Table of Contents

runpriordependentTS.m 1
Inputs: 1
Outputs: 2
Code: 2

runpriordependentTS.m

```
% Runs prior dependent TS algorithm and returns regret and fraction of
% pulls.
%
% This code runs the prior dependent TS algorithm adapted to be used
 in our
% setting. For more information, on this algorithm see Algorithm 4 of
%
% - https://pubsonline.informs.org/doi/abs/10.1287/moor.2014.0650.
%
% For using the algorithm in this paper, which is designed for linear
% we can concatenate all the arm parameters to get a theta,
% parameter (of interest) in R^{k*d}.
% The k actions at time period t, with context x_t are given by:
% A_t = {[X_t, 0, 0, ..., 0], [0, X_t, 0, 0, ..., 0], ... ,
% [0, 0, ..., X_t]}. After having this mapping, the algorithm builds
% confidence sets around theta and picks the action (corresponds to
 the
% same arm) that maximizes the optimistic reward.

% Our implementation uses Sherman-Morisson formula for fast rank one
 update
% of arm parameters. As construction of confidence intervals requires
% the knowledge of noise parameters, if this parameter is not
 provided,
% we use observations to estimate such parameter.

% Based on our observations, it is possible that the covariance
 matrices
% become ill-conditioned sometimes (or close to that) and hence we use
% Cholesky factorization to fixate this issue.
%
```

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 (or subgaussianity parameter).
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.
prior_scale: The scaling factor for the prior (for the original
 version
    set this equal to 1).
prior_mu: Prior mean vector of gaussians.
prior_sig: Prior covariance matrix of gaussians.
sigma_start: The noise parameter (or subgaussanity parameter) to start
with. If true sigma is provided, this parameter is not used.
use_true_sigma_e: Whether to use true noise parameter, sigma_e, for
    construction of confidence sets or not.
to_estimate_sigma_e: Whether to estimate sigma_e using observations.

 This is only effective if the true noise parameter is not provided.
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.
```

Code:

```
function [regret, fractions] = runpriordependentTS(k, T, d, b, ...
    sigma_e, sigma_x, xmax, prior_mu, ...
    prior_sig , sigma_start, use_true_sigma_e,
 to_estimate_sigma_e, ...
    verbose, varargin)

warning('off','all');
sigma_e_hat = sigma_start;

if nargin==13 % 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

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);

cov_matrices_inv = zeros(d, d, k); % Covariance matrices of Gaussians.
mean_vector = zeros(d, k);
cov_inv_times_mean = zeros(d, k);
%Compute v

for i=1:k
    cov_matrices_inv(:, :, i) = inv(prior_sig);  % Initialize with
 prior.
    mean_vector(:, i) = prior_mu ;  % Mean of Gaussians.
    cov_inv_times_mean(:, i) = cov_matrices_inv(:, :, i) * ...
        mean_vector(:, i);
end

sampled_vectors = zeros(d,k);  % Samples drawn according to posterior.
residuals = zeros(T,1);


for t=1:T
    x = X(t,:)';
    % First: draw samples.
    for i=1:k
        cov_matrices_inv(:,:,i) = (cov_matrices_inv(:, :, i) ...
            + cov_matrices_inv(:, :, i)') / 2;
        [U, V]=eig(cov_matrices_inv(:, :, i));
        h=min(diag(V));
        if(min(h)<0)
            if(verbose == 0)
                fprintf('PD-TS: Inv. cov mat is ill-conditioned. \n');
                % Cap the eigenvalues to 1e-9.
                V = diag(max(h, 1e-9));
                cov_matrices_inv(:, :, i) = U * V * inv(U);
            end
        end
        sampled_vectors(:, i) = mean_vector(:, i) + transpose(...
            randn(1, d) * chol(inv(cov_matrices_inv(:, :, i))) ...
            );
    end
```

```
    % Second: find which arm to play.
    [~, arm_pulled]=max(x' * sampled_vectors);
    pull_ind(t, arm_pulled) = 1;

    %Third: compute reward and 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

    %Fourth: update parameters.
    if(noise_input==1)
        reward_vector(t, arm_pulled) = ourreward + e(t);
    else
        reward_vector(t, arm_pulled) = rewards(t, arm_pulled);
    end

    if(use_true_sigma_e==1)
        R=sigma_e;
    else
        R=sigma_e_hat;
    end


    sigma_post = inv(cov_matrices_inv(:, :, arm_pulled) + (1 /
R^2) ...
        * (x * x'));
    mu_post = sigma_post*((reward_vector(t, arm_pulled) / R^2) * x
+ ...
        cov_matrices_inv(:, :, arm_pulled) * mean_vector(:,
arm_pulled));
    cov_matrices_inv(:, :, arm_pulled) = inv(sigma_post);
    mean_vector(:,arm_pulled)= mu_post;


    %Fifth: update estimate of sigma_e.
    if(to_estimate_sigma_e==1)
         residuals(t) = reward_vector(t, arm_pulled) ...
             - x' * mean_vector(:, arm_pulled);
         if (t>k*d+1)
            sigma_e_hat = sqrt(sum(residuals.^2)/(t-d));
         end
    end
    if(verbose==1)
```

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

Published with MATLAB® R2015a