Table of Contents

runOLSbandit.m

```
% Runs OLS Bandit algorithm and returns regret and fraction of pulls.
%
% This code implements the OLS bandit algorithm. For more information,
 see:

% - https://pubsonline.informs.org/doi/abs/10.1287/11-SSY032.
% - https://pubsonline.informs.org/doi/10.1287/opre.2019.1902.

% Our implementation of OLS bandit uses Sherman-Morrison formula for
 fast
% updates of least squares estimations of arms. This
% fast update is applied once the covariance matrices become
 invertible.
% Before having invertible covariance matrices, the algorithm applies
 ridge
% regression for more accurate estimations.
%
```

Inputs:

```
k: Number of arms.
T: Time horizon.
d: Dimension of covariates.
b: A k*d matrix of arm parameters.
sigma_e: Standard deviation of noise (used only for reward
 generation).
    This parameter is unused if noise and contexts are provided.
sigma_x: Standard deviation of covariates
    (used only for context generation for Gaussian contexts).
    This parameter is unused if noise and contexts are provided.
xmax: Maximum of l2-norm of covariates
    (used only for context generation). This parameter is unused if
    noise and contexts are provided.
h: The sub-optimality
 gap considered for filtering arms based on forced
    sampling estimates.
q: Number of forced sampling rounds per arm in each of forced sampling
    phases.
verbose: Whether to print outputs or not.
varargin: Additional arguments. In particular, if these are not
    provided the noise and contexts will be generated according to
```

```
    Gaussian and truncated Gaussian distributions.
    In case they are provided,
 there should exactly be THREE additional
    arguments. The first one is contexts. The second one is a binary
    input, called noise_input. If noise_input = 1, this means the last
    argument will be noise e=(Y-X*beta). On the other hand, if
    noise_input = 0, then the last argument will be Y or
    rewards. Note that the noise should be T*1 while rewards should be
    T*k.
```

Outputs:

```
regret: Cumulative regret as a running sum over regret terms.
fractions: Fractions of pulls of different arms.
pull_ind: Matrix of pulls.
```

Code:

```
function [regret, fractions, pull_ind] = runOLSbandit(k, T, d, b, ...
    sigma_e, sigma_x, xmax, h, ...
    q, verbose, varargin)

warning('off','all');

if nargin==10 % Context and noise are NOT provided, so generate those.
    % Noise is Gaussian with std sigma_e.
    e = randn(T,1)*sigma_e;
    noise_input = 1;

    % Contexts follow truncated gaussian distributions with l-infinity
 norm
    % at most xmax.
    X = max(-xmax, min(xmax, mvnrnd(zeros(d, 1), sigma_x, T)));
else
    X = varargin{1};
    noise_input = varargin{2};
    if(noise_input==1)
        e = varargin{3};
    else
        rewards = varargin{3};
    end
end

% Binary matrix, keeps track of forced sampling rounds.
is_forced_sampling = zeros(T,k);

for i=1:d
     intvl = ((i-1)*k + 1):(i*k);
     is_forced_sampling(intvl, :) = eye(k);
end

maxn = ceil(log2(T/(k*q)));  % Number of forced sampling phases.
```

```
for n=0:maxn
    for i=1:k  % Arm i.
        for j=(q*(i-1)+1):(q*i)  % Periods where arm i is forced
 sampled.
            index = (2^n-1)*k*q+j;
            if (index<=T && index>k*d)
                is_forced_sampling(index, i)=1;
            end
        end
    end
end


reward_vector = zeros(T, k);  % Vector of all (potential) rewards.
pull_ind = zeros(T, k);  % Binary indicator whether each is pulled.

regret = zeros(1, T);

betahat_all = b * 0;  % Initialize all sample estimators.
betahat_forced = b * 0; % Initialize forced sample estimators.

XtopX_inv = zeros(d, d, k);

% Whether direct LS calculation using normal equations is needed.
no_LS_calculations = zeros(k, 1);
for t=1:T

    x = X(t,:)';
    explore = 0;

    %------ First, choose which arm we should pull in this round.
    forced_sampled_t = sum(is_forced_sampling(t,:));
    if (forced_sampled_t>0)  % If we should force sample.
        i = find(is_forced_sampling(t,:));
        pull_ind(t, i) = 1;
        arm_pulled = i;
        explore = 1;
    else
        reward_forced_sample = betahat_forced*x;
        [max_forced_sample_reward, ~] = max(reward_forced_sample);

        % Filter arms that are within h/2 of max reward.
        arms_to_use = reward_forced_sample >
 max_forced_sample_reward ...
            - h / 2;

        % Find estimated reward of all sample estimators.
        reward_all_sample = betahat_all * x;

        % Remove those who are filtered out by the forced sample
 estimator.
        reward_all_sample(~arms_to_use) = -inf;

        % Pull the arm with maximum reward among reward_all_sample.
```

```
        opt_arms = find(reward_all_sample==max(reward_all_sample));
        % Break ties randomly.
        arm_pulled = opt_arms(randi(length(opt_arms)));

        pull_ind(t, arm_pulled)=1;
    end
    %------ Second, calculate the regret.
    if(noise_input==1)
        bx = b*x;
        ourreward = bx(arm_pulled);
        bestreward = max(bx);
    else
        ourreward = rewards(t, arm_pulled);
        bestreward = max(rewards(t, :));
    end

    if (t==1)
        regret(t) = bestreward - ourreward;
    else
        regret(t) = regret(t-1) + bestreward - ourreward;
    end

    %------ Third, update estimates.
    if(noise_input==1)
        reward_vector(t, arm_pulled) = ourreward + e(t);
    else
        reward_vector(t, arm_pulled) = rewards(t, arm_pulled);
    end

    if (explore==1)
        % Update forced sample estimator.
        lsX = X( (is_forced_sampling(:, arm_pulled)==1) & ...
            (pull_ind(:, arm_pulled)==1), :);  % Design matrix.
        lsY = reward_vector((is_forced_sampling(:, arm_pulled)==1)
& ...
            (pull_ind(:, arm_pulled)==1), arm_pulled);  % Reward
vector.

        if (size(lsX, 1)>=d)
            betahat_forced(arm_pulled, :) = lsX\lsY;
        end

        % Update all sample estimator. First filter observations.
        obs_filt = find((pull_ind(:, arm_pulled)==1));

        lsX_AS = X(obs_filt, :); % Design matrix.
        lsY_AS = reward_vector(obs_filt, arm_pulled);
        if (size(lsX_AS, 1)>=d)
            betahat_all(arm_pulled, :)=lsX_AS\lsY_AS;
        end
    else
        if (no_LS_calculations(arm_pulled)==0)
           obs_filt=find((pull_ind(:, arm_pulled)==1));
           lsX_AS = X(obs_filt, :); % Design matrix.
```

```
            lsY_AS = reward_vector(obs_filt, arm_pulled);
            if (size(lsX_AS,1)>=d && rank(lsX_AS) >=d)
                XtopX_inv(:, :, arm_pulled)=inv(lsX_AS'*lsX_AS);
                betahat_all(arm_pulled, :) = lsX_AS\lsY_AS;
                no_LS_calculations(arm_pulled) = 1;
            end
         else
            [XtopX_inv(:,:,arm_pulled),
 betahat_vertical]=rankoneupdate(...
                XtopX_inv(:, :, arm_pulled), ...
                betahat_all(arm_pulled, :)', x, ...
                reward_vector(t, arm_pulled));
            betahat_all(arm_pulled,:) = betahat_vertical';  %
 Transpose it.
         end


    end

    if(verbose == 1)
        if (mod(t,500)==0)
            fprintf('OLS: t=%d, parameter estimation error = %f.
 \n', ...
                t, norm(b - betahat_all, 'fro'))
        end
    end
end
fractions = mean(pull_ind);  % Fraction of times each arm is pulled.
if(verbose == 1)
    fprintf('OLS: Total parameter estimation error = %f. \n', ...
        norm(b - betahat_all, 'fro'))
    fprintf('OLS: Fraction of pulls = %f\n', fractions)
    fprintf('OLS: Total regret occured = %f.\n', regret(end))
end
end
```

Published with MATLAB® R2015a