COMP90051 Statistical Machine Learning
Project 2 Description

Due date: 4:00pm Thursday, 17th October 2019 **Weight:** 25%[1]

Multi-armed bandits (MABs) are a powerful tool in statistical machine learning: they bridge decision making, control, optimisation and learning; they address practical problems of sequential decision making while backed by elegant theoretical guarantees; they are relatively easily implemented, efficient to run, and are used in many industrial applications. They are neither fully supervised nor unsupervised, being partial supervised by indirect rewards—as a subroutine they employ supervised learning for predicting future rewards. *Exploitation* behaviour in MABs optimises short-term rewards by acting greedily based on current knowledge; but this must be balanced against imprecision in knowledge by *exploration*; and when effectively balanced, MABs optimise for long-term cumulative reward. In this project, you will work *individually* (not in teams) to implement several MAB learners. Some will be directly from class, while others will be more advanced and come out of papers that you will have to read and understand yourself.

By the end of the project you should have developed

ILO1. A deeper understanding of the MAB setting and common MAB approaches, and an appreciation of how MABs are applied;

ILO2. Better understanding of how the Bayesian paradigm can support machine learning;

ILO3. Demonstrable ability to implement ML approaches in code; and

ILO4. An ability to pick up recent machine learning publications in the literature, understand their focus, contributions, and algorithms enough to be able to implement and apply them. (And being able to ignore other presented details not needed for your task.)

Overview

Through the 2000s Yahoo! Research led the way in applying MABs to problems in online advertising, information retrieval, and media recommendation. One of their many applications was to Yahoo! News, in deciding what news items to recommend to users based on article content, user profile, and the historical engagement of the user with articles. Given decision making in this setting is sequential—what do we show next—and feedback is only available for articles shown, Yahoo! researchers observed a perfect formulation for MABs like those (ϵ-Greedy and UCB) learned about in class. Going further, however, they realised that incorporating some element of user-article state requires *contextual bandits*: articles are arms; context per round incorporates information about both user and article (arm); and $\{0, 1\}$-valued rewards represent clicks. Therefore the per round cumulative reward represents click-through-rate (CTR) which is exactly what services like Yahoo! News want to maximise to drive user engagement and advertising revenue. You will be implementing these approaches, noting that you need not necessarily complete the entire project.

Required Resources The LMS page for project 2 comprises

- `project2.pdf` this spec;
- `proj2.ipynb` Jupyter notebook: skeleton in Python; and
- `dataset.txt` A text-file dataset (see below for details).

[1] Forming a combined hurdle with project 1.

You will implement code in Python Jupyter notebooks, which after running on your machine you will submit via LMS. Further detailed rules about what is expected with code are available towards the end of this spec. We appreciate that while some have entered COMP90051 with little/no prior Python experience, many workshops so far have exercised and built up basic Python and Jupyter knowledge.

Part 1: Implementing ϵ-Greedy and UCB [3 marks total]

Implement Python classes `EpsGreedy` and `UCB` for both ϵ-Greedy and UCB learners as covered in class. You should use inheritance: make your classes sub-classes of the abstract `MAB` base class. Include components:

- All necessary properties for storing MAB state
- `__init__` constructor methods for initialising MAB state with respective signatures:
  - `def __init__(self, narms, epsilon, Q0)` for positive integer `narms`, floating-point probability `epsilon`, real-valued `Q0` taking by default `numpy.inf`; and
  - `def __init__(self, narms, rho, Q0)` for positive integer `narms`, positive real `rho`, real-valued `Q0` taking by default `numpy.inf`.
- Additional methods (where in your implementations `context` will go unused)
  - `def play(self, tround, context)` for positive integer `tround`, and unused (for now) `context`. This should return an arm integer in $\{1, \ldots,$ `self.narms`$\}$; and
  - `def update(self, arm, reward, context)` for positive integer `arm` no larger than property `self.narms`, floating-point `reward`, and unused (for now) context. This method should not return anything.

Tie-breaking in `play()` should be completed uniformly-at-random among value-maximising arms.

Part 2: The Basic Thompson Bandit [5 marks total]

Your next task is to implement a third bandit learner, one that you haven't seen in class. *Thompson sampling* is named after Thompson who discovered the idea in 1933 (before the advent of machine learning), and went unnoticed by the machine learning community until relatively recently. It is now regarded as a leading MAB technique, that uses a Bayesian model of rewards. The simplest Thompson sampler models rewards in $\{0, 1\}$ as Bernoulli draws with different parameters per arm each starting with a common Beta prior.

In this part you are to implement a Python class for the Beta-Bernoulli Thompson MAB as described in Algorithm 1 "Thompson Sampling for Bernoulli bandits" from the paper:

> Shipra Agrawal and Navin Goyal, 'Analysis of Thompson sampling for the multi-armed bandit problem', in *Proceedings of the Conference on Learning Theory (COLT 2012)*, 2012.
> `http://proceedings.mlr.press/v23/agrawal12/agrawal12.pdf`

While the COLT'2012 Algorithm 1 only considers a uniform $Beta(1, 1)$ prior, you are to implement a more flexible $Beta(\alpha_0, \beta_0)$ prior for any given $\alpha_0, \beta_0 > 0$. A point that may be missed on first reading, is that while each arm begins with the same prior, each arm updates its own posterior.

Your class `BetaThompson` should sub-class abstract `MAB` base class with components similar to above:

- All necessary properties for storing MAB state
- Constructor for initialising MAB state with signature `__init__(self, narms, alpha0, beta0)` for positive integer `narms`, positive reals `alpha0, beta0` taking by default `1`;
- Additional methods (where again `context` will go unused) of `play()` and `update()` as above.

Tie-breaking in `play()` should again be completed uniformly-at-random among value-maximising arms.

Part 3: Off-Policy Evaluation [3 marks total]

A major practical challenge for industry deployments of MAB learners has been the requirement to let the learner loose on real data. Inevitably bandits begin with little knowledge about arm reward structure, and so a bandit must necessarily suffer poor rewards in beginning rounds. For a company trying out and evaluating dozens of bandits in their data science groups, this is potentially very expensive.

A breakthrough was made when it was realised that MABs can be evaluated *offline* or *off policy*. The idea being that you collect just once a dataset of uniformly-random arm pulls and resulting rewards. Then you evaluate any possible future bandit learner of interest on that one historical data—there is no need to run bandits online in order to evaluate them! In this part you are to implement a Python function for offline/off-policy evaluation.

You must implement the algorithm first described as Algorithm 3 "Policy_Evaluator" from the paper:

> Lihong Li, Wei Chu, John Langford, Robert E. Schapire, 'A Contextual-Bandit Approach to Personalized News Article Recommendation', in *Proceedings of the Nineteenth International Conference on World Wide Web (WWW 2010)*, Raleigh, NC, USA, 2010.
> `https://arxiv.org/pdf/1003.0146.pdf`

You should begin by reading Section 4 of the WWW2010 paper which describes the algorithm. You may find it helpful to read the rest of the paper up to this point for background (skipping Sec 3.2) as this also relates to Part 4. If you require further detail of the algorithm you may find the follow-up paper useful (particularly Sec 3.1):

> Lihong Li, Wei Chu, John Langford, and Xuanhui Wang. 'Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms.' In *Proceedings of the Fourth ACM International Conference on Web Search and Data Mining (WSDM'2011)*, pp. 297-306. ACM, 2011.
> `https://arxiv.org/pdf/1003.5956.pdf`

Note that what is not made clear in the pseudo-code of Algorithm 3, is that after the bandit plays (written as function or *policy* π) an arm that matches a given log, you should not only note down the reward as if the bandit really received this reward, but you should also *update* the bandit with the played arm a, reward r_a, and later in the project the context $\mathbf{x}_1, \ldots, \mathbf{x}_K$ over the K arms. Bandits that do not make use of context—such as your Part 1 and 2 bandits—can still take context as an argument even if unused.

A second point that is implied in the pseudo-code, but may be missed, is that when asking the bandit to play an arm, the supplied round number should not be the current round in the log file, but instead the length of history recorded so far, plus one. That is, after playing a matching arm on the first logged event, a bandit may play different arms for events 2, 3, and 4, and on the 5th event may for a second time play a matching arm. For the function calls to `play()` for each of events 2, 3, 4, 5 you would pass as the `tround` argument the value 2. You would then increment to 3 for `tround` from the 6th event.

Implement your function (nominally outside any Python class) with signature

> `def offlineEvaluate(mab, arms, rewards, contexts, nrounds=None)`
> for a MAB class object `mab` such as `EpsGreedy`, `UCB`, `BetaThompson` (and the classes implemented in later project Parts), a (numpy) array `arms` of values in $\{1, \ldots,$ `mab.narms`$\}$, an array of scalar numeric `rewards` of the same length as `arms`, a numeric 2D array `contexts` with number of rows equal to the length of `arms` and number of columns equal to a positive multiple of `mab.narms`, a positive integer `nrounds` with default value `None`.

Here `arms` corresponds to the arms played by a uniformly-random policy recorded in a dataset of say M events. While `rewards` corresponds to the resulting observed M rewards. In the next Part we will consider contextual bandits, in which each arm may have a feature vector representing its state/context (and potentially

factoring in the context of the user also). So that if each of the K arms have d features, each row of `contexts` will have these feature vectors flattened as the d features of arm 1 followed by the d features of arm 2, all the way up to arm K so that we have $d \times K$ features (a multiple of K).

Finally `nrounds` is the desired number of matching events we would like to evaluate bandit `mab` on. Once your function finds this many matching arm plays, it should stop and return the per round rewards—and *not* their sum as in the WWW'2010 Algorithm 3. If it reaches the end of the logged dataset without reaching the required number (or in the case of default `None`) then it should return the per round rewards recorded.

Dataset: The LMS page for project 2 contains a 2 MB `dataset.txt` suitable for validating MAB implementations. You may download this file and familiarise yourself with its format:

- 10,000 lines (i.e., rows) corresponding to distinct site visits by users—events in the language of this part;
- Each row comprises 102 space-delimited columns of integers:
  - Column 1: The arm played by a uniformly-random policy out of 10 arms (news articles);
  - Column 2: The reward received from the arm played—1 if the user clicked 0 otherwise; and
  - Columns 3–102: The 100-dim flattened context: 10 features per arm (incorporating the content of the article and its match with the visiting user), first the features for arm 1, then arm 2, etc. up to arm 10.

Your function should be able to run on this file where column 1 forms `arms`, column 2 forms `rewards`, and columns 3–102 form `contexts`. On both classes you've implemented thus far. You should output the result of running

```
mab = EpsGreedy(10, 0.05)
results_EpsGreedy = offlineEvaluate(mab, arms, rewards, contexts, 800)
print("EpsGreedy average reward ", np.mean(results_EpsGreedy))

mab = UCB(10, 1.0)
results_UCB = offlineEvaluate(mab, arms, rewards, contexts, 800)
print("UCB average reward ", np.mean(results_UCB))

mab = BetaThompson(10, 1.0, 1.0)
results_BetaThompson = offlineEvaluate(mab, arms, rewards, contexts, 800)
print("BetaThompson average reward ", np.mean(results_BetaThompson))
```

Part 4: Contextual Bandits—LinUCB [5 marks total]

In this part you are to implement a fourth MAB learner as a fourth Python class. This time you are to read up to and including Section 3.1 of the WWW'2010 paper to understand and then implement the LinUCB learner with disjoint linear models (Algorithm 1). This is a contextual bandit—likely the first you've seen—however it's workings are a direct mashup of UCB and ridge regression both of which you've seen in class. Practicing reading and implementing papers is the best way to turbo-charge your ML skills. Your class `LinUCB` should have methods

- `def __init__(self, narms, ndims, alpha)` constructor for positive integer `narms` the number of arms, positive integer `ndims` the number of dimensions for each arm's context, positive real-valued `alpha` a hyperparameter balancing exploration-exploitation
- `def play(self, tround, context)` as for your other classes. For positive integer `tround`, and `context` being a numeric array of length `self.ndims * self.narms`; and

- `def update(self, arm, reward, context)` as for your other classes. For positive integer `arm` no larger than property `self.narms`, floating-point `reward`, and `context` as previous.

While the idea of understanding LinUCB enough to implement it correctly may seem daunting, the WWW'2010 paper is written for a non-ML audience and is complete in its description. The pseudo-code is detailed. There is one unfortunate typo however: pg. 3, column. 2, line 3 of the linked arXiv version should read $\mathbf{c}_a$ rather than $\mathbf{b}_a$. The pseudo-code uses the latter (correctly) as shorthand for the former times the contexts.

Note also one piece of language you may not have encountered: "design matrix" means a feature matrix in the statistics literature.

After you have implemented your class, include and run an evaluation on the given dataset with

```
mab = LinUCB(10, 10, 1.0)
results_LinUCB = offlineEvaluate(mab, arms, rewards, contexts, 800)
print("LinUCB average reward ", np.mean(results_LinUCB))
```

Part 5: Contextual Bandits—LinThompson [6 marks total]

Just as LinUCB mashes up ridge regression (with confidential intervals) with UCB for the contextual MAB problem, we may plug Bayesian linear regression into the Thompson sampling framework to tackle contextual bandit learning. This idea is considered by the paper:

> Shipra Agrawal and Navin Goyal, 'Thompson sampling for contextual bandits with linear payoffs', in *Proc. International Conference on Machine Learning (ICML 2013)*, pp. 127-135. 2013. `http://proceedings.mlr.press/v28/agrawal13.pdf`

In this Part you are to implement as a fifth MAB class `LinThompson`, the ICML'2013 Algorithm 1 (described in Section 2.2). Your class `LinThompson` should have methods

- `def __init__(self, narms, ndims, v)` constructor for positive integer `narms` the number of arms, positive integer `ndims` the number of dimensions for each arm's context, `v` a hyperparameter controlling exploration vs. exploitation;
- `def play(self, tround, context)` as for your other classes. For positive integer `tround`, and `context` being a numeric array of length `self.ndims * self.narms`; and
- `def update(self, arm, reward, context)` as for your other classes. For positive integer `arm` no larger than property `self.narms`, floating-point `reward`, and `context` as previous.

While the ICML'2013 intro does re-introduce the Thompson sampling framework explored in Part 2, it can be very much skimmed. Section 2.1 introduces the setting formally—information on regret and the assumptions[2] are not important for simple experimentation. Section 2.2 and Algorithm 1 'Thompson Sampling for Contextual bandits' is the key place to find the described algorithm to be implemented. Note that the first 14 lines of Section 2.3 explains the hyperparameters ϵ, δ further: the latter controls our confidence of regret being provably low (and so we might imagine taking it to be 0.05 as in typical confidence intervals); while advice is given for setting ϵ when you know the total number of rounds to be played. All that said, R, ϵ, δ only feature in the expression v, while we wouldn't really know R. And so you should just use v as a hyperparameter to control exploration balance as you have in the previous part with α.

After you have implemented your class, include and run an evaluation on the given dataset with

[2]Sub-Gaussianity is a generalisation of Normally distributed rewards. I.e., while they don't assume the rewards are Normal, they assume something Normal-like in order to obtain theoretical guarantees. R takes the role of $1/\sigma$ and controls how fast likelihood of extreme rewards decays. You can ignore all this–phew!

```
mab = LinThompson(10, 10, 1.0)
results_LinThompson = offlineEvaluate(mab, arms, rewards, contexts, 800)
print("LinThompson average reward ", np.mean(results_LinThompson))
```

Part 6: Evaluation [3 marks total]

In this part you are to delve deeper into the performance of your implemented bandit learners. This part's first sub-part does not necessarily require completion of Parts 4 and 5.

Part 6(a) [1 marks]: Run `offlineEvaluate` on each of your Python classes just with hyperparameters as when you ran `offlineEvaluate` above. This time plot the running per-round cumulative reward i.e. $T^{-1}\sum_{t=1}^{T} r_{t,a}$ for $T = 1..800$ as a function of round T, all on one overlayed plot. Your plot will have up to 5 curves, clearly labelled.

Part 6(b) [2 marks]: How can you optimise hyperparameters? Devise grid-search based strategies to select the α and v hyperparameters in LinUCB and LinThompson, as Python code in your notebook. Output the result of this strategy—which could be a graph, number, etc.

Project Submission

Preserving its structure, you must (1) rename `proj2.ipynb` as `username.ipynb` using your username[3] (2) flesh out with your project solutions with cells, (3) run on your local machine prior to submission so that outputs and plots are preserved (you are strongly recommended to open your notebook again prior to upload to double check. We may not run your notebook; given your environment might subtly differ to ours, it is your responsibility to ensure results are contained), and then (4) submit in LMS.

Marks: graders will perform code reviews of your implementations. In general a portion will be available for correctness, a portion will be available for code structure and style (primarily the former). Code should have necessary commenting to understand interfaces, and major points of inner working, basic checks of well-formed input, clear variable names and readable statements.

Further Rules. You may discuss the bandit learning deck or Python at a high-level with others, but do not collaborate on solutions. You may consult resources to understand bandits conceptually, but do not make any use of online code *whatsoever*. (We will run code comparisons against online partial implementations to enforce these rules.) You must use the environment (Anaconda3, Python 3.6 or higher) as used in labs. (You may use your own machine of course, but we strongly recommend you check code operation on lab machines prior to submission. In case we run code.) You may only use the packages already imported in the provided `proj2.ipynb` notebook. You should use `matplotlib` for plotting. Late submissions will be accepted to 4 days with -3 penalty per day.

[3] LMS/UniMelb usernames look like `brubinstein`, not to be confused with email such as `benjamin.rubinstein`.