All gists GitHub New gist

anonymous / **bandit_simulations.py**
Created 3 years ago

bandit_simulations.py

```python
import numpy as np
from matplotlib import pylab as plt
#from mpltools import style # uncomment for prettier plots
#style.use(['ggplot'])

'''
function definitions
'''
# generate all bernoulli rewards ahead of time
def generate_bernoulli_bandit_data(num_samples,K):
    CTRs_that_generated_data = np.tile(np.random.rand(K),(num_samples,1))
    true_rewards = np.random.rand(num_samples,K) < CTRs_that_generated_data
    return true_rewards,CTRs_that_generated_data

# totally random
def random(estimated_beta_params):
    return np.random.randint(0,len(estimated_beta_params))

# the naive algorithm
def naive(estimated_beta_params,number_to_explore=100):
    totals = estimated_beta_params.sum(1) # totals
    if np.any(totals < number_to_explore): # if have been explored less than specified
        least_explored = np.argmin(totals) # return the one least explored
        return least_explored
    else: # return the best mean forever
        successes = estimated_beta_params[:,0] # successes
        estimated_means = successes/totals # the current means
        best_mean = np.argmax(estimated_means) # the best mean
        return best_mean

# the epsilon greedy algorithm
def epsilon_greedy(estimated_beta_params,epsilon=0.01):
    totals = estimated_beta_params.sum(1) # totals
    successes = estimated_beta_params[:,0] # successes
    estimated_means = successes/totals # the current means
    best_mean = np.argmax(estimated_means) # the best mean
    be_exporatory = np.random.rand() < epsilon # should we explore?
    if be_exporatory: # totally random, excluding the best_mean
        other_choice = np.random.randint(0,len(estimated_beta_params))
        while other_choice == best_mean:
            other_choice = np.random.randint(0,len(estimated_beta_params))
        return other_choice
    else: # take the best mean
        return best_mean

# the UCB algorithm using
# (1 - 1/t) confidence interval using Chernoff-Hoeffding bound)
# for details of this particular confidence bound, see the UCB1-TUNED section, slide 18, of:
# http://lane.compbio.cmu.edu/courses/slides_ucb.pdf
def UCB(estimated_beta_params):
    t = float(estimated_beta_params.sum()) # total number of rounds so far
    totals = estimated_beta_params.sum(1)
    successes = estimated_beta_params[:,0]
    estimated_means = successes/totals # sample mean
    estimated_variances = estimated_means - estimated_means**2
    UCB = estimated_means + np.sqrt( np.minimum( estimated_variances + np.sqrt(2*np.log(t)/totals), 0.25 ) * np.log(t)/totals )
    return np.argmax(UCB)

# the UCB algorithm - using fixed 95% confidence intervals
# see slide 8 for details:
# http://dept.stat.lsa.umich.edu/~kshedden/Courses/Stat485/Notes/binomial_confidence_intervals.pdf
def UCB_bernoulli(estimated_beta_params):
    totals = estimated_beta_params.sum(1) # totals
    successes = estimated_beta_params[:,0] # successes
```

```python
    estimated_means = successes/totals # sample mean
    estimated_variances = estimated_means - estimated_means**2
    UCB = estimated_means + 1.96*np.sqrt(estimated_variances/totals)
    return np.argmax(UCB)


# the bandit algorithm
def run_bandit_dynamic_alg(true_rewards,CTRs_that_generated_data,choice_func):
    num_samples,K = true_rewards.shape
    # seed the estimated params (to avoid )
    prior_a = 1. # aka successes
    prior_b = 1. # aka failures
    estimated_beta_params = np.zeros((K,2))
    estimated_beta_params[:,0] += prior_a # allocating the initial conditions
    estimated_beta_params[:,1] += prior_b
    regret = np.zeros(num_samples) # one for each of the 3 algorithms

    for i in range(0,num_samples):
        # pulling a lever & updating estimated_beta_params
        this_choice = choice_func(estimated_beta_params)

        # update parameters
        if true_rewards[i,this_choice] == 1:
            update_ind = 0
        else:
            update_ind = 1

        estimated_beta_params[this_choice,update_ind] += 1

        # updated expected regret
        regret[i] = np.max(CTRs_that_generated_data[i,:]) - CTRs_that_generated_data[i,this_choice]

    cum_regret = np.cumsum(regret)

    return cum_regret

'''
main code
'''
# define number of samples and number of choices
num_samples = 10000
K = 5 # number of arms
number_experiments = 100

regret_accumulator = np.zeros((num_samples,5))
for i in range(number_experiments):
    print "Running experiment:", i+1
    true_rewards,CTRs_that_generated_data = generate_bernoulli_bandit_data(num_samples,K)
    regret_accumulator[:,0] += run_bandit_dynamic_alg(true_rewards,CTRs_that_generated_data,random)
    regret_accumulator[:,1] += run_bandit_dynamic_alg(true_rewards,CTRs_that_generated_data,naive)
    regret_accumulator[:,2] += run_bandit_dynamic_alg(true_rewards,CTRs_that_generated_data,epsilon_greedy)
    regret_accumulator[:,3] += run_bandit_dynamic_alg(true_rewards,CTRs_that_generated_data,UCB)
    regret_accumulator[:,4] += run_bandit_dynamic_alg(true_rewards,CTRs_that_generated_data,UCB_bernoulli)

plt.semilogy(regret_accumulator/number_experiments)
plt.title('Simulated Bandit Performance for K = 5')
plt.ylabel('Cumulative Expected Regret')
plt.xlabel('Round Index')
plt.legend(('Random','Naive','Epsilon-Greedy','(1 - 1/t) UCB','95% UCB'),loc='lower right')
plt.show()
```