striatum Documentation

Release 0.0.1

Y.-Y. Yang, Y.-A. Lin

November 07, 2016

Contents

Contents:

CHAPTER 1

API Reference

1.1 striatum.bandit package

1.1.1 Submodules

1.1.2 striatum.bandit.exp3 module

Exp3: Exponential-weight algorithm for Exploration and Exploitation This module contains a class that implements EXP3, a bandit algorithm that randomly choose an action according to a learned probability distribution.

class `striatum.bandit.exp3.`**`Exp3`**(*history_storage*, *model_storage*, *action_storage*, *gamma=0.3*, *random_state=None*)

Bases: `striatum.bandit.bandit.BaseBandit`

Exp3 algorithm.

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

gamma: float, 0 < gamma <= 1

The parameter used to control the minimum chosen probability for each action.

References

[R1]

Attributes

`history_storage`

Methods

`add_action`(actions)	Add new actions (if needed).
`calculate_avg_reward`()	Calculate average reward with respect to time.
`calculate_cum_reward`()	Calculate cumulative reward with respect to time.
`get_action`([context, n_actions])	Return the action to perform
`plot_avg_regret`()	Plot average regret with respect to time.
`plot_avg_reward`()	Plot average reward with respect to time.
`reward`(history_id, rewards)	Reward the previous action with reward.

add_action(*actions*)

Add new actions (if needed).

Parameters **actions** : iterable

A list of Action objects for recommendation

get_action(*context=None*, *n_actions=None*)

Return the action to perform

Parameters **context** : {array-like, None}

The context of current state, None if no context available.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns **history_id** : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward(*history_id*, *rewards*)

Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

1.1.3 striatum.bandit.exp4p module

EXP4.P: An extention to exponential-weight algorithm for exploration and exploitation. This module contains a class that implements EXP4.P, a contextual bandit algorithm with expert advice.

class `striatum.bandit.exp4p.`**Exp4P**(*actions*, *historystorage*, *modelstorage*, *delta=0.1*, *p_min=None*, *max_rounds=10000*)

Bases: `striatum.bandit.bandit.BaseBandit`

Exp4.P with pre-trained supervised learning algorithm.

Parameters **actions** : list of Action objects

List of actions to be chosen from.

historystorage: a HistoryStorage object

The place where we store the histories of contexts and rewards.

modelstorage: a ModelStorage object

The place where we store the model parameters.

delta: float, 0 < delta <= 1

With probability 1 - delta, LinThompSamp satisfies the theoretical regret bound.

p_min: float, 0 < p_min < 1/k

The minimum probability to choose each action.

References

[R2]

Attributes

`history_storage`

Methods

`add_action`(actions)	Add new actions (if needed).
`calculate_avg_reward`()	Calculate average reward with respect to time.
`calculate_cum_reward`()	Calculate cumulative reward with respect to time.
`get_action`([context, n_actions])	Return the action to perform
`plot_avg_regret`()	Plot average regret with respect to time.
`plot_avg_reward`()	Plot average reward with respect to time.
`reward`(history_id, rewards)	Reward the previous action with reward.

get_action(*context=None*, *n_actions=1*)

Return the action to perform

Parameters context : dictionary

Contexts {expert_id: {action_id: expert_prediction}} of different actions.

n_actions: int

Number of actions wanted to recommend users.

Returns history_id : int

The history id of the action.

action_recommendation : list of dictionaries

In each dictionary, it will contains {Action object, estimated_reward, uncertainty}.

reward(*history_id*, *rewards*)

Reward the previous action with reward.

Parameters history_id : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

1.1.4 striatum.bandit.linthompsamp module

Thompson Sampling with Linear Payoff In This module contains a class that implements Thompson Sampling with Linear Payoff. Thompson Sampling with linear payoff is a contexutal multi-armed bandit algorithm which assume the underlying relationship between rewards and contexts is linear. The sampling method is used to balance the exploration and exploitation. Please check the reference for more details.

class `striatum.bandit.linthompsamp.`**`LinThompSamp`**(*history_storage*, *model_storage*, *action_storage*, *context_dimension=128*, *delta=0.5*, *R=0.01*, *epsilon=0.5*, *random_state=None*)

Bases: `striatum.bandit.bandit.BaseBandit`

Thompson sampling with linear payoff.

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

delta: float, 0 < delta < 1

With probability 1 - delta, LinThompSamp satisfies the theoretical regret bound.

R: float, R >= 0

Assume that the residual $ri(t) - bi(t)^T \hat{\mu}$ is R-sub-gaussian. In this case, R^2 represents the variance for residuals of the linear model $bi(t)^T$.

epsilon: float, 0 < epsilon < 1

A parameter used by the Thompson Sampling algorithm. If the total trials T is known, we can choose epsilon = 1/ln(T).

References

[R3]

Attributes

`history_storage`

Methods

add_action(actions)	Add new actions (if needed).
calculate_avg_reward()	Calculate average reward with respect to time.
calculate_cum_reward()	Calculate cumulative reward with respect to time.
get_action(context[, n_actions])	Return the action to perform
plot_avg_regret()	Plot average regret with respect to time.
plot_avg_reward()	Plot average reward with respect to time.
reward(history_id, rewards)	Reward the previous action with reward.

add_action (*actions*)
Add new actions (if needed).

Parameters actions : iterable

A list of Action oBjects for recommendation

get_action (*context*, *n_actions=None*)
Return the action to perform

Parameters context : dictionary

Contexts {action_id: context} of different actions.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns history_id : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward (*history_id*, *rewards*)
Reward the previous action with reward.

Parameters history_id : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

1.1.5 striatum.bandit.linucb module

LinUCB with Disjoint Linear Models

This module contains a class that implements LinUCB with disjoint linear model, a contextual bandit algorithm assuming the reward function is a linear function of the context.

class striatum.bandit.linucb.**LinUCB** (*history_storage*, *model_storage*, *action_storage*, *context_dimension=128*, *alpha=0.5*)
Bases: striatum.bandit.bandit.BaseBandit

LinUCB with Disjoint Linear Models

Parameters history_storage : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

alpha: float

The constant determines the width of the upper confidence bound.

context_dimension: int

The dimension of the context.

References

[R4]

Attributes

`history_storage`

Methods

add_action(actions)	Add new actions (if needed).
`calculate_avg_reward()`	Calculate average reward with respect to time.
`calculate_cum_reward()`	Calculate cumulative reward with respect to time.
get_action(context[, n_actions])	Return the action to perform
`plot_avg_regret()`	Plot average regret with respect to time.
`plot_avg_reward()`	Plot average reward with respect to time.
reward(history_id, rewards)	Reward the previous action with reward.

add_action (*actions*)

Add new actions (if needed).

Parameters actions : iterable

A list of Action objects for recommendation

get_action (*context*, *n_actions=None*)

Return the action to perform

Parameters context : dict

Contexts {action_id: context} of different actions.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns history_id : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward (*history_id*, *rewards*)
Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

1.1.6 striatum.bandit.ucb1 module

Upper Confidence Bound 1 This module contains a class that implements UCB1 algorithm, a famous multi-armed bandit algorithm without context.

class `striatum.bandit.ucb1.`**`UCB1`** (*history_storage*, *model_storage*, *action_storage*)
Bases: `striatum.bandit.bandit.BaseBandit`

Upper Confidence Bound 1

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

References

[R5]

Attributes

`history_storage`

Methods

`add_action`(actions)	Add new actions (if needed).
`calculate_avg_reward`()	Calculate average reward with respect to time.
`calculate_cum_reward`()	Calculate cumulative reward with respect to time.
`get_action`([context, n_actions])	Return the action to perform
`plot_avg_regret`()	Plot average regret with respect to time.
`plot_avg_reward`()	Plot average reward with respect to time.
`reward`(history_id, rewards)	Reward the previous action with reward.

add_action (*actions*)
Add new actions (if needed).

Parameters **actions** : iterable

A list of Action objects for recommendation

get_action(*context=None*, *n_actions=None*)
Return the action to perform

Parameters **context** : {array-like, None}

The context of current state, None if no context available.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns **history_id** : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward(*history_id*, *rewards*)
Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

1.1.7 Module contents

Bandit algorithm classes

class `striatum.bandit.`**Exp3**(*history_storage*, *model_storage*, *action_storage*, *gamma=0.3*, *random_state=None*)
Bases: `striatum.bandit.bandit.BaseBandit`

Exp3 algorithm.

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

gamma: float, 0 < gamma <= 1

The parameter used to control the minimum chosen probability for each action.

References

[R6]

Attributes

`history_storage`

Methods

add_action(actions)	Add new actions (if needed).
`calculate_avg_reward()`	Calculate average reward with respect to time.
`calculate_cum_reward()`	Calculate cumulative reward with respect to time.
get_action([context, n_actions])	Return the action to perform
`plot_avg_regret()`	Plot average regret with respect to time.
`plot_avg_reward()`	Plot average reward with respect to time.
reward(history_id, rewards)	Reward the previous action with reward.

add_action(*actions*)

Add new actions (if needed).

Parameters **actions** : iterable

A list of Action objects for recommendation

get_action(*context=None*, *n_actions=None*)

Return the action to perform

Parameters **context** : {array-like, None}

The context of current state, None if no context available.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns **history_id** : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward(*history_id*, *rewards*)

Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

class `striatum.bandit.`**Exp4P**(*actions*, *historystorage*, *modelstorage*, *delta=0.1*, *p_min=None*, *max_rounds=10000*)

Bases: `striatum.bandit.bandit.BaseBandit`

Exp4.P with pre-trained supervised learning algorithm.

Parameters **actions** : list of Action objects

List of actions to be chosen from.

historystorage: a HistoryStorage object

The place where we store the histories of contexts and rewards.

modelstorage: a ModelStorage object

The place where we store the model parameters.

delta: float, 0 < delta <= 1

With probability 1 - delta, LinThompSamp satisfies the theoretical regret bound.

p_min: float, 0 < p_min < 1/k

The minimum probability to choose each action.

References

[R7]

Attributes

`history_storage`

Methods

`add_action`(actions)	Add new actions (if needed).
`calculate_avg_reward`()	Calculate average reward with respect to time.
`calculate_cum_reward`()	Calculate cumulative reward with respect to time.
`get_action`([context, n_actions])	Return the action to perform
`plot_avg_regret`()	Plot average regret with respect to time.
`plot_avg_reward`()	Plot average reward with respect to time.
`reward`(history_id, rewards)	Reward the previous action with reward.

`get_action`(*context=None*, *n_actions=1*)

Return the action to perform

Parameters **context** : dictionary

Contexts {expert_id: {action_id: expert_prediction}} of different actions.

n_actions: int

Number of actions wanted to recommend users.

Returns **history_id** : int

The history id of the action.

action_recommendation : list of dictionaries

In each dictionary, it will contains {Action object, estimated_reward, uncertainty}.

`reward`(*history_id*, *rewards*)

Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

class `striatum.bandit.`**`LinThompSamp`**(*history_storage*, *model_storage*, *action_storage*, *context_dimension=128*, *delta=0.5*, *R=0.01*, *epsilon=0.5*, *random_state=None*)

Bases: `striatum.bandit.bandit.BaseBandit`

Thompson sampling with linear payoff.

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

delta: float, 0 < delta < 1

With probability 1 - delta, LinThompSamp satisfies the theoretical regret bound.

R: float, R >= 0

Assume that the residual $ri(t) - bi(t)^T \hat{\mu}$ is R-sub-gaussian. In this case, R^2 represents the variance for residuals of the linear model $bi(t)^T$.

epsilon: float, 0 < epsilon < 1

A parameter used by the Thompson Sampling algorithm. If the total trials T is known, we can choose epsilon = 1/ln(T).

References

[R8]

Attributes

`history_storage`

Methods

`add_action`(actions)	Add new actions (if needed).
`calculate_avg_reward`()	Calculate average reward with respect to time.
`calculate_cum_reward`()	Calculate cumulative reward with respect to time.
`get_action`(context[, n_actions])	Return the action to perform
`plot_avg_regret`()	Plot average regret with respect to time.
`plot_avg_reward`()	Plot average reward with respect to time.
`reward`(history_id, rewards)	Reward the previous action with reward.

add_action (*actions*)
Add new actions (if needed).

Parameters **actions** : iterable

A list of Action oBjects for recommendation

get_action (*context*, *n_actions=None*)
Return the action to perform

Parameters **context** : dictionary

Contexts {action_id: context} of different actions.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns **history_id** : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward (*history_id*, *rewards*)
Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

class `striatum.bandit.`**LinUCB** (*history_storage*, *model_storage*, *action_storage*, *context_dimension=128*, *alpha=0.5*)
Bases: `striatum.bandit.bandit.BaseBandit`

LinUCB with Disjoint Linear Models

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

alpha: float

The constant determines the width of the upper confidence bound.

context_dimension: int

The dimension of the context.

References

[R9]

Attributes

history_storage

Methods

add_action(actions)	Add new actions (if needed).
calculate_avg_reward()	Calculate average reward with respect to time.
calculate_cum_reward()	Calculate cumulative reward with respect to time.
get_action(context[, n_actions])	Return the action to perform
plot_avg_regret()	Plot average regret with respect to time.
plot_avg_reward()	Plot average reward with respect to time.
reward(history_id, rewards)	Reward the previous action with reward.

add_action(*actions*)

Add new actions (if needed).

Parameters **actions** : iterable

A list of Action objects for recommendation

get_action(*context*, *n_actions=None*)

Return the action to perform

Parameters **context** : dict

Contexts {action_id: context} of different actions.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns **history_id** : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward(*history_id*, *rewards*)

Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

class striatum.bandit.**UCB1**(*history_storage*, *model_storage*, *action_storage*)

Bases: `striatum.bandit.bandit.BaseBandit`

Upper Confidence Bound 1

Parameters **history_storage** : HistoryStorage object

The HistoryStorage object to store history context, actions and rewards.

model_storage : ModelStorage object

The ModelStorage object to store model parameters.

action_storage : ActionStorage object

The ActionStorage object to store actions.

References

[R10]

Attributes

`history_storage`

Methods

add_action(actions)	Add new actions (if needed).
`calculate_avg_reward()`	Calculate average reward with respect to time.
`calculate_cum_reward()`	Calculate cumulative reward with respect to time.
get_action([context, n_actions])	Return the action to perform
`plot_avg_regret()`	Plot average regret with respect to time.
`plot_avg_reward()`	Plot average reward with respect to time.
reward(history_id, rewards)	Reward the previous action with reward.

add_action(*actions*)

Add new actions (if needed).

Parameters **actions** : iterable

A list of Action objects for recommendation

get_action(*context=None*, *n_actions=None*)

Return the action to perform

Parameters **context** : {array-like, None}

The context of current state, None if no context available.

n_actions: int (default: None)

Number of actions wanted to recommend users. If None, only return one action. If -1, get all actions.

Returns **history_id** : int

The history id of the action.

recommendations : list of dict

Each dict contains {Action object, estimated_reward, uncertainty}.

reward(*history_id*, *rewards*)

Reward the previous action with reward.

Parameters **history_id** : int

The history id of the action to reward.

rewards : dictionary

The dictionary {action_id, reward}, where reward is a float.

1.2 striatum.storage package

1.2.1 Submodules

1.2.2 striatum.storage.model module

Model storage

class `striatum.storage.model.`**`MemoryModelStorage`**

Bases: *`striatum.storage.model.ModelStorage`*

Store the model in memory.

Methods

`get_model`()	
`save_model`(model)	

`get_model`()

`save_model`(*model*)

class `striatum.storage.model.`**`ModelStorage`**

Bases: `object`

The object to store the model.

Methods

`get_model`()	Get model
`save_model`()	Save model

`get_model`()

Get model

`save_model`()

Save model

1.2.3 striatum.storage.history module

History storage

class `striatum.storage.history.`**`History`**(*history_id*, *context*, *recommendations*, *created_at*, *rewards=None*, *rewarded_at=None*)

Bases: `object`

action/reward history entry.

Parameters **history_id** : int

context : {dict of list of float, None}

recommendations : {Recommendation, list of Recommendation}

created_at : datetime

rewards : {float, dict of float, None}

rewarded_at : {datetime, None}

Methods

update_reward(rewards, rewarded_at)	Update reward_time and rewards.

update_reward(*rewards*, *rewarded_at*)
Update reward_time and rewards.

Parameters **rewards** : {float, dict of float, None}

rewarded_at : {datetime, None}

class striatum.storage.history.**HistoryStorage**
Bases: object

The object to store the history of context, recommendations and rewards.

Methods

add_history(context, recommendations[, rewards])	Add a history record.
add_reward(history_id, rewards)	Add reward to a history record.
get_history(history_id)	Get the previous context, recommendations and rewards with history_id.
get_unrewarded_history(history_id)	Get the previous unrewarded context, recommendations and rewards with h

add_history(*context*, *recommendations*, *rewards=None*)
Add a history record.

Parameters **context** : {dict of list of float, None}

recommendations : {Recommendation, list of Recommendation}

rewards : {float, dict of float, None}

add_reward(*history_id*, *rewards*)
Add reward to a history record.

Parameters **history_id** : int

The history id of the history record to retrieve.

rewards : {float, dict of float, None}

get_history(*history_id*)
Get the previous context, recommendations and rewards with history_id.

Parameters **history_id** : int

The history id of the history record to retrieve.

Returns history: History

get_unrewarded_history (*history_id*)
Get the previous unrewarded context, recommendations and rewards with history_id.

Parameters **history_id** : int

The history id of the history record to retrieve.

Returns history: History

class striatum.storage.history.**MemoryHistoryStorage**
Bases: *striatum.storage.history.HistoryStorage*

HistoryStorage that store all data in memory

Methods

add_history(context, recommendations[, rewards])	Add a history record.
add_reward(history_id, rewards)	Add reward to a history record.
get_history(history_id)	Get the previous context, recommendations and rewards with history_id.
get_unrewarded_history(history_id)	Get the previous unrewarded context, recommendations and rewards with h

add_history (*context*, *recommendations*, *rewards=None*)
Add a history record.

Parameters **context** : {dict of list of float, None}

recommendations : {Recommendation, list of Recommendation}

rewards : {float, dict of float, None}

add_reward (*history_id*, *rewards*)
Add reward to a history record.

Parameters **history_id** : int

The history id of the history record to retrieve.

rewards : {float, dict of float, None}

get_history (*history_id*)
Get the previous context, recommendations and rewards with history_id.

Parameters **history_id** : int

The history id of the history record to retrieve.

Returns history: History

get_unrewarded_history (*history_id*)
Get the previous unrewarded context, recommendations and rewards with history_id.

Parameters **history_id** : int

The history id of the history record to retrieve.

Returns history: History

1.2.4 Module contents

Storage classes

CHAPTER 2

Gallery of Examples

2.1 General examples

General-purpose and introductory examples from the sphinx-gallery

2.1.1 prerpocess MovieLens dataset

In this script, we pre-process the MovieLens 10M Dataset to get the right format of contextual bandit algorithms. This data set is released by GroupLens at 1/2009. Please fist download the dataset from http://grouplens.org/datasets/movielens/, then unzipped the file 'ml-1m.zip' to the examples folder.

```
import pandas as pd
import numpy as np
import itertools


def movie_preprocessing(movie):
    movie_col = list(movie.columns)
    movie_tag = [doc.split('|') for doc in movie['tag']]
    tag_table = {token: idx for idx, token in enumerate(set(itertools.chain.from_iterable(movie_tag))
    movie_tag = pd.DataFrame(movie_tag)
    tag_table = pd.DataFrame(tag_table.items())
    tag_table.columns = ['Tag', 'Index']

    # use one-hot encoding for movie genres (here called tag)
    tag_dummy = np.zeros([len(movie), len(tag_table)])

    for i in range(len(movie)):
        for j in range(len(tag_table)):
            if tag_table['Tag'][j] in list(movie_tag.iloc[i, :]):
                tag_dummy[i, j] = 1

    # combine the tag_dummy one-hot encoding table to original movie files
    movie = pd.concat([movie, pd.DataFrame(tag_dummy)], 1)
    movie_col.extend(['tag' + str(i) for i in range(len(tag_table))])
    movie.columns = movie_col
    movie = movie.drop('tag', 1)
    return movie


def feature_extraction(data):
```

```python
    # actions: we use top 50 movies as our actions for recommendations
    actions = data.groupby('movie_id').size().sort_values(ascending=False)[:50]
    actions = list(actions.index)

    # user_feature: tags they've watched for non-top-50 movies normalized per user
    user_feature = data[~data['movie_id'].isin(actions)]
    user_feature = user_feature.groupby('user_id').aggregate(np.sum)
    user_feature = user_feature.drop(['movie_id', 'rating', 'timestamp'], 1)
    user_feature = user_feature.div(user_feature.sum(axis=1), axis=0)

    # streaming_batch: the result for testing bandit algrorithms
    top50_data = data[data['movie_id'].isin(actions)]
    top50_data = top50_data.sort('timestamp', ascending=1)
    streaming_batch = top50_data['user_id']

    # reward_list: if rating >=3, the user will watch the movie
    top50_data['reward'] = np.where(top50_data['rating'] >= 3, 1, 0)
    reward_list = top50_data[['user_id', 'movie_id', 'reward']]
    reward_list = reward_list[reward_list['reward'] == 1]
    return streaming_batch, user_feature, actions, reward_list


def main():
    # read and preprocess the movie data
    movie = pd.read_table('movies.dat', sep='::', names=['movie_id', 'movie_name', 'tag'], engine='py
    movie = movie_preprocessing(movie)

    # read the ratings data and merge it with movie data
    rating = pd.read_table("ratings.dat", sep="::",
                           names=["user_id", "movie_id", "rating", "timestamp"], engine='python')
    data = pd.merge(rating, movie, on="movie_id")

    # extract feature from our data set
    streaming_batch, user_feature, actions, reward_list = feature_extraction(data)
    streaming_batch.to_csv("streaming_batch.csv", sep='\t', index=False)
    user_feature.to_csv("user_feature.csv", sep='\t')
    pd.DataFrame(actions, columns=['movie_id']).to_csv("actions.csv", sep='\t', index=False)
    reward_list.to_csv("reward_list.csv", sep='\t', index=False)

    action_context = movie[movie['movie_id'].isin(actions)]
    action_context.to_csv("action_context.csv", sep='\t', index = False)

if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

`Download Python source code: movielens_preprocess.py`

`Download Jupyter notebook: movielens_preprocess.ipynb`

Generated by Sphinx-Gallery

2.1.2 Contextual bandit on MovieLens

The script uses real-world data to conduct contextual bandit experiments. Here we use MovieLens 10M Dataset, which is released by GroupLens at 1/2009. Please fist pre-process datasets (use "movielens_preprocess.py"), and then you can run this example.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from striatum.storage import history
from striatum.storage import model
from striatum.bandit import ucb1
from striatum.bandit import linucb
from striatum.bandit import linthompsamp
from striatum.bandit import exp4p
from striatum.bandit import exp3
from striatum.bandit.bandit import Action
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.multiclass import OneVsRestClassifier


def get_data():
    streaming_batch = pd.read_csv('streaming_batch.csv', sep='\t', names=['user_id'], engine='c')
    user_feature = pd.read_csv('user_feature.csv', sep='\t', header=0, index_col=0, engine='c')
    actions_id = list(pd.read_csv('actions.csv', sep='\t', header=0, engine='c')['movie_id'])
    reward_list = pd.read_csv('reward_list.csv', sep='\t', header=0, engine='c')
    action_context = pd.read_csv('action_context.csv', sep='\t', header=0, engine='c')

    actions = []
    for key in actions_id:
        action = Action(key)
        actions.append(action)
    return streaming_batch, user_feature, actions, reward_list, action_context


def train_expert(action_context):
    logreg = OneVsRestClassifier(LogisticRegression())
    mnb = OneVsRestClassifier(MultinomialNB(), )
    logreg.fit(action_context.iloc[:, 2:], action_context.iloc[:, 1])
    mnb.fit(action_context.iloc[:, 2:], action_context.iloc[:, 1])
    return [logreg, mnb]


def get_advice(context, actions_id, experts):
    advice = {}
    for time in context.keys():
        advice[time] = {}
        for i in range(len(experts)):
            prob = experts[i].predict_proba(context[time])[0]
            advice[time][i] = {}
            for j in range(len(prob)):
                advice[time][i][actions_id[j]] = prob[j]
    return advice


def policy_generation(bandit, actions):
    historystorage = history.MemoryHistoryStorage()
    modelstorage = model.MemoryModelStorage()

    if bandit == 'Exp4P':
        policy = exp4p.Exp4P(actions, historystorage, modelstorage, delta=0.5, pmin=None)

    elif bandit == 'LinUCB':
```

```
        policy = linucb.LinUCB(actions, historystorage, modelstorage, 0.3, 20)

    elif bandit == 'LinThompSamp':
        policy = linthompsamp.LinThompSamp(actions, historystorage, modelstorage,
                                           d=20, delta=0.61, r=0.01, epsilon=0.71)

    elif bandit == 'UCB1':
        policy = ucb1.UCB1(actions, historystorage, modelstorage)

    elif bandit == 'Exp3':
        policy = exp3.Exp3(actions, historystorage, modelstorage, gamma=0.2)

    elif bandit == 'random':
        policy = 0

    return policy


def policy_evaluation(policy, bandit, streaming_batch, user_feature, reward_list, actions, action_con
    times = len(streaming_batch)
    seq_error = np.zeros(shape=(times, 1))
    actions_id = [actions[i].action_id for i in range(len(actions))]
    if bandit in ['LinUCB', 'LinThompSamp', 'UCB1', 'Exp3']:
        for t in range(times):
            feature = np.array(user_feature[user_feature.index == streaming_batch.iloc[t, 0]])[0]
            full_context = {}
            for action_id in actions_id:
                full_context[action_id] = feature
            history_id, action = policy.get_action(full_context, 1)
            watched_list = reward_list[reward_list['user_id'] == streaming_batch.iloc[t, 0]]

            if action[0]['action'].action_id not in list(watched_list['movie_id']):
                policy.reward(history_id, {action[0]['action'].action_id: 0.0})
                if t == 0:
                    seq_error[t] = 1.0
                else:
                    seq_error[t] = seq_error[t - 1] + 1.0

            else:
                policy.reward(history_id, {action[0]['action'].action_id: 1.0})
                if t > 0:
                    seq_error[t] = seq_error[t - 1]

    elif bandit == 'Exp4P':
        for t in range(times):
            feature = user_feature[user_feature.index == streaming_batch.iloc[t, 0]]
            experts = train_expert(action_context)
            advice = {}
            for i in range(len(experts)):
                prob = experts[i].predict_proba(feature)[0]
                advice[i] = {}
                for j in range(len(prob)):
                    advice[i][actions_id[j]] = prob[j]
            history_id, action = policy.get_action(advice)
            watched_list = reward_list[reward_list['user_id'] == streaming_batch.iloc[t, 0]]

            if action[0]['action'].action_id not in list(watched_list['movie_id']):
                policy.reward(history_id, {action[0]['action'].action_id: 0.0})
```

```
                if t == 0:
                    seq_error[t] = 1.0
                else:
                    seq_error[t] = seq_error[t - 1] + 1.0

            else:
                policy.reward(history_id, {action[0]['action'].action_id: 1.0})
                if t > 0:
                    seq_error[t] = seq_error[t - 1]

    elif bandit == 'random':
        for t in range(times):
            action = actions_id[np.random.randint(0, len(actions)-1)]
            watched_list = reward_list[reward_list['user_id'] == streaming_batch.iloc[t, 0]]

            if action not in list(watched_list['movie_id']):
                if t == 0:
                    seq_error[t] = 1.0
                else:
                    seq_error[t] = seq_error[t - 1] + 1.0

            else:
                if t > 0:
                    seq_error[t] = seq_error[t - 1]

    return seq_error


def regret_calculation(seq_error):
    t = len(seq_error)
    regret = [x / y for x, y in zip(seq_error, range(1, t + 1))]
    return regret


def main():
    streaming_batch, user_feature, actions, reward_list, action_context = get_data()
    streaming_batch_small = streaming_batch.iloc[0:10000]

    # conduct regret analyses
    experiment_bandit = ['LinUCB', 'LinThompSamp', 'Exp4P', 'UCB1', 'Exp3', 'random']
    regret = {}
    col = ['b', 'g', 'r', 'c', 'm', 'y', 'k', 'w']
    i = 0
    for bandit in experiment_bandit:
        policy = policy_generation(bandit, actions)
        seq_error = policy_evaluation(policy, bandit, streaming_batch_small, user_feature, reward_lis
                                      actions, action_context)
        regret[bandit] = regret_calculation(seq_error)
        plt.plot(range(len(streaming_batch_small)), regret[bandit], c=col[i], ls='-', label=bandit)
        plt.xlabel('time')
        plt.ylabel('regret')
        plt.legend(bbox_to_anchor=(1.05, 1), loc=2, borderaxespad=0.)
        axes = plt.gca()
        axes.set_ylim([0, 1])
        plt.title("Regret Bound with respect to T")
        i += 1
    plt.show()
```

```
if __name__ == '__main__':
    main()
```

Total running time of the script: (0 minutes 0.000 seconds)

`Download Python source code: movielens_bandit.py`

`Download Jupyter notebook: movielens_bandit.ipynb`

Generated by Sphinx-Gallery

`Download all examples in Python source code: auto_examples_python.zip`

`Download all examples in Jupyter notebooks: auto_examples_jupyter.zip`

Generated by Sphinx-Gallery

- genindex
- modindex
- search

Bibliography

[R1] Peter Auer, Nicolo Cesa-Bianchi, et al. "The non-stochastic multi-armed bandit problem ." SIAM Journal of Computing. 2002.

[R2] Beygelzimer, Alina, et al. "Contextual bandit algorithms with supervised learning guarantees." International Conference on Artificial Intelligence and Statistics (AISTATS). 2011u.

[R3] Shipra Agrawal, and Navin Goyal. "Thompson Sampling for Contextual Bandits with Linear Payoffs." Advances in Neural Information Processing Systems 24. 2011.

[R4] Lihong Li, et al. "A Contextual-Bandit Approach to Personalized News Article Recommendation." In Proceedings of the 19th International Conference on World Wide Web (WWW), 2010.

[R5] Peter Auer, et al. "Finite-time Analysis of the Multiarmed Bandit Problem." Machine Learning, 47. 2002.

[R6] Peter Auer, Nicolo Cesa-Bianchi, et al. "The non-stochastic multi-armed bandit problem ." SIAM Journal of Computing. 2002.

[R7] Beygelzimer, Alina, et al. "Contextual bandit algorithms with supervised learning guarantees." International Conference on Artificial Intelligence and Statistics (AISTATS). 2011u.

[R8] Shipra Agrawal, and Navin Goyal. "Thompson Sampling for Contextual Bandits with Linear Payoffs." Advances in Neural Information Processing Systems 24. 2011.

[R9] Lihong Li, et al. "A Contextual-Bandit Approach to Personalized News Article Recommendation." In Proceedings of the 19th International Conference on World Wide Web (WWW), 2010.

[R10] Peter Auer, et al. "Finite-time Analysis of the Multiarmed Bandit Problem." Machine Learning, 47. 2002.

Python Module Index

S

A

E

G

H

L

M

R

S

U