Jianneng Li, Zhiyuan Lin, Ashkon Soroudi
Department of Computer Science
University of California, Berkeley
{jiannengli, lzy94, ashkon77}@berkeley.edu

Improving Ensemble of Trees in MLlib

Background and Context

- Decision tree and random forest are popular machine learning algorithms for both classification and regression
- A decision tree grows by making splits using the <feature, split value> pair that minimizes some impurity function
- Random forest trains an ensemble of decision trees with a few modifications
  - Bagging (for each tree, use subset of data by sampling with replacement)
  - Compare random subset rather than all features when splitting
- We analyze the implementation of decision tree and random forest in MLlib, a machine learning library built on top of Apache Spark, and make algorithmic as well as parallelization improvements

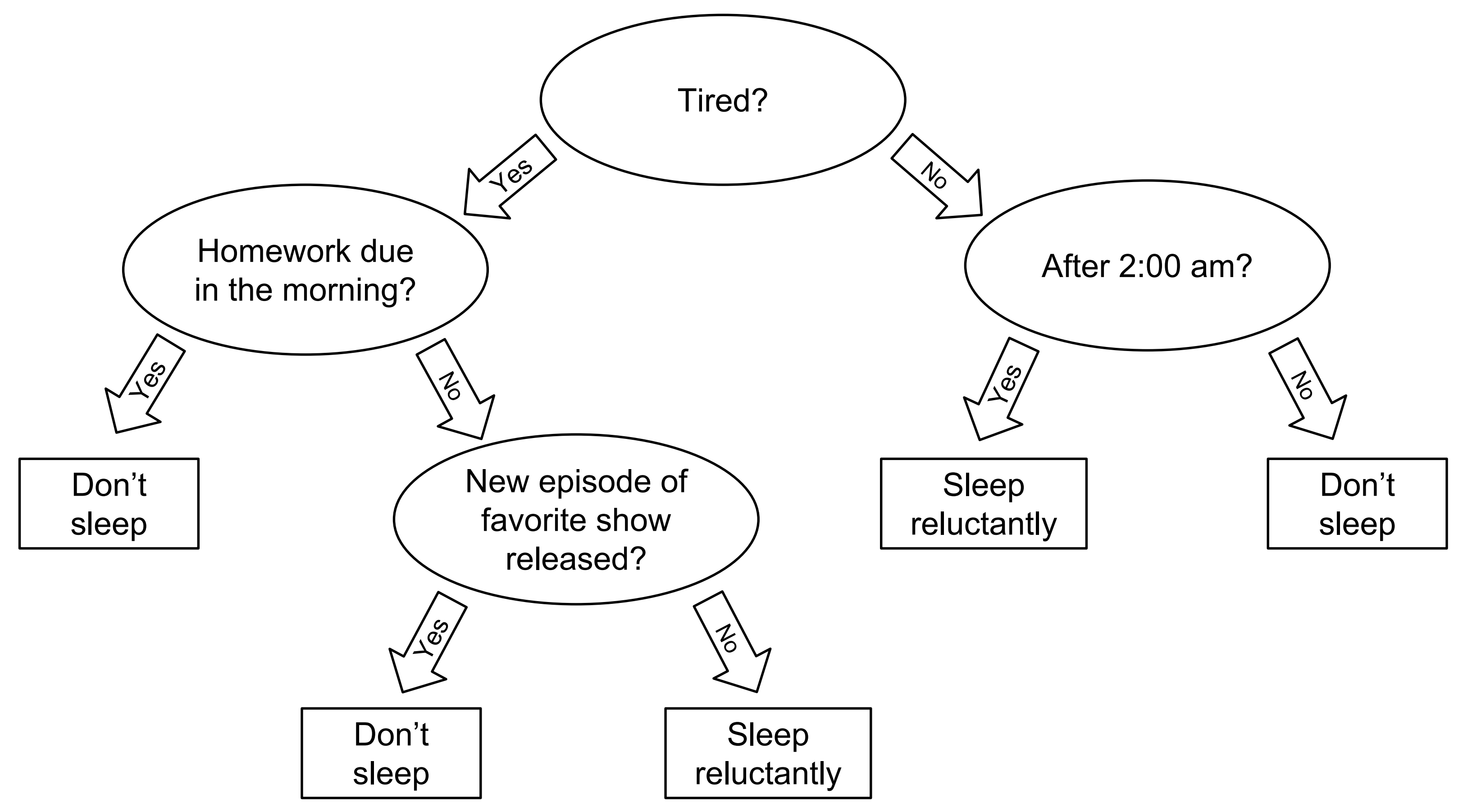

Figure 1: An example of a decision tree. Ever contemplated whether you should sleep?

Contribution #2: Implementing Extra-Trees

- Motivation
  - Sampling and sorting feature values before training is expensive
  - Both decision tree and random forest are prone to overfitting
- Comparison of extra-trees to random forest
  - Differences
    - No bagging
    - When selecting a split, choose a single random candidate per feature instead of multiple candidates
  - Advantages
    - Does not require calculation of feature split bins
    - Takes less memory and time to split a node
    - Less prone to overfitting
  - Disadvantages
    - Shallower trees may perform worse due to more randomization
    - Deeper trees may have more nodes, leading to longer training time

Contribution #1: Parallelizing Calculation of Feature Split Bins

- Motivation
  - Before training, MLlib's decision tree samples input data, and puts sampled values of each feature into bins; the boundaries between these bins are used as split candidates during training
  - However, the current implementation in MLlib performs binning on a single thread, leading to long initialization times for datasets with large number of features
- Implementation
  - Convert the data structure that will hold the binning information into a resilient distributed dataset (RDD), Spark's abstraction on a collection of objects partitioned across a set of machines
  - Perform the calculation of feature split bins in parallel using RDD operations
  - Aggregate results to rebuild the data structure in one place

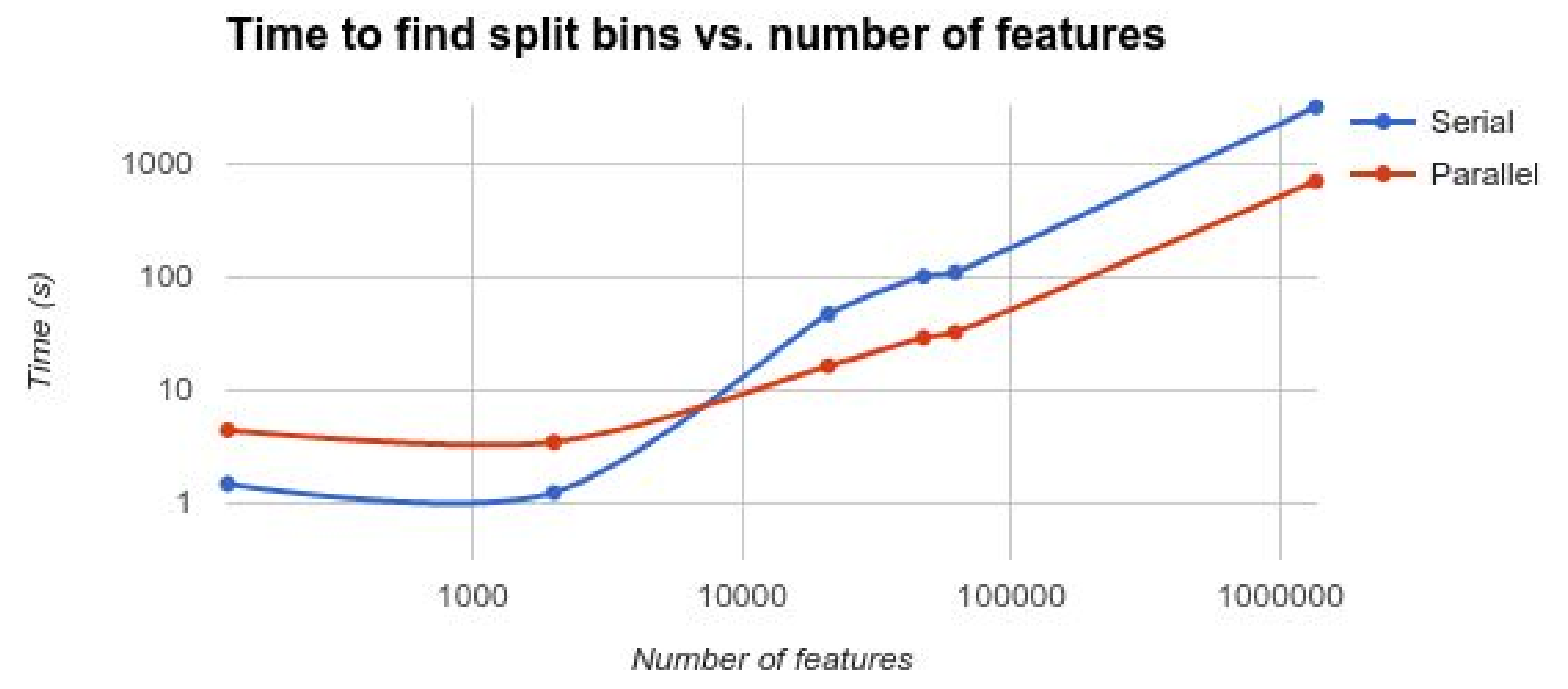

Figure 2: Performance comparison between the serial and parallel implementations of finding feature split bins. The parallel version ran on 4 machines with 2 cores each. For fewer than 10,000 features, the parallel implementation performs worse, because of overhead in parallelizing the computation. Between 10,000 and 100,000 features, the parallel implementation finishes approximately 3 times as fast. For more than 1,000,000 features, the parallel implementation is more than 4 times faster than the serial implementation.

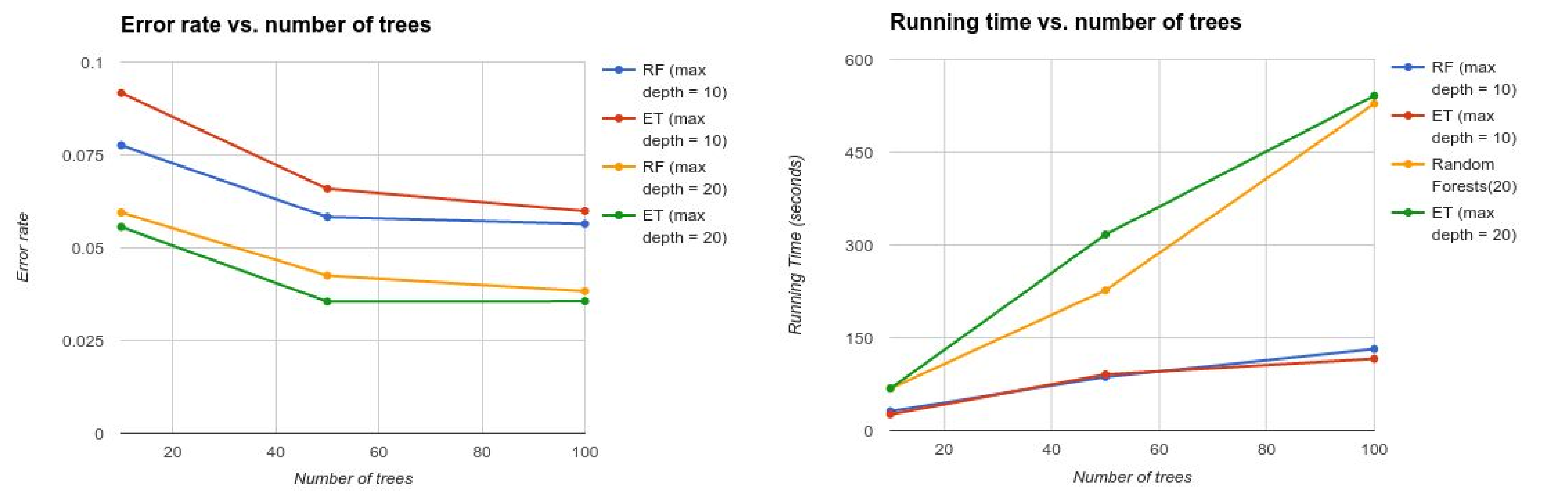

Figure 3: Performance comparison between random forest (RF) and extra-trees (ET) on the MNIST digits dataset (784 features). With max depth of 10, ET is less accurate than RF, likely due to its randomization. The benefit of randomization shows when the max depth is 20, where ET outperforms RF. Both algorithms take similar times to train on max depth of 10, because the number of tree nodes is small. ET takes longer to train on max depth of 20, since on average it can have up to more than twice the number of tree nodes than RF.