


All tasks are to be completed in the provided Python Jupyter notebook `proj2.ipynb`.³ Detailed instructions for each task are included in this document. These will require you to consult one or more academic papers. We provide helpful pointers in this project spec to guide your reading and to correct any ambiguities, with margin mark symbol .


MAB algorithms. The project’s tasks 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, ..., rng)`: Initialises the MAB with the given number of arms `n_arms` and pseudo-random number generator `rng`. Arms are indexed by integers in the set $\{0, \dots, n_arms - 1\}$. All MAB algorithms take additional algorithm-specific parameters in place of ‘...’. *Hint: When instantiating a MAB, you should set `rng` to be the [random generator](#) initialised for you in the notebook. This will ensure your results are reproducible.*
- `play(self, context)`: Plays an arm based on the provided `context` (a multi-dimensional array that encodes user and arm features). The method must return the integer index of the played arm in the set $\{0, \dots, n_arms - 1\}$. *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, context, reward)`: Updates the internal state of the MAB after playing an arm with integer index `arm` for given `context`, receiving an (optional) real-valued `reward`. Only class `MlinUCB` of Task 2 will handle missing rewards, all other classes should do nothing if updated with a missing reward.

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. Your classes may also use additional private methods to better structure your code. And you may decide how to use class inheritance.

Evaluating MAB classes. Each task directs you to evaluate your implementations in some way. You will need to use the `offline_eval()` function and dataset file provided. See Appendix A for details.

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 ≥ 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 highlighted here with margin marking . 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, including via Piazza, but do not collaborate with anyone on direct 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. See ‘academic misconduct’ statement above.)

Submission Checklist

You must complete all your work in the provided `proj2.ipynb` Jupyter notebook. When you are ready to submit, follow these steps. You may submit multiple times. We will mark your last attempt. *Hint: it is a good idea to submit early as a backup. Try to complete Task 1 in the first week and submit it; it will help you understand other tasks and be a fantastic start!*

³We appreciate that while some have entered COMP90051 feeling less confident with Python, many workshops so far have exercised and built up basic Python and Jupyter knowledge. Both are industry standard tools for the data sciences.

1. Restart your Jupyter kernel and run all cells consecutively.
2. Ensure outputs are saved in the `ipynb` file, as we may choose not to run your notebook when grading.
3. Rename your completed notebook from `proj2.ipynb` to `username.ipynb` where `username` is your university central username⁴.
4. Upload your submission to the Project 2 Canvas page.

Marking

Projects will be marked out of 25. Overall approximately 50%, 30%, 20% of available marks will come from correctness, code structure & style, and experimentation. Markers will perform code reviews of your submissions with **indicative** focus on the following. We will endeavour to provide (indicative not exhaustive) feedback.

1. *Correctness*: Faithful implementation of the algorithm as specified in the reference or clarified in the specification with possible updates in the Canvas 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).
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.
3. *Experimentation*: Each task you choose to complete directs you to perform some experimentation with your implementation, such as evaluation, tuning, or comparison. You will need to choose a reasonable approach to your experimentation, based on your acquired understanding of the MAB learners.

Late submission policy. Late submissions will be accepted to 4 days at -2.5 penalty per day or part day.

Task Descriptions

Task 1: Implement LinUCB Contextual MAB [7 marks total]

In this task, you will implement a MAB learner as a Python class. You are to read up to and including Section 3.1 of this WWW'2010 paper to understand and then implement the LinUCB learner with disjoint (not shared) linear models. LinUCB is described in 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 we cover in lecture.

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>

Your implementation of WWW'2010 Algorithm 1 should be done within the provided skeleton code for the `LinUCB` class. 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`. Note that tie-breaking in `play` should be done uniformly-at-random among value-maximising arms.

While the idea of understanding LinUCB enough to implement it correctly may seem daunting, the WWW paper is written for a non-ML audience and is complete in its description. The pseudo-code is detailed. There

⁴LMS/UniMelb usernames look like `brubinstein`, not to be confused with email such as `benjamin.rubinstein`.

is one unfortunate typo however: pg. 3, column 2, line 3 of the linked arXiv version linked above 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.

💡 *Another hint: for students who haven't seen "design matrix" before, it means a feature matrix.*

Experiments. Once your class is implemented, it is time to perform some basic experimentation.

- (a) Include and run an evaluation on the given dataset where column 1 forms arms, column 2 forms rewards (and missing indicator column 3 is ignored), and columns 4–103 form contexts:

```
mab = LinUCB(10, 10, 1.0, rng)
LinUCB_rewards, _ = offline_eval(mab, arms, rewards, contexts, 800)
print("LinUCB average reward ", np.mean(LinUCB_rewards))
```

- 💡 (b) Run offline evaluation as above, but now plot the 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 . We have imported `matplotlib.pyplot` to help here.
- (c) Can you tune your MAB's hyperparameters? Devise and run a grid-search based strategy to select `alpha` for `LinUCB` as Python code in your notebook. Output the result of this strategy—which could be a graph, number, etc. of your choice.

Task 2: Implement MLinUCB [9 marks total]

In this task, you will implement a contextual MAB that extends `LinUCB` (Task 1) for settings where observing rewards may not always be possible. For example, rewards may be lost due to errors in a physical process, they may be costly, or withheld due to privacy. The idea of this `MLinUCB` (`LinUCB` with missing rewards) is similar to semi-supervised learning: even when `MLinUCB` doesn't observe a reward in a round, it still tries to learn from the context feature vector by guessing the missing reward using clustering of past context-reward pairs. You are to read up to but not including Theorem 1 of this 2020 arXiv paper:

Djallel Bouneffouf, Sohini Upadhyay, and Yasaman Khazaeni. 'Contextual Bandit with Missing Rewards.' arXiv:2007.06368 [cs.LG]. 2020. <https://arxiv.org/pdf/2007.06368.pdf>

💡 Your implementation of Algorithm 2 should be done within the provided skeleton code for the `MLinUCB` class. You will need to implement the `__init__`, `play`, and `update` methods. Recall that `update()` will receive `reward=None` for rounds with no observed reward. The parameters and return values for each method are specified in docstrings in `proj2.ipynb`. Your implementation should make use of *k*-means clustering as implemented here in `scikit-learn` and imported for you in the `proj2.ipynb` skeleton. You will need to initialise using MAB hyperparameters for N, m from the paper. Except for the number of clusters, you should just use all of `KMeans` defaults.

💡 *Hint: A number of minor errors and omissions in the paper might cause confusion. To help out, we've clarified these and summarised the clarifications in a new pseudo-code algorithm, all in Appendix B.*

Experiments. Repeat the same set of experiments from Task 1 here, except for (respectively):

- (a) Here run your new class, this time with missing rewards as follows:

```
mab = MLinUCB(10, 10, 1.0, 10, 3)
MLinUCB_rewards, MLinUCB_ids = offline_eval(mab, arms, rewards_missing, contexts, 800)
print('MLinUCB average reward', np.mean(rewards[MLinUCB_ids]))
```

Then compare this with `LinUCB` also evaluated with the same missing reward version of the data.

- (b) Once again plot your results for both bandits. This time with the missing reward data.
- (c) This time consider tuning `alpha`, `N` and `m`.

Task 3: Implement SquareCB [9 marks total]

In this task, you will implement a final learner for the regular contextual MAB setting (the same setting as Task 1), for ‘general purpose’ contextual MABs. Where LinUCB is designed to work only with ridge regression estimating arm rewards, general-purpose MABs aim to be able to use *any supervised learner* instead. The following ICML’2020 paper describes a state-of-the-art approach to general-purpose MABs. (Note in this paper ‘oracle’ or ‘online regression oracle’ refers to this user-defined supervised learning algorithm. It is an oracle because the bandit can call it for answers without knowing how the oracle works.)

Dylan Foster and Alexander Rakhlin. ‘Beyond UCB: Optimal and Efficient Contextual Bandits with Regression Oracles.’ In *Proceedings of the 37th International Conference on Machine Learning (ICML’2020)*, pp. 3199-3210, PMLR, 2020. <http://proceedings.mlr.press/v119/foster20a/foster20a.pdf>

You are to read this paper up to and not including Theorem 1, then the first paragraph of Section 2.2 and Algorithm 1 (*i.e.*, you might skip Theorem 1 to Section 2.1, and most of Section 2.2 after the algorithm is described; these skipped discussions are really interesting theoretical developments but not needed for the task).

You should implement the paper’s Algorithm 1 within the provided skeleton code for the **SquareCB** class. As before, 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 paper allows for any ‘online regression oracle’ to be input into Algorithm 1 as parameter ‘SqAlg’, you should hard-card your oracle to be this [logistic regression](#) implementation using `scikit-learn` which is imported for you in the `proj2.ipynb` skeleton. Use the logistic regression class’s defaults. We have picked logistic regression since the data in this project has rewards for clicks.

While this paper appears to be free of errors, so we do not provide a full appendix of clarifications, we offer pointers now. *Hint: this paper deals with losses not rewards. To fit this into our framework, it might help to convert a loss to a reward by using its negative value. Hint: Line 6 of the algorithm defines $p_{t,a}$ of a distribution but in general this might not sum to one to form a proper distribution. You should hard-code parameter μ to be the number of arms to fix this.* Finally, you should be sure to use a separate model per arm.

Experiments. Repeat the same set of experiments as above, except for (respectively):

- (a) Here run `mab = SquareCB(10, 10, 18.0, rng)` with all rewards (ignoring the missing reward column 3), save results in `SquareCB_rewards` instead. Once again also run LinUCB on the same data. *Hint: Where does $\gamma = 18$ come from? It was examined by a reference, Abe & Long.*
- (b) When plotting the results on all data (ignore missing rewards), include LinUCB’s curve too.
- (c) This time consider tuning `gamma`.

A Appendix: Details for Off-Policy Evaluation and Dataset

You are directed to experiment with your bandits in each project task. To help with this, we have provided in the skeleton notebook a useful `offline_eval()` function, and a dataset. This section describes both.

Off-policy evaluation: The `offline_eval()` function evaluates bandit classes on previously collected data. The parameters and return value of the `offline_eval` function—its interface—are specified in the function’s docstring. Recall that bandits are interactive methods: to train, they must interact with their environment. This is true even for evaluating a bandit, as it must be trained at the same time as it is evaluated. But this requirement to let a bandit learner loose on real data, was a major practical challenge for industry deployments of MAB. Typically 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 would be potentially prohibitively expensive. A breakthrough was made when it was realised that MABs can be evaluated *offline* or *off-policy*. (‘Policy’ is the function outputting an arm given a context.) With off-policy evaluation: you collect just one dataset of uniformly-random arm pulls and resulting rewards; then you evaluate any interesting future bandit learners on that one historical dataset! We have provided this functionality in `offline_eval()` for you, which implements Algorithm 3 “Policy_Evaluator” of the WWW’2010 paper referenced in Task 1. In order to understand this algorithm, you could read Section 4 of WWW’2010.⁵

Dataset. Alongside the skeleton notebook, we provide a 2 MB `dataset.txt` suitable for validating contextual MAB implementations. You should download this file and place it in the same directory as your local `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 task;
- Each row comprises 103 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;
 - Column 3: An indicator used in Task 2, only. In that task you will be considering a MAB that can operate when some rewards are missing. This indicator value says whether the reward should be missing (1) or available (0); and
 - Columns 4–103: 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.

⁵What might not be clear in the pseudo-code of Algorithm 3 of WWW’2010, is that after the MAB plays (policy written as π) an arm that matches the next logged record, off-policy evaluation notes down the reward as if the MAB really received this reward—for the purposes of the actual evaluation calculation; it also runs `update()` of the MAB with the played arm a , reward r_a , and the context $\mathbf{x}_1, \dots, \mathbf{x}_K$ over the K arms.

Algorithm 1 MLinUCB correcting Algorithm 2 of Bouneffouf *et al.* (2020)

```
1: Input: value for  $\alpha, \mathbf{b}_0, \mathbf{A}_0, N, m$ 
2: for all  $k \in K$  do
3:    $\mathbf{A}_k \leftarrow \mathbf{A}_0, \mathbf{b}_k \leftarrow \mathbf{b}_0$ 
4: end for
5: for  $t = 1$  to  $T$  do
6:   Let  $\{\mathbf{x}_1, \dots, \mathbf{x}_s\}$  be the updated contexts up to round  $t$ 
7:   Cluster  $\{\mathbf{x}_1, \dots, \mathbf{x}_s\}$  into  $\min\{s, N\}$  clusters
8:   for all  $k \in K$  do
9:      $\theta_k \leftarrow \mathbf{A}_k^{-1} \mathbf{b}_k$ 
10:     $p_k \leftarrow \theta_k^T \mathbf{x}_{t,k} + \alpha \sqrt{\mathbf{x}_{t,k}^T \mathbf{A}_k^{-1} \mathbf{x}_{t,k}}$ 
11:   end for
12:   Choose arm  $k_t = \arg \max_{k \in K} p_k$ , and observe real-valued payoff  $r_t$ 
13:   if  $r_t$  available from data then
14:     retrieve  $r_t$  from data
15:   else
16:      $m' \leftarrow \min\{m, s\}$ 
17:      $r_t \leftarrow g(\mathbf{x}_t) = \frac{\sum_{j=1}^{m'} \frac{\bar{r}_j}{0.1+d_j}}{\sum_{j=1}^{m'} \frac{1}{0.1+d_j}}$ 
18:   end if
19:   if  $s > 0$  or  $r_t$  successfully retrieved then
20:      $\mathbf{A}_k \leftarrow \mathbf{A}_k + \mathbf{x}_{t,k} \mathbf{x}_{t,k}^T$ 
21:      $\mathbf{b}_k \leftarrow \mathbf{b}_k + r_t \mathbf{x}_{t,k}$ 
22:   end if
23: end for
```

B Appendix: Corrected Pseudo-Code for MLinUCB

We now list some corrections to Algorithm 2 of Bouneffouf *et al.* (2020) for Task 2's MLinUCB. These are summarised in our pseudo-code Algorithm 1.

The purpose of inputs $\mathbf{A}_0, \mathbf{b}_0$ (matrix and vector) are to initialise the model parameters per arm (new line 3) which wasn't done originally.

There are several places where index k_t was used but undefined, and can often just be k for arm. While in some places the arm's specific context $\mathbf{x}_{t,k}$ should be used not just the overall context \mathbf{x}_t —these are largely corrected now.

A detail omitted in the paper: you can't have more clusters than the number of observations being clustered.

💡 *Hint: if s denotes the number of instances passed to clustering, then form $\min\{s, N\}$ clusters instead of N , and find the closest $\min\{s, m\}$ clusters instead of m (new lines 7 and 16).*

To prevent division by zero in the definition of function $g()$ (Equation 2 of the paper) you should add constant 0.1 to the denominators (new line 17).

You might need to play/update an arm without seeing a reward for it. You cannot update model parameters in this case (new line 19).