COMP90051 Statistical Machine Learning
Project 2 Description

Due date: 8:00pm Thursday, 22nd October 2020 **Weight:** 30%[1]

Multi-armed bandits (MABs) are a simple but powerful framework for sequential decision making under uncertainty. 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, which is exactly what services like Yahoo! News want to maximise to drive user engagement and advertising revenue. In this project, you will work *individually* (not in teams) to implement several MAB algorithms. 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;

ILO2. An appreciation of how MABs are applied;

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

You will be completing the following tasks, noting that you need not complete the entire project to achieve a high mark:

1. Implement ϵ-greedy and UCB MABs (StatML lecture 16) [4 marks]
2. Implement off-policy evaluation (Li et al., 2010, 2011) [4 marks]
3. Implement LinUCB contextual MAB (Li et al., 2010) [7 marks]
4. Implement TreeBootstrap contextual MAB (Elmachtoub et al., 2017) [7 marks]
5. Evaluation and hyperparameter tuning for LinUCB [3 marks]
6. Implement KernelUCB contextual MAB (Valko et al., 2013) [5 marks]

All tasks are to be completed in the provided Python Jupyter notebook `proj2.ipynb`.[2] Detailed instructions for each task are included at the end of this document. Most tasks will require you to consult one or more references (e.g. academic papers or lecture slides)—we provide helpful pointers to guide your reading and to correct any ambiguities.

[1]Forming a combined hurdle with project 1.

[2]We appreciate that while some have entered COMP90051 with little Python experience, many workshops so far have exercised and built up basic Python and Jupyter knowledge.

MAB algorithms. Tasks 1, 3, 4 and 6 require you to implement MAB algorithms by completing provided skeleton code in `proj2.ipynb`. All of the MAB algorithms must be implemented as sub-classes of a base `MAB` class (defined for you). This ensures all MAB algorithms inherit the same interface, with the following methods:

- `__init__(self, n_arms, ...)`: Initialises the MAB with the given number of arms `n_arms`. Arms are indexed by integers in the set $\{0, \ldots, \texttt{n\_arms} - 1\}$. All MAB algorithms take additional algorithm-specific parameters in place of '`...`'.
- `play(self, context)`: Plays an arm based on the provided `context` (a multi-dimensional array that encodes user and arm features). For non-contextual bandits (e.g. ϵ-greedy and UCB), this method should accept `context=None`. The method must return the integer index of the played arm in the set $\{0, \ldots, \texttt{n\_arms} - 1\}$. *Note: this method should not update the internal state of the MAB. All play methods should tie-break uniformly at random in this project.*
- `update(self, arm, reward, context)`: Updates the internal state of the MAB after playing an arm with integer index `arm` for some `context`, and receiving a real-valued `reward`. For non-contextual bandits, this method should accept `context=None`.

Your implementations *must* conform to this interface. You may implement some functionality in the base `MAB` class if you desire—e.g. to avoid duplicating common functionality in each sub-class.

Python environment. You must use the Python environment used in workshops to ensure markers can reproduce your results if required. We assume you are using Python $\geq$ 3.8, numpy $\geq$ 1.19.0, scikit-learn $\geq$ 0.23.0 and matplotlib $\geq$ 3.2.0.

Other constraints. You may not use functionality from external libraries/packages, beyond what is imported in the provided Jupyter notebook. You must preserve the structure of the skeleton code—please only insert your own code where specified. You should not add new cells to the notebook. You may discuss the bandit learning slide 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.)

Submission Instructions

You must complete all your work in the provided `proj2.ipynb` Jupyter notebook. When you are ready to submit, you should restart the kernel and run all cells consecutively. You must ensure outputs are saved in the `ipynb` file, as we may not run your notebook when grading. Finally, rename your completed notebook from `proj2.ipynb` to `username.ipynb` where `username` is your university central username[3] and upload to the Project 2 LMS page.

Marking

Your project will be marked out of 30, with 80% (20%) of your mark coming from correctness (respectively code structure and style). Markers will perform code reviews of your implementations with **indicative** focus on:

1. *Correctness:* Faithful implementation of the algorithm as specified in the reference or clarified in the specification with possible updates in the LMS changelog. It is important that your code performs other basic functions such as: raising errors if the input is incorrect, working for any dataset that meets the requirements (i.e. not hard-coded).

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

2. *Code structure and style:* Efficient code (e.g. making use of vectorised functions, avoiding recalculation of expensive results); self-documenting variable names and/or comments; avoiding inappropriate data structures, duplicated code and illegal package imports.

Late submission policy. Late submissions will be accepted to 4 days with −3 penalty per day.

Task Descriptions

Task 1: Implement ϵ-Greedy and UCB MABs [4 marks total]

Your first task is to implement the ϵ-Greedy and UCB learners *as covered in class*, by completing the skeleton code for the `EpsGreedy` and `UCB` Python classes. You will need to implement the `__init__`, `play`, and `update` methods for each class. The parameters and return values for each method are specified in docstrings in `proj2.ipynb`. Note that tie-breaking in `play` should be done uniformly-at-random among value-maximising arms.

Task 2: Implement Off-Policy Evaluation [4 marks total]

In this task, you will need to implement the provided skeleton code for the `offlineEvaluate` function. The parameters and return value of the function are specified in the function's docstring.

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 one 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.

Your second task is to implement 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

In order to understand Algorithm 3, you should begin by reading Section 4 of the WWW'2010 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 Task 3. 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 MAB 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 MAB really received this reward, but you should also *update* the MAB 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. MABs that do not make use of context—such as ϵ-greedy and UCB—can still take context as an argument even if unused.

Dataset. The LMS page for project 2 contains a 2 MB `dataset.txt` suitable for validating MAB implementations. You should download this file and place it in the same directory as `proj2.ipynb`. It is formatted as follows:

- 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 `offlineEvaluate` function should be able to run on data from this file where column 1 forms `arms`, column 2 forms `rewards`, and columns 3–102 form `contexts`. You should evaluate both MAB classes you've implemented thus far by 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))
```

Task 3: Implement LinUCB Contextual MAB [7 marks total]

In this task, you will implement a third MAB learner as a third 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. Your implementation of Algorithm 1 should be done within the provided skeleton code for the `LinUCB` class. As for the previous MAB classes, you will need to implement the `__init__`, `play`, and `update` methods. The parameters and return values for each method are specified in docstrings in `proj2.ipynb`.

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 the `LinUCB` 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))
```

Task 4: Implement TreeBootstrap Contextual MAB [7 marks total]

While LinUCB can exploit contextual information to optimise long-term rewards, it is limited in the kinds of patterns it can learn, as it assumes a *linear* model for the expected rewards conditional on the context. For this fourth task, you will implement a more flexible MAB learner called TreeBootstrap which uses decision trees to predict future rewards (assumed to be binary). The learner is defined in Algorithm 2 of the following paper:

Adam N. Elmachtoub, Ryan McNellis, Sechan Oh and Marek Petrik, 'A Practical Method for Solving Contextual Bandit Problems Using Decision Trees', in *Proceedings of the Thirty-Third Conference on Uncertainty in Artificial Intelligence (UAI'2017)*, Sydney, Australia, 2017.
`http://auai.org/uai2017/proceedings/papers/171.pdf`

In order to understand the context for Algorithm 2, you should read the first half of the paper, stopping at the end of Sec 5.1. You should pay particular attention to how the explore-exploit trade-off is managed (Secs 4.2 and 5.1), as the approach differs markedly from ϵ-greedy and UCB-style algorithms.

While reading the paper, you may notice that the formulation of the contextual MAB problem differs from the one we have been using so far. In particular, the authors assume the bandit receives a *single* context vector $x_t \in \mathbb{R}^M$ which corresponds to the user's features at round t in a recommender system application. However, we would like to allow for a *collection* of context vectors $\{x_{t,a}\}$ (one for each arm $a \in 1, \ldots, K$) which may encode user *and* arm-specific features. Fortunately, it is relatively straightforward to generalise Algorithm 2 in the paper to the multi-context vector case. Concretely, we change the definition of the historical data collected for arm a at round t to be $D_{t,a} = \{(x_{s,a}, r_{s,a}) : s \leq t-1, a_s = a\}$. We also modify the estimate of the success probability $\hat{p}(\tilde{\theta}_{t,a}, x_t)$ for each arm. Rather than passing a single context vector x_t to the decision tree for each arm, we instead pass the arm-specific context $x_{t,a}$. These changes are realised in a generalised Algorithm 2 provided below. The generalised Algorithm 2 also includes two additional clarifications:

- Line 5 specifies how to proceed when no historical data is available for an arm.
- Lines 12-14 manage the problem discussed at the end of Sec 5.1 in the paper, by adding fabricated prior data of one success and one failure for each arm when it is first pulled.

Algorithm 2 TreeBootstrap

```
1: for $t = 1, \ldots, T$ do
2:     Observe context vectors $\{x_{t,a}\}$
3:     for $a = 1, \ldots K$ do
4:         if $|D_{t,a}| = 0$ then
5:             Set decision tree $\tilde{\theta}_{t,a}$ to predict 1 for any input
6:         else
7:             Sample bootstrapped dataset $\tilde{D}_{t,a}$ from $D_{t,a}$
8:             Fit decision tree $\tilde{\theta}_{t,a}$ to $\tilde{D}_{t,a}$
9:         end if
10:    end for
11:    Choose action $a_t = \arg\max_a \hat{p}(\tilde{\theta}_{t,a}, x_{t,a})$
12:    if $|D_{t,a_t}| = 0$ then
13:        Update $D_{t,a_t}$ with $(x_{t,a_t}, 0)$ and $(x_{t,a_t}, 1)$
14:    end if
15:    Update $D_{t,a_t}$ with $(x_{t,a_t}, r_{t,a_t})$
16: end for
```

You should implement the generalised TreeBootstrap algorithm by completing the provided skeleton code for the `TreeBootstrap` class. As for the previous MAB classes, you will need to implement the `__init__`, `play`, and `update` methods. The parameters and return values for each method are specified in docstrings in `proj2.ipynb`. You should use the `DecisionTreeClassifier` from scikit-learn to instantiate the decision trees $\tilde{\theta}_{t,a}$ referenced in the TreeBootstrap algorithm.

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

```
mab = TreeBootstrap(10, 10)
results_TreeBootstrap = offlineEvaluate(mab, arms, rewards, contexts, 800)
print("TreeBootstrap average reward ", np.mean(results_TreeBootstrap))
```

Task 5: Evaluation and Hyperparameter Tuning [3 marks total]

In this task, you are to delve deeper into the performance of the four MABs algorithm you have implemented so far. This tasks's first sub-task does not necessarily require completion of Tasks 3 and 4.

Task 5(a) [1 marks]: Run `offlineEvaluate` on each of your Python classes with the same 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 should have up to four curves, clearly labelled.

Task 5(b) [2 marks]: How can you optimise hyperparameters? Devise grid-search based strategies to select the α hyperparameter in LinUCB as Python code in your notebook. Output the result of this strategy—which could be a graph, number, etc.

Task 6: KernelUCB [5 marks total]

You may skip this final task if short on time. So far you have built on knowledge of bandits and ridge regression. In this part you would make use of the kernel methods part of class in a mashup of all three concepts by implementing as a fifth Python class the KernelUCB (with online updates) that is Algorithm 1 of this paper:

> Michal Valko, Nathan Korda, Rémi Munos, Ilias Flaounas, and Nello Cristianini, 'Finite-time analysis of kernelised contextual bandits.' In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence (UAI'13)*, pp. 654-663. AUAI Press, 2013.
> `http://auai.org/uai2013/prints/papers/161.pdf`

You will need to judiciously skim more theoretical parts of the introduction as not relevant to understanding the crux of the algorithm. To help: you will find Sec 2.2 sets up the MAB problem (and therefore notation in the paper), Sec 2.3 reviews LinUCB, while the first two columns of Sec 3 plus Algorithm 1, explain KernelUCB. Unfortunately, there are several inconsistencies in Algorithm 1 as it appears in the paper. We have provided a corrected version below, which you should use instead.

Note that the notation $[K_{11}, K_{12}; K_{21}, K_{22}]$ is block-matrix notation defining the first rows of the matrix by submatrices, then the final rows. That is, the algorithm iteratively updates the kernel matrix by first constructing submatrices, and then merging them together. Also note that the 'RKHS $\mathcal{H}$' can be thought of as the feature space associated with the kernel function.

```
Algorithm 1 KernelUCB with online updates (corrected)
    Input: $N$ the number of actions, $T$ the number of pulls, $\gamma$, $\eta$ regularization
    and exploration parameters, $k(\cdot, \cdot)$ kernel function
 1: for $t = 1$ to $T$ do
 2:     Receive contexts $\{x_{1,t}, \ldots, x_{N,t}\}$
 3:     if $t = 1$ then
 4:         $u_t \leftarrow [1, 0, \ldots, 0]^\top$
 5:     else
 6:         for $a = 1$ to $N$ do
 7:             $\sigma_{a,t} \leftarrow \left[k(x_{a,t}, x_{a,t}) - k_{x_{a,t},t}^\top K_t^{-1} k_{x_{a,t},t}\right]^{\frac{1}{2}}$
 8:             $u_{a,t} \leftarrow k_{x_{a,t},t}^\top K_t^{-1} y_t + \frac{\eta}{\gamma^{1/2}} \sigma_{a,t}$
 9:         end for
10:     end if
11:     Choose action $a_t \leftarrow \arg\max u_t$ and get reward $r_t$
12:     Store context for action $a_t$: $x_t \leftarrow x_{a_t,t}$
13:     Update reward history: $y_{t+1} \leftarrow [r_1, \ldots, r_t]^\top$
14:     if $t = 1$ then                          ▷ initialise kernel matrix inverse
15:         $K_{t+1}^{-1} \leftarrow (k(x_t, x_t) + \gamma)^{-1}$
16:     else                              ▷ online update of kernel matrix inverse
17:         $b \leftarrow k_{x_t,t}$
18:         $K_{22} \leftarrow \left(k(x_t, x_t) + \gamma - b^\top K_t^{-1} b\right)^{-1}$
19:         $K_{11} \leftarrow K_t^{-1} + K_{22} K_t^{-1} b b^\top K_t^{-1}$
20:         $K_{12} \leftarrow -K_{22} K_t^{-1} b$
21:         $K_{21} \leftarrow -K_{22} b^\top K_t^{-1}$
22:         $K_{t+1}^{-1} \leftarrow [K_{11}, K_{12}; K_{21}, K_{22}]$
23:     end if
24: end for
```

You should implement the (corrected) KernelUCB algorithm by completing the provided skeleton code for the `KernelUCB` class. As for the previous MAB classes, you will need to implement the `__init__`, `play`, and `update` methods. The parameters and return values for each method are specified in docstrings in `proj2.ipynb`.

Once done, demonstrate a MAB hyperparameter setting using the RBF kernel, and generate a plot like in Task 5(a) demonstrating competitive performance relative to LinUCB. Note that the RBF kernel is imported for you as `rbf_kernel` in the given notebook. Note that you may like to try different kernel hyperparameters, and beware that while the RBF's hyperparameter is also called 'gamma', it is distinct to the 'gamma' in KernelUCB.