Submodular maximization for random streams

Paul Liu, Aviad Rubinstein, Jan Vondrák, Junyao Zhao

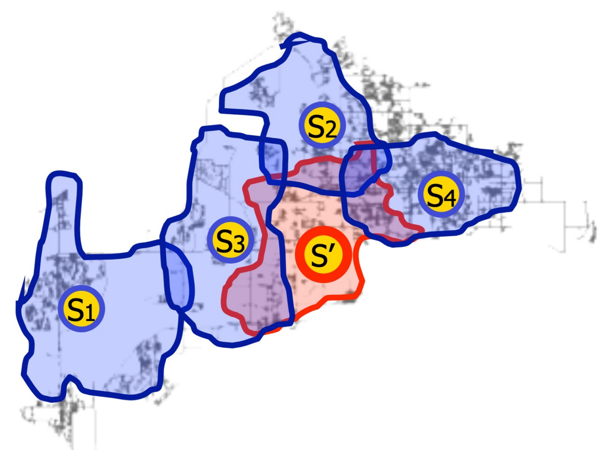
Stanford Computer Science

NEURAL INFORMATION PROCESSING SYSTEMS

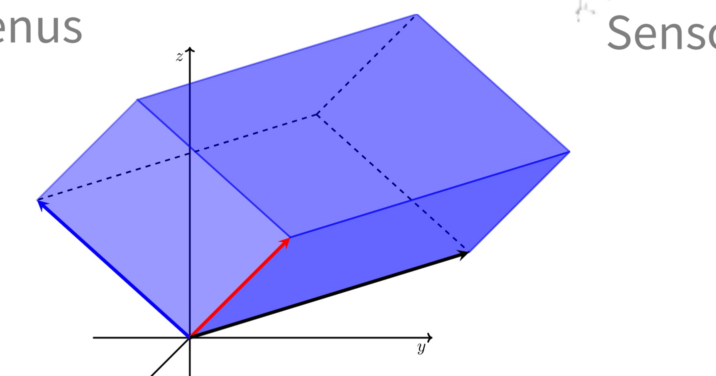
MOTIVATION

Are submodular functions in your backyard?

McDonald's Menus



Sensor placement



$f(S) = \log\det A_S^T A_S$

Determinantal point processes

Submodularity: $f(S\cup\{e\})-f(S) \geq f(T\cup\{e\})-f(T)$ when $S\subseteq T$.
Monotone: $f(S\cup\{e\}) \geq f(S)$
Typical application: $\max_{|S|\leq k} f(S)$ for a given k.

- "cardinality constrained submodular maximization"

REQUIREMENTS OF MODERN SYSTEMS

Modern data often comes in the form of a stream.

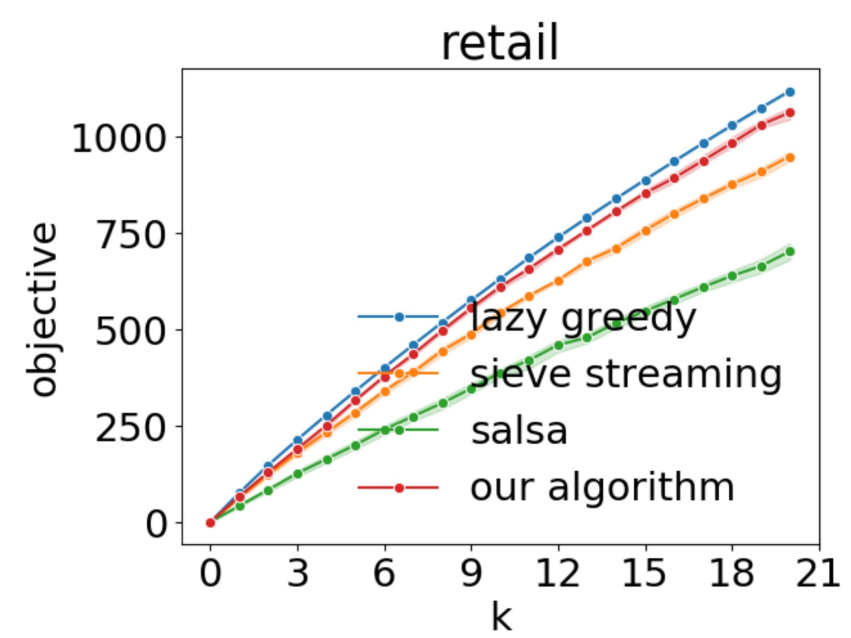
Characteristics of typical systems:

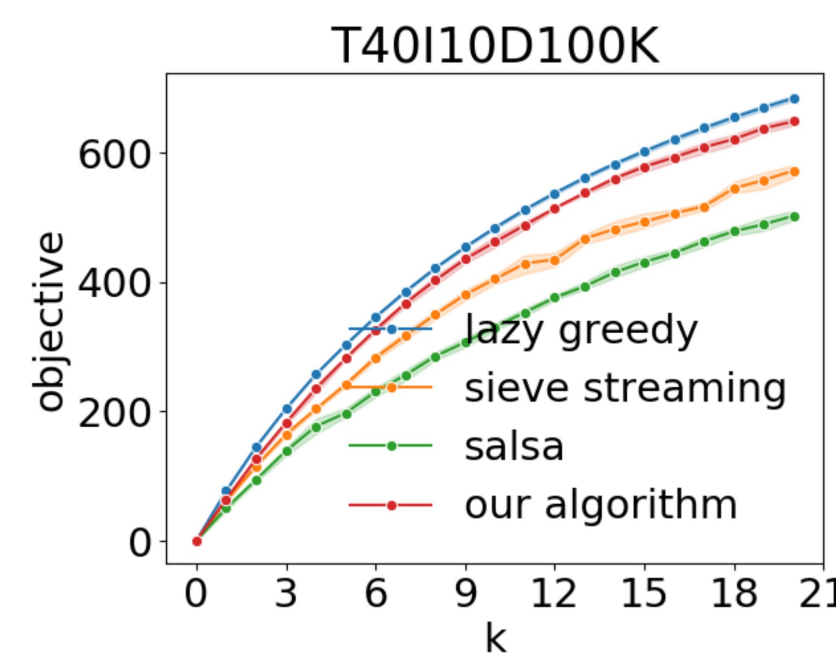
- Memory is limited
- Random access is not possible
- Data is sampled from some underlying distribution

Classical algorithms require storing the entire dataset. Can we do better?

OUR CONTRIBUTIONS

Our algorithms take exponentially less memory than current state-of-the-art while achieving better approximations in practice.





Our algorithm on real-world data. "Lazy greedy" is the best known *offline* solution.

Authors	Memory	Approx.	Notes
Sieve streaming, Badanidiyuru et al. '14	$O\left(\frac{k\log k}{\epsilon}\right)$	$\frac{1}{2}-\epsilon$	Adversarial order
Salsa, Norouzi-Fard et al. '18	$O(k\log k)$	$\frac{1}{2}+c$, $0<c \leq 10^{-13}$	Random order, also gives 1/2 l.b. for adversarial order
Agrawal et al. '19	$O(k\exp(\mathrm{poly}(1/\epsilon))$	$1-\frac{1}{e}-\epsilon$	Optimal approx.; constant of $> 2^{100}$ for $\epsilon < 0.2$.
Our paper	$O(k/\epsilon)$	$1-\frac{1}{e}-\epsilon$	Works in practice; also $\frac{1}{e}$ for non-mono.

THE COMPUTING MODEL

We only require black-box access to the submodular function and the ability to store data set elements.

Formally, we assume the following:

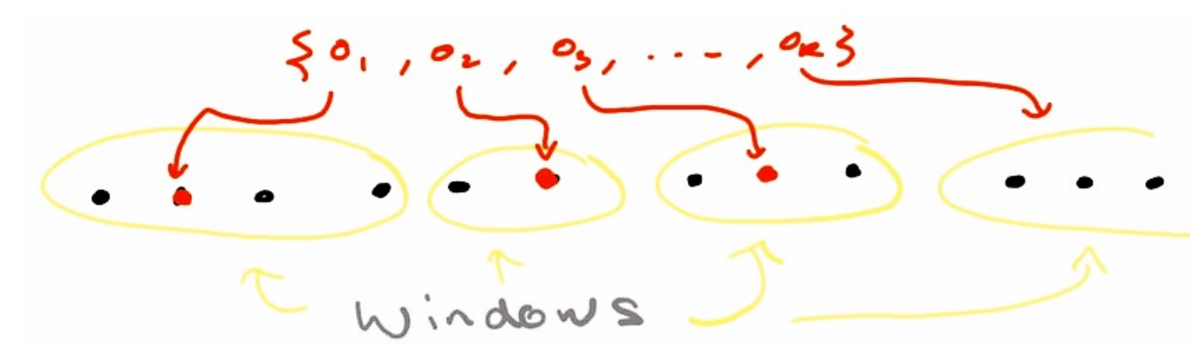
- Elements from a ground set T is streamed in to the algorithm, with order according to a uniformly random permutation.
- The algorithm chooses to store or throw away the element the moment it is seen.
- Only $\tilde{O}(k)$ elements can be stored.

ALGORITHM OUTLINE

Our algorithm relies on two ingredients: a *solution cascade* and a *random window partitioning*.

Random window partitioning:

- Randomly partition the stream into $O(k)$ windows.
- Most optimal elements go into their own window.
- In these windows we can make progress.



Solution cascade:

- A pyramid of solutions $\{L_1, L_2, \ldots, L_k\}$ with $|L_i| = i$.
- Greedily add one element to each level per window, choosing from window + elements added in previous windows.

Add e (and replace L_{i+1}) if $f(L_i \cup \{e\}) \geq f(L_{i+1})$.