querying.query_optimizer.**QueryOptimizerDiscreteTrajectorySet**(*trajectory_set:* aprel.basics.trajectory.TrajectorySet)
Bases: *aprel.querying.query_optimizer.QueryOptimizer*

Query optimization framework that assumes a discrete set of trajectories is available. The query optimization is then performed over this discrete set.

Parameters **trajectory_set** (TrajectorySet) – The set of trajectories from which the queries will be optimized. This set defines the possible set of trajectories that may show up in the optimized query.

trajectory_set
The set of trajectories from which the queries are optimized. This set defines the possible set of trajectories that may show up in the optimized query.

Type *TrajectorySet*

argplanner(*user:* aprel.learning.user_models.User) → int
Given a user model, returns the index of the trajectory that best fits the user in the trajectory set.

Parameters **user** (User) – The user object for whom the optimal trajectory is being searched.

Returns The index of the optimal trajectory in the trajectory set.

Return type int

boundary_medoids_batch(*acquisition_func: Callable*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, *batch_size: int*, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]
Uses the boundary medoids method to find a batch of queries. See Batch Active Preference-Based Learning of Reward Functions for more information about the method.

Parameters

- **acquisition_func** (*Callable*) – the acquisition function to be maximized by each individual query.
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- **batch_size** (*int*) – the batch size of the output.
- ****kwargs** – Hyperparameters *reduced_size*, *distance*, and extra arguments needed for specific acquisition functions.
  - *reduced_size* (int): The hyperparameter B in the original method. This method first greedily chooses B queries from the feasible set of queries out of the trajectory set, and then applies the boundary medoids selection. Defaults to 100.
  - *distance* (Callable): A distance function which returns a pairwise distance matrix (numpy.array) when inputted a list of queries. Defaults to *aprel.utils.batch_utils.default_query_distance()*.

Returns

- List[Query]: The optimized batch of queries as a list.
- numpy.array: An array of floats that keep the acquisition function values corresponding to the output queries.

Return type 2-tuple

dpp_batch(*acquisition_func: Callable*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, *batch_size: int*, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]
Uses the determinantal point process (DPP) based method to find a batch of queries. See Batch Active

Learning Using Determinantal Point Processes for more information about the method.

Parameters

- **acquisition_func** (*Callable*) – the acquisition function to be maximized by each individual query.
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- **batch_size** (*int*) – the batch size of the output.
- ****kwargs** – Hyperparameters *reduced_size*, *distance*, *gamma*, and extra arguments needed for specific acquisition functions.
  - *reduced_size* (int): The hyperparameter B in the original method. This method first greedily chooses B queries from the feasible set of queries out of the trajectory set, and then applies the boundary medoids selection. Defaults to 100.
  - *distance* (Callable): A distance function which returns a pairwise distance matrix (numpy.array) when inputted a list of queries. Defaults to *aprel.utils.batch_utils.default_query_distance()*.
  - *gamma* (float): The hyperparameter *gamma* in the original method. The higher gamma, the more important the acquisition function values. The lower gamma, the more important the diversity of queries. Defaults to 1.

Returns

- List[Query]: The optimized batch of queries as a list.
- numpy.array: An array of floats that keep the acquisition function values corresponding to the output queries.

Return type 2-tuple

exhaustive_search(*acquisition_func: Callable*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]

Searches over the possible queries to find the singular most optimal query.

Parameters

- **acquisition_func** (*Callable*) – the acquisition function to be maximized.
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- ****kwargs** – extra arguments needed for specific acquisition functions.

Returns

- List[Query]: The optimal query as a list of one *Query*.
- numpy.array: An array of floats that keep the acquisition function value corresponding to the output query.

Return type 2-tuple

greedy_batch(*acquisition_func: Callable*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, *batch_size: int*, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]

Uses the greedy method to find a batch of queries by selecting the `batch_size` individually most optimal queries.

Parameters

- **acquisition_func** (*Callable*) – the acquisition function to be maximized by each individual query.
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- **batch_size** (*int*) – the batch size of the output.
- ****kwargs** – extra arguments needed for specific acquisition functions.

Returns

- List[Query]: The optimized batch of queries as a list.
- numpy.array: An array of floats that keep the acquisition function values corresponding to the output queries.

Return type 2-tuple

medoids_batch(*acquisition_func: Callable*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, *batch_size: int*, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]

Uses the medoids method to find a batch of queries. See Batch Active Preference-Based Learning of Reward Functions for more information about the method.

Parameters

- **acquisition_func** (*Callable*) – the acquisition function to be maximized by each individual query.
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- **batch_size** (*int*) – the batch size of the output.
- ****kwargs** – Hyperparameters *reduced_size*, *distance*, and extra arguments needed for specific acquisition functions.
  - *reduced_size* (int): The hyperparameter B in the original method. This method first greedily chooses B queries from the feasible set of queries out of the trajectory set, and then applies the medoids selection. Defaults to 100.
  - *distance* (Callable): A distance function which returns a pairwise distance matrix (numpy.array) when inputted a list of queries. Defaults to *aprel.utils.batch_utils.default_query_distance()*.

Returns

- List[Query]: The optimized batch of queries as a list.
- numpy.array: An array of floats that keep the acquisition function values corresponding to the output queries.

Return type 2-tuple

optimize(*acquisition_func_str: str*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, *batch_size: int = 1*, *optimization_method: str = 'exhaustive_search'*, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]

This function generates the optimal query or the batch of queries to ask to the user given a belief distribution about them. It also returns the acquisition function values of the optimized queries.

Parameters

- **acquisition_func_str** (*str*) – the name of the acquisition function used to decide the value of each query. Currently implemented options are:
  - *disagreement*: Based on Katz. et al. (2019).
  - *mutual_information*: Based on Bıyık et al. (2019).
  - *random*: Randomly chooses a query.
  - *regret*: Based on Wilde et al. (2020).
  - *thompson*: Based on Tucker et al. (2019).
  - *volume_removal*: Based on Sadigh et al. (2017) and Bıyık et al..
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- **batch_size** (*int*) – the number of queries to return.
- **optimization_method** (*str*) – the name of the method used to select queries. Currently implemented options are:
  - *exhaustive_search*: Used for exhaustively searching a single query.
  - *greedy*: Exhaustively searches for the top `batch_size` queries in terms of the acquisition function.
  - *medoids*: Batch generation method based on Bıyık et al. (2018).
  - *boundary_medoids*: Batch generation method based on Bıyık et al. (2018).
  - *successive_elimination*: Batch generation method based on Bıyık et al. (2018).
  - *dpp*: Batch generation method based on Bıyık et al. (2019).
- ****kwargs** – extra arguments needed for specific optimization methods or acquisition functions.
- **Returns** – 2-tuple:
  - List[Query]: The list of optimized queries. **Note**: Even if `batch_size` is 1, a list is returned.
  - numpy.array: An array of floats that keep the acquisition function values corresponding to the output queries.

planner(*user:* aprel.learning.user_models.User) → *aprel.basics.trajectory.Trajectory*

Given a user model, returns the trajectory in the trajectory set that best fits the user.

Parameters **user** (User) – The user object for whom the optimal trajectory is being searched.

Returns The optimal trajectory in the trajectory set.

Return type *Trajectory*

successive_elimination_batch(*acquisition_func: Callable*, *belief:* aprel.learning.belief_models.Belief, *initial_query:* aprel.learning.data_types.Query, *batch_size: int*, ***kwargs*) → Tuple[List[*aprel.learning.data_types.Query*], numpy.array]

Uses the successive elimination method to find a batch of queries. See Batch Active Preference-Based Learning of Reward Functions for more information about the method.

Parameters

- **acquisition_func** (*Callable*) – the acquisition function to be maximized by each individual query.
- **belief** (Belief) – the current belief distribution over the user.
- **initial_query** (Query) – an initial query such that the output query will have the same type.
- **batch_size** (*int*) – the batch size of the output.
- ****kwargs** – Hyperparameters *reduced_size*, *distance*, and extra arguments needed for specific acquisition functions.
  - *reduced_size* (int): The hyperparameter B in the original method. This method first greedily chooses B queries from the feasible set of queries out of the trajectory set, and then applies the boundary medoids selection. Defaults to 100.
  - *distance* (Callable): A distance function which returns a pairwise distance matrix (numpy.array) when inputted a list of queries. Defaults to *aprel.utils.batch_utils.default_query_distance()*.

Returns

- List[Query]: The optimized batch of queries as a list.
- numpy.array: An array of floats that keep the acquisition function values corresponding to the output queries.

Return type 2-tuple

5.1.5 aprel.utils package

aprel.utils.batch_utils module

Utility functions for active batch generation.

aprel.utils.batch_utils.**default_query_distance**(*queries: List[*aprel.learning.data_types.Query*]*, ***kwargs*) → numpy.array

Given a set of m queries, returns an m-by-m matrix, each entry representing the distance between the corresponding queries.

Parameters

- **queries** (*List[Query]*) – list of m queries for which the distances will be computed
- ****kwargs** – The hyperparameters.
  - *metric* (str): The distance metric can be specified with this argument. Defaults to 'euclidean'. See https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.distance.pdist.html for the set of available metrics.

Returns an m-by-m numpy array that consists of the pairwise distances between the queries.

Return type numpy.array

Raises **AssertionError** – if the query is not a compatible type. Currently, the compatible types are: *FullRankingQuery*, *PreferenceQuery*, and *WeakComparisonQuery* (all for a slate size of 2).

aprel.utils.dpp module

This module handles greedy estimation of the mode of a determinantal point process (DPP). The technique is based on Biyik et al. (2019). The code is adopted from https://github.com/Stanford-ILIAD/DPP-Batch-Active-Learning/blob/master/reward_learning/dpp_sampler.py.

class aprel.utils.dpp.**Kernel**

Bases: object

getKernel(*ps*, *qs*)

class aprel.utils.dpp.**Sampler**(*kernel*, *distances*, *k*)

Bases: object

addGreedy()

append(*ind*)

clear()

ratios(*item_ids=None*)

sample()

warmStart()

class aprel.utils.dpp.**ScoredKernel**(*R*, *distances*, *scores*)

Bases: *aprel.utils.dpp.Kernel*

getKernel(*p_ids*, *q_ids*)

aprel.utils.dpp.**dpp_mode**(*distances*, *scores*, *k*)

aprel.utils.dpp.**sample_ids_mc**(*distances*, *scores*, *k*)

aprel.utils.dpp.**setup_sampler**(*distances*, *scores*, *k*)

aprel.utils.generate_trajectories module

This module stores the functions for trajectory set generation.

aprel.utils.generate_trajectories.**generate_trajectories_randomly**(*env: aprel.basics.environment.Environment*, *num_trajectories: int*, *max_episode_length: Optional[int] = None*, *file_name: Optional[str] = None*, *restore: bool = False*, *headless: bool = False*, *seed: Optional[int] = None*) → *aprel.basics.trajectory.TrajectorySet*

Generates num_trajectories random trajectories, or loads (some of) them from the given file.

Parameters

- **env** (Environment) – An *Environment* instance containing the OpenAI Gym environment to be simulated.
- **num_trajectories** (*int*) – the number of trajectories to generate.
- **max_episode_length** (*int*) – the maximum number of time steps for the new trajectories. No limit is assumed if None (or not given).
- **file_name** (*str*) – the file name to save the generated trajectory set and/or restore the trajectory set from. :Note: If `restore` is true and so a set is being restored, then the restored file will be overwritten with the new set.
- **restore** (*bool*) – If true, it will first try to load the trajectories from `file_name`. If the file has fewer trajectories than needed, then more trajectories will be generated to compensate the difference.
- **headless** (*bool*) – If true, the trajectory set will be saved and returned with no visualization. This makes trajectory generation faster, but it might be difficult for real humans to compare trajectories only based on the features without any visualization.
- **seed** (*int*) – Seed for the randomness of action selection. :Note: Environment should be separately seeded. This seed is only for the action selection.

Returns A set of `num_trajectories` randomly generated trajectories.

Return type *TrajectorySet*

Raises **AssertionError** – if `restore` is true, but no `file_name` is given.

aprel.utils.kmedoids module

Function for K-Medoids algorithm.

aprel.utils.kmedoids.**kMedoids**(*D: numpy.array*, *k: int*, *tmax: int = 100*) → numpy.array

Runs the K-Medoids algorithm to return the indices of the medoids. This is based on Bauckhage (2015). And the implementation is adopted from https://github.com/letiantian/kmedoids.

Parameters

- **D** (*numpy.array*) – a distance matrix, where D[a][b] is the distance between points a and b.
- **k** (*int*) – the number of medoids to return.
- **tmax** (*int*) – the maximum number of steps to take in forming clusters.

Returns an array that keeps the indices of the `k` selected queries.

Return type numpy.array

aprel.utils.sampling_utils module

This module contains functions that are useful for the sampling in *SamplingBasedBelief*.

aprel.utils.sampling_utils.**gaussian_proposal**(*point: Dict*) → Dict

For the Metropolis-Hastings sampling algorithm, this function generates the next step in the Markov chain, with a Gaussian distribution of standard deviation 0.05.

Parameters **point** (*Dict*) – the current point in the Markov chain.

Returns the next point in the Markov chain.

Return type Dict

aprel.utils.sampling_utils.**uniform_logprior**(*params: Dict*) → float
This is a log prior belief over the user. Specifically, it is a uniform distribution over ||weights|| <= 1.

Parameters **params** (*Dict*) – parameters of the user for which the log prior is going to be calculated.

Returns the (unnormalized) log probability of weights, which is 0 (as 0 = log 1) if ||weights|| <= 1, and negative infitiny otherwise.

Return type float

aprel.utils.util_functions module

General utility functions.

aprel.utils.util_functions.**get_random_normalized_vector**(*dim: int*) → numpy.array
Returns a random normalized vector with the given dimensions.

Parameters **dim** (*int*) – The dimensionality of the output vector.

Returns A random normalized vector that lies on the surface of the dim-dimensional hypersphere.

Return type numpy.array

CHAPTER

SIX

ACKNOWLEDGEMENTS

We benefitted from the robosuite library while building this documentation.

CHAPTER

SEVEN

REFERENCES

- A Simple and Fast Algorithm for K-medoids Clustering. Hae-Sang Park, Chi-Hyuck Jun
- Active Preference-Based Learning of Reward Functions . Dorsa Sadigh, Anca D. Dragan, Shankar Sastry, and Sanjit A. Seshia
- Active Preference Learning Using Maximum Regret . Nils Wilde, Dana Kulic, Stephen L. Smith
- Asking Easy Questions: A User-Friendly Approach to Active Reward Learning. Erdem Bıyık, Malayandi Palan, Nicholas C. Landolfi, Dylan P. Losey, Dorsa Sadigh
- Batch Active Preference-Based Learning of Reward Functions. Erdem Bıyık, Dorsa Sadigh
- Batch Active Learning Using Determinantal Point Processes. Erdem Bıyık, Kenneth Wang, Nima Anari, Dorsa Sadigh
- Bayesian Inverse Reinforcement Learning. Deepak Ramachandran, Eyal Amir
- Determinantal point processes for machine learning. Alex Kulesza, Ben Taskar
- Learning an Urban Air Mobility Encounter Model from Expert Preferences. Sydney M. Katz, Anne-Claire Le Bihan, Mykel J. Kochenderfer
- Learning Reward Functions by Integrating Human Demonstrations and Preferences. Malayandi Palan, Nicholas C. Landolfi, Gleb Shevchuk, Dorsa Sadigh
- Learning Reward Functions from Diverse Sources of Human Feedback: Optimally Integrating Demonstrations and Preferences. Erdem Bıyık, Dylan P. Losey, Malayandi Palan, Nicholas C. Landolfi, Gleb Shevchuk, Dorsa Sadigh
- NumPy / SciPy Recipes for Data Science: k-Medoids Clustering. Christian Bauckhage
- OpenAI Gym. Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, Wojciech Zaremba
- Preference-Based Learning for Exoskeleton Gait Optimization . Maegan Tucker, Ellen Novoseller, Claudia Kann, Yanan Sui, Yisong Yue, Joel Burdick, Aaron D. Ames
- Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor . Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, Sergey Levine
- The Green Choice: Learning and Influencing Human Deci-sions on Shared Roads. Erdem Bıyık, Daniel A. Lazar, Dorsa Sadigh, Ramtin Pedarsani

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

a

INDEX

Q

R

S

T

U

V

W