

# Decision Forests at Google

Richard Stotz

Fast Machine Learning for Science, 2025

# Google Decision Forests



Richard  
**Stotz**






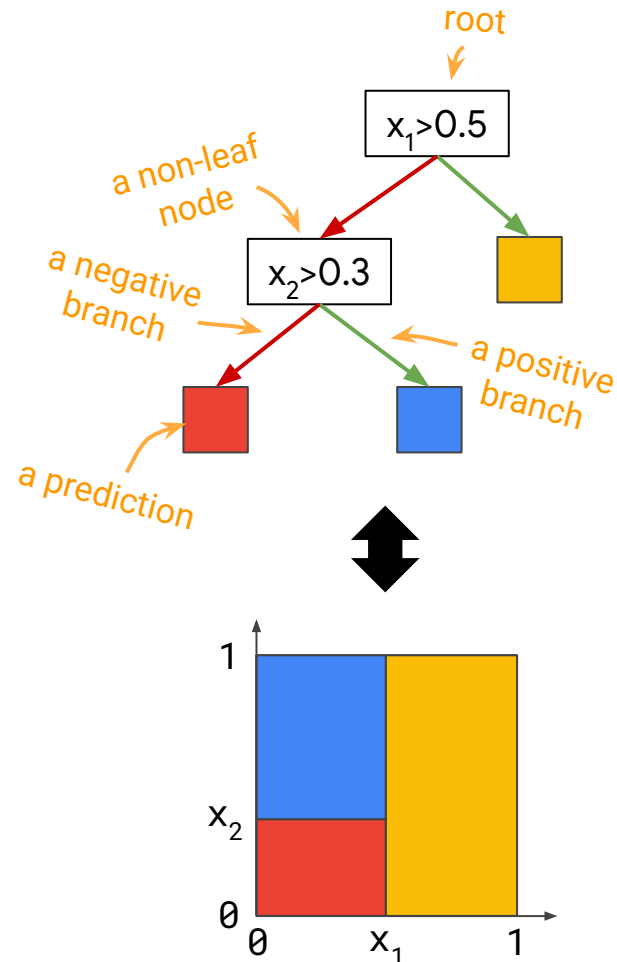
Mathieu  
**Guillame-Bert**

# Agenda

1. What are Decision Forests
2. Decision Forests in Practice
3. Fast inference for Decision Forests

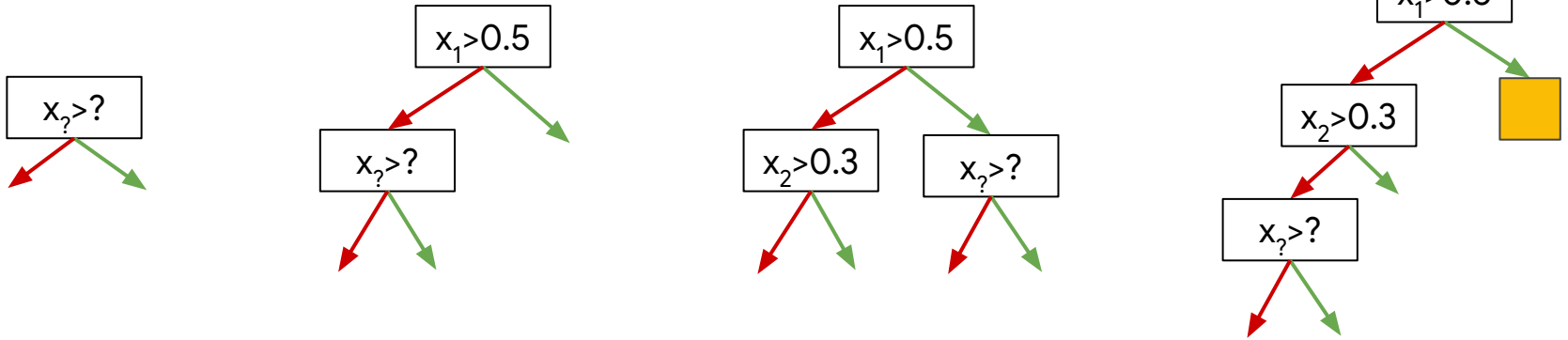
# What is a decision tree (DT)?

- A DT is a **recursive partitioning** of the feature space represented as a tree.
- Each non-leaf node contains a **split** (or condition), e.g.,  $x_1 > 0.5$
- Each leaf node contains a prediction, e.g.,   
- **Often, conditions are binary** and on a single attribute (**axis aligned**).
- **Individual decision trees** are no longer state of the art.



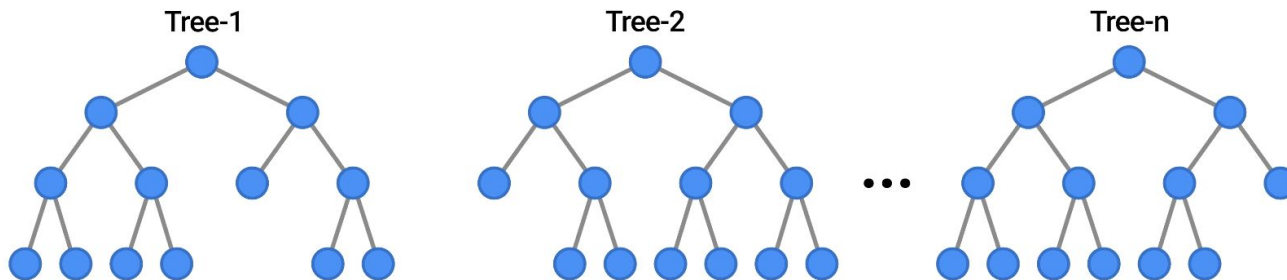
# Decision Tree Training

- Often, trees are trained with a **greedy divide and conquer** algorithm.
- **Main challenge:** How to find the best split efficiently?
- Famous learning algorithms: CART, ID3.



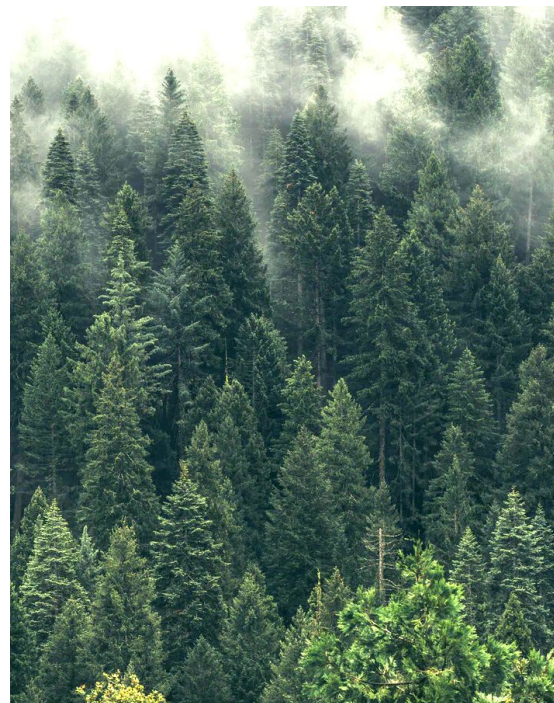
# What are decision forests?

- A **decision forest** (DF) is a collection of decision trees
- Often, the output of a DF is the **sum of the outputs** of its DTs.
- Popular Decision Forests:
  - **Gradient Boosted Trees** (Friedman, 1999)
  - **Random Forests** (Breiman, 2001)
  - AdaBoost (Freund & Schapire, 1995)



# What makes Decision Forests great?

- **State-of-the-art performance** on tabular data
  - [McElfresh et al. NeurIPS'23](#)
- Strong **out-of-the-box** performance
  - Robust to outliers
  - Handle heterogeneous features
- **Fast training**
- **Very fast and flexible inference**
  - $\sim 1 \mu\text{s}$  per example on CPU
- **Explainability and interpretability**



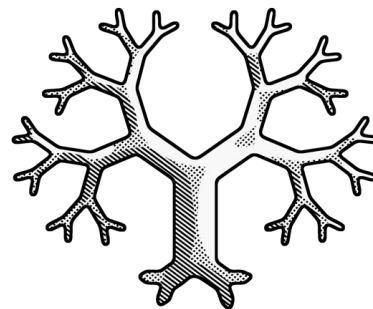
# Decision Forests in Practice

# Decision Forests libraries

- Classic Implementations
  - **R Random Forest (2002)**
  - **Scikit-learn (2007)**
  
- Modern Boosted Trees and Random Forests
  - **XGBoost (2014)**
  - **LightGBM (2016; Microsoft)**
  - **CatBoost (2017; Yandex)**
  - **Yggdrasil Decision Forests (2018, open source in 2021; Google)**

# Yggdrasil Decision Forests (YDF)

- A Python & C++ library for **training**, **servicing**, and **interpreting** decision forests.
- Key features
  - **Simple and safe API** for training, evaluation, servicing and interpretation.
  - Original **Feature Semantics and Algorithms**
  - **Fast model inference** on CPU.
  - **Integration** with other ML libraries.



**Yggdrasil**  
Decision Forests

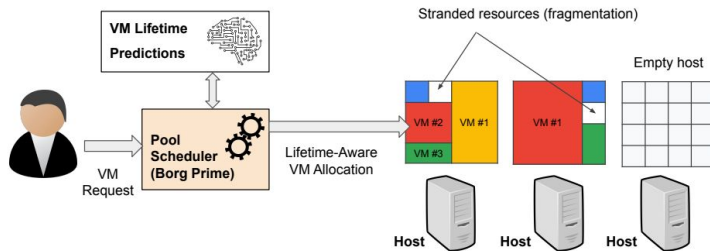
<https://ydf.readthedocs.io>

# Decision Forests at Google

- Wide adoption across the company
- Selected by AutoML ~1/4 of the time.

## Recent (public) usage of Decision Forest models

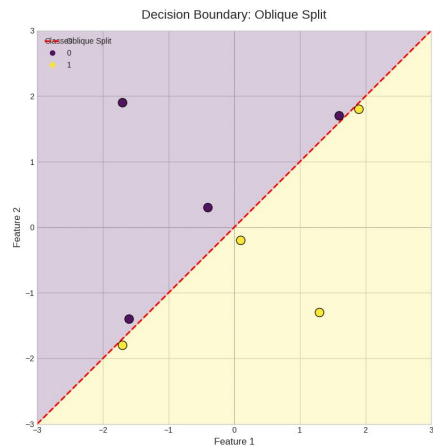
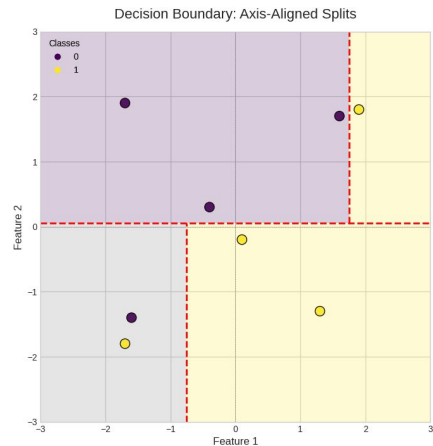
- Advertiser Understanding <https://arxiv.org/pdf/2504.18785>
- Speculative Testing <https://doi.ieeecomputersociety.org/10.1109/ICST62969.2025.10988976>
- Lifetime-aware VM Allocation <https://arxiv.org/abs/2412.09840>
- Efficient On-Chip Memory Allocation <https://dl.acm.org/doi/abs/10.1145/3567955.3567961>



Source: <https://arxiv.org/abs/2412.09840>

# Modern training algorithms

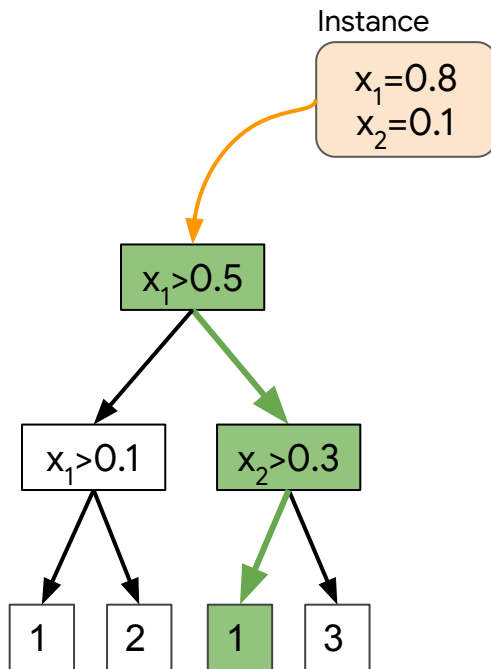
- **Sparse Oblique splits** ([Tomita et al, 2015](#))
  - Random sparse linear feature combinations
  - **Consistently outperforms axis-aligned splits**
- **Embedding consumption**
  - Baseline: Embedding dimensions as **individual features**
  - Better: Exploit **embedding geometry**: (approximate) Nearest neighbors, Embedding distances
  - YDF: **Vector Sequences**: Consume arbitrary-length sequences of embeddings (e.g. LLM internal states)
- Many other ideas



# Decision Forest Inference

# Naive decision tree inference

- **Route** the instance down the tree to the active leaf.
- Evaluate conditions **sequentially**
- Theoretical time complexity of  $\log N$  with  $N$  the number of nodes. **Great!**
- In practice, this is **slow**. **Why?**



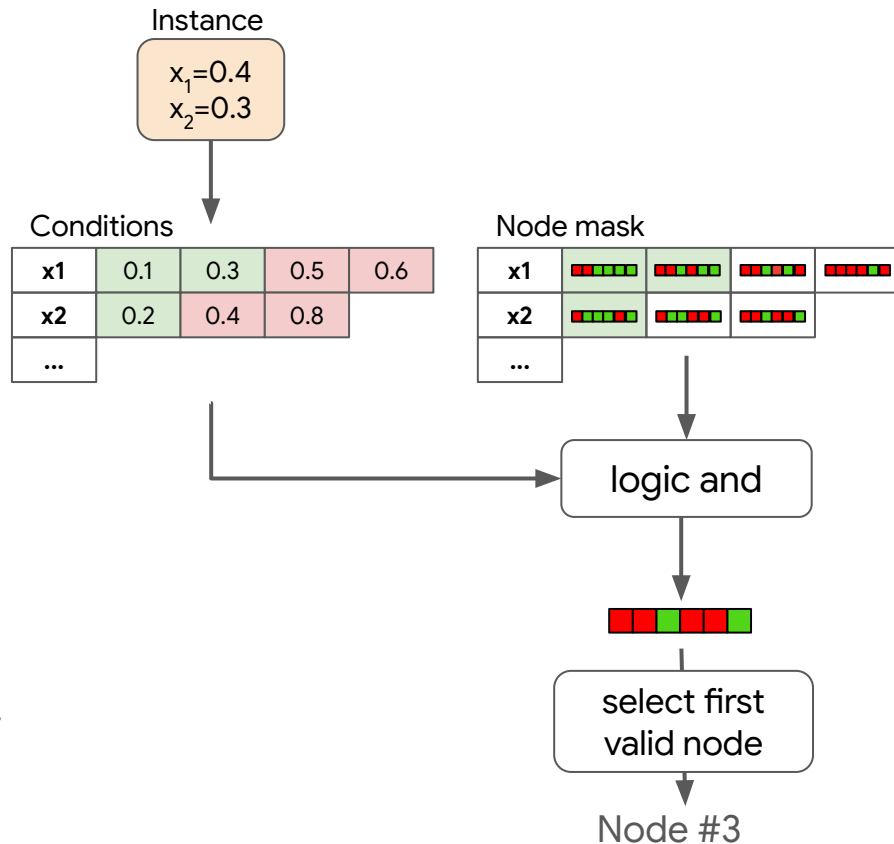
# Why is naive decision tree inference slow?

- Modern CPU speed (approximate)
  - **0.05ns**: Run an instruction
  - **50 bytes / ns**: RAM bandwidth.
  - **100ns**: Get a random byte from RAM.  
= time to run **2000** instructions.
- Modern CPU optimizations (informal)
  - When loading a piece of data from RAM, load the data that follows.
  - While waiting for a RAM fetch, continue executing the following instructions.
- Efficient algorithms
  - Access data **sequentially**.
  - **Limit branches** (i.e., "if"s)
- The naive decision tree routing algorithm spends the majority of its time **waiting for RAM**.
- Alternative approach: **Evaluate all the conditions**, and then find the active leaf with bitmap.

# Quick Scorer approach

[Lucchese et al. 2017](#)

- Evaluate all the conditions i.e. process the whole model.
- Find the active leaf using bitmap masking
- Extra optimizations
  - Don't evaluate all the conditions.
  - Encode masks efficiently.
- **Very efficient on x86 architectures with wide SIMD registers**
- Naive routing
  - Process  $100 \times 6 = 600$  nodes per call.
  - $3.2 \mu\text{s}$  / example / cpu core
- QuickScorer
  - Process  $2^6 \times 100 = 6400$  nodes per call.
  - $0.61 \mu\text{s}$  / example / cpu core



# Pure C++ export

```
inline Label Predict(const Instance& instance) {  
    float accumulator {-1.1631};  
  
    const Node* root = nodes;  
    const Node* node;  
    const char* raw_instance = (const char*)&instance;  
    uint8_t eval;  
    for (uint8_t tree_idx = 0; tree_idx != kNumTrees; tree_idx++) {  
        node = root;  
        while(node->pos) {  
            if (condition_types[node->cond.feats] == 0) {  
                int32_t numerical_feature;  
                std::memcpy(&numerical_feature, raw_instance + node->cond.feats * sizeof(int32_t), sizeof(int32_t));  
                eval = numerical_feature >= node->cond.thr;  
            } else if (condition_types[node->cond.feats] == 1) {  
                uint32_t categorical_feature;  
                std::memcpy(&categorical_feature, raw_instance + node->cond.feats * sizeof(uint32_t), sizeof(uint32_t));  
                eval = categorical_bank[categorical_feature + node->cond.cat];  
            } else {  
                assert(false);  
            }  
            node += (node->pos & -eval) + 1;  
        }  
        accumulator += node->leaf.val;  
        root += root_deltas[tree_idx];  
    }  
    return static_cast<Label>(accumulator >= 0);  
}
```

Predict in 30 lines

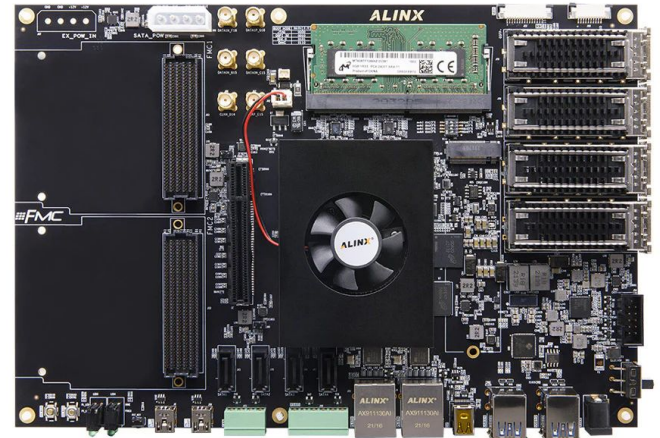
Efficient encoding

Strong tabular model in 1-5 KB

# FPGA Synthesis with Conifer

- Decision Tree to FPGA
- Extremely low latency (100ns)
- Supports a wide range of backends

<https://github.com/thesps/conifer>



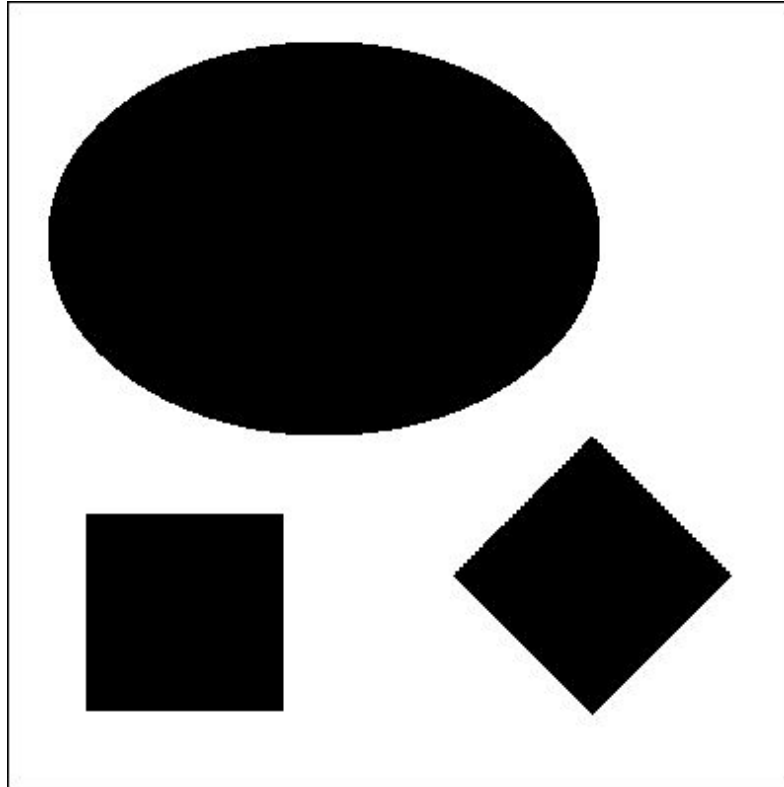
# Conclusion

- Decision Forests shine due to their flexibility
- Modern Training algorithms offer strong performance benefits
- FPGA and GPU story actively developing

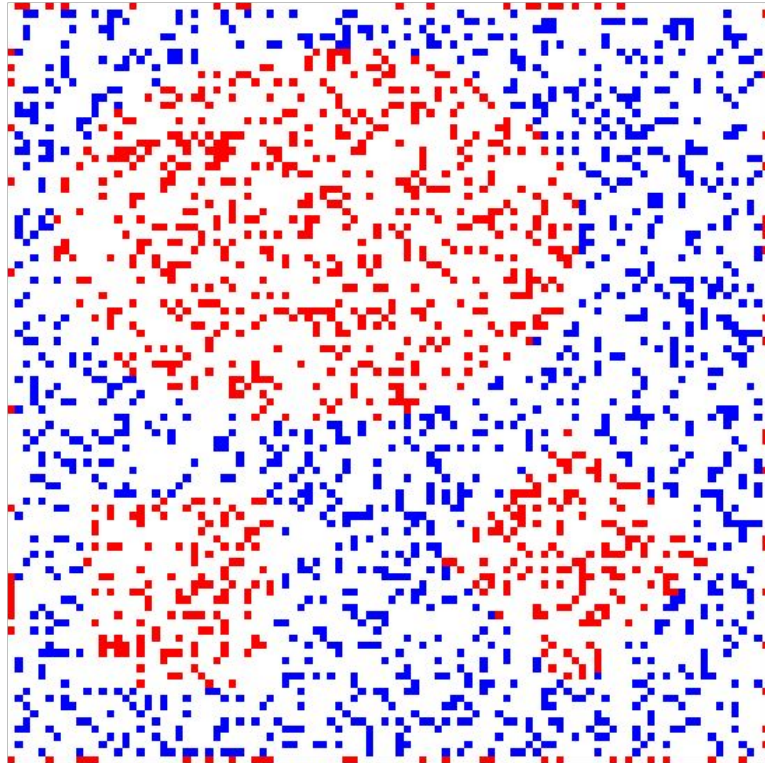


Thank you for your attention

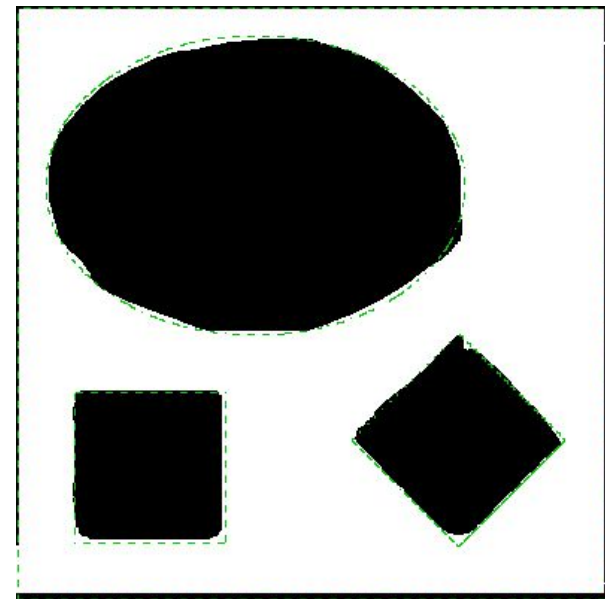
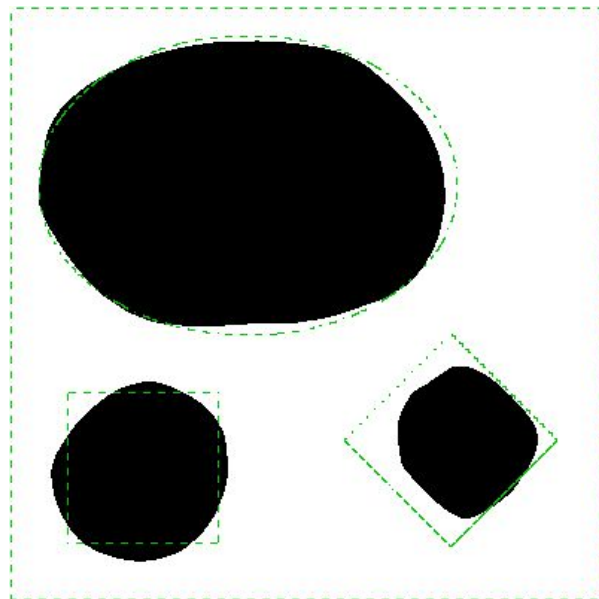
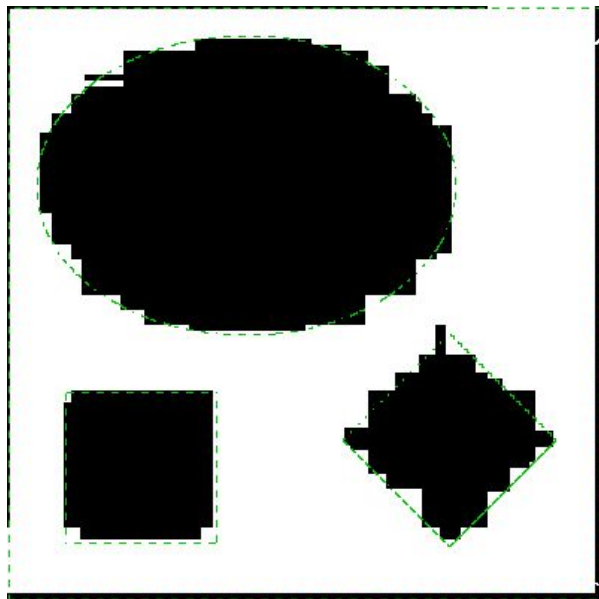
# Identify the Algorithm



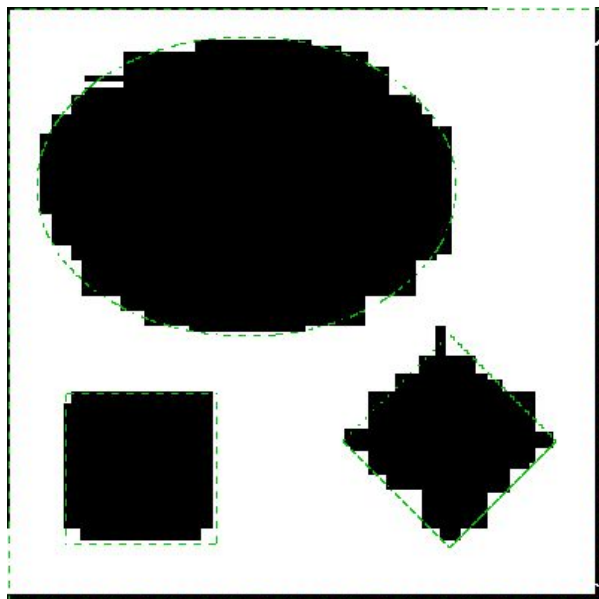
# Identify the Algorithm



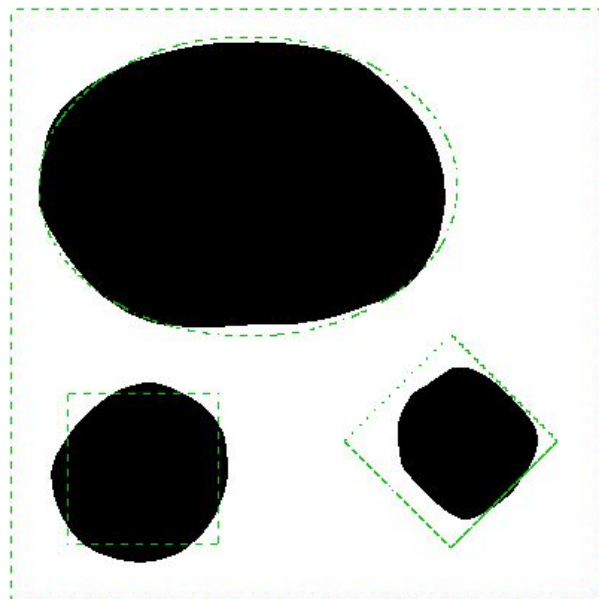
# Identify the Algorithm



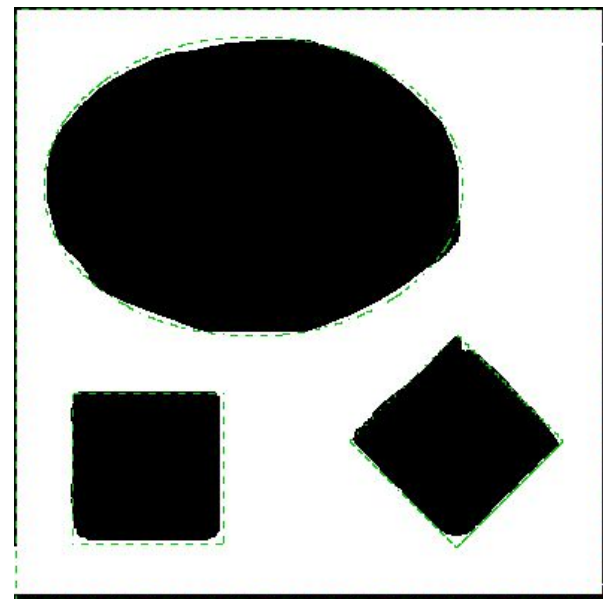
# Identify the Algorithm



Decision Tree  
Blocky, axis-aligned



MLP (8 x 200)  
Smooth, non-linear



Oblique Decision Forest  
Diagonal Splits

# Yggdrasil Decision Forests (YDF)

```
!pip install ydf

import ydf
import pandas as pd

train_ds = pd.read_csv("adult_train.csv")
test_ds = pd.read_csv("adult_test.csv")

# Train a model
model = ydf.GradientBoostedTreesLearner(label="income").train(train_ds)

# Look at the model (input features, training logs, structure, etc.)
model.describe()


# Evaluate model
model.evaluate(test_ds)

# Make predictions
model.predict(test_ds)

# Study model.
model.analyze(test_ds)
model.analyze_prediction(test_ds.iloc[:1])
model.benchmark(test_ds)

# Export model
model.save("my_model")
model.to_tensorflow_saved_model("tf_model")
model.to_cpp()
```

Automatic model  
configuration and  
feature ingestion



Rich HTML reports



# What about GPU?

- A modern GPU (informally)
  - Thousands of slow cores running in parallel.
  - Globally, an order of magnitude more RAM bandwidth.
- Naive decision tree branching algorithm: ~30x speed-up over a single core.

