MAS_4

November 29, 2020

Leonard Vorbeck

No Group

Student Number : 2709813

[34]:
```
import pandas as pd
import numpy as np
from scipy.stats import pearsonr as r
import matplotlib.pyplot as plt
from scipy.stats import norm, uniform
%config InlineBackend.figure_format = 'retina'
plt.rcParams["figure.dpi"] = 93
plt.rcParams["figure.figsize"] = (12,7)
plt.style.use("default")
```

0.0.1 4.1.1

For i.i.d sample $X_1, X_2, \ldots, X_n \sim N(0,1)$ with pdf f we have

$$\bar{X}_n = \frac{1}{n}\sum_{i=1}^{n} X_i$$

with

$$E[\bar{X}_n] = \mu = 0$$

and

$$Var[\bar{X}_n] = \frac{\sigma^2}{n} = \frac{1}{n}$$

For $g(x) = \cos^2(x)$ the MC-estimate with samples drawn from f is

$$E[g(X)] \approx \widetilde{g_n}(x) = \frac{1}{n}\sum_{i=1}^{n} g(x_i)$$

However, the MC-estimator is a random variable itself :

$$\widetilde{g_n}(X) \sim N(\mu, \frac{\sigma^2}{n})$$

The uncertainty of the MC-estimator is given by its variance:

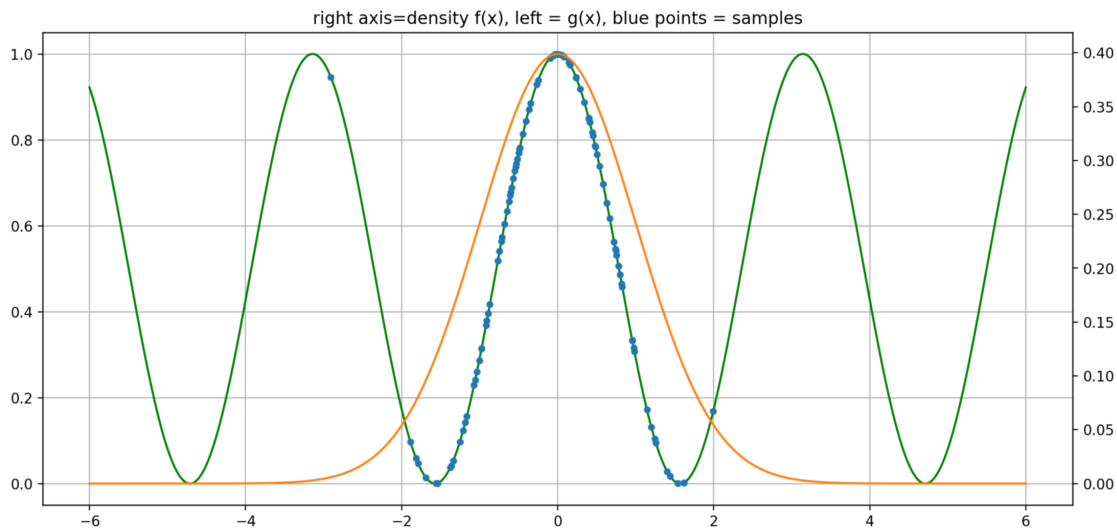
$$\mathrm{Var}\left(\widetilde{g_n}(X)\right) = Var[E[g(X)]] \approx \frac{\sigma^2}{n}$$

[2]:
```
# Define function
def g(x): return np.cos(x)**2
# Instantiate density
N = norm(0, np.sqrt(1))
```

[36]:
```
n = 100
draws = pd.Series(N.rvs(n))
sim = pd.Series([g(x) for x in draws.values], index=draws.values)
func = pd.Series([g(x) for x in np.linspace(-6, 6, 10_000)],
                 index=np.linspace(-6, 6, 10_000))
pdf_x = pd.Series([N.pdf(x) for x in np.linspace(-6, 6, 10_000)],
                 index=np.linspace(-6, 6, 10_000))
```

[37]:
```
ax = func.plot(color="green",figsize=(13,6))
sim.plot(ax=ax, style=".",ms=7.73)
pdf_x.plot(ax=ax, secondary_y=True,grid=True,title="right axis=density f(x),␣
↪left = g(x), blue points = samples")
```

[37]: <AxesSubplot:label='be37b585-0741-4cc6-9e12-66495072af7c'>



[21]:
```python
sim.mean(), sim.var()
```

[21]:
```
(0.5642320564431826, 0.11467367026497269)
```

The blue dots represent samples $g(X_i)$. The orange line is the original density and the green line is $g(x)$. The chart shows nicely why the MC-estimate is sharp for large values of n.

The simulation reveals $E[g(x)] = 0.56$. The estimate has a variance of $Var[g(x)] = 0.12$.

A better way of estimating uncertainty is to look at the variance of the *estimator* and not the variance of the *estimate*.

[27]:
```python
n_draws = 100
n_sims = 50
sims =pd.Series([pd.Series([g(x) for x in N.rvs(n_draws)]).mean() for n_sim in␣
 ↪range(n_sims)])
```

Repeating the experiment 50 times ($n = 100$ each) yields the distribution of the MC-estimator, with

$$E[E[g(x)]] = 0.5675 = E[g(x)] = 0.56 \approx \mu Var[E[g(x)]] = 0.0012 = \frac{Var[g(x)]}{n} = \frac{0.12}{100} \approx \frac{\sigma^2}{n}$$

[28]:
```python
sims.mean(), sims.var() # For 50 simulation with n=100 each [MC-ESTIMATOR DIST]
```

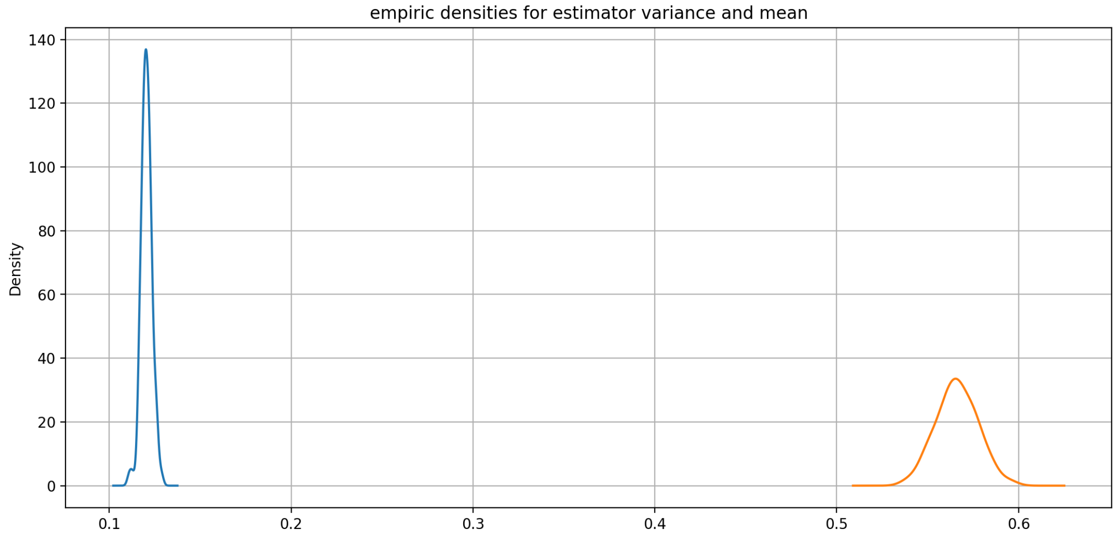
[28]:
```
(0.5595546837689471, 0.000924285604112229)
```

[29]:
```python
sim.mean(), sim.var() # Single simulation with n=100 [ESTIMATE DIST]
```

[29]:
```
(0.5642320564431826, 0.11467367026497269)
```

[30]:
```python
# CLT Induce, blue : VAR[g(x)], orange : AVG[g(x)]
ax = pd.Series([pd.Series([g(np.random.normal(0,1)) for _ in range(1000)]).
 ↪var() for _ in range(100)]).plot.density()
pd.Series([pd.Series([g(np.random.normal(0,1)) for _ in range(1000)]).mean()␣
 ↪for _ in range(100)]).plot.density(ax=ax,
                                                                          ␣
 ↪                                  grid=True,figsize=(13,6),
                                                                          ␣
 ↪                                title="empiric densities for estimator␣
 ↪variance and mean")
```

[30]:
```
<AxesSubplot:title={'center':'empiric densities for estimator variance and
mean'}, ylabel='Density'>
```



0.0.2 4.1.2

For the plain vanilla p-test we have $df = 10 - 2 = 8$ and critical values of -0.632 and 0.632, which translates to insignificant values for $r \in [-0.632, 0.632]$ and therefore also for $r = 0.3$. Note that p-values are never useful other than for synthetic data where the assumptions of the significance test are met.

Using MC, we can setup a simulation to evaluate how random the found result might be, without assuming the shapes of f_S and f_A.

Let $A \leftarrow [a_1, a_2, ..., a_{10}]$ and $S \leftarrow [s_1, s_2, ..., s_{10}]$ with linear correlation $r(A, S)$

Now to check if $r(A, S)$ is non-random (i.e significant), simulate the initial experiment n times to obtain N different values for $r(A, S)$, where in each of the N simulations, S (or both A and S) is permuted ($s_i^t \neq s_i^{t+1}$).

Now after obtaining N values $(r_1, r_2, ..., r_N)$, for a one-tailed test sort the correlations, for a two tailed test sort the absolute correlations. The result is a sorted set R indexed i. Now the c CI level corresponds to the $N \times c$-th element of R, which is $r_{N\times c}$, e.g, r_{990} for $N = 1000$ and $c = .99$.

0.0.3 4.2.1

Let $\varphi(X) = X^2$ and $X \sim N(0, 1)$ with density f and $g \sim U(-5, 5)$ with density g. Then

$$E_f[\varphi(X)] = E_g\left[\varphi(X)\frac{f(X)}{g(X)}\right] \approx \frac{1}{n}\sum_{i=1}^{n} \varphi\left(X_i\right)\frac{f\left(X_i\right)}{g\left(X_i\right)}$$

where X_i is drawn from g.

First of all, $g(X_i)$ will never be 0, while $f(X_i)$ gets very close to zero for unlikeli values, especially if $|X_i| > 1.96$.

In particular, any X_i drawn from U yields $g(X_i) = \frac{1}{10}$. Therefore likelihood ratio can be simplified

$$\frac{f\left(X_i\right)}{g\left(X_i\right)} \leftarrow f\left(X_i\right) \times 10$$

Therefore I predict the following *estimate*.

$$E_f[\varphi(X)] \approx \frac{1}{n} \sum_{i=1}^{n} 10\varphi\left(X_i\right) f\left(X_i\right)$$

and

$$VAR_f[\varphi(X)] \approx \frac{1}{n} \sum_{i=1}^{n} [10\varphi\left(X_i\right) f\left(X_i\right) - E_f[\varphi(X)]]^2$$

And subsequently the *estimator*

$$E[E_f[\varphi(X)]] = \mu Var[E_f[\varphi(X)]] = \frac{\sigma^2}{n}$$

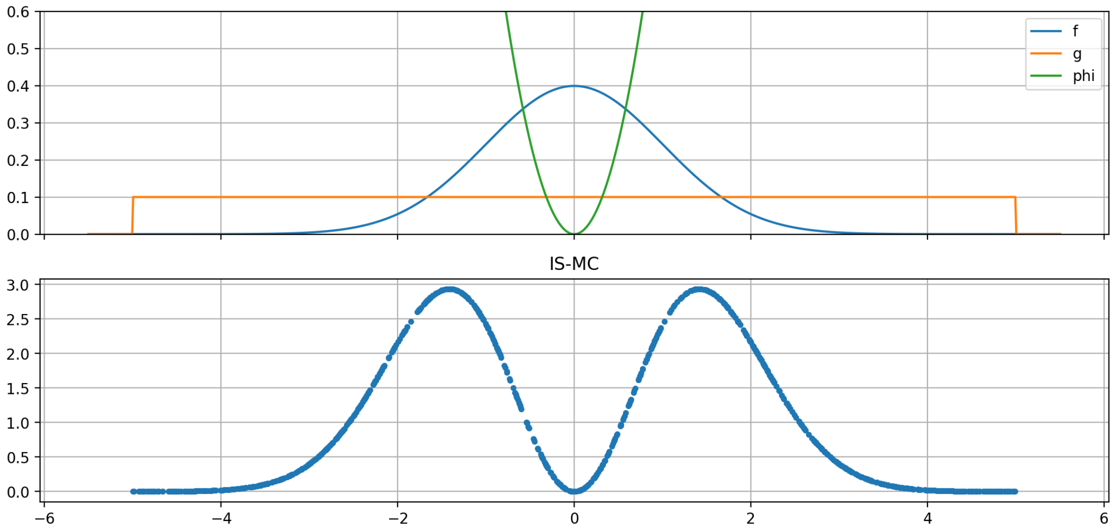
[39]:
```
# Instanciate densities
N = norm(0, np.sqrt(1))
U = uniform(loc=-5, scale=10)
```

[40]:
```
n_draws = 1000
draws = pd.Series(U.rvs(n_draws))
sim = pd.Series([(xi**2)*10*N.pdf(xi) for xi in draws],index=draws)
sim.mean(), sim.var()
```

[40]:
```
(1.0327807495961905, 1.1611606847404532)
```

[41]:
```
fig, axes = plt.subplots(ncols=1, nrows=2, figsize=(15,6), sharex=True)
df = pd.DataFrame({"f" : [N.pdf(x) for x in np.linspace(-5.5,5.5,1000)],
              "g" : [U.pdf(x) for x in np.linspace(-5.5,5.5,1000)],
              "phi" : [ x**2 for x in np.linspace(-5.5,5.5,1000)]},
             index=np.linspace(-5.5,5.5,1000))
ax = df.plot(grid=True, figsize=(13,6), ylim=(0,.6),ax=axes[0])
sim.plot(style=".",ax=axes[1],grid=True)
axes[1].set_title("IS-MC")
```

[41]:
```
Text(0.5, 1.0, 'IS-MC')
```



The IS-MC-estimate is $E_f[X^2] \approx 1$, with $VAR_f[X^2] \approx 1.05$. (Please note !!! : $E_f[X^2] = E_g\left[X^2\frac{f(X)}{g(X)}\right]$, similar to VAR, as described earlier).

[42]:
```
n_draws = 1000
n_sims = 50
sims = pd.Series([pd.Series([(xi**2)*10*N.pdf(xi) for xi in U.rvs(n_draws)]).
↪mean() for n_sim in range(n_sims)])
```

If we look at the distribution of the IS-MC-Estimator, for example 50 simulations, each $n = 100$ draws, we get :

[43]:
```
sims.mean(), sims.var() # For 50 simulation with n=1000 each [IS-MC-ESTIMATOR␣
↪DIST]
```

[43]: (1.004474884074828, 0.001067546583881102)

[44]:
```
sim.mean(), sim.var() # For single sim with n=1000 [IS-MC-ESTIMATE DIST]
```

[44]: (1.0327807495961905, 1.1611606847404532)

[45]:
```
sims.var() * n_draws # Fits theoretical Approximation
```

[45]: 1.067546583881102

Repeating the experiment 50 times ($n = 1000$ each) yields the distribution of the MC-estimator, with

$$E[E[g(x)]] = 1 = E[g(x)] = 1 \approx \mu Var[E[g(x)]] = 0.00099 = \frac{Var[g(x)]}{n} = \frac{1}{1000} \approx \frac{\sigma^2}{n}$$

0.0.4 4.2.2

From the exercise it is not clear whether f_X is the target or proposal distribution. I assume that you meant f_X as the target dist using again U to draw from. In this case, we should use

$$g \sim U(-1, 1)$$

because f_X is not defined for values outside of $[-1, 1]$.

Let $\varphi(X) = X^2$ and $x \in [-1, 1]$ with target density $f(x) = \frac{1+\cos(\pi x)}{2}$ and proposal density $g \sim U(-1, 1)$.

Again

$$E_f[\varphi(X)] = E_g\left[\varphi(X)\frac{f(X)}{g(X)}\right] \approx \frac{1}{n}\sum_{i=1}^{n}\varphi\left(X_i\right)\frac{f\left(X_i\right)}{g\left(X_i\right)}$$

For $g \sim (-1, 1)$ we have

$$\frac{f\left(X_i\right)}{g\left(X_i\right)} \leftarrow f\left(X_i\right) \times 2$$

s.t.

$$E_f[\varphi(X)] \approx \frac{1}{n}\sum_{i=1}^{n}2\varphi\left(X_i\right)f\left(X_i\right)$$

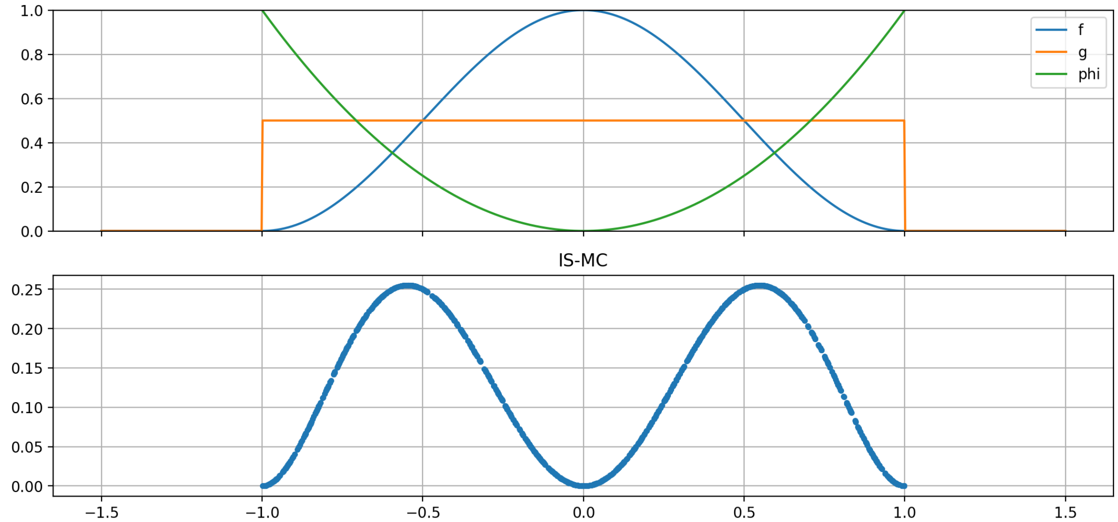
[46]:
```
# Define new target density
def fx(x) : return (1+np.cos(np.pi*x))/2 if -1 <= x <= 1 else 0
# Instanciate new proposal density
U = uniform(loc=-1, scale=2)
```

[47]:
```
n_draws = 1000
draws = pd.Series(U.rvs(n_draws))
sim = pd.Series([(xi**2)*2*fx(xi) for xi in draws],index=draws)
sim.mean(), sim.var()
```

[47]: (0.12879984635877567, 0.0083807124088555)

[48]:
```
fig, axes = plt.subplots(ncols=1, nrows=2, figsize=(15,6), sharex=True)
df = pd.DataFrame({"f" : [fx(x) for x in np.linspace(-1.5,1.5,1000)],
              "g" : [U.pdf(x) for x in np.linspace(-1.5,1.5,1000)],
              "phi" : [ x**2 for x in np.linspace(-1.5,1.5,1000)]},
             index=np.linspace(-1.5,1.5,1000))
ax = df.plot(grid=True, figsize=(13,6), ylim=(0,1),ax=axes[0])
sim.plot(style=".",ax=axes[1],grid=True)
axes[1].set_title("IS-MC")
```

[48]: Text(0.5, 1.0, 'IS-MC')



The IS-MC-estimate is $E_f[X^2] \approx 0.13$, with $VAR_f[X^2] \approx 0.008$. (Please note !!! : $E_f[X^2] = E_g\left[X^2 \frac{f(X)}{g(X)}\right]$, similar to VAR, as described earlier).

[49]:
```
n_draws = 1000
n_sims = 50
sims = pd.Series([pd.Series([(xi**2)*2*fx(xi) for xi in U.rvs(n_draws)]).mean()␣
↪for n_sim in range(n_sims)])
```

If we look at the distribution of the IS-MC-Estimator, for example 50 simulations, each $n = 1000$ draws, we get :

[52]:
```
sims.mean(), sims.var() # For 50 simulation with n=1000 each [IS-MC-ESTIMATOR␣
↪DIST]
```

[52]: (0.13101941825232205, 6.673497253745432e-06)

[53]:
```
sim.mean(), sim.var() # For single sim with n=1000 [IS-MC-ESTIMATE DIST]
```

[53]: (0.12879984635877567, 0.0083807124088555)

[54]:
```
sims.var() * n_draws # Fits theoretical Approximation
```

[54]: 0.006673497253745432

Repeating the experiment 50 times ($n = 1000$ each) yields the distribution of the MC-estimator, with

$$E[E[g(x)]] = 0.13 = E[g(x)] = 0.13 \approx \mu Var[E[g(x)]] = 9.322359799403895e^{-06} = \frac{Var[g(x)]}{n} = \frac{0.008}{1000} \approx \frac{\sigma^2}{n}$$

0.0.5 4.3.1

Let

$$KL(f,g) = \int_{-\infty}^{\infty} f(x) \log \frac{f(x)}{g(x)} dx f \sim N\left(\mu, \sigma^2\right) \text{ and } g \sim N\left(\nu, \tau^2\right)$$

plug the two univariate gaussians into the equation.

$$\ldots\ldots KL(f,g) = \log\left(\frac{\tau}{\sigma}\right) - \frac{1}{2} + [\sigma^2 + (\mu-\nu)^2] \times 2\tau^{-2}$$

yields something like that (derivation in latex is impossible..)

Obviously KL is not symmetric ($KL(f,g) \neq KL(g,f)$), $\log\left(\frac{\tau}{\sigma}\right)$ swaps coefficients

[220]:
```
np.log((b/a))
```

[220]: 0.49247648509779424

[55]:
```
# In code
def KL(p, q): return np.log(q.std()/p.std()) + (p.var() + (p.mean() - q.
↪mean())**2)/(2*q.var()) - (1/2)
```

0.0.6 4.3.2

MC-Estimators

$$\hat{KL}(f,g) = \frac{1}{n} \sum_{i \sim f}^{n} log \frac{f(x_i)}{g(x_i)}$$

$$\hat{KL}(g,f) = \frac{1}{n} \sum_{i \sim g}^{n} log \frac{g(x_i)}{f(x_i)}$$

Let

$$f_X \leftarrow N(10,2) g_X \leftarrow N(1,3)$$

[56]:
```
def KL(p, q): return np.log(q.std()/p.std()) + (p.var() + (p.mean() - q.
↪mean())**2)/(2*q.var()) - (1/2)
```

[57]:
```
# Instantiate densities
f = norm(10,np.sqrt(2))
g = norm(1,np.sqrt(3))
```

[58]:
```
# MC estimate for KL(f, g)
estimate = []
n=5000
```

```python
for xi in f.rvs(n):
    f_xi = f.pdf(xi)
    g_xi = g.pdf(xi)
    res = np.log(f_xi / g_xi)
    estimate.append(res)
pd.Series(estimate).mean()
```

[58]: 13.554201304736548

The MC-estimate of $KL(f, g) = \hat{KL}(f, g) = 13.548$. This is consistent when using the previously derived formula.

[59]:
```python
KL(f, g)
```

[59]: 13.536065887387418

[60]:
```python
# KL(g, f)
estimate = []
n=5000
for xi in g.rvs(n):
    f_xi = f.pdf(xi)
    g_xi = g.pdf(xi)
    res = np.log(g_xi / f_xi)
    estimate.append(res)
pd.Series(estimate).mean()
```

[60]: 20.155027335922334

[61]:
```python
KL(g, f)
```

[61]: 20.297267445945913

Again, $KL(g, f) = \hat{KL}(g, f)$

0.0.7 5.1 k-armed-bandit

Let

$$L(t) = E\left(\sum_{i=1}^{t} \left(q^* - q\left(a_i\right)\right)\right)$$

be the total (cumulative) regret at time t.

Let

$$Q_t(a) \simeq E\left[R_t \mid A_t = a\right] = \frac{\sum_{i=1}^{N_a} r_i}{N_a}$$

be the estimate for action a at time t. (i.e. the sample mean).

with update rule that is equivalent to sample-mean-sharpening (i.e linear)

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{N_a + 1}\left[r_{t+1} - Q_t(a)\right]$$

0.0.8 5.1.0 Theoretical optimal lower bound

$$L(t) \geq A\log(t) \quad \text{where} \quad A = \sum_{a:\Delta_a \neq 0} \frac{\Delta_a}{KL\left(f_a || f_a^*\right)} \quad \text{and} \quad \Delta_a = q^* - q(a)$$

[83]:
```
def L(t, k_dists):
    q_star = [_ for _ in k_dists if _.mean() == max([__.mean() for __ in␣
↪k_dists])][0]
    A = sum([(q_star.mean() - q_a.mean())/KL(q_a, q_star) for q_a in k_dists if␣
↪q_a != q_star])
    return np.log(t) * A
```

0.0.9 5.1.1 ϵ-greedy

Action probabilites are handy in this approach :

$$P(a^*) = (1 - \epsilon) \text{ for } a^* = \arg\max_a Q_t(a)$$
$$P(A_{-a^*}) = \epsilon$$

[63]:
```
def greedy_bandit(e,k, init_dists, T,
                  opt_init=1.05):
    k_dists = init_dists
    q_star = [_ for _ in k_dists if _.mean() == max([__.mean() for __ in␣
↪k_dists]) ][0]
    q_init = q_star.mean()*opt_init

    # Placeholder for Q values
    Q = { t : {a : q_init if t == 0 else None for a in range(k)} for t in␣
↪range(T) }
    # Placeholder for action choices
    N = { a : 1 for a in range(k)}
    # Placeholder for regrets
    L = { t : None for t in range(T)}
    # Placeholder for rewards
    R = { t : None for t in range(T)}

    for t in range(T-1):
        ## Choice policy
```

```
        argmax_a = [a for a in Q[t] if Q[t][a] == max([Q[t][a] for a in␣
↪Q[t]])][0]
        A_ = [a for a in Q[t] if a != argmax_a]
        choice = argmax_a if np.random.binomial(1, e) == 0 else np.random.
↪choice(A_)
        # Get reward from chosen density
        reward = k_dists[choice].rvs(1)[0]
        # Upadte chosen action estimate
        Q[t+1][choice] = (Q[t][choice] + (1/(N[choice]+1))*(reward -␣
↪Q[t][choice] ))
        # Add 1 obs to N_choice
        N[choice] += 1
        # Update all other
        for a in Q[t]:
            if a == choice:
                continue
            Q[t+1][a] = Q[t][a]
        ## Regret
        regret = q_star.mean() - k_dists[choice].mean()
        L[t] = regret
        ## Reward
        R[t] = reward

    return dict(Q=Q, R=R, L=L, N=N, dists=k_dists, q_star=q_star)
```

0.0.10 5.1.2 UCB

UCB is even simpler, with decision rule according to the most promising action according to its upper boundary.

Choose a s.t.

$$\arg\max_a \left(Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right)$$

[153]:
```
def ucb_bandit(c,k, init_dists, T):
    k_dists = init_dists
    q_star = [_ for _ in k_dists if _.mean() == max([__.mean() for __ in␣
↪k_dists]) ][0]
    q_init = 0
    u_init = 999
    # Placeholder for U values
    #U = { t : {a : u_init if t == 0 else None for a in range(k)} for t in␣
↪range(T) }
    # Placeholder for Q values
```

```python
    Q = { t : {a : q_init if t == 0 else None for a in range(k)} for t in range(T) }
    # Placeholder for action choices
    N = { a : 0 for a in range(k)}
    # Placeholder for regrets
    L = { t : None for t in range(T)}
    # Placeholder for rewards
    R = { t : None for t in range(T)}

    for t in range(T-1):
        ## Choice policy
        U = { a : c * np.sqrt((np.log(t)/N[a])) for a in range(k)}
        choice_values = [ U[a] + Q[t][a] for a in range(k)]
        ucb_argmax_a = choice_values.index(max(choice_values))
        choice = ucb_argmax_a
        A_ = [a for a in Q[t] if a != choice]
        # Get reward from chosen density
        reward = k_dists[choice].rvs(1)[0]
        # Upadte chosen action estimate
        Q[t+1][choice] = (Q[t][choice] + (1/(N[choice]+1))*(reward - Q[t][choice] ))
        # Add 1 obs to N_choice
        N[choice] += 1
        # Update all other
        for a in Q[t]:
            if a == choice:
                continue
            Q[t+1][a] = Q[t][a]
        ## Regret
        regret = q_star.mean() - k_dists[choice].mean()
        L[t] = regret
        ## Reward
        R[t] = reward

    return dict(Q=Q, R=R, L=L, N=N, dists=k_dists, q_star=q_star)
```

The arm densities are instantiated as follows :

[160]:
```python
k = 5
unit_std = 30
mean_space=list(range(-100,500,30))
k_dists = [norm(np.random.choice(mean_space),unit_std) for _ in range(k)]
[(a.mean(), a.std()) for a in k_dists]
```

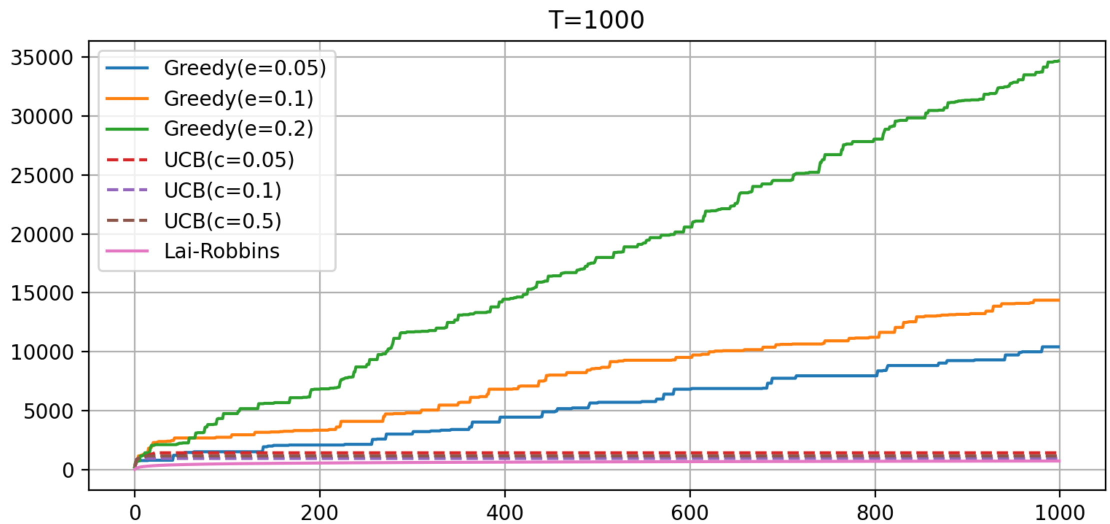
[160]: [(230.0, 30.0), (20.0, 30.0), (380.0, 30.0), (410.0, 30.0), (440.0, 30.0)]

[163]:
```
e_g = pd.DataFrame({"Greedy(e=%s)" % e :
↪greedy_bandit(e=e,k=k,init_dists=k_dists,T=1000)["L"] for e in [0.05, 0.1, 0.
↪2]})
ucb = pd.DataFrame({"UCB(c=%s)" % c :
↪ucb_bandit(c=c,k=k,init_dists=k_dists,T=1000)["L"] for c in [0.05, 0.1, 0.
↪5]})
df = e_g.merge(ucb, right_index=True, left_index=True)
df_cumul = df.cumsum()
df_cumul["Lai-Robbins"] = [L(t+1, k_dists) for t in df.index.values]
correct = df[df == 0].replace(0,1).replace(np.nan, 0)
```

```
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:19:
RuntimeWarning: divide by zero encountered in log
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:19:
RuntimeWarning: invalid value encountered in sqrt
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:19:
RuntimeWarning: invalid value encountered in double_scalars
/opt/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:19:
RuntimeWarning: divide by zero encountered in double_scalars
```

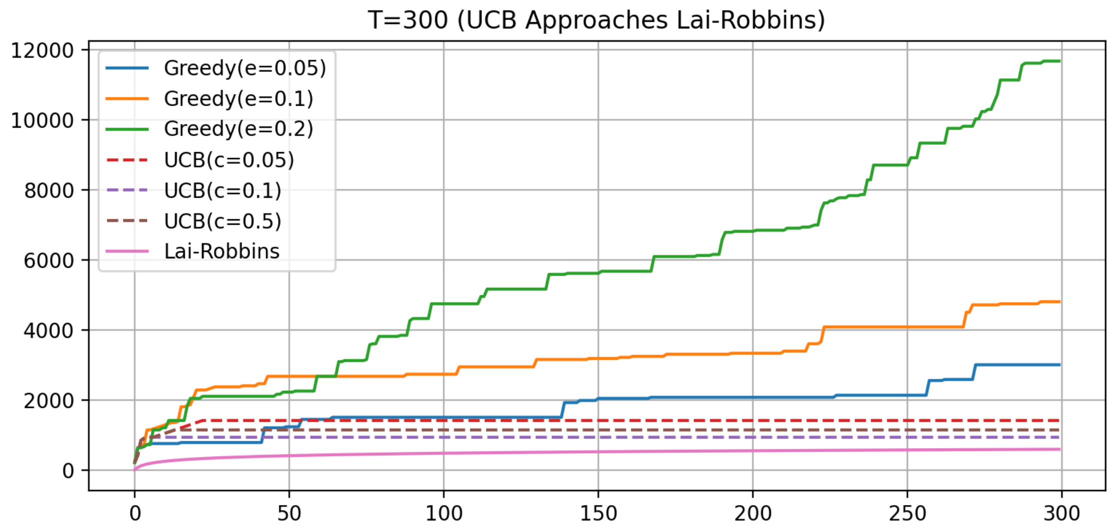
[164]:
```
df_cumul.plot(style=["-","-","-","--","--","--"],grid=True,
↪figsize=(9,4),title="T=1000")
```

[164]: <AxesSubplot:title={'center':'T=1000'}>



[216]:
```
df_cumul[0:300].plot(style=["-","-","-","--","--","--"],grid=True,
↪figsize=(9,4),title="T=300 (UCB Approaches Lai-Robbins)")
```

[216]: <AxesSubplot:title={'center':'T=300 (UCB Approaches Lai-Robbins)'}>



0.0.11 Correct choices

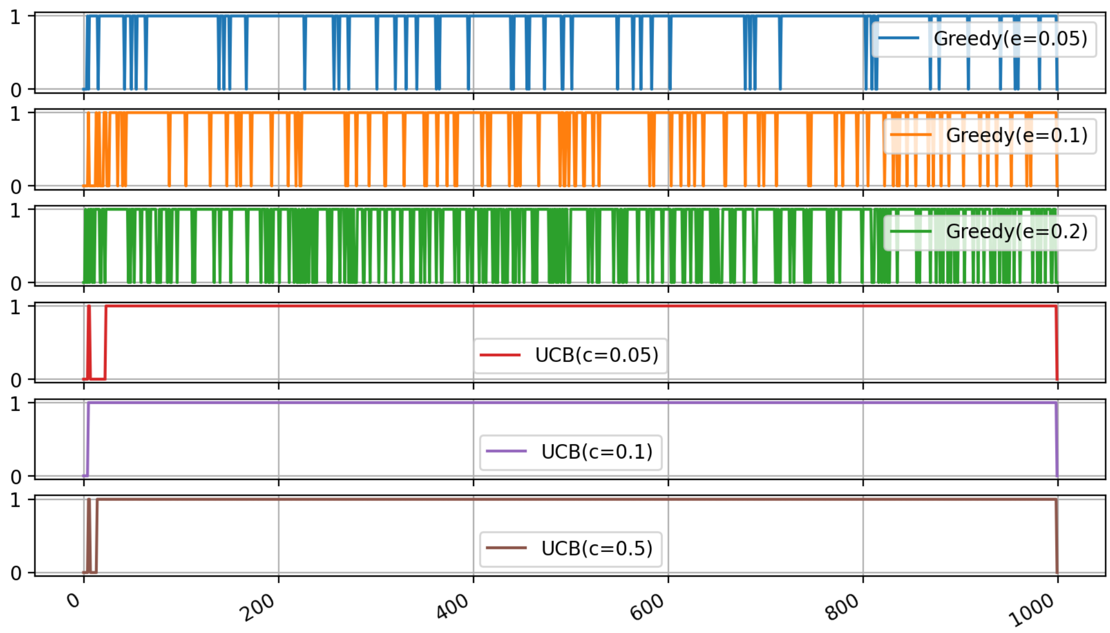
Distribution :

[167]:
```
correct.sum()/len(correct)
```

[167]:
```
Greedy(e=0.05)    0.946
Greedy(e=0.1)     0.896
Greedy(e=0.2)     0.798
UCB(c=0.05)       0.978
UCB(c=0.1)        0.994
UCB(c=0.5)        0.987
dtype: float64
```

[218]:
```
correct.plot(grid=True, style=".-",ms=0.2,figsize=(10,6),subplots=True)
```

[218]:
```
array([<AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>, <AxesSubplot:>,
       <AxesSubplot:>, <AxesSubplot:>], dtype=object)
```



1
0
Greedy(e=0.05)
Greedy(e=0.1)
Greedy(e=0.2)
UCB(c=0.05)
UCB(c=0.1)
UCB(c=0.5)
0
200
400
600
800
1000


```
[ ]:
```