MP #1 - Bandit Algorithms

Notations:

$\mathcal{A}$: Action set

t: Trials

n^k: How many times the k-th arm has been pulled

d: Contextual dimension

K: The number of arms

a_k: Pull the k-th arm

σ: Standard deviation(noise scale)

x_a: Feature vector for the k-th arm

Basic Settings:

Iterations: 400

Noise Scale: 0.2

User: 10

Arms: 25

Contextual Dimension: 25

Part 1: Multi-armed Bandit Problem

1.1 Implement Multi-armed Bandit Algorithms

1.1.1 Upper Confidence Bound

let $\hat{r_{a,t}} = 0, \forall a \in \mathcal{A}$

for $t = 1, \ldots, n$:

decide:

if $n_t^k = 0:$

$B_t(a_k) = +\infty$

else:

$B_t(a_k) = \sigma\sqrt{2\log t / n_t^k}$

$a_t = \arg\max_{a \in A}(\hat{r_{a,t}} + B_t(a))$

update:

$\hat{r_{a,t+1}} = (\hat{r_{a,t}} \times n_t^k + r_t)/(n_t^k + 1)$

$n_{t+1}^k = n_t^k + 1$

其中，由于已知噪声符合方差为σ^2的高斯分布，我们有理由认为置信区间的宽度与标准差成线性关系，就可以在$B_t(a)$的标准公式的基础上再乘σ这个常数倍数以更精确地估计UCB。

```
def updateParameters(self, articlePicked_id, reward):
    self.UserArmMean[articlePicked_id] = (self.UserArmMean[articlePicked_id]*self.UserArmTrials[articlePicked_id] + reward) / (self.UserArmTrials[articlePicked_id]+1)
    self.UserArmTrials[articlePicked_id] += 1

    self.time += 1
```

```
def decide(self, pool_articles):
    articlePicked = None
    maxPTA = float('-inf')

    for article in pool_articles:
        if self.UserArmTrials[article.id] == 0:
            return article
        article_pta = self.UserArmMean[article.id] + self.sigma * np.sqrt(2 * np.log(self.time) / self.UserArmTrials[article.id])
        if maxPTA < article_pta:
            articlePicked = article
            maxPTA = article_pta

    return articlePicked
```

1.1.2 Thompson Sampling

let $\hat{r_{a,1}} = 0, \hat{\sigma_{a,1}}^2 = 1, \forall a \in \mathcal{A}$

for $t = 1, \ldots, n$:

decide:

$$v_{a,t} \sim \mathcal{N}(\hat{r_{a,t}}, \hat{\sigma_{a,t}}^2)$$

$$a_t = \arg\max_{a \in A}(v_{a,t})$$

update:

$$\hat{r_{a,t+1}} = (\hat{r_{a,t}} \times n_t^k + r_t)/(n_t^k + 1)$$

$$n_{t+1}^k = n_t^k + 1$$

$$\hat{\sigma_{a,t+1}}^2 = (\sigma^2 \times \hat{\sigma_{a,t}}^2)/(\sigma^2 + \hat{\sigma_{a,t}}^2)$$

其中，由于已知噪声符合方差为σ^2的高斯分布，我们不需要使用更复杂的贝叶斯推断来更新先验均值和方差。而先验方差不直接使用σ^2而是逐步更新可以增加鲁棒性和表现。

而事实上，先验分布并不是一个高斯分布：用户的偏好向量θ取自$K = d$维的球面，reward的均值则是θ在文章特征向量上的投影，满足$\mathcal{B}eta(1/2, (k-1)/2)$，方差为$1/k$。但为了计算的便利性我们采用高斯分布作为先验分布，这种简化导致$\sigma^2 = 1/k$的效果不好，最终选择$\sigma^2 = 1$作为先验方差以鼓励探索。

```
def updateParameters(self, articlePicked_id, reward):
    self.UserArmMean[articlePicked_id] = (self.var * self.UserArmMean[articlePicked_id] + self.UserArmVar[articlePicked_id] * reward) / (self.var + self.UserArmVar[articlePicked_id])
    self.UserArmVar[articlePicked_id] = self.var * self.UserArmVar[articlePicked_id] / (self.var + self.UserArmVar[articlePicked_id])
```

```
def decide(self, pool_articles):
    maxPTA = float('-inf')
    articlePicked = None

    for article in pool_articles:
        # article_pta = np.random.normal(self.UserArmMean[article.id], self.sigma)
        article_pta = np.random.normal(self.UserArmMean[article.id], np.sqrt(self.UserArmVar[article.id]))
        if maxPTA < article_pta:
            articlePicked = article
            maxPTA = article_pta

    return articlePicked
```

1.1.3 Perturbed-history Exploration

let $\hat{r_{a,1}} = 0, \forall a \in \mathcal{A}$

for $t = 1, \ldots, n$:

decide:

if $n_t^k = 0$:

$$v_{a,t} = +\infty$$

else:

$$v_{a,t} = (\hat{r_{a,t}} \times n_t^k + \Sigma_{l=1}^{\alpha \times n_t^k} \mathcal{B}ernoulli(0.5))/(n_t^k \times (1+\alpha))$$

$$a_t = \arg\max_{a \in A}(v_{a,t})$$

update:

$$\hat{r_{a,t+1}} = (\hat{r_{a,t}} \times n_t^k + r_t)/(n_t^k + 1)$$

$$n_{t+1}^k = n_t^k + 1$$

该情景下由于reward较小，α过大会使得扰动项过大，探索过度，本次实验一般使用$\alpha = 1$。

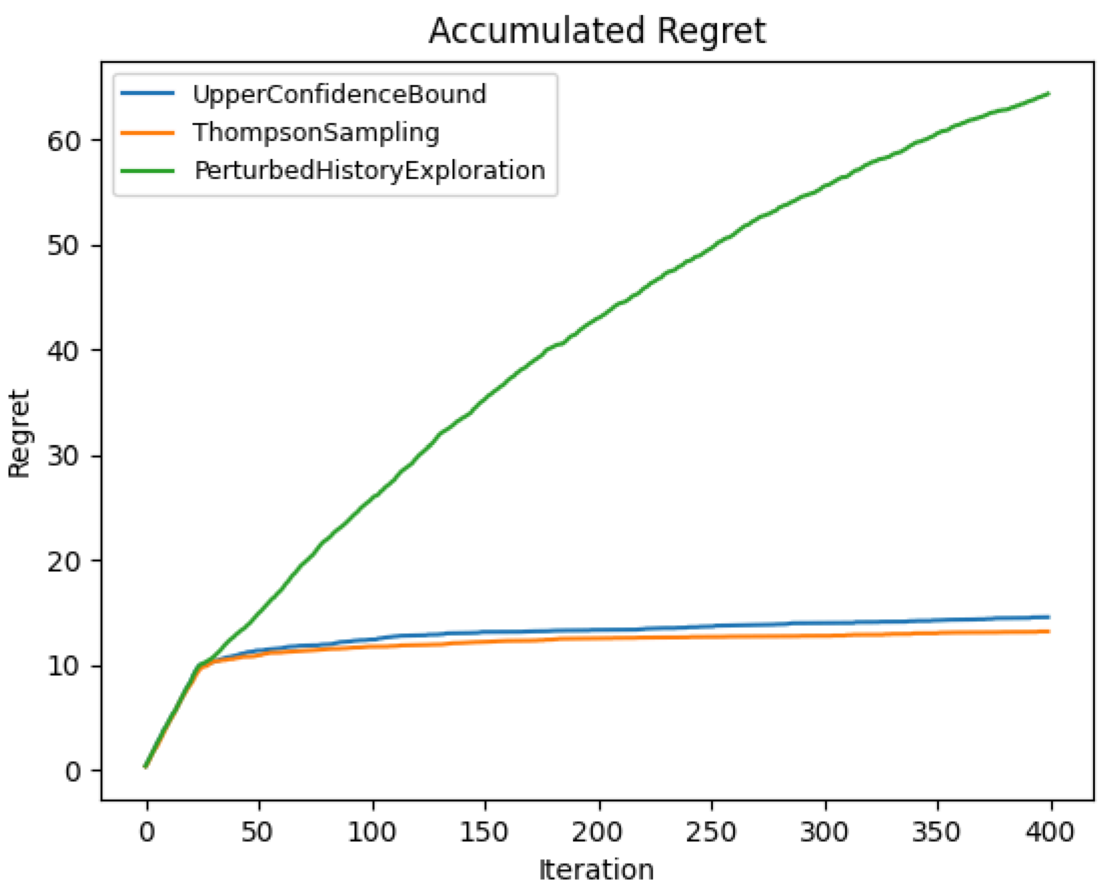
```
def updateParameters(self, articlePicked_id, reward):
    self.UserArmMean[articlePicked_id] = (self.UserArmMean[articlePicked_id]*self.UserArmTrials[articlePicked_id] + reward) / (self.UserArmTrials[articlePicked_id]+1)
    self.UserArmTrials[articlePicked_id] += 1
```

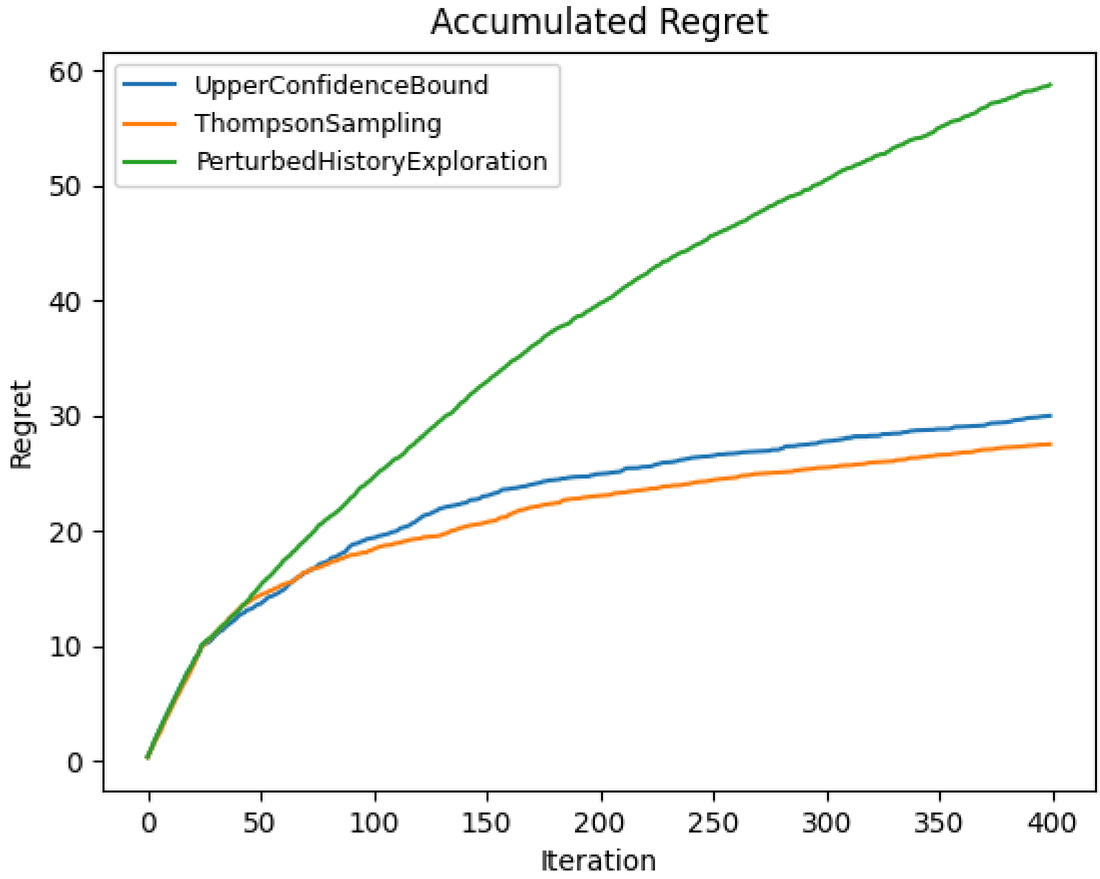
```
def decide(self, pool_articles):
    articlePicked = None
    maxPTA = float('-inf')

    for article in pool_articles:
        if self.UserArmTrials[article.id] == 0:
            return article

        perturb = np.sum(np.random.binomial(1, 0.5, int(self.alpha * self.UserArmTrials[article.id])))
        article_pta = (self.UserArmMean[article.id] * self.UserArmTrials[article.id] + float(perturb)) / ((1 + self.alpha) * self.UserArmTrials[article.id])
        if maxPTA < article_pta:
            articlePicked = article
            maxPTA = article_pta

    return articlePicked
```

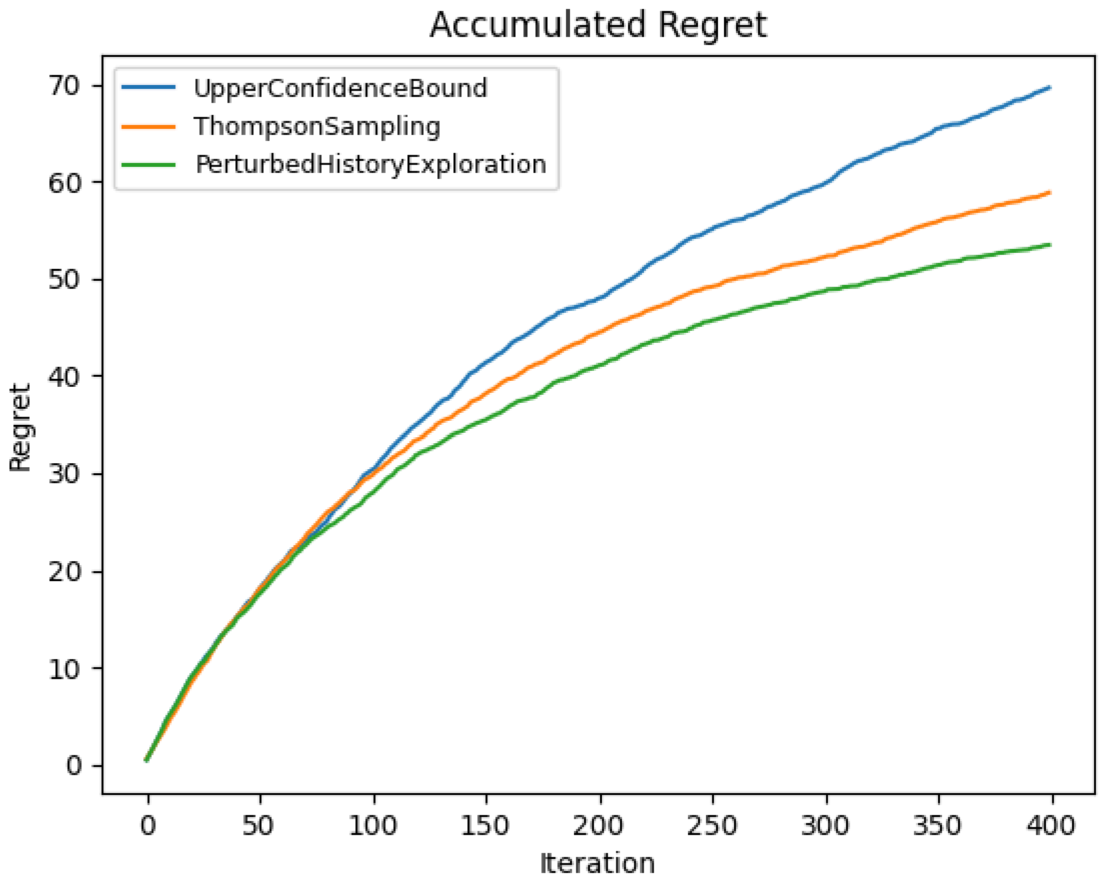
1.2 Comparison in Different Environment Settings

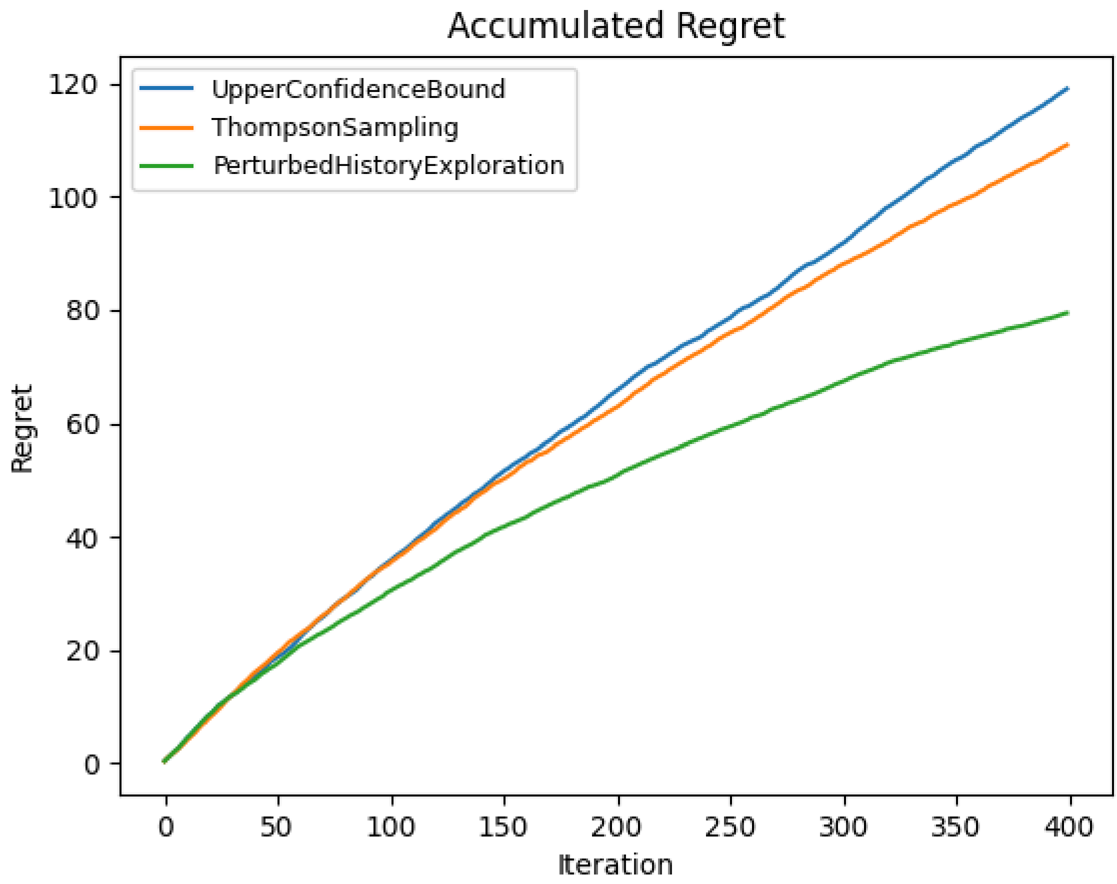
Noise Scale(NS)

基础设定下分别采用0.1 0.2 0.4 0.8进行实验。




Accumulated Regret
UpperConfidenceBound
ThompsonSampling
PerturbedHistoryExploration
Regret
Iteration
0
10
20
30
40
50
60
0
50
100
150
200
250
300
350
400


Accumulated Regret
UpperConfidenceBound
ThompsonSampling
PerturbedHistoryExploration
Regret
Iteration
0
10
20
30
40
50
60
70
0
50
100
150
200
250
300
350
400

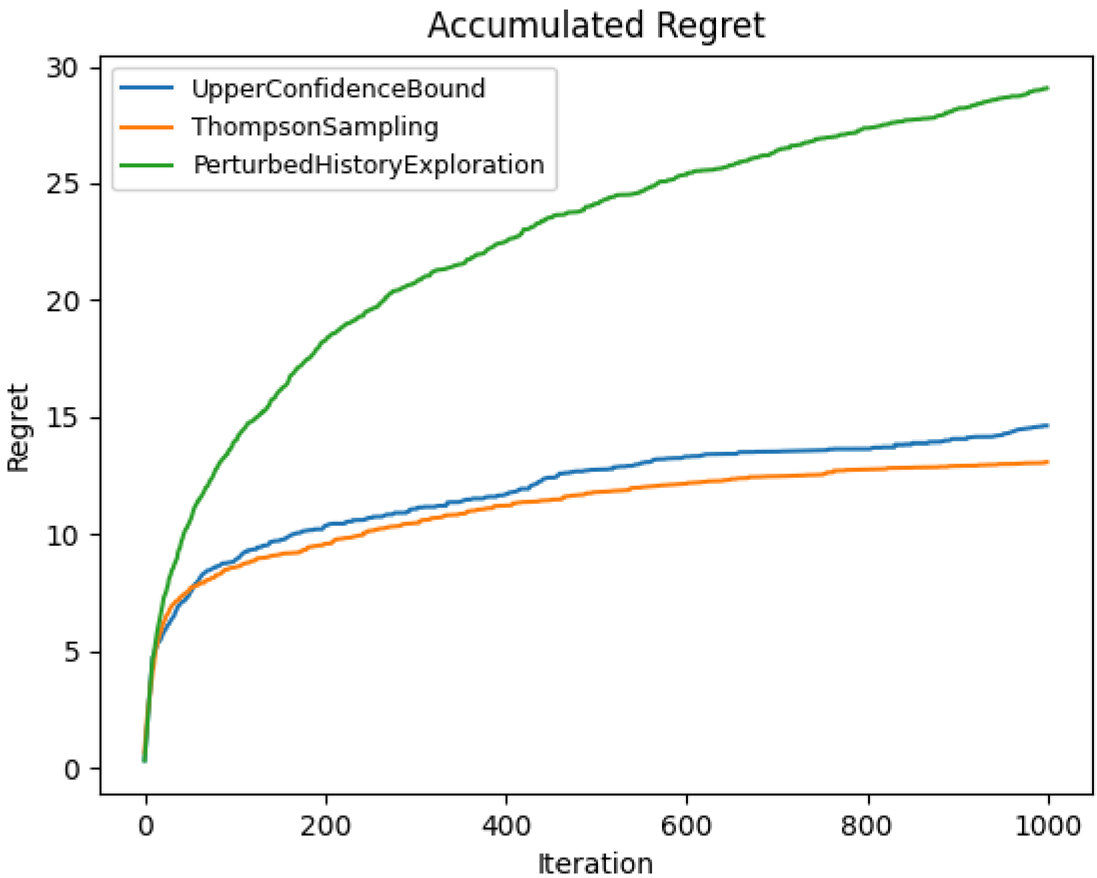


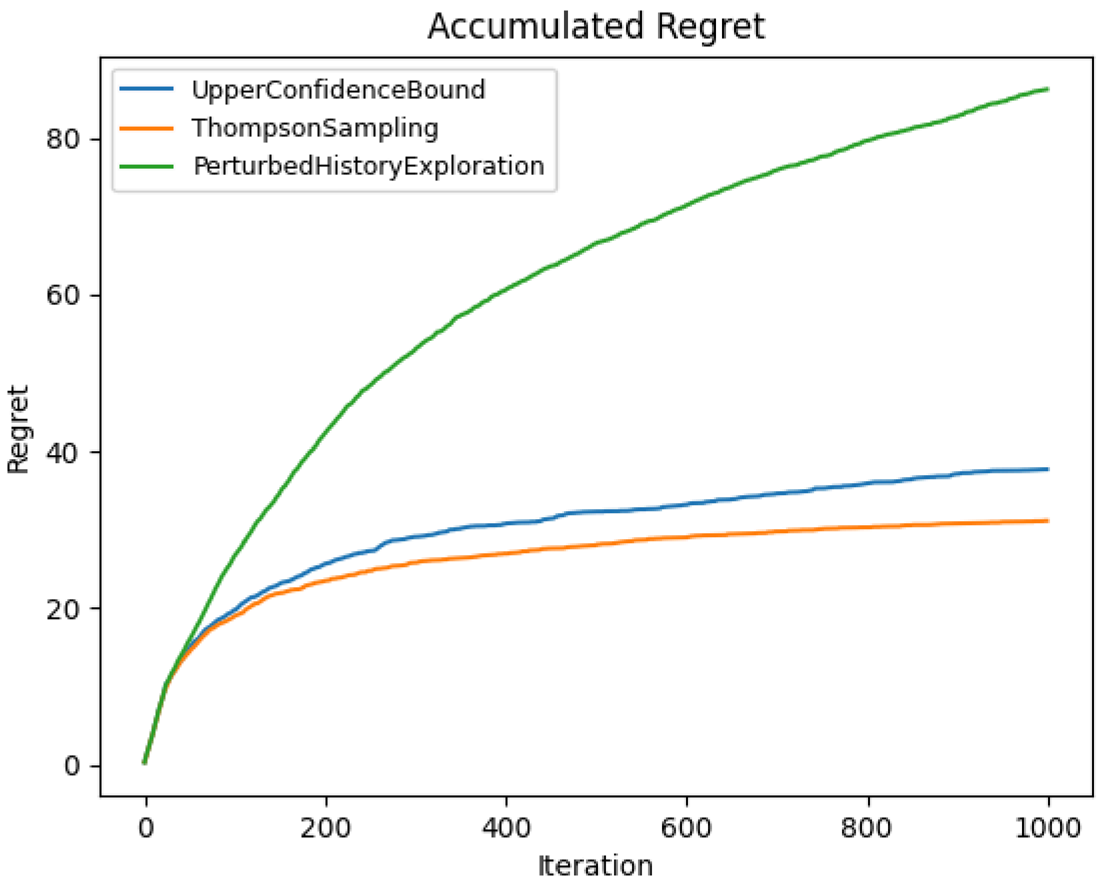
Analysis:

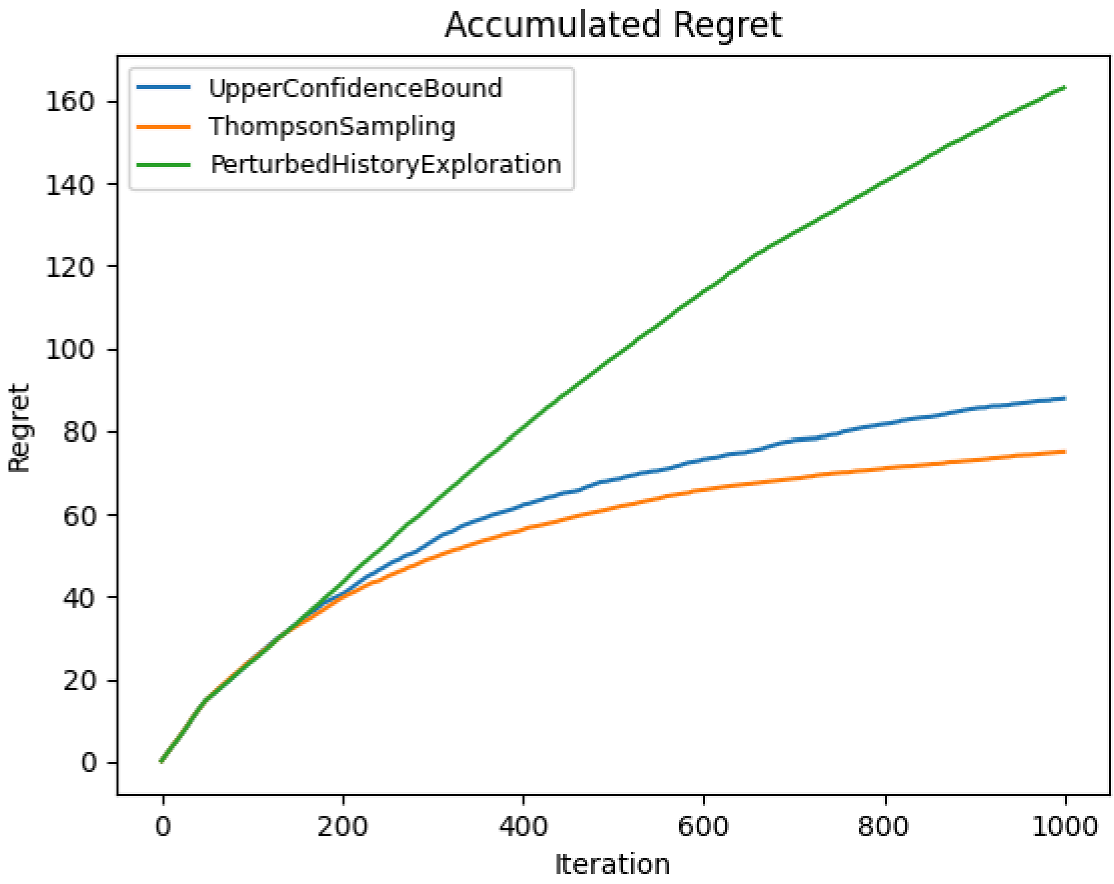
1. 所有算法都随NS增大而变差：方差变大，随机性变大，难以估计均值。
2. 低噪声下UCB和TS较好，PHE较差：均值更准、方差更小的情况下，UCB和TS的探索由于有方差的限制，更易选择最好的臂；PHE的扰动项显得过大，在不好的臂上浪费很多时间。
3. 高噪声下PHE > TS > UCB：方差较大时，PHE的扰动项从大减小，使得探索先多后少，稳定合理；TS的采样范围较大，利于探索但不太好利用；UCB的估计过于乐观，自身的随机性不足，再加上观测的不确定性，探索效果差。

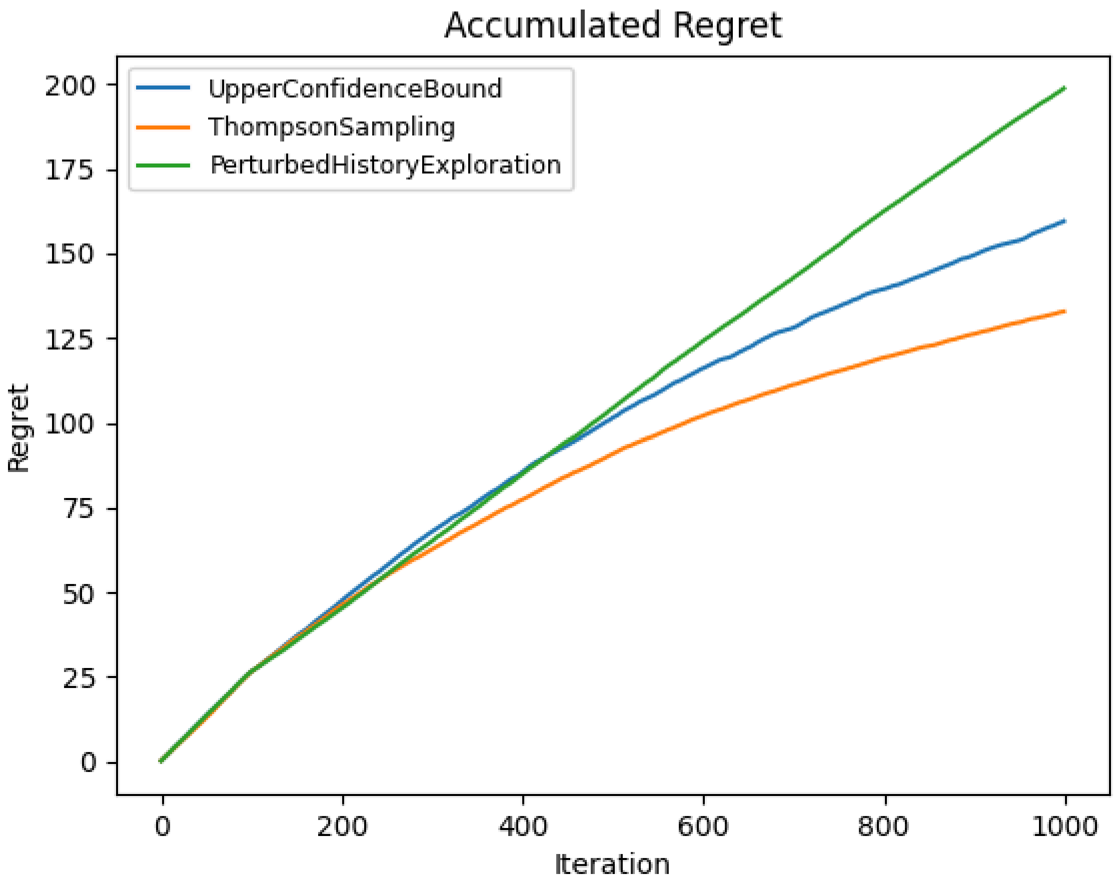
K

除Iteration = 1000外基础设定下分别采用10 25 50 100进行实验


Accumulated Regret
UpperConfidenceBound
ThompsonSampling
PerturbedHistoryExploration
Regret
Iteration
0
5
10
15
20
25
30
0
200
400
600
800
1000


Accumulated Regret
UpperConfidenceBound
ThompsonSampling
PerturbedHistoryExploration
Regret
Iteration
0
20
40
60
80
0
200
400
600
800
1000





Analysis:

1. 所有算法都随K增大而变差：臂的数量增多，需要耗费更多的时间进行探索。
2. TS > UCB >PHE：TS的探索与利用最为稳定；臂数量较多时，UCB对每个臂的探索次数变少，导致上置信界一直较大，探索稍多但对均值的估计稍差；PHE的扰动项使它更难找到表现好的臂。
3. PHE受影响较小：更高维度对探索的需求更大，PHE是最激进的，相对估计能力变强。

Part 2: Contextual Linear Bandit Problem

2.1 Implement Contextual Linear Bandit Algorithms

LinUCB

let $A_1 = \lambda I, b_1 = \mathbf{0}, \hat{\theta}_1 = \mathbf{0}$

for $t = 1, \ldots, n$:

decide:

$$\alpha_t = \alpha\sqrt{d\log\left(t/\lambda\right)}$$

$$B_t(a_k) = \sigma \times \alpha_t \times x_{a_k}^{\top} A_t^{-1} x_{a_k}$$

$$a_t = \arg\max_{a \in A}(x_{a_k}^{\top}\hat{\theta}_t + B_t(a))$$

update:

$$A_{t+1} = A_t + x_{a_t} x_{a_t}^{\top}$$

$$b_{t+1} = b_t + r_t x_{a_t}$$

$$\hat{\theta_{t+1}} = A_{t+1}^{-1} b_{t+1}$$

正则化参数$\lambda > 0$即可保证矩阵可逆、训练稳定，经调试选$\lambda = 0.1$；$\alpha > 0$是为$\alpha_t = O(\sqrt{dlog(t/\lambda)})$增加的参数，经调试选$\alpha = 0.1$。和普通UCB一样，我们在$B_t(a)$的标准公式的基础上再乘$\sigma$以更精确地估计UCB。

```
def updateParameters(self, articlePicked_FeatureVector, reward):
    feature_vector = torch.tensor(articlePicked_FeatureVector, device=self.device).float()
    self.A += torch.outer(feature_vector, feature_vector)
    self.b += feature_vector * reward
    # self.AInv = torch.inverse(self.A)
    AInvFeature = torch.matmul(self.AInv, feature_vector)
    self.AInv -= torch.outer(AInvFeature, AInvFeature) / (1 + torch.dot(feature_vector, AInvFeature))
    self.UserTheta = torch.matmul(self.AInv, self.b)
    self.time += 1
```

```
def decide(self, pool_articles):
    articlePicked = None
    maxPTA = float('-inf')

    for article in pool_articles:
        alpha = self.dim * np.log(self.time / self.lambda_)
        feature_vector = torch.tensor(article.featureVector, device=self.device).float()
        article_pta = torch.dot(self.UserTheta, feature_vector) + self.sigma * self.a * torch.sqrt(alpha * torch.dot(torch.matmul(self.AInv, feature_vector), feature_vector))
        if maxPTA < article_pta:
            articlePicked = article
            maxPTA = article_pta

    return articlePicked
```

LinTS

let $A_1 = \lambda I, b_1 = \mathbf{0}, \hat{\theta}_1 = \mathbf{0}$

for $t = 1, \ldots, n$:

decide:

$$v_{a_t} \sim \mathcal{N}(x_{a_t}^{\top}\hat{\theta}_t, x_{a_k}^{\top} A_t^{-1} x_{a_k})$$

$$a_t = \arg\max_{a \in A}(v_{a_t})$$

update:

$$A_{t+1} = A_t + x_{a_t} x_{a_t}^{\top}$$

$$b_{t+1} = b_t + r_t x_{a_t}$$

$$\hat{\theta_{t+1}} = A_{t+1}^{-1} b_{t+1}$$

与普通TS算法相同，这里的先验分布也用高斯分布近似，但这里取$\lambda = k$（真实先验分布方差）效果很好，不用更改。

```python
def updateParameters(self, articlePicked_FeatureVector, reward):
    feature_vector = torch.tensor(articlePicked_FeatureVector, device=self.device).float() / self.sigma
    self.A += torch.outer(feature_vector, feature_vector)
    self.b += (feature_vector * reward) / self.sigma
    # self.AInv = torch.inverse(self.A)
    AInvFeature = torch.matmul(self.AInv, feature_vector)
    self.AInv -= torch.outer(AInvFeature, AInvFeature) / (1 + torch.dot(feature_vector, AInvFeature))
    self.UserTheta = torch.matmul(self.AInv, self.b)
```

```python
def decide(self, pool_articles):
    articlePicked = None
    maxPTA = float('-inf')

    for article in pool_articles:
        feature_vector = torch.tensor(article.featureVector, device=self.device).float()
        article_pta = torch.normal(torch.dot(self.UserTheta, feature_vector), torch.sqrt(torch.dot(torch.matmul(self.AInv, feature_vector), feature_vector)))
        if maxPTA < article_pta:
            articlePicked = article
            maxPTA = article_pta

    return articlePicked
```

LinPHE

let $A_1 = \lambda I, b_1 = \mathbf{0}, \hat{\theta}_1 = \mathbf{0}$

for $t = 1, \ldots, n$:

decide:

if $t <= d$:

$$a_t = arm(t)$$

else:

$$b_t' = \Sigma_{(a_i, r_i) \in \mathcal{H}_t} (r_i + \Sigma_{l=1}^{\alpha} \mathcal{B}ernoulli(0.5)) x_{a_i}$$

$$\hat{\theta}_t' = A_t^{-1} b_t' / (1 + \alpha)$$

$$a_t = \arg\max_{a \in A} (x_{a_k}^\top \hat{\theta}_t')$$

update:

$$A_{t+1} = A_t + x_{a_t} x_{a_t}^\top$$

$$b_{t+1} = b_t + r_t x_{a_t}$$

$$\hat{\theta_{t+1}} = A_{t+1}^{-1} b_{t+1}$$

正则化参数$\lambda > 0$即可保证矩阵可逆、训练稳定，经调试选$\lambda = 0.1$；查阅原论文后得知，$\forall \alpha > 1$可以满足sub-linear regret，调试后选择2。

其中，$t < d$时直接选择，是为了在扰动之前先对偏好向量有大概的估计（不妨认为d个arm几乎可以组成一组基），避免扰动项干扰最初的探索。实际实现中选择第一个未被选择过的arm。

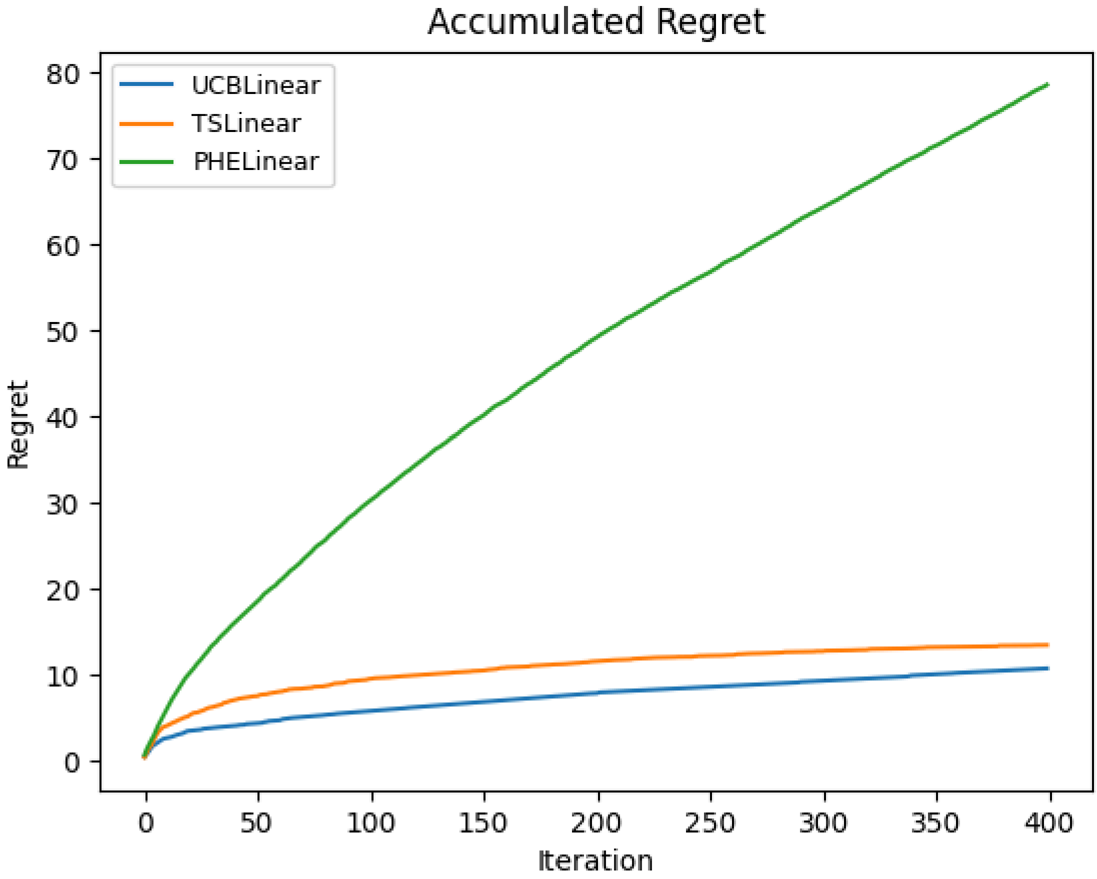
```python
def updateParameters(self, articlePicked_FeatureVector, reward):
    feature_vector = torch.tensor(articlePicked_FeatureVector, device=self.device).float()
    self.A += torch.outer(feature_vector, feature_vector)
    self.history.append((feature_vector, reward))
    b = torch.zeros(self.dim, device=self.device)
    for x, r in self.history:
        b += (r + torch.bernoulli(0.5 * torch.ones(self.alpha, device=self.device)).sum()) * x
    # self.AInv = torch.inverse(self.A * (1 + self.alpha))
    AInvFeature = torch.matmul(self.AInv, feature_vector)
    self.AInv -= torch.outer(AInvFeature, AInvFeature) / (1 + torch.dot(feature_vector, AInvFeature))
    self.UserTheta = torch.matmul(self.AInv, b) / (1 + self.alpha)
```

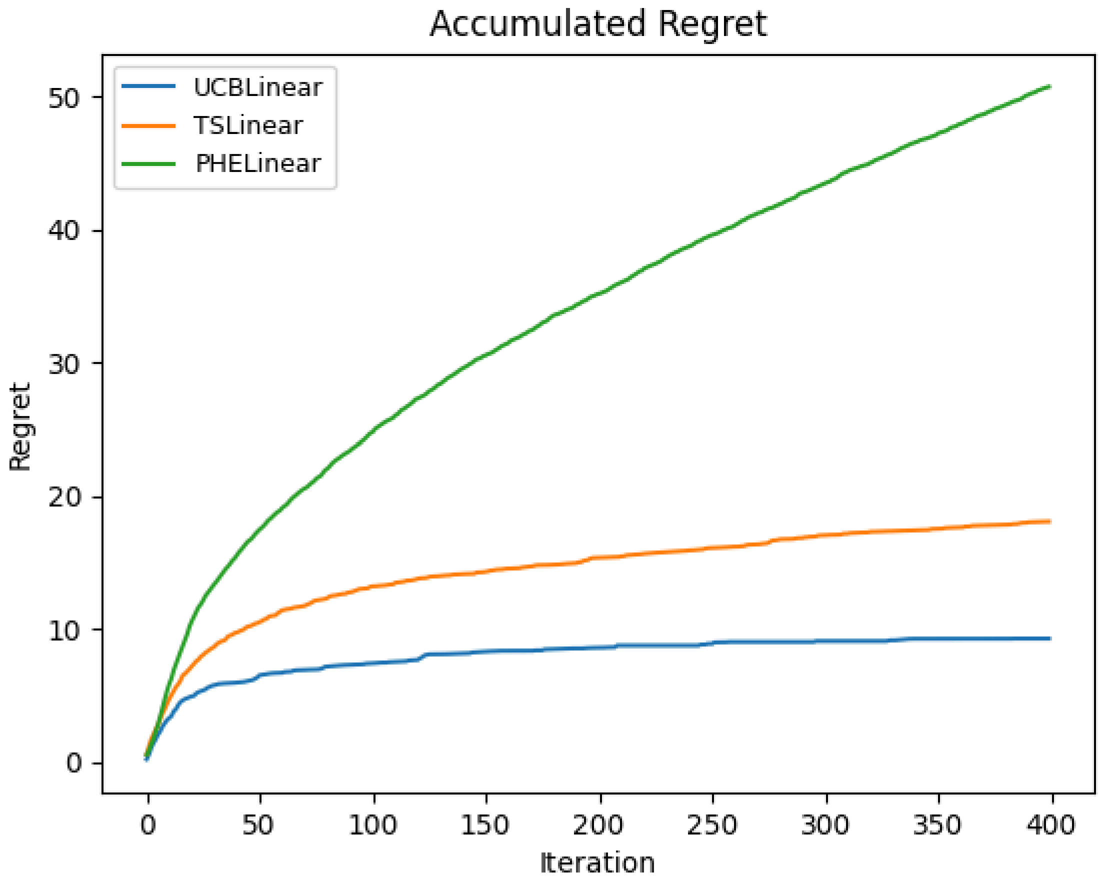
```python
def decide(self, pool_articles):
    articlePicked = None
    maxPTA = float('-inf')

    if self.time < self.dim and self.time < self.num and self.time < len(pool_articles):
        for article in pool_articles:
            if self.UserArmTrials[article.id] == 0:
                articlePicked = article
                break
    else:
        for article in pool_articles:
            feature_vector = torch.tensor(article.featureVector, device=self.device).float()
            article_pta = torch.dot(self.UserTheta, feature_vector)
            if maxPTA < article_pta:
                articlePicked = article
                maxPTA = article_pta

    self.UserArmTrials[articlePicked.id] += 1
    return articlePicked
```

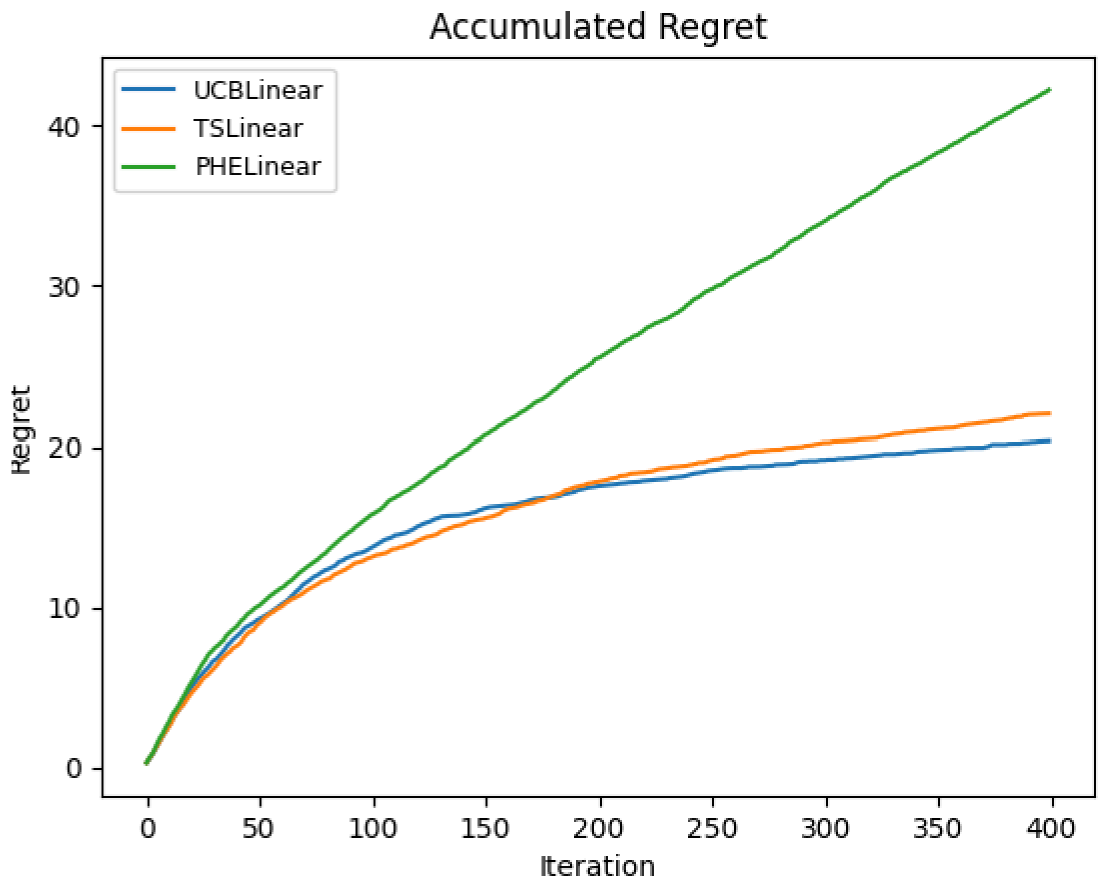
2.2 Comparison in Different Environment Settings

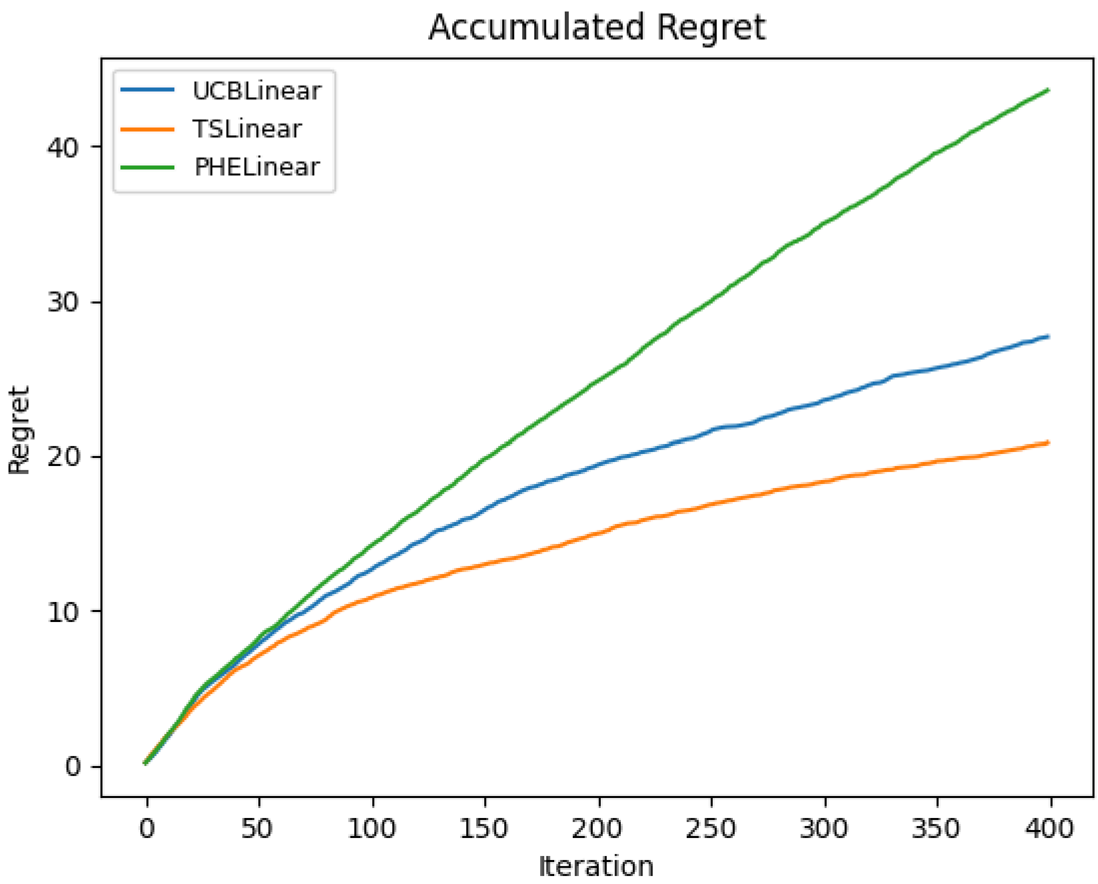
Contextual Dimension(CD)

基础设定下分别取10 20 50 100进行实验


Accumulated Regret
UCBLinear
TSLinear
PHELinear
Regret
Iteration
0
10
20
30
40
50
60
70
80
0
50
100
150
200
250
300
350
400


Accumulated Regret
UCBLinear
TSLinear
PHELinear
Regret
Iteration
0
10
20
30
40
50
0
50
100
150
200
250
300
350
400





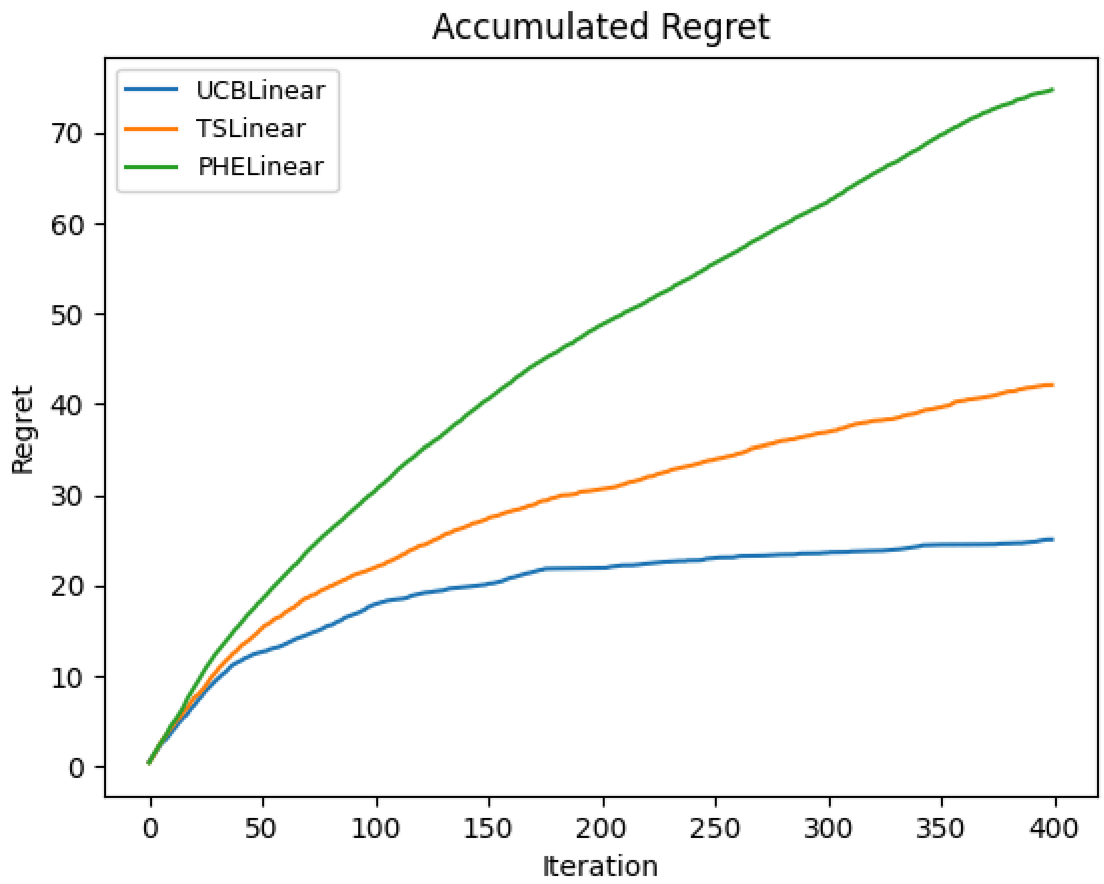
Analysis:

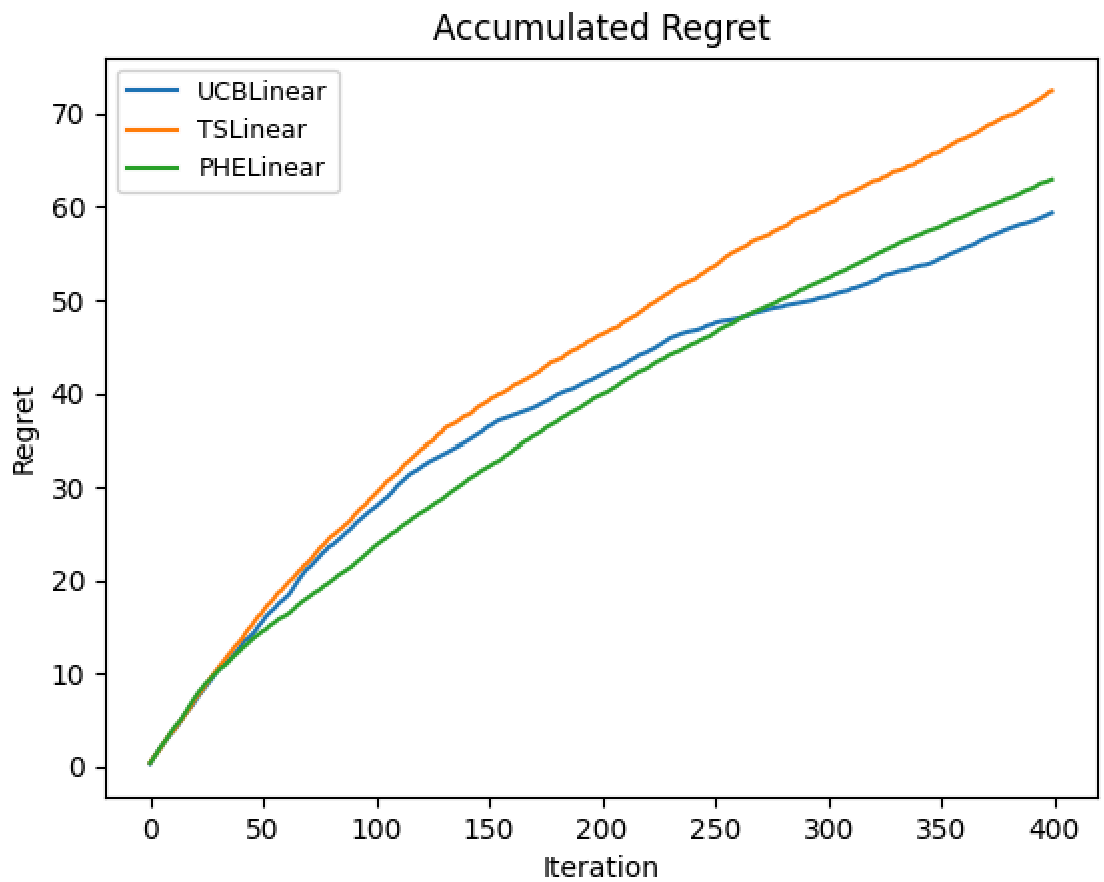
1. 大体随CD增大而变差：特征数变多，需要更多探索来学习。
2. PHE的反常：维度很小时，特征空间的信息量不足，PHE过于激进的探索并不能学到更多东西，难以找到最优解；维度较大时，PHE的探索会带来更多特征的信息，从而获得更好的表现，并缩短与其他二者的差距。

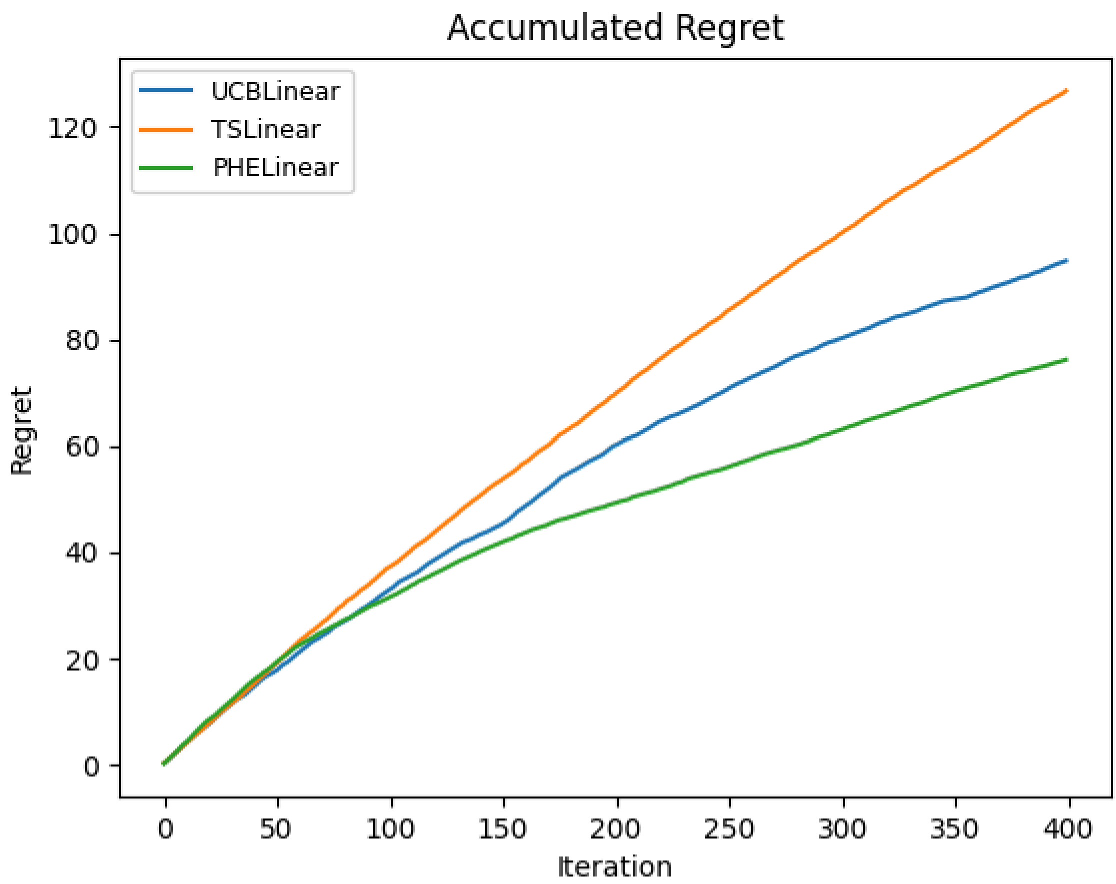
3. UCB与TS的比较：维度较低时UCB更好，是因为低维信息更简单，UCB的确定性策略能更快收敛；维度高时TS更好，因为在更复杂、不确定性更高的环境下随机采样能更好地学到特征信息。
4. 偏好向量估计PHE > TS > UCB：与探索强度一致。

Noise Scale

基础设定下采用0.4 0.8 1.5进行实验





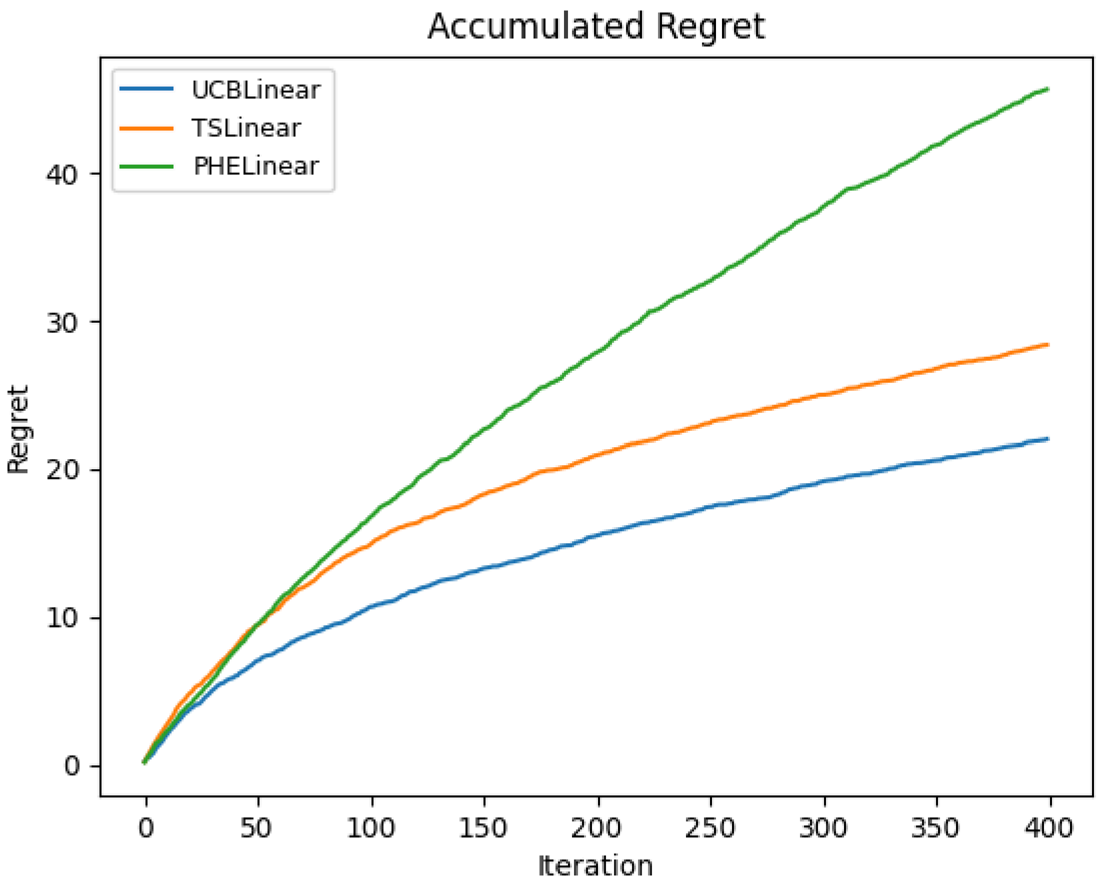


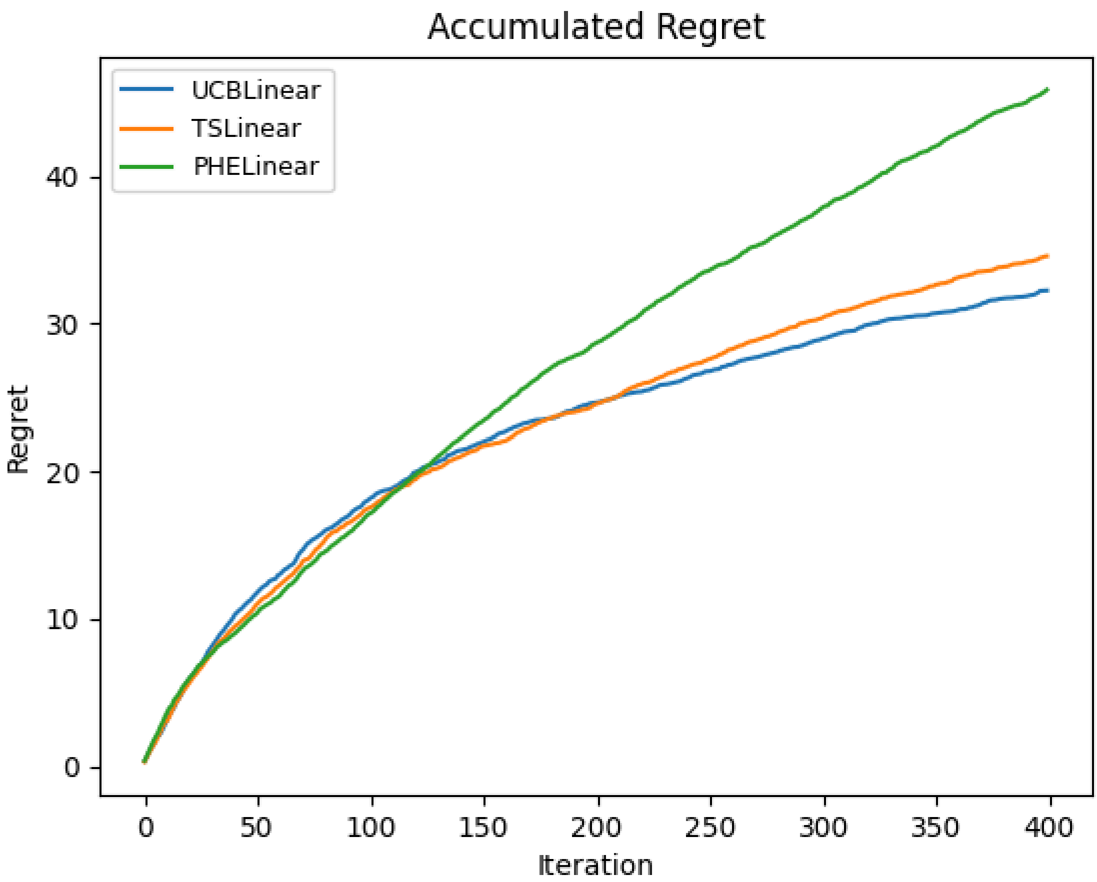
Analysis:

1. 基础分析与1.2相同。
2. 偏好向量估计TS > PHE > UCB：维度较低时TS基于贝叶斯更新的噪声处理的优越性超过PHE更激进的探索带来的优势，是反超的主要原因。

Action Set

除Noise Scale = 0.4外基础设定下分别采用5 15进行实验



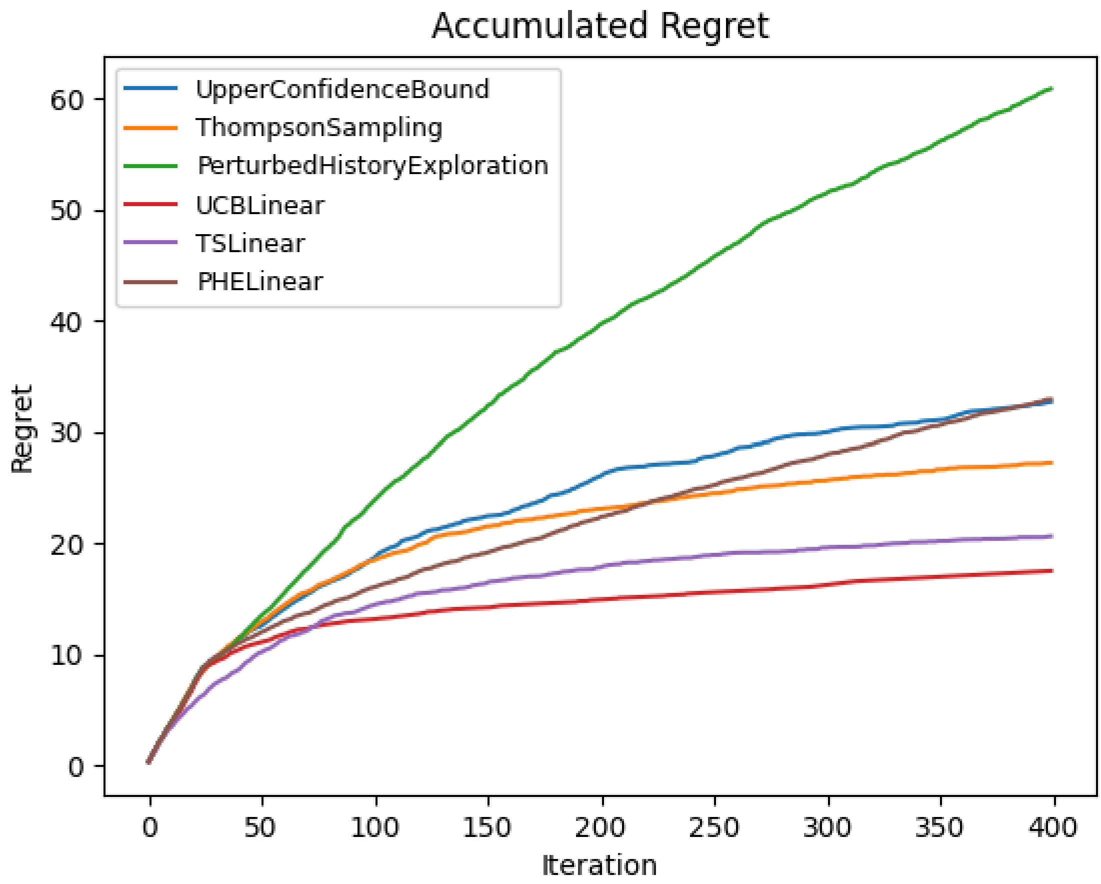


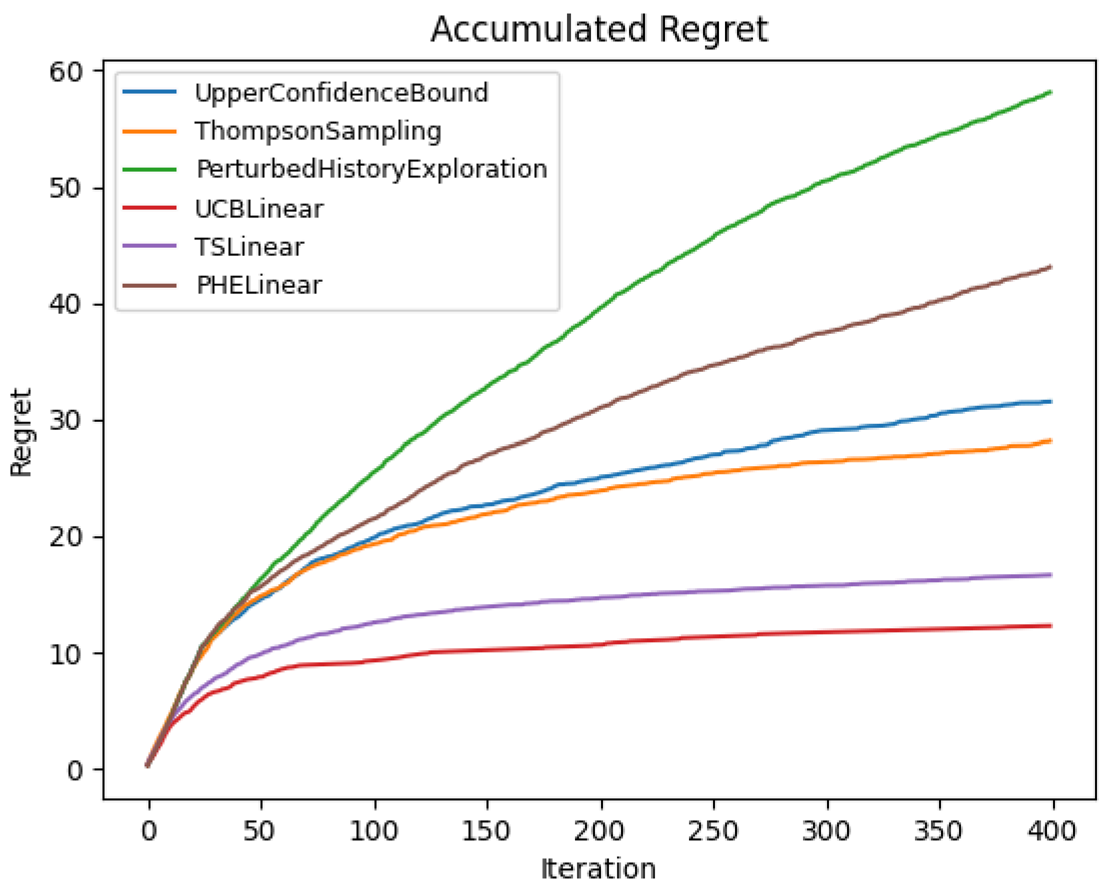
Analysis:

大体随size增加而变差：提供的臂数量越少，越容易找到最优解。偏好向量估计并没有变化，因为参数更新的过程并没有变化，表现完全取决于选项数量。

2.3 Influence of the Shape of Action Set

基础设定分别采用basis_vector和random进行实验





Analysis:

1. basis_vector下表现比random好：特征向量更规整，复杂度小。

2. random下linear算法明显更好：linear算法可以学习每个维度的特征信息。
3. basis_vector下表现差不多：特征向量正交，每个维度的信息消失，linear算法退化。