R11/Sarthak Mittal/200050129

March 19, 2023

This paper targets the problem of **personalized web services**, where the content is **dynamically** changing, and the **scale** demands quick computation and adaptation. The authors model the recommendation as a **contextual bandit problem** making use of **contextual information** of users and articles and adapting strategy based on **user-click feedback**. They propose a more computationally efficient contextual bandit **algorithm**, argue that any bandit algorithm can be evaluated **offline** using recorded random traffic, and apply this method to their new algorithm to obtain results.

In general, to model popularity and temporal changes, consumer feedback is collected from a fraction of the traffic **(exploration)** and the most popular content is **exploited** on the remaining traffic (ϵ-greedy). Personalization involves making use of current and past behaviour to deliver the individual best content. Due to vast possible differences in user views on the same content, various filtering and recommendation methods work well on an **individual** level. However, due to frequent changes, learning the **"goodness"** of the match between interests and content.

In the context of article recommendation, the expected payoff of an article is the **click-through rate (CTR)**, and maximizing this value is equivalent to maximizing expected clicks. Additionally, access to user information can help **generalize** CTR information across users. The authors also discuss existing bandit algorithms briefly.

The authors analyze the case where the payoff model is **linear** and call this algorithm **LinUCB**. They consider disjoint linear models (no sharing of parameters across arms), where the UCB formulated by them depends on the **design matrix**, d-dimensional **feature**, the **empirical estimate** of the coefficients (for expectation) obtained using ridge regression, and a probability parameter δ. Their algorithm **caches** the inverse of the matrix A to reduce computation and remains **efficient** as long as the set of arms or actions is not too large.

To extend the algorithm to hybrid models, they add an **additional linear term** to the expectation using a coefficient vector common to all arms. In this case, the UCB's derivation relies on **block matrix inversion**.

Due to the interactive nature of the problem, the authors would have been forced to use **"off-policy evaluation"**. The option to build a simulator was rejected due to the possibility of a **bias**. Instead, they suggest an evaluation method assuming individual events are **i.i.d.** and the logging policy chose

arms **uniformly at random**. Their solution can be modified using rejection sampling but with reduced efficiency. They assume a **distribution** of observed feature vectors and **hidden** payoffs and access to a large sequence of **logged events**. The evaluator **steps** through the logged events, and if the policy chooses the **same** action after considering the history and the current context, the history and payoff are updated. The proven theorem shows that the **policy evaluator**, when run with the policy and the stream of events, will produce the correct probability.

The paper also explains the experimental setup used for running the algorithm and evaluator on the **Yahoo! Today-Module**, involving data collection, feature construction and encoding, followed by a **comparison** of several algorithms under different model settings. The analysis revealed that when the parameter was **too large**, the algorithms over-explored, while **warm-start** proved helpful in finding a better match between interests and content. All online-learning policies outperformed the omniscient approach. ϵ-greedy algorithms had lower CTR in the **learning** setting (due to being unguided due to random choice). Even with the **variation** in data size, UCB algorithms outperformed ϵ-greedy ones, and LinUCB (hybrid) showed a lot of benefits when the data size was **small**. The authors felt that more features with **diverse components** (for example, using PCA) would be needed to distinguish LinUCB (disjoint) and UCB (seg).