Contents

1 Executive Summary

In this notebook, I approach the Multi-armed bandit problem and introducing its algorithms as better techniques comparing to the A/B testing approach. More precisely, I discuss how an algorithmic approach can solve some limitations of A/B testings in real life scnerios.

2 Introduction

2.1 What is A/B Testing

A/B testing, at its most basic, is a way to compare two versions of something to figure out which performs better. However, in real life scenarios (testing marketing strategies, etc), we usually need to handle more than two options at once.(i.e., A, B, C, D, E, F, G, etc). New options can be added or removed at any time, and we may use th ebest option until it stops being the best choice. So instead of us going back and forth to check how every change we made is changing our business and picking the best options we better follow algorithmic approaches instead of A/B teasting. This is actually where we can highlight the potential of Multi-armed bandit algorithms such as epsilon-greedy, epsilon-decreasing, epsilon-first, UCB, Bayesian UCB and EXP3.

2.2 Exploration vs Exploitation

Our daily life decision making involves a fundamental choice; **Exploration**, (think of it as exploring your surrounding enviornment randomly to find your answer) where we gather more information that might lead us to better decisions in the future or **Exploitation**, (think of it as exploiting what you think is the best at the moment) where we make the best decision given your current information.
In such settings we may think, ok we have only two options either choose a random action or do something which optimize our payoff. That is right but there are many ways we can do this optimization.

2.3 Multi-armed bandit algorithms

Imagin that we are faced repeatedly with a choice among k different options, or actions. After each choice we receive a numerical reward chosen from a stationary probability distribution that depends on the action we selected. Our objective is to maximize the expected total reward over some time period, for example 1000 action selections, or time steps. Each of the k actions has an expected or mean reward given that that action is selected. We can call this the value of that action. So our job is to concentrate on our best actions.But how?

1) Just pick actions randomly (No Exploitation)

2) Greedy Approach (No Exploration)

3) Epsilon-greedy Approach (10% Random action 90% go for the best action).

4) Epsilon-decreasing Approach(keeping epsilon dependent on time)

5) Espilon-first Approach(exploration only for predermined amount of time)

6) UCB Approach(the more uncertain we are about an action, the more important it becomes to explore that action.)

7) EXP Approach(keep exploration always in the loop even with small probabilities)

3 Project

To depict the usefullness and potentials of bandit algorithms, I consider an advertising scnerio in which I let two algorithms do their job in finding the best choice quickly among a set of digital advertisements presented on the first web page of a website. I will compare a purly random strategy versus UCB (Upper Confidence Bound Algorithm) to depict the benefits of bandit algorithms. I set the number of digital advertisement as 10.

3.0.1 Importing the required libraries

In [22]:
```
import numpy as np
import math
import random
import matplotlib.pyplot as plt
import pandas as pd
```

3.0.2 Loading the Data

In [138]:
```
df = pd.read_csv('Downloads/Ads_Selection_Optimisation.csv')
```

In [24]:
```
df.head()
```

Out[24]:

	Ad 1	Ad 2	Ad 3	Ad 4	Ad 5	Ad 6	Ad 7	Ad 8	Ad 9	Ad 10
0	1	0	0	0	1	0	0	0	1	0
1	0	0	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	0	0	0
3	0	1	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	0	0	0	0

In [27]:
```
df.values[1, 8]
```

Out[27]: 1

Someone (record number 1) picked ad9 and we can see that the value (reward) of cliking this ad has been 1 for the player. User 0 also clicked on ad9 so may be this is a high rewarding add for our business. We have to find out.

3.0.3 Purely Random Strategy

Contents

In [139]:
```
N = 10000 # Number of experiments
d = 10    # Number of Advertisements
ads_selected = []
total_reward = 0
for n in range(0, N): # Let us pick 10000 Ad randomly from the dataset and see what
    #could be our best reward out of following this strategy
    ad = random.randrange(d)
    ads_selected.append(ad)
    reward = df.values[n, ad]
    total_reward = total_reward + reward
print('total_reward: {:.2f}'.format(total_reward))
```

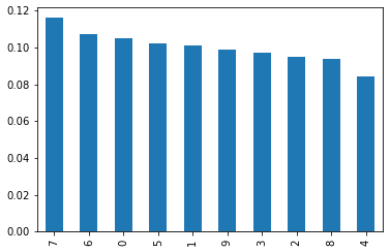
```
total_reward: 1245.00
```

In [140]:
```
# Let us look at the last 1000 trials
pd.Series(ads_selected).tail(1000).value_counts(normalize=True)
```

Out[140]:
```
7    0.116
6    0.107
0    0.105
5    0.102
1    0.101
9    0.099
3    0.097
2    0.095
8    0.094
4    0.084
dtype: float64
```

In [141]:
```
pd.Series(ads_selected).tail(1000).value_counts(normalize=True).plot(kind='bar')
```

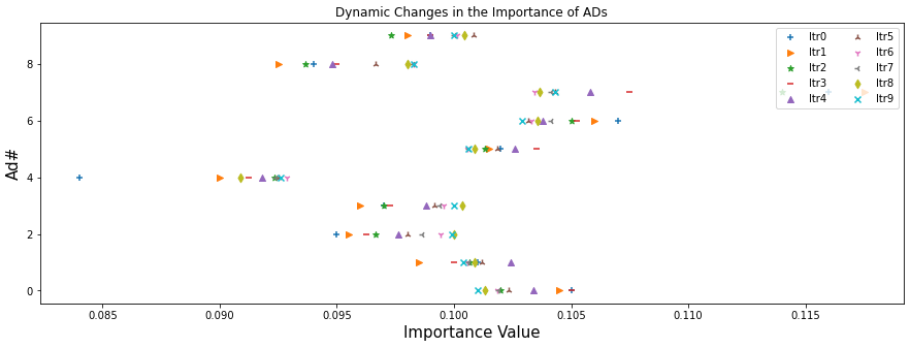
Out[141]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x214760a3188>
```



It seems that the pure random selection strategy treats the Ads selection equality, So the algorithm basiclaly doenot learn anything here. The highest rank goes to Ad# 7 which is not the right answer for our business.

In [142]:
```
rows, cols = (10, 10)
data= []
arr = [[0 for i in range(cols)] for j in range(rows)]
for n in range(0, 10):
    for i in range(0, d):
        X = pd.Series(ads_selected).tail((n + 1)*1000).value_counts(normalize=True)
        arr[n][i] = X
        data.append(X)
```

In [143]:
```
markerlist= ['+', ">", "*", "_", "^", "2", "1", "3", "d" ,"x"]
fig, ax = plt.subplots(figsize=(15,5))
for j in range(cols):
    ax.scatter(arr[j][0], y=arr[0][0].index, label= "Itr" + str(j),  marker=markerlist[j])
ax.legend(loc='upper right', ncol=2)
ax.set_xlabel('Importance Value', fontsize=15)
ax.set_ylabel('Ad#', fontsize=15)
ax.set_title('Dynamic Changes in the Importance of ADs')
plt.show()
```



As we an see the dunamics of changes in the importance of Ads do not follow a certain pattern.

Therefore we can conlcude that With a purly random selection strategy we are unable to find the true ranking of Ads. We will try UCB algorithm next.

3.0.4 UCB

Contents

In [144]:
```
N = 10000  # Number of experiments
d = 10     # Number of Advertisements
ads_selected = []
numbers_of_selections = [0] * d
sums_of_reward = [0] * d
total_reward = 0
for n in range(0, N):
    ad = 0
    max_upper_bound = 0
    for i in range(0, d):
        if (numbers_of_selections[i] > 0):
            average_reward = sums_of_reward[i] / numbers_of_selections[i]
            delta_i = math.sqrt(2 * math.log(n+1) / numbers_of_selections[i])
            upper_bound = average_reward + delta_i
        else:
            upper_bound = 1e400
        if upper_bound > max_upper_bound:
            max_upper_bound = upper_bound
            ad = i
    ads_selected.append(ad)
    numbers_of_selections[ad] += 1
    reward = dataset.values[n, ad]
    sums_of_reward[ad] += reward
    total_reward += reward
print('total_reward: {:.2f}'.format(total_reward))
```

```
total_reward: 2125.00
```

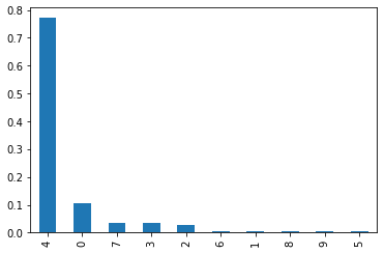
The calculated total_reward out of UCB algorithm is bigger than the pure random seletion stratgey (2125.00 > 1243) so for sure the UCB algorithm is learning some patterns about our data.

In [145]:
```
pd.Series(ads_selected).tail(1000).value_counts(normalize=True)
```

Out[145]:
```
4    0.771
0    0.106
7    0.034
3    0.034
2    0.026
6    0.007
1    0.007
8    0.006
9    0.005
5    0.004
dtype: float64
```

In [146]:
```
pd.Series(ads_selected).tail(1000).value_counts(normalize=True).plot(kind='bar')
```

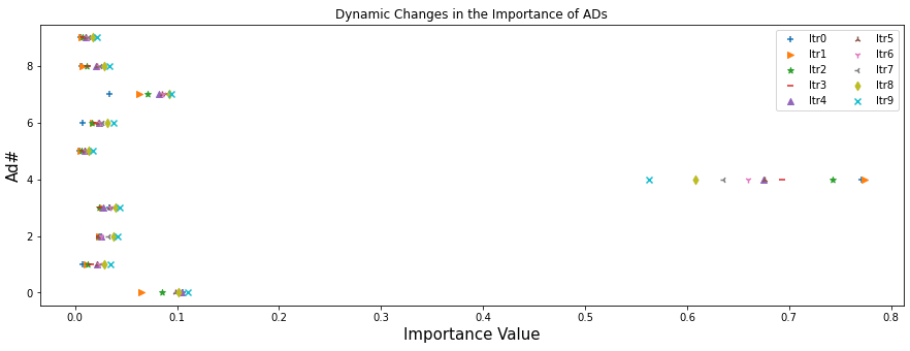
Out[146]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2147660a088>
```



The results show that Ad4 is the most reward generating Ads. But let us have a look at the dynamic changes of the calculated total reward for both algorithms.

In [147]:
```
rows, cols = (10, 10)
data= []
arr = [[0 for i in range(cols)] for j in range(rows)]
for n in range(0, 10):
    for i in range(0, d):
        X = pd.Series(ads_selected).tail((n + 1)*1000).value_counts(normalize=True)
        arr[n][i] = X
        data.append(X)
```

In [148]:
```
markerlist= ['+', ">", "*", "_", "^", "2", "1", "3", "d" ,"x"]
fig, ax = plt.subplots(figsize=(15,5))
for j in range(cols):
    ax.scatter(arr[j][0], y=arr[0][0].index, label= "Itr" + str(j),  marker=markerlist[j])
ax.legend(loc='upper right', ncol=2)
ax.set_xlabel('Importance Value', fontsize=15)
ax.set_ylabel('Ad#', fontsize=15)
ax.set_title('Dynamic Changes in the Importance of ADs')
plt.show()
```



As we can see in the last iteration of the simulation Ad# 4 is regognized to be the most wanted Ad. Aslo the plot shows that UCB has been able to identify the importance of Ad# 4 very early during the first 1000 rounds of the simulation.

4 Conclusion

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward.

Multi-armed bandit problems are some of the simplest reinforcement learning (RL) problems to solve. Multi-Armed Bandit Algorithm Bandit algorithm balances exploration and exploitation. We have an agent which we allow to choose actions, and each action has a reward that is returned according to a given, underlying probability distribution. The game is played over many episodes and the goal of the agent is to maximize its reward. What we observed was that when using function approximation, UCB without any knowledge about the problem, actually systematically performs really quite well.

In [ ]: