Ierot Documentation

Release 2.0

uva

June 24, 2016

Contents

CHAPTER 1

Packages

1.1 lerot.analysis

class lerot.analysis.**HeatmapAnalysis**(**parms*)
Bases: lerot.analysis.AbstractAnalysis.AbstractAnalysis

finish()

class lerot.analysis.**SummarizeAnalysis**(**parms*)
Bases: lerot.analysis.AbstractAnalysis.AbstractAnalysis

finish()

1.2 lerot.comparison

class lerot.comparison.**BalancedInterleave**(*arg_str=None*)
Bases: lerot.comparison.AbstractInterleavedComparison.AbstractInterleavedComparison

Interleave and compare rankers using the original balanced interleave method.

infer_outcome(*l*, *a*, *c*, *query*)

interleave(*r1*, *r2*, *query*, *length*)

class lerot.comparison.**DocumentConstraints**(*arg_str='random'*)
Bases: lerot.comparison.AbstractInterleavedComparison.AbstractInterleavedComparison

Interleave using balanced interleave, compare using document constraints.

check_constraints(*l*, *a*, *click_ids*)

infer_outcome(*l*, *a*, *c*, *query*)

interleave(*r1*, *r2*, *query*, *length*)

class lerot.comparison.**HistBalancedInterleave**(*arg_str=None*)
Bases: lerot.comparison.AbstractHistInterleavedComparison.AbstractHistInterleavedComparis

Balanced interleave method, applied to historical data.

infer_outcome(*l*, *a*, *c*, *target_r1*, *target_r2*, *query*)
count clicks within the top-k interleaved list

class lerot.comparison.**HistDocumentConstraints**(*arg_str=None*)
Bases: lerot.comparison.AbstractHistInterleavedComparison.AbstractHistInterleavedCompari

Document constraints method, applied to historical data.

infer_outcome(*l*, *a*, *c*, *target_r1*, *target_r2*, *query*)
count clicks within the top-k interleaved list

class lerot.comparison.**HistProbabilisticInterleave**(*arg_str=None*)
Bases: lerot.comparison.AbstractHistInterleavedComparison.AbstractHistInterleavedCompari

Probabilistic interleaving using historical data

infer_outcome(*l*, *source_context*, *c*, *target_r1*, *target_r2*, *query*)

class lerot.comparison.**HistTeamDraft**(*arg_str=None*)
Bases: lerot.comparison.AbstractHistInterleavedComparison.AbstractHistInterleavedCompari

Team draft method, applied to historical data.

infer_outcome(*l*, *a*, *c*, *target_r1*, *target_r2*, *query*)
assign clicks for contributed documents

class lerot.comparison.**OptimizedInterleave**(*arg_str=''*)
Bases: lerot.comparison.AbstractInterleavedComparison.AbstractInterleavedComparison

An implementation of Optimized Interleave as described in:

@see: Radlinski, F., & Craswell, N. (2013, February). Optimized interleaving for online retrieval evaluation. In Proceedings of the sixth ACM international conference on Web search and data mining (pp. 245-254).

@author: Anne Schuth @contact: anne.schuth@uva.nl @since: February 2013 @requires: Gurobi from http://www.gurobi.com/

binary_credit(*li*, *rankA*, *rankB*)

f(*i*)

infer_outcome(*l*, *credit*, *clicks*, *query*)

interleave(*r1*, *r2*, *query*, *length*, *bias=0*)

interleave_n(*r1*, *r2*, *query*, *length*, *num_repeat*, *bias=0*)

inverse_credit(*li*, *rankA*, *rankB*)

linear_credit(*li*, *rankA*, *rankB*)

perm_given_index(*alist*, *apermindex*)
See http://stackoverflow.com/questions/5602488/random-picks-from-permutation-generator

precompute_rank(*R*)

prefix_constraint(*rankings*, *length*)

prefix_constraint_bound(*rankings*, *length*, *prefix_bound*)

rank(*li*, *R*)

reject(*l*, *rankings*)

sample(*docs*, *length*)

sample_prefix_constraint(*rankings*, *length*)

sample_prefix_constraint_constructive(*rankings*, *length*)

class lerot.comparison.**OptimizedInterleaveVa**(*arg_str=None*)
Bases: lerot.comparison.OptimizedInterleave.OptimizedInterleave

precompute_rank_va(*R*)

prefix_constraint_va(*rankings*, *length*)

class lerot.comparison.**ProbabilisticInterleave**(*arg_str=None*)
Bases: lerot.comparison.AbstractInterleavedComparison.AbstractInterleavedComparison

Probabilistic interleaving, marginalizes over assignments

get_probability_of_list(*result_list*, *context*, *query*)

infer_outcome(*l*, *a*, *c*, *query*)

interleave(*r1*, *r2*, *query*, *length*)

class lerot.comparison.**ProbabilisticInterleaveWithHistory**(*arg_str*)
Bases: lerot.comparison.ProbabilisticInterleave.ProbabilisticInterleave

Probabilistic interleaving that reuses historic data (with importance sampling).

infer_outcome(*l*, *context*, *c*, *query*)

class lerot.comparison.**StochasticBalancedInterleave**(*arg_str*)
Bases: lerot.comparison.AbstractInterleavedComparison.AbstractInterleavedComparison

Interleave and compare rankers using the stochastic interleave method introduced in Hofmann et al. ECIR'11.

infer_outcome(*l*, *a*, *c*, *query*)

interleave(*r1*, *r2*, *query*, *length*)

class lerot.comparison.**TeamDraft**(*arg_str=None*)
Bases: lerot.comparison.AbstractInterleavedComparison.AbstractInterleavedComparison

Baseline team draft method.

infer_outcome(*l*, *a*, *c*, *query*)
assign clicks for contributed documents

interleave(*r1*, *r2*, *query*, *length1=None*)
updated to match the original method

class lerot.comparison.**VaTdi**(*arg_str=None*)
Bases: lerot.comparison.TeamDraft.TeamDraft

Algorithm described in https://bitbucket.org/varepsilon/tois2013-interleaving

interleave(*r1*, *r2*, *query*, *length=None*)

static sampleSmoothly(*a*, *b*, *maxVal*)

1.3 lerot.document

class lerot.document.**Document**(*docid*, *doctype='Web'*)
Bases: object

get_id()

get_type()

set_type(*doctype*)

1.4 lerot.evaluation

class lerot.evaluation.**AsRbpEval**(*alpha=10*, *beta=0.8*)
Bases: lerot.evaluation.AbstractEval.AbstractEval

Compute AS_RBP metric as described in [1].

[1] Zhou, K. et al. 2012. Evaluating aggregated search pages. SIGIR. (2012).

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*)

class lerot.evaluation.**DcgEval**
Bases: lerot.evaluation.AbstractEval.AbstractEval

Compute DCG (with gain = 2**rel-1 and log2 discount).

evaluate_ranking(*ranking*, *query*, *cutoff=-1*)
Compute DCG for the provided ranking. The ranking is expected to contain document ids in rank order.

get_dcg(*ranked_labels*, *cutoff=-1*)
Get the dcg value of a list ranking. Does not check if the numer for ranked labels is smaller than cutoff.

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*)
Compute the value of the metric - ranking contains the list of documents to evaluate - labels are the relevance labels for all the documents, even those

that are not in the ranking; labels[doc.get_id()] is the relevance of doc

•orientations contains orientation values for the verticals; orientations[doc.get_type()] is the orientation value for the doc (from 0 to 1).

class lerot.evaluation.**NdcgEval**
Bases: lerot.evaluation.DcgEval.DcgEval

Compute NDCG (with gain = 2**rel-1 and log2 discount).

evaluate_ranking(*ranking*, *query*, *cutoff=-1*)
Compute NDCG for the provided ranking. The ranking is expected to contain document ids in rank order.

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*)

class lerot.evaluation.**LetorNdcgEval**
Bases: lerot.evaluation.NdcgEval.NdcgEval

Compute NDCG as implemented in the Letor toolkit.

get_dcg(*labels*, *cutoff=-1*)

class lerot.evaluation.**VSEval**
Bases: lerot.evaluation.AbstractEval.AbstractEval

Simple vertical selection (VS) metric, a.k.a. prec_v.

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*)

class lerot.evaluation.**VDEval**
Bases: lerot.evaluation.AbstractEval.AbstractEval

Simple vertical selection (VD) metric, a.k.a. rec_v.

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*)

class lerot.evaluation.**ISEval**
Bases: lerot.evaluation.AbstractEval.AbstractEval

Simple vertical selection (IS) metric, a.k.a. mean-prec.

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*)

class lerot.evaluation.**RPEval**
Bases: lerot.evaluation.AbstractEval.AbstractEval

Simple vertical selection (RP) metric, a.k.a. corr.

get_value(*ranking*, *labels*, *orientations*, *cutoff=-1*, *ideal_ranking=None*)

class lerot.evaluation.**LivingLabsEval**

get_performance()

get_win()

update_score(*wins*)

class lerot.evaluation.**PAKEval**
Bases: lerot.evaluation.AbstractEval.AbstractEval

Precision at k evaluation. Relevant document in ranking up to index k

evaluate_ranking(*ranking*, *query*, *cutoff=-1*)

1.5 lerot.environment

class lerot.environment.**CascadeUserModel**(*arg_str*)
Bases: lerot.environment.AbstractUserModel.AbstractUserModel

Defines a cascade user model, simulating a user that inspects results starting from the top of a result list.

get_clicks(*result_list*, *labels*, ***kwargs*)
simulate clicks on list l

class lerot.environment.**FederatedClickModel**(*arg_str*)
Bases: lerot.environment.AbstractUserModel.AbstractUserModel

b(*i*, *vert*)

static getParamRescaled(*rank*, *serp_len*, *param_vector*)

static getVertClass(*vert_type*)

get_clicks(*result_list*, *labels*, ***kwargs*)
Simulate clicks on the result_list. - labels contain relevance labels indexed by the docid

get_examination_prob(*result_list*, ***kwargs*)

h(*i*, *serp_len*, *vert*)

p(*i*, *serp_len*)

class lerot.environment.**PositionBasedUserModel**(*p*)
Bases: lerot.environment.AbstractUserModel.AbstractUserModel

Defines a positions based user model.

get_clicks(*result_list*, *labels*, ***kwargs*)
simulate clicks on list l

get_examination_prob(*result_list*, ***kwargs*)

p(*i*)

class lerot.environment.**RandomClickModel**(*p=0.5*)
Bases: lerot.environment.AbstractUserModel.AbstractUserModel

Defines a positions based user model.

get_clicks(*result_list*, *labels*, ***kwargs*)
simulate clicks on list l

class lerot.environment.**LivingLabsRealUser**(*key*, *doc_ids*)
Bases: lerot.environment.AbstractUserModel.AbstractUserModel

KEY = ''

get_clicks(*result_list*, *labels*, ***kwargs*)

get_win(*query*, *feedback_list*, *lerot_ranked_list*)
Used for seznam site which interleaves ranked list with it's own list Returns 'ranked list winner' with number of clicks of each ranker e.g. [0 2] where [lerot_list_score seznam_list_score]

runs = {}

upload_run(*query*, *upload_list*, *runid*)
Uploads a run to living-labs api.

class lerot.environment.**RelevantUserModel**(*arg_str*)
Bases: lerot.environment.AbstractUserModel.AbstractUserModel

Defines a user model that clicks on all relevant documents in a list with an optional limit

get_clicks(*result_list*, *labels*, ***kwargs*)

1.6 lerot.experiment

class lerot.experiment.**GenericExperiment**(*args_str=None*)

run()

run_experiment(*aux_log_fh*)

class lerot.experiment.**LearningExperiment**(*training_queries*, *test_queries*, *feature_count*, *log_fh*, *args*)
Bases: lerot.experiment.AbstractLearningExperiment.AbstractLearningExperiment

Represents an experiment in which a retrieval system learns from implicit user feedback. The experiment is initialized as specified in the provided arguments, or config file.

run()
A single run of the experiment.

class lerot.experiment.**MetaExperiment**

apply(*conf*)

finish_analytics()

run_celery()

run_conf()

run_local()

store(*conf*, *r*)

update_analytics()

update_analytics_file(*log_file*)

class lerot.experiment.**PrudentLearningExperiment**(*training_queries*, *test_queries*, *feature_count*, *log_fh*, *args*)
Bases: lerot.experiment.AbstractLearningExperiment.AbstractLearningExperiment

Represents an experiment in which a retrieval system learns from implicit user feedback. The experiment is initialized as specified in the provided arguments, or config file.

run()
Run the experiment num_runs times.

class lerot.experiment.**HistoricalComparisonExperiment**(*queries*, *feature_count*, *log_fh*, *args*)
Represents an experiment in which rankers are compared using interleaved comparisons with live and historic click data.

run()
Run the experiment for num_queries queries.

class lerot.experiment.**SingleQueryComparisonExperiment**(*query_dir*, *feature_count*, *log_fh*, *args*)
Represents an experiment in which rankers are compared using interleaved comparisons on a single query.

run()
Run the experiment for num_queries queries.

class lerot.experiment.**SyntheticComparisonExperiment**(*log_fh*, *args*)
Represents an experiment in which synthetic rankers are compared to investigate theoretical properties / guarantees.

run()
Run the experiment for num_queries queries.

class lerot.experiment.**VASyntheticComparisonExperiment**(*log_fh*, *args*)
Represents an experiment in which synthetic rankers are compared to investigate theoretical properties / guarantees.

static block_counts(*l*)

static block_position1(*l*, *result_length*)

static block_sizes(*l*)

static generate_ranking_pair(*result_length*, *num_relevant*, *pos_method='beyondten'*, *vert_rel='non-relevant'*, *block_size=3*, *verticals=None*, *fixed=False*, *dominates=<function <lambda>>*)
Generate pair of synthetic rankings. Appendix A, https://bitbucket.org/varepsilon/tois2013-interleaving

static get_online_metrics(*clicks*, *ranking*)

init_rankers(*query*)
Init rankers for a query

Since the ranker may be stateful, we need to init it every time we access its documents.

run()

1.7 lerot.ranker

class lerot.ranker.**DeterministicRankingFunction**(*ranker_arg_str*, *ties*, *feature_count*, *init='random'*, *sample='sample_unit_sphere'*)

Bases: lerot.ranker.AbstractRankingFunction.AbstractRankingFunction

document_count()

getDocs(*numdocs=None*)
Copied from StatelessRankingFunction.

get_document_probability(*docid*)
get probability of producing doc as the next document drawn

init_ranking(*query*)

next()
produce the next document

next_det()

next_random()
produce a random next document

rm_document(*docid*)
remove doc from list of available docs and adjust probabilities

class lerot.ranker.**ModelRankingFunction**

Bases: lerot.ranker.StatelessRankingFunction.StatelessRankingFunction

add_doc_for_query(*query*, *doc*)

init_ranking(*query*)

update_weights(*new_weights*)

class lerot.ranker.**ProbabilisticRankingFunction**(*ranker_arg_str*, *ties*, *feature_count*, *init='random'*, *sample='sample_unit_sphere'*)

Bases: lerot.ranker.AbstractRankingFunction.AbstractRankingFunction

document_count()

getDocs(*numdocs=None*)
Copied from StatelessRankingFunction.

get_document_probability(*docid*)
get probability of producing doc as the next document drawn

get_ranking()

init_ranking(*query*)

next()
produce the next document by random sampling, or deterministically

next_det()

next_random()
produce a random next document

rm_document(*docid*)
remove doc from list of available docs and adjust probabilities

class lerot.ranker.**StatelessRankingFunction**(*ranker_arg_str*, *ties*, *feature_count*, *init='random'*, *sample='sample_unit_sphere'*)
Bases: lerot.ranker.AbstractRankingFunction.AbstractRankingFunction

document_count()

getDocs(*numdocs=None*)
More efficient and less error-prone version of getDocs.

init_ranking(*query*)
Initialize ranking for particular query.

Since AbstractRankingFunction has a next() function that changes a state, we need to have a support for that. You need to set self.docs and the only stateful object self.doc_idx

next()

next_det()

next_random()

rm_document(*doc*)

verticals(*length=None*)

class lerot.ranker.**SyntheticDeterministicRankingFunction**(*synthetic_docs*)
Bases: lerot.ranker.StatelessRankingFunction.StatelessRankingFunction

Synthetic deterministic ranker.

get_document_probability(*doc*)
Get probability of producing doc as the next document drawn.

init_ranking(*query*)

update_weights(*new_weights*)

class lerot.ranker.**SyntheticProbabilisticRankingFunction**(*ranker_arg_str*, *ties='random'*)
Bases: lerot.ranker.ProbabilisticRankingFunction.ProbabilisticRankingFunction

Synthetic ranker for use in this experiment only

get_document_probability(*docid*)
get probability of producing doc as the next document drawn

init_ranking(*synthetic_docids*)

rm_document(*docid*)
remove doc from list of available docs, adjust probabilities

update_weights(*new_weights*)
not required under synthetic data

1.8 lerot.retrieval_system

class lerot.retrieval_system.**ListwiseLearningSystem**(*feature_count*, *arg_str*)
Bases: lerot.retrieval_system.AbstractLearningSystem.AbstractLearningSystem

A retrieval system that learns online from listwise comparisons. The system keeps track of all necessary state variables (current query, weights, etc.) so that comparison and learning classes can be stateless (implement only static / class methods).

get_ranked_list(*query*, *getNewCandidate=True*)

get_solution()

update_solution(*clicks*)

class lerot.retrieval_system.**PrudentListwiseLearningSystem**(*feature_count*, *arg_str*)
Bases: lerot.retrieval_system.AbstractLearningSystem.AbstractLearningSystem

A retrieval system that learns online from listwise comparisons. The system keeps track of all necessary state variables (current query, weights, etc.) so that comparison and learning classes can be stateless (implement only static / class methods).

get_outcome(*clicks*)

get_ranked_list(*query*, *getNewCandidate=True*)

get_solution()

update_solution()

class lerot.retrieval_system.**ListwiseLearningSystemWithCandidateSelection**(*feature_count*, *arg_str*)
Bases: lerot.retrieval_system.ListwiseLearningSystem.ListwiseLearningSystem

A retrieval system that learns online from listwise comparisons, and pre-selects exploratory rankers using historic data.

select_candidate_beat_the_mean(*candidate_us*)

select_candidate_random(*candidates*)

select_candidate_repeated(*candidates*)
Selects a ranker in randomized matches. Ranker pairs are sampled uniformly and compared over a number of historical samples. The outcomes observed over these samples are averaged (with / without importance sampling). The worse-performing ranker is removed from the pool. If no preference is found, the ranker to be removed is selected randomly. The final ranker in the pool is returned. This selection method assumes transitivity.

select_candidate_simple(*candidates*)
Selects a ranker in randomized matches. For each historic data point two rankers are randomly selected from the pool and compared. If a ranker loses the comparison, it is removed from the pool. If there is more than one ranker left when the history is exhausted, a ranker is randomly selected from the remaining pool. This selection method assumes transitivity (a ranker that loses against one ranker is assumed to not be the best ranker).

class lerot.retrieval_system.**PairwiseLearningSystem**(*feature_count*, *arg_str*)
Bases: lerot.retrieval_system.AbstractLearningSystem.AbstractLearningSystem

A retrieval system that learns online from pairwise comparisons. The system keeps track of all necessary state variables (current query, weights, etc.).

get_ranked_list(*query*)

get_solution()

initialize_weights(*method*, *feature_count*)

sample_fixed(*n*)

sample_unit_sphere(*n*)
See http://mathoverflow.net/questions/24688/efficiently-sampling- points-uniformly-from-the-surface-of-an-n-sphere

update_solution(*clicks*)
"Ranker weights are updated after each observed document pair. This means that a pair may have been

misranked when the result list was gen- erated, but is correctly labeled after an earlier update based on a higher-ranked pair from the same list.

class lerot.retrieval_system.**SamplerSystem**(*feature_count*, *arg_str*, *run_count=''*)
Bases: lerot.retrieval_system.AbstractLearningSystem.AbstractLearningSystem

get_ranked_list(*query*)

get_solution()

update_solution(*clicks*)

class lerot.retrieval_system.**PerturbationLearningSystem**(*feature_count*, *arg_str*)
Bases: lerot.retrieval_system.AbstractLearningSystem.AbstractLearningSystem

A retrieval system that learns online from pairwise comparisons. The system keeps track of all necessary state variables (current query, weights, etc.) so that comparison and learning classes can be stateless (implement only static / class methods).

get_ranked_list(*query*)

get_solution()

update_solution(*clicks*)
Update the ranker weights

while keeping in mind that documents with a relevance of > 1 are clicked more than once

update_solution_once(*clicks*)
Update the ranker weights without regard to multiple clicks on a single link

1.9 lerot.query

Interface to query data with functionality for reading queries from svmlight format, both sequentially and in batch mode.

class lerot.query.**Query**(*qid*, *feature_vectors*, *labels=None*, *comments=None*)

get_comment(*docid*)

get_comments()

get_docids()

get_document_count()

get_feature_vector(*docid*)

get_feature_vectors()

get_ideal()

get_label(*docid*)

get_labels()

get_prediction(*docid*)

get_predictions()

get_qid()

has_ideal()

set_feature_vector(*docid*, *feature_vector*)

set_ideal(*ideal*)

set_label(*docid*, *label*)

set_labels(*labels*)

set_predictions(*predictions*)

write_to(*fh*, *sparse=False*)

class lerot.query.**Queries**(*fh*, *num_features*, *preserve_comments=False*)
a list of queries with some convenience functions

get_feature_vectors()

get_labels()

get_predictions()

get_qids()

get_query(*index*)

get_size()

keys()

set_predictions()

values()

class lerot.query.**QueryStream**(*fh*, *num_features*, *preserve_comments=False*)
iterate over a stream of queries, only keeping one query at a time

next()

read_all()

lerot.query.**load_queries**(*filename*, *features*, *preserve_comments=False*)
Utility method for loading queries from a file.

lerot.query.**write_queries**(*filename*, *queries*)
Utility method for writing queries to a file. Returns the number of queries written

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

l

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W