Motivation. “Classical” algorithms (think 6.046) are often efficient in practice while providing worst-case guarantees. However, many of them have limited adaptivity to inputs\(^1\), meaning that they do not use the fact that some inputs are easier to solve than others. As a result, they often need to be fine-tuned via algorithm engineering to be practical. On the other hand, machine learning has been remarkably successful in identifying useful data properties, to the point that many recent papers directly train machine learning classifiers to solve well-defined algorithmic problems. However, we do not have a good understanding of what those “inferred” algorithms do and when (and how badly) they fail. This leads to the question of whether we can achieve the best of both worlds\(^2\), namely design algorithms that use machine learning to adapt to input data, while providing meaningful guarantees for their correctness and performance. Such algorithms, called learning-augmented algorithms, are the main focus of this class.

Lecture format. After spending few lectures on the basics of machine learning, we will have mostly invited speakers to present their recent results on design of learning-augmented algorithms\(^2\).

A few clarifications

- In this course we assume basic knowledge of machine learning. So if your goal is to learn ML, this is not the best course for you to take. That said, we will spend the next two lectures on machine learning essentials, just in case. We will also have a crash course on cloud computing later in the semester.

- In this course, we would like to understand how to employ machine-learning approaches to improve runtime or space consumption or accuracy of existing algorithms. In other words, our focus is on “machine learning for algorithms”. This should be contrasted with recent work on improving/analyzing modern machine learning methods, which could be described as “algorithms for machine learning”.

Next, we describe the first example of a learning-augmented algorithm, for the Bloom Filter data structure.

1 Bloom Filter

Bloom filter is a space-efficient probabilistic data-structure for “set-membership” introduced by Bloom [Blo70] and has applications in different domains such as networking and security. In set-

---

\(^1\)We note that there are many approaches in TCS focused on designing adaptive algorithm, e.g., self-adjusting data structures, self-improving algorithms or instance-optimal algorithms. However, those approaches typically do not use modern machine learning methods, unlike the algorithms covered in this class.

\(^2\)Most of the material covered in this course are from last two years.
membership, given a set of elements \( S = \{x_1, \cdots, x_n\} \), the goal is to determine whether a given \( x \) is in \( S \) or not. To achieve low-space, Bloom filter allows a small amount of error in its solution: if a given key is not in the set, a Bloom filter may mistakenly say that it belongs to the set (this is called a false positive).

A Bloom filter consists of two components: 1) an array \( B \) of \( m \) bits and 2) \( k \) independent hash functions \( h_1, \cdots, h_k \) each mapping a very large universe \( U \supset S \) to the set \( R_m = \{1, \cdots, m\} \). We assume that hash functions \( h_i \) behave as independent random functions i.e. each hash function \( h_i \) maps each key \( y \in U \) to each element of \( R_m \) with probability \( \frac{1}{m} \).

In the initialization phase, for each \( x \in S, i \in [k] \), a Bloom filter sets \( B[h_i(x)] = 1 \). Then, for a given query \( y \), to decide whether \( y \in S \), the Bloom filter outputs Yes iff for all \( i \in [k], B[h_i(y)] = 1 \). See Figure 1 for an example.

![Figure 1: An example of a Bloom filter with \( m = 18 \) and \( k = 3 \) representing the set \( \{x, y, z\} \).](image)

An important property of Bloom filters is the following: while a Bloom filter may have false positives which means that for an element \( y \notin S \) the Bloom filter outputs Yes, it never has false negatives. In other words, for all elements in \( S \) the Bloom filter always output Yes. This immediately follows from the initialization phase of Bloom filters. This property is crucial for many applications, e.g. detecting malicious URLs, where one does not want to report an unsafe URL as safe.

Next, we analyze the false positive rate of Bloom filters.

**Lemma 1.** For any query element \( y \notin S \), \( \mathbb{P}[\text{Bloom filter outputs Yes on } y] \leq \left( \frac{kn}{m} \right)^k \) where \( n = |S| \) and \( m \) is the size of the Bloom filter.

**Proof.** Recall that we assume elements are mapped to each position in \( B \) with equal probability. Since in the initialization phase, \( kn \) bits are set to 1, the probability that a certain bit in \( B \) is set to 1 is \( 1 - \left( 1 - \frac{1}{m} \right)^{kn} \leq \frac{kn}{m} \). Since for each element the Bloom filter checks the values of \( B[h_1(y)], \cdots, B[h_k(y)] \), the probability that all are set to 1 is at most \( \left( \frac{kn}{m} \right)^k \). \( \square \)

So far, we saw how the standard Bloom filter works. Now the question is how machine learning can help in the improvement of Bloom filters (i.e. achieve even smaller space).

Ideally, we would like to train a function \( f \) that given an element \( x \), \( f(x) \) indicates whether \( x \in S \). But then \( f \) is just a machine learning based implementation of set-membership data structure of \( S \). Note that the machine learning based approach cannot solve the set-membership problem exactly.
(unless the set is really simple/compressible), and it may have both non-zero false positives and false negatives. So if we only rely on the machine learned oracle we might have to give up on the zero false negative property of Bloom filters, which would be a bummer.

Now the question is how to preserve the zero false negative rate of Bloom filters in machine learned oracles for set membership.

## 2 Machine-Learned Bloom Filter

The result of this section is due to Kraska et al. [KBC+18] and is shown in Figure 2. To achieve a zero false negative rate, in the construction of their data structure (which is called machine-learned Bloom filter), the learned oracle $f$ is augmented with a standard Bloom filter (which is expected to have much smaller size compared to the Bloom filter for the whole set $S$). More precisely, the Bloom filter is constructed on the set of false negatives of $f$ which is denoted as $S^- := \{x \in S | f(x) = 0\}$. Hence, in the initialization phase, for each element $x$ we first compute $f(x)$ and if it returns 0 then we add $x$ to the Bloom filter of $S^-$. It is straightforward to verify that the new construction has zero false negatives as in the standard Bloom filters.

**Lemma 2.** Machine-learned Bloom filter has no false negatives.

Note that compared to the standard Bloom filter whose total space is $\text{space}(\text{BF}(S))$, in the machine-learned Bloom filter the memory space is $\text{space}(f) + \text{space}(\text{BF}(S^-))$ where $\text{space}(f)$ denote the total space to store and evaluate function $f$ and $\text{space}(\text{BF}(X))$ denotes the required space to construct a Bloom filter on set $X$ with the desired accuracy. [KBC+18] show empirically that $\text{space}(f)$ is negligible compared to $\text{space}(\text{BF}(S))$. Hence, if the oracle has small false negative rate $\text{FNR}_f$, then the space for the Bloom filter on $S^-$ that contains $|S| \cdot \text{FNR}_f$ elements is significantly smaller than $S$. Moreover, note that it is always the case that $|S^-| \leq |S|$.

![Figure 2: This picture show the structure of the machine-learned Bloom filter. Oracle $f$ is a machine-earned oracle for set-membership task of $S$.](image)

Next, we bound the false positive rate of the machine-learned Bloom filter. We remark that unlike the guarantee for the standard Bloom filter which bounds the false-positive probability for each element $y \notin S$, here we assume some distribution $\mathcal{D}$ over $\mathcal{U}$ and bound the false positive rate according to $\mathcal{D}$. Specifically, the false negative and false positive rates of $f$ are defined as

$$\text{FNR}_f = \sum_{x \in S^-} \mathbb{P}(f(x) = 0),$$

$$\text{FPR}_f = \sum_{x \in S^c} \mathbb{P}(f(x) = 1).$$
\[ \text{FNR}_f = \mathbb{P}_{x \sim \mathcal{D}}[f(x) = 0| x \in S], \text{ and } \text{FPR}_f = \mathbb{P}_{x \sim \mathcal{D}}[f(x) = 1| x \notin S]. \] See [Mitz18] for further discussion of this point.

**Lemma 3.** Let \( \text{FPR}_\text{BF} \) be the false positive rate of \( \text{BF}(S^-) \). The false positive rate of the machine-learned Bloom filter is \( \text{FPR}_f + (1 - \text{FPR}_f) \cdot \text{FPR}_\text{BF} \).

**Proof.** Let \( \text{MBF} \) denote the machine-learned Bloom filter. Then,

\[
\text{FPR}_{\text{MBF}} = \mathbb{P}_{y \sim \mathcal{D}}[\text{MBF returns Yes on } y| y \notin S]
= \mathbb{P}_{y \sim \mathcal{D}}[f(y) = 1| y \notin S] + \mathbb{P}_{y \sim \mathcal{D}}[f(y) = 0| y \notin S] \cdot \mathbb{P}[	ext{BF}(S^-) \text{ returns Yes}| y \notin S^-]
= \text{FPR}_f + (1 - \text{FPR}_f) \cdot \text{FPR}_\text{BF}
\]

So, \( \text{FPR}_{\text{MBF}} \) is higher than \( \text{FPR}_f \), but \( \text{MBF} \) uses less space than \( \text{BF} \) over \( S \), which means that we get a better error-space tradeoff. This is exemplified by an empirical evaluation of [KBC+18], which shows significant improvement of memory consumption of machine-learned Bloom filters compared to Bloom filters on a data set of malicious URLs (Figure 3). For details on how \( f \) is implemented and trained, see the original paper.

![Figure 3: The comparison of the space-accuracy trade-offs the standard and machine-learned Bloom filters. Here \( W \) and \( E \) denote parameters of the learned function \( f \).](image)

We emphasize that while machine-learned Bloom filter provides better space-accuracy trade-off, 1) it requires training which can be time consuming and expensive, 2) it could be slower in the evaluation phase (unless GPUs are used) and 3) it provides a false positive guarantee over distribution \( \mathcal{D} \) rather than “for each” \( x \).

**References**
