Practical Secondary Stack Optimization on Search Pages: A Lightweight Contextual Bandit Approach*

Eva C. Song[1,*], Jun Zhao[1], Junchao Zheng[1] and Vivek Agrawal[2]

[1]Walmart Global Tech, Hoboken NJ, USA
[2]Walmart Global Tech, Sunnyvale CA, USA

Abstract

Secondary Stacks – recommender modules displayed alongside core search results, are a key driver of engagement and conversion on E-commerce search pages. A secondary stack suitable for display should generally demonstrate relevance to the query and good potential for engagement. An E-commerce website can have many of such secondary stack modules that feature different themes and are powered by different algorithms for the content generation. Due to space limitations, not all available stacks can be displayed, making it necessary to prioritize those most likely to perform well under the current search context, such as the query. Additionally, retrieval for many stacks is computationally non-trivial. Hence, it is often beneficial to conduct a pre-retrieval selection, when only partial context is known, to limit the number of stacks to retrieve, and then prioritize based on the full context only among the retrieved stacks. Lastly, stack performance is dynamic, varying with evolving content and user behavior. We introduce a lightweight, highly flexible solution that frames secondary stack prioritization as a contextual multi-armed bandit problem. Our approach features a low-cost offline data pipeline and a real-time selection algorithm based on Thompson sampling, enabling seamless deployment at any stage of the search flow, such as pre-retrieval, post-ranking, or anywhere contextually appropriate. The system dynamically adapts to shifting contexts without requiring complex retraining, and thrives where large models often overfit, such as with low-visibility stacks that receive sparse impressions. AB tests on Walmart.com show that our method delivers significant gains in user engagement metrics, validating a solution that is fast to implement, cost-efficient to serve, and robust in production. The algorithm has been successfully launched and continues to drive improved search experiences at scale.

Keywords

Multi-armed bandit, Search Page Optimization, Secondary Stack, Thompson sampling

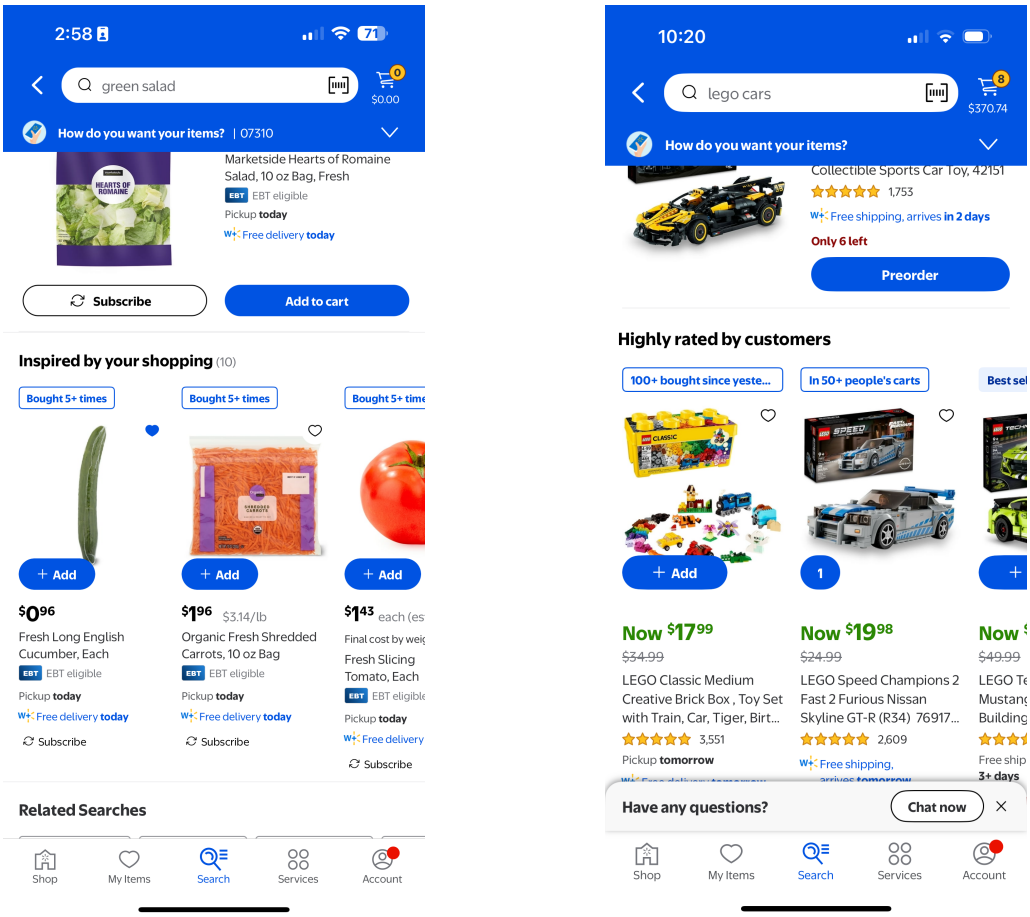
1. Introduction

In E-commerce search experiences, Secondary Stacks – recommender modules displayed alongside or below primary search results play a vital role in driving user engagement, discovery, and conversion. With limited real estate on the search page and a growing number of available stacks powered by diverse algorithms, efficiently prioritizing which modules to display has become increasingly critical.

On the Walmart.com website, a secondary stack can be shown at the bottom of the first page search result. At the time of writing, there are ~ 10 types of secondary stacks live in production and several more under development. Each of these secondary stacks features a different aspect of business consideration and the content within each secondary stack is generated with a separate algorithm. For example, some secondary stacks are personalized based on users' past shopping history and the other ones are generic to all users. However, they all share a very similar front-end treatment and the same space on the search page. As the number of different types of secondary stack increases, several stacks may be qualified to be shown on the same search page and consequently, a choice must be made. It is worth noting that retrieval for many stacks are non-trivial, with some as costly as that of the primary search retrieval. Therefore, the ideal framework should be invokable at both pre-retrieval stage, where it needs to operate on partial context, and post-retrieval stage, to conduct the final selection based on the full context. Prior to this work, the selection of the secondary stacks was based on a

ECOM'25: SIGIR Workshop on eCommerce, Jul 17, 2025, Padua, Italy
*Corresponding author.
eva.song@walmart.com (E. C. Song); jun.zhao3@walmart.com (J. Zhao); junchao.zheng@walmart.com (J. Zheng); vivek.agrawal@walmart.com (V. Agrawal)

set of business heuristic rules. Under these rules, the prioritization of the stacks is predetermined. This rule-based prioritization is suboptimal as each secondary stack can demonstrate different user engagement patterns under different contexts. One of such contexts is the query itself. An example illustrating different stacks should be prioritized under different contexts is shown in Figure 1. In addition to the lack of contextual awareness, the engagement pattern of each individual stack can also change over time. In order to address this issue, we treat it as a *contextual multi-armed bandit problem*, where an explore-exploit algorithm learns the dynamics from all the secondary stacks under various contexts by continuously collecting users' engagement data. When multiple secondary stacks are qualified to be displayed on the same search page, the explore-exploit algorithm decides whether to show the most promising stack based on past users' engagement data (exploit) or show another stack so that we continue taking measurements on the other stacks in order to capture any change in stacks' behaviors over time (explore). An efficient explore-exploit algorithm allows us to achieve the best engagement from all secondary stacks on average in the long run.



(a) A stack featuring personalized cross-selling options

(b) A stack featuring highly rated items

Figure 1: Example that shows different stacks are more useful under different contexts, such as the query

The multi-armed bandit is a framework that handles the tradeoff between explore and exploit. In this setting, at each time instance, an arm is selected and a reward (regret) from choosing the arm is observed. The selected arm and its corresponding reward then contribute as historical observations and are used to make the next decision. This area has been intensively studied in the literature since 1933 [1]. Many algorithms are well established in the literature, including ϵ-greedy, Upper Confidence Bound (UCB) [2, 3, 4] and Thompson sampling [5, 6, 7]. The corresponding theoretical bounds on expected regret in time have also been studied [2, 8, 9, 5, 10]. The contextual multi-armed bandit further assumes that the selection of the best arm and the corresponding reward depend on another random variable,

the context. This setting has been applied to many areas including in recommender systems [11]. In recent years, deep contextual bandits [12] are also intensively studied and applied to similar problems and demonstrate great results.

Traditional complex modeling approaches often require high-quality, large-scale data and heavy infrastructure, making them impractical for the dynamic and noisy environment where many stacks can suffer from low visibility and sparse impressions. In this work, we present a lightweight, flexible, and production-ready solution based on contextual multi-armed bandits that efficiently selects high-performing secondary stacks with minimal cost.

The main contributions of this paper include:

- Framing the problem of search page secondary stack selection as a contextual multi-armed bandit and designing the offline data pipeline to maintain contextual statistics and online selection using Thompson sampling.
- Designing the context tree data structure with
  - operational flexibility that allows the sampling to be conducted with full or partial context;
  - robustness that allows tracing back to higher levels of the tree when the confidence in leaf-node statistics is low due to low impressions.

2. Formulation and Related Work

2.1. Contextual Multi-Armed Bandit Setting

We identify the possible arms as different types of search page secondary stack (recommender)

$$\mathcal{S} = \{\text{recommender 1}, \ldots, \text{recommender K}\}.$$

We emphasize that we focus on the selection of the secondary stack, rather than the content generation within each individual secondary stack. The content within each secondary stack is assumed to have been generated. All these secondary stacks share many common characteristics, such as similar front-end User Interface (UI) treatment and position on the search page where they can be displayed when selected. We focus on making the optimal selection of a stack for a fixed position on the search page.

We define the user engagement (number of item clicks) from a secondary stack as the reward R. From historical observations, we can form a data set as

$$D^t = \{(z_i, s_i, r_i)\}_{i=1}^t \tag{1}$$

where the context z_i itself can be a vector, the corresponding selected stack (arm) s_i, and observed reward r_i are recorded. We then obtain an induced distribution

$$P_{S_t,R_t|Z_t,D^{t-1}} = P_{S_t|Z_t,D^{t-1}} P_{R_t|Z_t,S_t,D^{t-1}}, \;\; t = 1, 2, \ldots$$

Note that the arm selection scheme $P_{S_t|Z_t,D^{t-1}}$ and the corresponding reward $P_{R_t|Z_t,S_t,D^{t-1}}$ can be stochastic. The goal is to obtain an optimal arm selection scheme $P_{S_t|Z_t,D^{t-1}}$ that maximizes the expected reward over time, i.e.

$$\max_{\left\{P_{S_t|Z_t,D^{t-1}}\right\}_t} \sum_{t=1}^{T} \mathbb{E}\left[R_t|\, Z_t, D^{t-1}\right]. \tag{2}$$

As for the reward, we use the user engagement from the secondary stack by assuming the number of engaged items X from a search page secondary stack follows a binomial distribution

$$X \sim \mathrm{B}(n, \Theta), \tag{3}$$

where n is the maximum number of engaged items from the secondary stack, and Θ is the success rate of each item from the secondary stack. We let the parameter Θ itself be a random variable

$$\Theta \sim \text{Beta}(\alpha, \beta), \tag{4}$$

where α and β are hyperparameters. Since Θ is a conjugate prior for X, after t observations, we have a computationally efficient way of obtaining the posterior distribution

$$P_{\Theta|X^t}(\theta|x^t) \sim \text{Beta}\left(\alpha + \sum_{i=1}^{t} x_i, \beta + \sum_{i=1}^{t}(n - x_i)\right). \tag{5}$$

Since we choose the reward as the number of engaged items from the selected secondary stack, which in turn is modeled according to the binomial distribution given in (3) and n is a fixed parameter to be determined later, the problem simplifies as follows.

$$\begin{aligned} R_t|(Z_t = z, S_t = s, D^{t-1} = d) &\sim \text{B}\left(n, \Theta_{t-1}(z, s)\right) \\ \Theta_t(z, s) &\sim \text{Beta}\left(\alpha_0 + \sum_{i=1}^{t} x_i(z, s), \beta_0 + \sum_{i=1}^{t}(n - x_i(z, s))\right), \end{aligned} \tag{6}$$

where α_0 and β_0 are universal initial values to be determined experimentally, and $x_i(z, s)$ denotes the number of clicked items from the secondary stack s under context z. The corresponding optimization given in (2) becomes

$$\max_{\left\{P_{S_t|Z_t,\Theta_{t-1}}\right\}_t} \sum_{t=1}^{T} \mathbb{E}\left[R_t|\, Z_t, \Theta_{t-1}\right] \tag{7}$$

where the expectation is w.r.t. the joint distribution

$$P_{Z_t,S_t,R_t,\Theta_{t-1}} = P_{Z_t,\Theta_{t-1}} P_{S_t|Z_t,\Theta_{t-1}} P_{R_t|Z_t,S_t,\Theta_{t-1}}.$$

2.2. Related Work and Our Approach

In our setting (7), we formulate the problem as contextual multi-armed bandit in order to capture each secondary stack's performance under different search contexts, such as queries. In addition, we adopt a Bayesian approach to exploit the user engagement in order to update the posterior distributions of the parameters in (6).

Ref. [11] proposes a contextualized approach LinUCB based on the traditional UCB where the expected reward is assumed to take a linear form of the context vector with some unknown parameters to be estimated, i.e. $\mathbb{E}\left[R_t|Z_t, \Theta\right] = Z_t^\top \Theta$. This result also has a natural Bayesian interpretation when Θ follows a Gaussian distribution. Similarly, the linear expected reward treatment is also studied using Thompson sampling [13]. However, both approaches intrinsincally assume a continuous unbounded form of reward. These formulations are not directly applicable to our problem since our reward is discrete.

Without the intricacies of the context, the problem can be addressed based on the Bernoulli Bandit setting. Thompson sampling is particularly efficient in conducting the explore-exploit of Bernoulli Bandit. However, Bernoulli Bandit is not as flexible in handling the inclusion of contexts. One way of handling the contextualization is by quantization, such as Lipschitz contextual bandits [14, 15, 16, 17]. The general idea is that the distributions of the parameters are maintained separately for a finite set of context realizations after the quantization of the context space. When the cardinality of the context is small, it is possible to skip the quantization. This is the approach we adopt by making sure that only the most prominent contexts are kept through a context preselection.

To solve for (7), we use Thompson sampling [18]. The simplified version of the protocol is summarized as follows.

Algorithm 1 Thompson sampling for stack selection

Require: n

$\alpha_{z,s}, \beta_{z,s} \leftarrow 1$ for all $z \in \mathcal{Z}, s \in \mathcal{S}$
for $t = 1, 2, \ldots$ **do**
    observe context z_t
    for $s \in \mathcal{S}$ **do**
        sample $\theta_t(z, s) \sim \text{Beta}(\alpha_{z,s}, \beta_{z,s})$
    end for
    pick a secondary stack $s_t^* = \arg\max_{s \in \mathcal{S}} \theta_t(z, s)$
    observe reward r_t^* from selecting s_t^*
    write (z_t, s_t^*, r_t^*) to log
    $\alpha_{z,s_t^*} \leftarrow \alpha_{z,s_t^*} + x_t(z_t, s_t^*)$
    $\beta_{z,s_t^*} \leftarrow \beta_{z,s_t^*} + (n - x_t(z_t, s_t^*))$
end for

Algorithm 1 only serves as a high-level overview of the flow. The actual realization of this algorithm does a batch update with delay and is discussed in the following sections. In the rest of the paper, we first discuss the methodology of identifying the context for our business problem and the data structure we use to maintain the statistics under such contexts. We then present the secondary stack selection algorithm which contains an offline data processing part and an online stack selection part including data logging. We also discuss the implementation details that address several practical considerations, including an initial data collection mode, handling data delays, hashing, and intricacies in conducting AB tests.

3. Context Tree

From business intuition and domain knowledge, we start by identifying the following contexts that can affect the performance (user engagement rate) of a secondary stack. Such contexts broadly include but are not limited to:

a) search context, such as query and Product Type (PT) associated with the query and how many exact matched items are retrieved;

b) session context, such as device platform and search filters applied;

c) customer context, such as customer's membership status

The use of context trees is well known in the literature of data compression [19]. Here we use the data structure to maintain each stack's statistics on the context level.

3.1. Building the Context Tree

To keep the notation generic, assume we have the following m contexts ranked from the most to the least important after context preselection:

$$z = (\text{ctx}_1, \ldots, \text{ctx}_m)$$

Each context ctx_i takes on a finite number of values $|\mathcal{C}_i|$. The structure of a context tree is shown in Figure 2. As a point of reference, our context tree is constructed with six layers.

We next explain the nodes in the context tree. A node at depth d contains the engagement statistics of each secondary stack under a realization of context $\text{ctx}_1, \ldots, \text{ctx}_d$, which can be traversed from the root node. Consequently, the statistics contained at any given node is an aggregation of the statistics of all its direct child nodes.

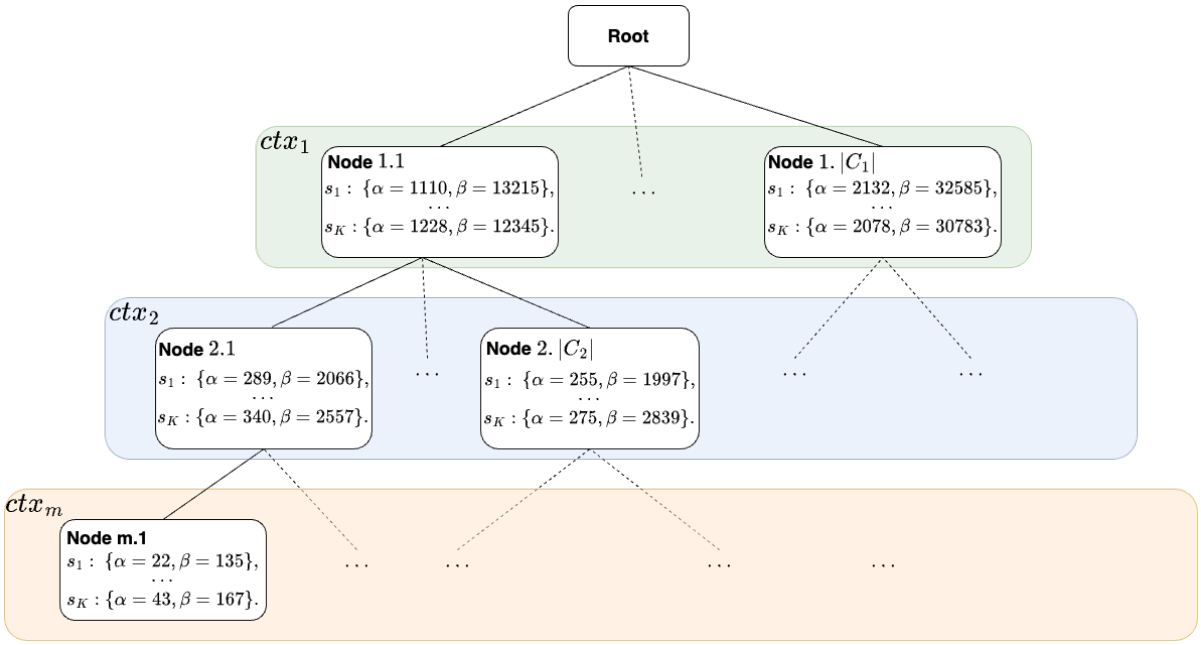


Figure 2: Sample context tree

Within each node, the engagement statistics of each secondary stack is given by two parameters which correspond to the hyperparameters of the beta distribution given in (5). That is, by looking at both parameters, we can efficiently obtain the posterior distribution of $\Theta(z, s)$ for a given secondary stack s and context realization z that is indicated by the traversal trajectory that leads to the node from the root.

To construct the tree based on a historical data set, we can first construct all the leaf nodes in the tree and then recursively compute the statistics at all the other nodes bottom up. This is presented in Algorithm 2.

In Algorithm 2, the maximum number of engaged items from a secondary stack n, as mentioned in (3), is set to 3. This is chosen by our empirical study of clicks from secondary stacks. In the rest of the algorithm, we first construct all the leaf nodes under $|\mathcal{C}_1| \times \ldots \times |\mathcal{C}_m|$ realizations. As a result of the simple form of Bayesian update in (5), we can maintain only two parameters per stack and context realization by counting the engagement directly in L.7-8. Consequently, the count of engagement at a given node is simply the sum of engagement from all its child nodes. This allows us to easily compute the statistics by recursion.

Note that the constructed context tree after executing Algorithm 2 cannot be directly used as the Bayesian statistics yet, because we have not assign any prior distribution. If we assume a uniform prior of Θ from (3) on $[0, 1]$, we can set $\alpha = \beta = 1$ in (4). Therefore, with the uniform prior assumption, we can use the context tree from Algorithm 2 as the Bayesian statistics by adding 1 to all the α and β's at each node.

3.2. Updating the Context Tree

In Algorithm 2, we provide a way of constructing a context tree from one batch of data. Next, we present a scheme to keep updating the Bayesian statistics with new observations.

The frequency of updating the context tree can be application driven. We focus on a batch update scheme, where we choose one day as the batch size or update frequency. However, the scheme is generalizable to any update batch size. We provide an algorithm to update the context tree on a daily basis in Algorithm 3.

Observe that at any point of the traversal, oldTree and oneDayTree have exactly the same structure. Therefore, we can traverse the two trees together based on the child node key of either tree. In

Algorithm 2 Construct context tree from historical data

$n \leftarrow 3$
CTX $\leftarrow [[\text{ctx}_m^{(1)}, \ldots, \text{ctx}_m^{(k_m)}], \ldots, [\text{ctx}_1^{(1)}, \ldots, \text{ctx}_1^{(k_1)}]]$
while ctx $\in \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$ **do** ▷ construct leaf nodes
    stats $\leftarrow \{\}$
    while $s \in \mathcal{S}$ **do**
        $m_{\text{impression}} \leftarrow$ number of times stack s is seen under ctx
        $\alpha \leftarrow$ number of total item engagement from stack s
        $\beta \leftarrow m_{\text{impression}} * n - \alpha$
        stats$[s] \leftarrow \{\text{“}\alpha\text{”} : \alpha, \text{“}\beta\text{”} : \beta\}$
    end while
    leaves[ctx] $\leftarrow$ stats
end while
tree $\leftarrow$ buildTreeRecursive$(m,$ leaves, $\varnothing)$

function BUILDTREERECURSIVE(curLevel, leafMap, levelKey)
    if curLevel<1 **then** ▷ base case
        return {"stats": leafMap[levelKey]}
    end if
    nodes, stats $\leftarrow \{\}$ ▷ begin recursive step
    while ctxVal $\in$ CTX[curLevel] **do**
        nodes[ctxVal] $\leftarrow$ buildTreeRecursive$(\text{curLevel} - 1, \text{leafMap}, \text{ctxVal} \times \text{levelKey})$
    end while
    while $s \in \mathcal{S}$ **do** ▷ compose stats from child nodes
        $\alpha, \beta \leftarrow 0$
        while ctxVal $\in$ nodes **do**
            $\alpha \leftarrow \alpha +$ nodes[ctxVal]["stats"]$[s][\text{“}\alpha\text{”}]$
            $\beta \leftarrow \beta +$ nodes[ctxVal]["stats"]$[s][\text{“}\beta\text{”}]$
        end while
        stats$[s] \leftarrow \{\text{“}\alpha\text{”} : \alpha, \text{“}\beta\text{”} : \beta\}$
    end while
    return {"stats" : stats, "nodes" : nodes}
end function

Algorithm 3, we provide a way of efficiently updating the posterior distribution of $\Theta(z, s)$ by batch. If it is the first batch of data, oldTree is initialized to reflect uniform distributions of $\Theta(z, s)$ for every secondary stack s and context realization z, i.e. Beta$(1, 1)$. Exploiting the identical tree structure, each node with the corresponding statistics can be traversed and updated using Depth First Search (DFS).

In order to balance the "Explore" capability in Thompson sampling, a discount factor $\lambda \in [0, 1]$ is enforced to prevent the parameters α and β's from growing unboundedly. This is to ensure that L.5 in Algorithm 1 does not become deterministic.

The rationale for the parameter update in L.20-21 of Algorithm 3 is the following result in Proposition 1.

Proposition 1. *Given two random variables*

$$\Phi_0 \sim \textit{Beta}(\alpha_0, \beta_0) \textit{ and } \Phi_1 \sim \textit{Beta}(\alpha_1, \beta_1)$$

with the corresponding probability density functions $f_{\alpha_0,\beta_0}(\cdot)$ and $f_{\alpha_1,\beta_1}(\cdot)$, respectively, the distribution defined by a new probability density function $f^{(\lambda)}(\cdot)$ of the tilted *form*

$$\log f^{(\lambda)}(x) \triangleq (1 - \lambda) \log f_{\alpha_0,\beta_0}(x) + \lambda \log f_{\alpha_1,\beta_1}(x) + \textit{constant}$$

is again a beta distribution

$$Beta\left((1-\lambda)\alpha_0 + \lambda\alpha_1, (1-\lambda)\beta_0 + \lambda\beta_1\right).$$

Note that the constant term itself is equal to $(1-\lambda)D_\lambda(\Phi_1\|\Phi_0)$, where $D_\lambda(\Phi_1\|\Phi_0)$ is known as the Rényi divergence [20].

By updating the hyperparameters as a convex combination of the ones from the oldTree and the oneDayTree as in L.20-21, we obtain the *tilted* distribution.

The discount factor λ itself can be adaptive. We discuss more on the adaptation scheme in Section 5.

Algorithm 3 Update context tree by batch

```
1: oldTree ← load existing context tree
2: oneDayTree ← construct tree with Algorithm 2 with one day of data
3: if oldTree == ∅ then                                  ▷ first day, no previous tree
4:     oldTree ← uniform tree
5: end if
6: updatedTree ← updateTree(oldTree, oneDayTree, λ)

7: function UPDATETREE(oldTree, oneDayTree, λ)
8:     if oldTree == ∅ then                                                ▷ base case
9:         return ∅
10:    end if
11:    newTreeNodes, stats ← {}
12:    childNodes ← oldTree["nodes"]
13:    while ctxVal ∈ childNodes do                                          ▷ DFS
14:        newSubTree ← updateTree(childNodes[ctxVal], oneDayTree["nodes"][ctxVal], λ)
15:        newTreeNodes[ctxVal] ← newSubTree
16:    end while
17:    while s ∈ 𝒮 do
18:        α0, β0 ← oldTree["stats"][s]["α"], ["β"]
19:        α1, β1 ← oneDayTree["stats"][s]["α"], ["β"]
20:        α ← (1 − λ) ∗ α0 + λ ∗ α1
21:        β ← (1 − λ) ∗ β0 + λ ∗ β1
22:        stats[s] ← {"α" : α, "β" : β}
23:    end while
24:    return {"stats" : stats, "nodes" : newTreeNodes}
25: end function
```

3.3. Context Preselection

If the sample size were not an issue, including more contexts could let us learn each stack's performance more accurately under various situations. However, recall that in (6), we are maintaining each stack parameters on a per context vector realization level. The sample size under each context vector realization decreases exponentially with the number of contexts, thereby eventually leading to overfitting. To combat the overfitting issue, we propose a context preselection method. To allow this context preselection, we enable an initial data collection phase with a small percentage of traffic, where all the qualifying secondary stacks are selected uniformly at random, and the corresponding contexts and user engagement are logged. This is also known as ϵ-greedy in the literature of Explore and Exploit. However, we use this phase only for context preselection and context tree initial construction. The details of experiment setup for the data collection phase are presented in Section 5.

In order to shortlist the context and avoid overfitting, we design a simple scheme using the stack engagement data from the data collection phase. We split the data chronologically, i.e. first time window

is used for training and the second time window is used for evaluation. We treat it as a winning stack prediction problem. For each context or set of context, we use the statistics of each secondary stack under the given context to predict the winning stack. When evaluated on the testing data set, the prediction accuracy should increase when we include one more useful context and decrease when we include too many contexts and start overfitting.

3.4. Operational Flexibility

Observe that with the context tree structure, it is possible to invoke the stack selection algorithm at different stage of the search flow. For example, given a context tree (after context preselection) with structure “query -> platform -> filter -> recall size”, the stack selection algorithm can be applied pre-retrieval with the context up to “query -> platform -> filter” as well as post-ranking, i.e. after all the stacks have already been generated using the full context. This flexibility makes the solution applicable to a wide range of use cases. In this example, invoking the algorithm pre-retrieval can reduce cost by limiting the number of secondary stacks to generate; whereas invoking the algorithm post-ranking can refine the selection based on the search result.

4. Secondary Stack Selection Algorithm

4.1. Architecture

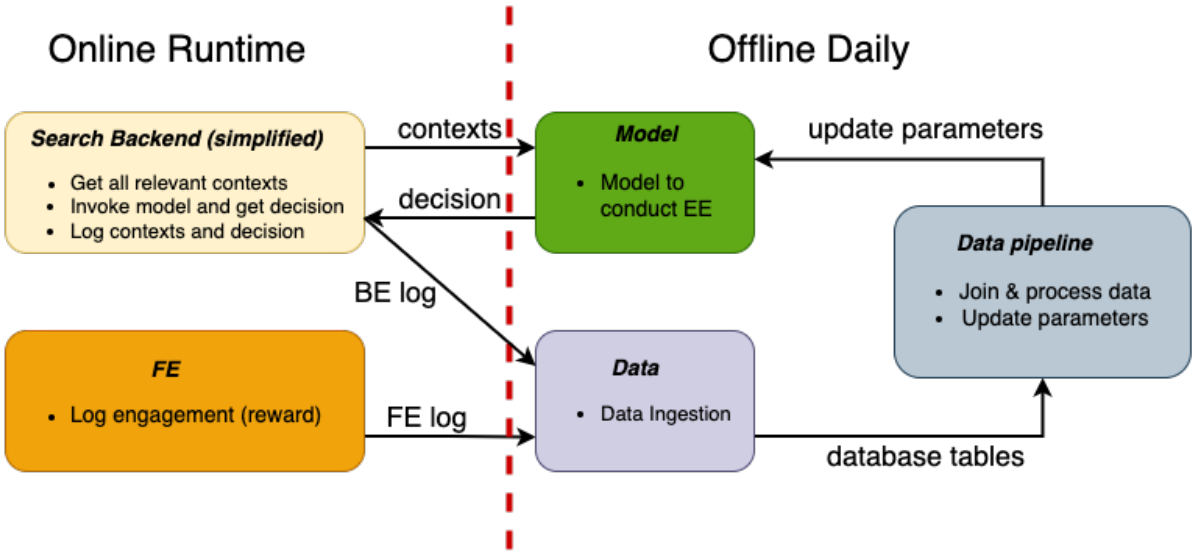
In this section, we present the complete secondary stack selection logic using Thompson sampling. Recall that our objective is to maximize the secondary stack user engagement in the long run by selecting the best performing stack among all based on the context "most of the time" while continuously exploring the potentially underperforming stack "from time to time" to capture any non-stationarity. The overall flow of the secondary stack selection procedure is illustrated in Figure 3. The flow contains two major components: offline and online.

In the offline component, we manage the context tree update according to the schemes in Section 3. This is implemented through a daily automatic data pipeline which reads the backend log and frontend log, and joins the information to update the context tree. Recall the data format in (1), the context z_i and selected secondary stack s_i are from the backend log and the engagement data, i.e. the reward, r_i is from the frontend log. The updated context tree effectively acts as the "Model" as it contains all the parameters that are needed for the stack selection.

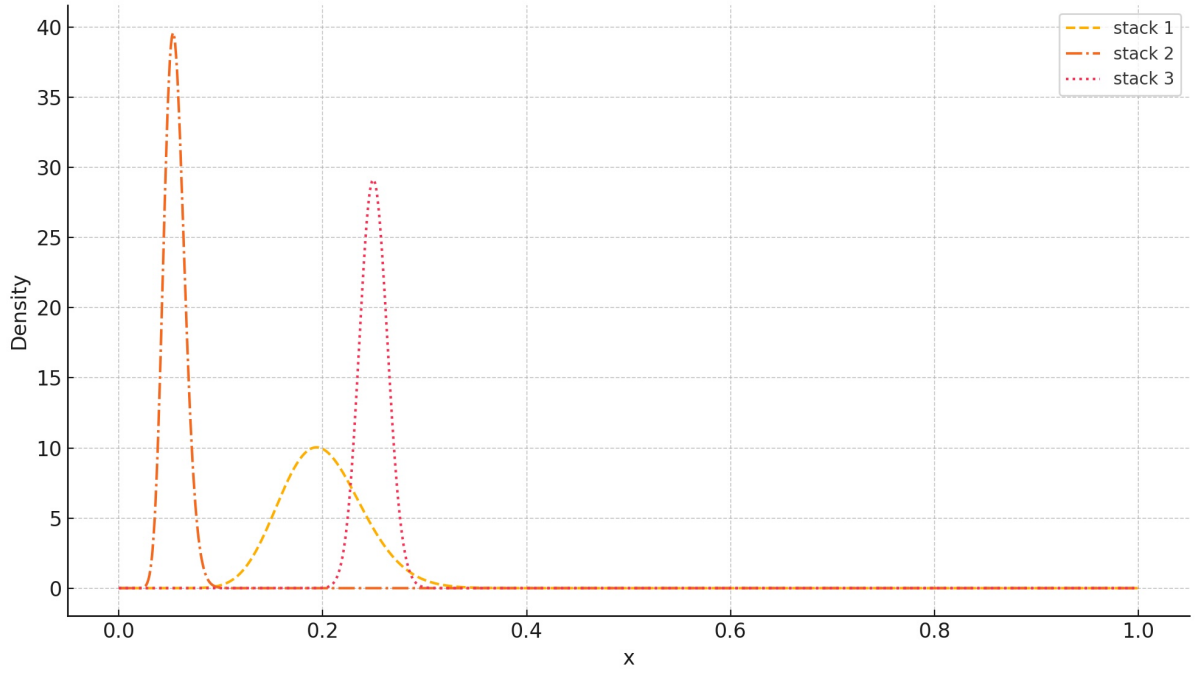
In the online part, we conduct the Explore and Exploit operation by using Thompson sampling. At runtime, when a search request is triggered, different qualifying secondary stacks are computed and generated in the background. The resulting stacks are returned to the backend engine. At this point, all the required contexts are available in the backend engine as well. Based on the most updated parameters published by the data pipeline and the context, the backend engine must make a decision on which secondary stack to select when there are multiple options. The general scheme to conduct Thompson sampling is already given in Algorithm 1. However, there are more practical considerations to be accounted for during implementation.

4.2. Node statistics reliability

In normal circumstances, we should always invoke the statistics at a leaf node by traversing down the context tree shown in Figure 2 by using the context obtained at runtime. This allows us to retrieve the most accurate stack performance history given a context combination. However, the context preselection in 3.3 only eliminates overfitting on average. There can be leaf nodes under which certain stacks have not been seen enough number of times. Theoretically, we can let Thompson sampling handle the tradeoff between explore and exploit and gradually accumulate more observations. Because of the site facing nature of our application, we place a further regularization based on the reliability of the statistics at the node. When the confidence is low due to low impressions, we trace back one level



(a) Architecture



(b) Example of beta distributions of three stacks with (α, β) set to $(20, 80)$, $(27.5, 472.5)$ and $(250, 750)$, respectively. In this example, stack 3 is confidently better than stack 2, with high concentrations. During the Thompson sampling, stack 3 is preferred to stack 2 almost deterministically (exploit). However, between stack 1 and stack 3, the average engagement rate is not too different. With the lower concentration from stack 1, there will be more explore between the two stacks during Thompson sampling.

Figure 3: Architecture and sample beta distributions to conduct Thompson sampling

by going back to the parent node and invoke the statistics carried by the parent node. We repeat it on the parent node if the impression is still low until we reach an ancestor node on the path where the impression exceeds a chosen threshold $\tau_{\text{impression}}$ or we hit the first level context node.

In addition, we enforce another guardrail to prevent the worst secondary stack from showing. The reason for this consideration is that there is no guarantee on the recall quality from each individual stack's qualifying condition. Due to the random nature of Thompson sampling, there is a non-zero probability

that a bad secondary stack is selected for exploration. Theoretically, we can rely on the algorithm itself to converge based on the engagement statistics it collects. However, since we are operating in production, we need to mitigate the risk of harming business metrics or breaking customers' trust from doing the exploration. To address this issue, we apply a regularization by setting a dynamic threshold using the mean of the beta distribution corresponding to the second worst stack. If the winner stack from Thompson sampling is worse than the overall performance of the second worst stack among all, the winner stack is dropped and consequently no secondary stack is shown.

4.3. Runtime algorithm

We next outline the main algorithm that conducts the stack selection per search request with Thompson sampling with the practical considerations that are pointed out earlier.

Algorithm 4 Runtime stack selection with Thompson sampling

1: **Input**: contextTree, currentContext, eligibleStacks, sessionId
2: **Ouput**: selectedStack

3: nodePath $\leftarrow$ traverseTreeByCtx(currentContext) ▷ returns a list of nodes from root to leaf
4: $\theta^*, n_{\text{impression}} \leftarrow 0, i \leftarrow d$, node $\leftarrow \varnothing$, ctrList $\leftarrow []$ ▷ d is the depth of the context tree
5: **while** $n_{\text{impression}} < \tau_{\text{impression}}$ and node is not root **do** ▷ find operating node
6: $i \leftarrow i - 1$
7: node $\leftarrow$ nodePath[i]
8: $n_{\text{impression}} \leftarrow$ sumImpressionsFromNode(node)$/n$
9: **end while**
10: **while** $s \in$ eligibleStacks **do**
11: seed $\leftarrow$ hash(sessionId + query + s)
12: $\alpha, \beta \leftarrow$ readStatsFromNode(node)
13: $\theta \leftarrow$ beta(seed, α, β) ▷ generate a random number
14: **if** $\theta > \theta^*$ **then**
15: selectedStack $\leftarrow s$
16: $\theta^* \leftarrow \theta$
17: **end if**
18: ctrList.add $\left(\frac{\alpha}{\alpha+\beta}\right)$
19: **end while**
20: **if** θ^* < secondSmallest(ctrList) **then**
21: selectedStack $\leftarrow \varnothing$
22: **else**
23: logging(requestId, currentContext, selectedStack)
▷ requestId is used as the join key with FE log
24: **end if**

The algorithm operates on a per search request level. For each search request, we obtain its corresponding context and execute Algorithm 4 on all the qualifying secondary stacks to select the final one to be displayed.

Due to the random nature of Thompson sampling, without further regulation, it is possible for the same user to experience different selections of secondary stacks under the same query, e.g. a customer may search for the same query twice. This can lead to confusing customer experience, especially within a short time window. To prevent such situation from happening, we set a constraint so that a user always sees the same choice of secondary stack under the same query within a session. A session is defined as a thirty-minute window uniquely assigned to each visitor. This is achieved in L.11 by concatenating the session ID, the query and the stack name, which are all strings respectively, and using the hash value of the concatenated string as seed.

5. Experimental Setup and Results

In this section, we explain how the AB tests are conducted in production on Walmart.com, which include a data collection phase and a full explore-exploit phase.

There are a total of ~ 10 secondary stacks live in production. In the control group, the stack selection logic is exactly the same as in production, i.e. following a manually configured priority that is universal w.r.t. context. Consequently, one stack is always preferred over another stack according to the manual priority rule regardless of the context. As previously mentioned, we have two phases. In the data collection phase, the goal is to collect unbiased data to initialize the context tree. It can be also interpreted as learning the prior distributions. No significant business metrics improvement is expected in the variant group. In the full explore-exploit phase, the goal is to show that the stack selection algorithm Algorithm 4 together with the context tree has an effect on improving the Converted Visitor (CV) rate.

5.1. Data collection phase

In this phase, we set up an AB test with a small percentage of the production traffic, 3% in both control and variant group. In the variant group, we aim to collect unbiased data to initialize the context tree. To achieve that, we allow a uniform random triggering among all the qualifying secondary stacks. The corresponding context encountered and the actual user engagement are logged together with the selected secondary stack. In theory, we should only need the data from the variant group. However, we set up the control group to monitor the impact from random triggering. If the uniform random triggering introduce too much suboptimality that it starts to show statistically significant loss in CV, we must stop or reduce the traffic of this experiment in order not to harm the business. This experiment is set up on three platforms: web, iOS, and Android, respectively. The metrics that are closely monitored. During the data collection period, no statistically significant lift or drop is detected in the business metrics. Therefore, we keep this test running at low traffic over the course of four weeks.

As pointed out in Section 3.3, the goal of data collection is to preselect the useful context and initialize the context tree. We start with the following context based on business intuition: query or PT, platform, whether a fulfillment based filter is applied, and customer's Walmart membership status. We briefly explain the query / PT context. Since the secondary stack is shown on the search page, we believe the query plays an important role. However, it is not always possible to aggregate the statistics on a per query level as the query itself is free text and the number of unique queries is unbounded. For frequently searched queries, we consider the normalized query[1] as one context. For infrequently searched queries, we use its PT as the corresponding context. A query PT can be understood as a quantization of the actual queries where the cardinality is bounded ($\sim$ 8000 in Walmart production). By conducting the context preselection scheme discussed in 3.3, we find out only two levels of context, query / PT and platform contribute positively to the overall prediction accuracy. The various combinations of contexts and their corresponding prediction accuracy is reported in Table 1. The context preselection is learned based on ten consecutive days of data and evaluated on next five days of data. After the context preselection, we initialize the context tree using the four weeks of data collected by executing Algorithm 2 and 3 day by day.

Note that in the context tree initialization, in order to quickly establish the statistics, we have an adaptation scheme for the discount factor λ in Algorithm 3, where we replace λ with λ_1 and $1-\lambda$ with λ_0. This is summarized as

$$\begin{aligned}
&\lambda_0 \leftarrow 0.99, \lambda_1 \leftarrow 1, \text{ if } n_{\text{impression}} < 300 \\
&\lambda_0 \leftarrow 0.9, \lambda_1 \leftarrow 0.1, \text{ if } n_{\text{impression}} > 3000 \\
&\lambda_0, \lambda_1 \leftarrow \text{linear interpolation in } [0.99, 0.9] \text{ and } [1, 0.1] \text{ otherwise}
\end{aligned}$$

where $n_{\text{impression}}$ is the total number of impressions from all observed secondary stacks at the first level (query or PT) node.

[1] A normalized query is by standardizing the expression based on stemming, such as singular/plural forms.

Table 1
Context Preselection

Single context	Accuracy	Combined context	Accuracy
no context	0.7758	query + platform	0.8758
query	0.8684	query + platform + filter	0.8629
PT	0.8477	PT + platform	0.8604
platform	0.7802	PT + platform + filter	0.8546
filter	0.7761		

Table 2
Search metrics in % lift

Metrics	Web (%)	iOS (%)	Android (%)
Search ATC / Visitor	0.23	0.33^{+}	0.14
Search Direct ATC / Visitor	0.29	0.34^{+}	0.24
Search Rf Item Page ATC / Visitor	-0.07	0.29	-0.43
Clicks To ATC	-0.16	-0.23^{+}	0.16

Table 3
Secondary stack metrics in % lift

Metrics	Web (%)	iOS (%)	Android (%)
Secondary stack surface rate	-6.10^{-}	-9.38^{-}	-8.60^{-}
Secondary stack ATC / Visitor	5.41^{+}	11.14^{+}	4.10^{+}

Table 4
Engagement Metrics from Explore-Exploit in % lift

Metrics	Web	p-value	iOS	p-value	Android	p-value
CV	0.17%	0.3249	$0.22\%^{+}$	0.0750	0.01%	0.9695
UNITS	0.47%	0.1267	$0.31\%^{+}$	0.0991	0.06%	0.8313

5.2. Explore-Exploit phase

At this point, we have all the components to start the full explore-exploit AB test. In this phase, we set up a regular AB test with higher production search traffic, 15% each in control and variant group. The goal of this test is to show that the scheme can move up business and search metrics. In both the control and the variant group, we have the same set of secondary stacks. Recall that in the control group, the selection of the secondary stack is pre-configured given a prioritization order regardless of any context. Similar to the data collection phase, the experiment is set up on three platforms: web, iOS and Android, respectively. In the variant group, we test the explore-exploit algorithm by conducting Thompson sampling in Algorithm 4. The tests are run for a two-week period in production. We report the search metrics, secondary stack specific metrics, and engagement metrics in Table 2, 3 and 4, respectively. The numbers in color (or with superscript) indicate statistical significance with p-value < 0.1, where color green (or superscript +) generally indicates a change in the "good" direction and color red (or superscript -) generally indicates a change in a "bad" direction.

We next explain how to interpret the metrics encountered in the tables. From the search metrics table, ATC stands for "Add To Cart". With this definition, it is intuitively easy to understand Search ATC / Visitor (the higher the better). The next two metrics Search Direct ATC / Visitor and Search Rf Item Page ATC / Visitor measure the ATC from two sources: the search page directly and the item page after the customer clicks on an item from the search page (the higher the better). The last metric Clicks to ATC is a search efficiency metric, defined as the average number of clicks it takes for a customer to

add an item to cart (the lower the better).

From the secondary stack specific metrics, we report the stack surface rate and number of secondary stack add to cart per visitor. Note that the decrease in surface rate of secondary stacks is expected because of the guardrail we impose during Thompson sampling as was discussed in 4.2. The key observation here is that the number of ATC from secondary stacks has significantly increased across platforms despite a lower surface rate. This shows that the algorithm is triggering the secondary stacks less but target customers more accurately.

Overall, the experimental results have shown that the algorithm is highly effective in targeting the right secondary stack selection based on the secondary stack metrics. The improvement on search metrics and business metrics is mostly reflected in the test on iOS. Our conjecture is that these are more general metrics and therefore, require longer time (more data) for the effect to propagate. Since the traffic on iOS is relatively large, we already observe the effect.

We discuss briefly a few practical considerations in productionalization of the algorithm, including how to onboard a new secondary stack and conduct AB tests with new stacks. In order to keep the maintenance of the context tree manageable, we keep only one version of the tree at any time. When a new secondary stack enters the competition, an AB test needs to be conducted to evaluate the effectiveness of the new stack. An initial statistics, such as a uniform distribution on $[0, \epsilon]$ where $\epsilon \leq 1$, can be assigned based on the average of all qualifying stacks in production to the new stack on Day 1. Once the experiment has started, the engagement data and corresponding context of the new stack will be collected. In both control and variant group, we conduct Thompson sampling by accessing the same context tree. However, in the control group, the new stack is not considered as a qualifying stack. The data collected from control and variant groups contributes equally to updating the context tree.

6. Conclusion

We optimize the search page secondary stack (recommender) selection using a low-cost and lightweight contextual multi-armed bandit approach. The algorithm has two major components: an offline data pipeline that maintains the statistics of each secondary stack under various contexts, and an online stack selection using Thompson sampling. We design a context tree data structure that allows Thompson sampling to be conducted with full or partial contexts. This flexibility makes the solution general enough to be invoked at various stage of the search flow depending on the use case. We achieve the results by conducting the experiment in two phases. In the data collection phase, we collect unbiased data of all the secondary stacks by enabling a uniform selection among all qualified stacks. In the explore-exploit phase, we use Thompson sampling to handle the tradeoff between explore and exploit. The AB test results show positive impact on business metrics, search metrics and secondary stack metrics. Particularly, within the secondary stack metrics, we are able to target the triggering of the stack more accurately, reflected in a significant increase in the secondary stack Add To Cart per Visitor rate.

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

[1] W. R. Thompson, On the likelihood that one unknown probability exceeds another in view of the evidence of two samples, Biometrika 25 (1933) 285–294.

[2] T. L. Lai, H. Robbins, Asymptotically efficient adaptive allocation rules, Advances in applied mathematics 6 (1985) 4–22.

[3] O. Cappé, A. Garivier, O.-A. Maillard, R. Munos, G. Stoltz, Kullback-leibler upper confidence bounds for optimal sequential allocation, The Annals of Statistics (2013) 1516–1541.
[4] E. Kaufmann, O. Cappé, A. Garivier, On bayesian upper confidence bounds for bandit problems, in: Artificial intelligence and statistics, PMLR, 2012, pp. 592–600.
[5] S. Agrawal, N. Goyal, Analysis of thompson sampling for the multi-armed bandit problem, in: Conference on learning theory, JMLR Workshop and Conference Proceedings, 2012, pp. 39–1.
[6] S. Agrawal, V. Avadhanula, V. Goyal, A. Zeevi, Thompson sampling for the mnl-bandit, in: Conference on learning theory, PMLR, 2017, pp. 76–78.
[7] O. Chapelle, L. Li, An empirical evaluation of thompson sampling, Advances in neural information processing systems 24 (2011).
[8] S. Agrawal, N. Goyal, Near-optimal regret bounds for thompson sampling, Journal of the ACM (JACM) 64 (2017) 1–24.
[9] O.-C. Granmo, Solving two-armed bernoulli bandit problems using a bayesian learning automaton, International Journal of Intelligent Computing and Cybernetics 3 (2010) 207–234.
[10] B. C. May, N. Korda, A. Lee, D. S. Leslie, Optimistic bayesian sampling in contextual-bandit problems, Journal of Machine Learning Research 13 (2012) 2069–2106.
[11] L. Li, W. Chu, J. Langford, R. E. Schapire, A contextual-bandit approach to personalized news article recommendation, in: Proceedings of the 19th international conference on World wide web, 2010, pp. 661–670.
[12] R. Zhu, M. Rigotti, Deep bandits show-off: Simple and efficient exploration with deep networks, Advances in Neural Information Processing Systems 34 (2021) 17592–17603.
[13] S. Agrawal, N. Goyal, Thompson sampling for contextual bandits with linear payoffs, in: International conference on machine learning, PMLR, 2013, pp. 127–135.
[14] R. Kleinberg, Nearly tight bounds for the continuum-armed bandit problem, Advances in Neural Information Processing Systems 17 (2004).
[15] S. Magureanu, R. Combes, A. Proutiere, Lipschitz bandits: Regret lower bound and optimal algorithms, in: Conference on Learning Theory, PMLR, 2014, pp. 975–999.
[16] T. Lu, D. Pál, M. Pál, Showing relevant ads via context multi-armed bandits, in: Proceedings of AISTATS, 2009.
[17] A. Slivkins, et al., Introduction to multi-armed bandits, Foundations and Trends® in Machine Learning 12 (2019) 1–286.
[18] D. J. Russo, B. Van Roy, A. Kazerouni, I. Osband, Z. Wen, et al., A tutorial on thompson sampling, Foundations and Trends® in Machine Learning 11 (2018) 1–96.
[19] F. Willems, Y. Shtarkov, T. Tjalkens, The context-tree weighting method: basic properties, IEEE Transactions on Information Theory 41 (1995) 653–664. doi:10.1109/18.382012.
[20] A. Rényi, On measures of entropy and information, in: Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics, volume 4, University of California Press, 1961, pp. 547–562.