A case study: implementing ID3 decision trees to be as fast as possible

Ryan R. Curtin
ryan@ratml.org

December 9, 2017

You find yourself in a dimly lit room with a DESK. On the DESK is a PAPER describing an algorithm and a NOTE from your boss indicating the PAPER must be implemented. The IMPLEMENTATION must be fast or you may lose your JOB. Exits to the NORTH.

? _

You find yourself in a dimly lit room with a DESK. On the DESK is a PAPER describing an algorithm and a NOTE from your boss indicating the PAPER must be implemented. The IMPLEMENTATION must be fast or you may lose your JOB. Exits to the NORTH.

?

You find yourself in a dimly lit room with a DESK. On the DESK is a PAPER describing an algorithm and a NOTE from your boss indicating the PAPER must be implemented. The IMPLEMENTATION must be fast or you may lose your JOB. Exits to the NORTH.

? why is this my life

I don't understand WHY.

? _

You find yourself in a dimly lit room with a DESK. On the DESK is a PAPER describing an algorithm and a NOTE from your boss indicating the PAPER must be implemented. The IMPLEMENTATION must be fast or you may lose your JOB. Exits to the NORTH.

? why is this my life

I don't understand WHY.

?

You find yourself in a dimly lit room with a DESK. On the DESK is a PAPER describing an algorithm and a NOTE from your boss indicating the PAPER must be implemented. The IMPLEMENTATION must be fast or you may lose your JOB. Exits to the NORTH.

? why is this my life

I don't understand WHY.

? get paper

You pick up the PAPER off the DESK and look at it...

Machine Learning 1: 81–106, 1986

Induction of Decision Trees

J.R. QUINLAN (munnari!nswitgould.oz!quinlan@seismo.css.gov)
Centre for Advanced Computing Sciences, New South Wales Institute of Technology, Sydney 2007, Australia

(Received August 1, 1985)

Key words: classification, induction, decision trees, information theory, knowledge acquisition, expert systems

Abstract. The technology for building knowledge-based systems by inductive inference from examples has been demonstrated successfully in several practical applications. This paper summarizes an approach to synthesizing decision trees that has been used in a variety of systems, and it describes one such system, ID3, in detail. Results from recent studies show ways in which the methodology can be modified to deal with information that is noisy and/or incomplete. A reported shortcoming of the basic algorithm is discussed and two means of overcoming it are compared. The paper concludes with illustrations of current research directions.

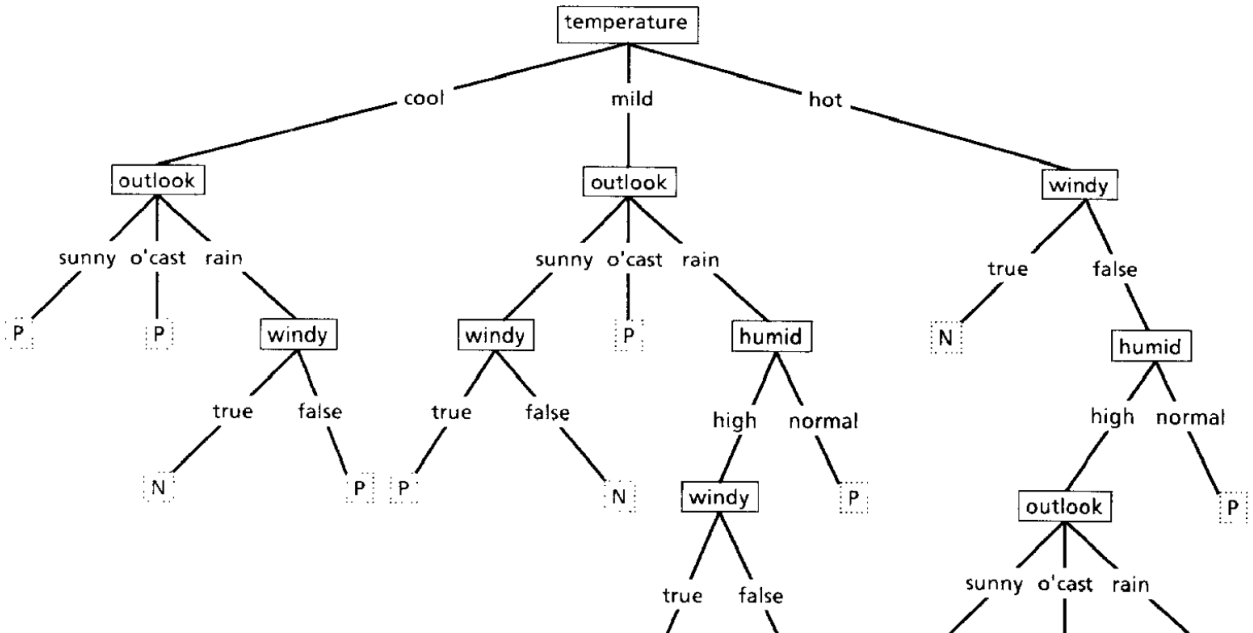
1. Introduction

Since artificial intelligence first achieved recognition as a discipline in the mid 1950's,

4. ID3

One approach to the induction task above would be to generate all possible decision trees that correctly classify the training set and to select the simplest of them. The

[2] The preference for simpler trees, presented here as a commonsense application of Occam's Razor, is also supported by analysis. Pearl (1978b) and Quinlan (1983a) have derived upper bounds on the expected error using different formalisms for generalizing from a set of known cases. For a training set of predetermined size, these bounds increase with the complexity of the induced generalization.



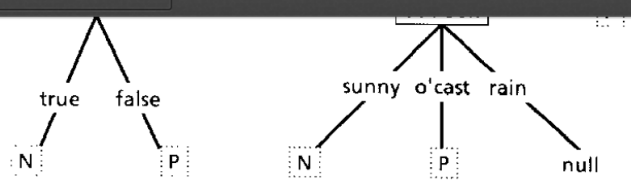


Figure 3. A complex decision tree.

number of such trees is finite but very large, so this approach would only be feasible for small induction tasks. ID3 was designed for the other end of the spectrum, where there are many attributes and the training set contains many objects, but where a reasonably good decision tree is required without much computation. It has generally been found to construct simple decision trees, but the approach it uses cannot guarantee that better trees have not been overlooked.

The basic structure of ID3 is iterative. A subset of the training set called the *window* is chosen at random and a decision tree formed from it; this tree correctly classifies all objects in the window. All other objects in the training set are then classified using the tree. If the tree gives the correct answer for all these objects then it is correct for the entire training set and the process terminates. If not, a selection of the incorrectly classified objects is added to the window and the process continues. In this way, correct decision trees have been found after only a few iterations for training sets of up to thirty thousand objects described in terms of up to 50 attributes. Empirical evidence suggests that a correct decision tree is usually found more quickly by this iterative method than by forming a tree directly from the entire training set. However, O'Keefe (1983) has noted that the iterative framework cannot be guaranteed to converge on a final tree unless the window can grow to include the entire training set. This potential limitation has not yet arisen in practice.

The crux of the problem is how to form a decision tree for an arbitrary collection C of objects. If C is empty or contains only objects of one class, the simplest decision tree is just a leaf labelled with the class. Otherwise, let T be any test on an object with possible outcomes $O_1, O_2, \ldots O_w$. Each object in C will give one of these outcomes for T, so T produces a partition $\{C_1, C_2, \ldots C_w\}$ of C with C_i containing those ob-

possible outcomes $O_1, O_2, \ldots O_w$. Each object in C will give one of these outcomes for T, so T produces a partition $\{C_1, C_2, \ldots C_w\}$ of C with C_i containing those ob-

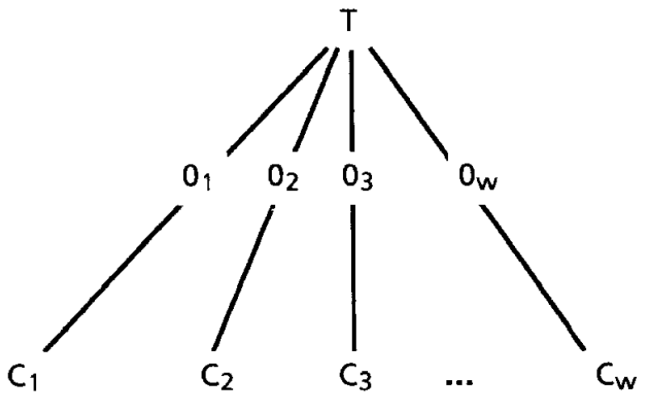


Figure 4. A tree structuring of the objects in C.

jects having outcome O_i. This is represented graphically by the tree form of Figure 4. If each subset C_i in this figure could be replaced by a decision tree for C_i, the result would be a decision tree for all of C. Moreover, so long as two or more C_i's are non-empty, each C_i is smaller than C. In the worst case, this divide-and-conquer strategy will yield single-object subsets that satisfy the one-class requirement for a leaf. Thus, provided that a test can always be found that gives a non-trivial partition of any set of objects, this procedure will always produce a decision tree that correctly classifies each object in C.

The choice of test is crucial if the decision tree is to be simple. For the moment,

will yield single-object subsets that satisfy the one-class requirement for a leaf. Thus, provided that a test can always be found that gives a non-trivial partition of any set of objects, this procedure will always produce a decision tree that correctly classifies each object in C.

The choice of test is crucial if the decision tree is to be simple. For the moment, a test will be restricted to branching on the values of an attribute, so choosing a test comes down to selecting an attribute for the root of the tree. The first induction programs in the ID series used a seat-of-the-pants evaluation function that worked reasonably well. Following a suggestion of Peter Gacs, ID3 adopted an information-based method that depends on two assumptions. Let C contain p objects of class P and n of class N. The assumptions are:

(1) Any correct decision tree for C will classify objects in the same proportion as their representation in C. An arbitrary object will be determined to belong to class P with probability $p/(p+n)$ and to class N with probability $n/(p+n)$.
(2) When a decision tree is used to classify an object, it returns a class. A decision tree can thus be regarded as a source of a message 'P' or 'N', with the expected information needed to generate this message given by

$$I(p, n) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

If attribute A with values $\{A_1, A_2, \ldots A_v\}$ is used for the root of the decision tree, it will partition C into $\{C_1, C_2, \ldots C_v\}$ where C_i contains those objects in C that have value A_i of A. Let C_i contain p_i objects of class P and n_i of class N. The expected

If attribute A with values $\{A_1, A_2, \ldots A_v\}$ is used for the root of the decision tree, it will partition C into $\{C_1, C_2, \ldots C_v\}$ where C_i contains those objects in C that have value A_i of A. Let C_i contain p_i objects of class P and n_i of class N. The expected

information required for the subtree for C_i is $I(p_i, n_i)$. The expected information required for the tree with A as root is then obtained as the weighted average

$$E(A) = \sum_{i=1}^{v} \frac{p_i + n_i}{p + n} I(p_i, n_i)$$

where the weight for the ith branch is the proportion of the objects in C that belong to C_i. The information gained by branching on A is therefore

$$\text{gain}(A) = I(p, n) - E(A)$$

A good rule of thumb would seem to be to choose that attribute to branch on which gains the most information.[3] ID3 examines all candidate attributes and chooses A to maximize gain(A), forms the tree as above, and then uses the same process recursively to form decision trees for the residual subsets $C_1, C_2, \ldots C_v$.

To illustrate the idea, let C be the set of objects in Table 1. Of the 14 objects, 9 are of class P and 5 are of class N, so the information required for classification is

$$I(p, n) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940 \text{ bits}$$

maximize gain(A), forms the tree as above, and then uses the same process recursively to form decision trees for the residual subsets $C_1, C_2, \ldots C_v$.

To illustrate the idea, let C be the set of objects in Table 1. Of the 14 objects, 9 are of class P and 5 are of class N, so the information required for classification is

$$I(p, n) = -\frac{9}{14}\log_2\frac{9}{14} - \frac{5}{14}\log_2\frac{5}{14} = 0.940 \text{ bits}$$

Now consider the outlook attribute with values {sunny, overcast, rain}. Five of the 14 objects in C have the first value (sunny), two of them from class P and three from class N, so

$$p_1 = 2 \quad n_1 = 3 \quad I(p_1, n_1) = 0.971$$

and similarly

$$p_2 = 4 \quad n_2 = 0 \quad I(p_2, n_2) = 0$$
$$p_3 = 3 \quad n_3 = 2 \quad I(p_3, n_3) = 0.971$$

The expected information requirement after testing this attribute is therefore

$$E(\text{outlook}) = \frac{5}{14} I(p_1, n_1) + \frac{4}{14} I(p_2, n_2) + \frac{5}{14} I(p_3, n_3)$$
$$= 0.694 \text{ bits}$$

[3] Since I(p,n) is constant for all attributes, maximizing the gain is equivalent to minimizing E(A), which is the mutual information of the attribute A and the class. Pearl (1978a) contains an excellent account of the rationale of information-based heuristics.

[3] Since I(p,n) is constant for all attributes, maximizing the gain is equivalent to minimizing E(A), which is the mutual information of the attribute A and the class. Pearl (1978a) contains an excellent account of the rationale of information-based heuristics.

The gain of this attribute is then

$$\text{gain(outlook)} = 0.940 - \text{E(outlook)} = 0.246 \text{ bits}$$

Similar analysis gives

$$\text{gain(temperature)} = 0.029 \text{ bits}$$
$$\text{gain(humidity)} = 0.151 \text{ bits}$$
$$\text{gain(windy)} = 0.048 \text{ bits}$$

so the tree-forming method used in ID3 would choose outlook as the attribute for the root of the decision tree. The objects would then be divided into subsets according to their values of the outlook attribute and a decision tree for each subset would be induced in a similar fashion. In fact, Figure 2 shows the actual decision tree generated by ID3 from this training set.

A special case arises if C contains no objects with some particular value A_j of A, giving an empty C_j. ID3 labels such a leaf as 'null' so that it fails to classify any object arriving at that leaf. A better solution would generalize from the set C from which C_j came, and assign this leaf the more frequent class in C.

The worth of ID3's attribute-selecting heuristic can be assessed by the simplicity of the resulting decision trees, or, more to the point, by how well those trees express

induced in a similar fashion. In fact, Figure 2 shows the actual decision tree generated by ID3 from this training set.

A special case arises if C contains no objects with some particular value A_j of A, giving an empty C_j. ID3 labels such a leaf as 'null' so that it fails to classify any object arriving at that leaf. A better solution would generalize from the set C from which C_j came, and assign this leaf the more frequent class in C.

The worth of ID3's attribute-selecting heuristic can be assessed by the simplicity of the resulting decision trees, or, more to the point, by how well those trees express real relationships between class and attributes as demonstrated by the accuracy with which they classify objects other than those in the training set (their *predictive accuracy*). A straightforward method of assessing this predictive accuracy is to use only part of the given set of objects as a training set, and to check the resulting decision tree on the remainder.

Several experiments of this kind have been carried out. In one domain, 1.4 million chess positions described in terms of 49 binary-valued attributes gave rise to 715 distinct objects divided 65%:35% between the classes. This domain is relatively complex since a correct decision tree for all 715 objects contains about 150 nodes. When training sets containing 20% of these 715 objects were chosen at random, they produced decision trees that correctly classified over 84% of the unseen objects. In another version of the same domain, 39 attributes gave 551 distinct objects with a correct decision tree of similar size; training sets of 20% of these 551 objects gave decision trees of almost identical accuracy. In a simpler domain (1,987 objects with a correct decision tree of 48 nodes), randomly-selected training sets containing 20% of the objects gave decision trees that correctly classified 98% of the unseen objects. In all three cases, it is clear that the decision trees reflect useful (as opposed to random) relationships present in the data.

This discussion of ID3 is rounded off by looking at the computational requirements of the procedure. At each non-leaf node of the decision tree, the gain of each untested attribute A must be determined. This gain in turn depends on the values p_i

correct decision tree of similar size, training sets of 20% of these 551 objects gave decision trees of almost identical accuracy. In a simpler domain (1,987 objects with a correct decision tree of 48 nodes), randomly-selected training sets containing 20% of the objects gave decision trees that correctly classified 98% of the unseen objects. In all three cases, it is clear that the decision trees reflect useful (as opposed to random) relationships present in the data.

This discussion of ID3 is rounded off by looking at the computational requirements of the procedure. At each non-leaf node of the decision tree, the gain of each untested attribute A must be determined. This gain in turn depends on the values p_i

and n_i for each value A_i of A, so every object in C must be examined to determine its class and its value of A. Consequently, the computational complexity of the procedure at each such node is $O(|C| \cdot |A|)$, where $|A|$ is the number of attributes above. ID3's total computational requirement per iteration is thus proportional to the product of the size of the training set, the number of attributes and the number of non-leaf nodes in the decision tree. The same relationship appears to extend to the entire induction process, even when several iterations are performed. No exponential growth in time or space has been observed as the dimensions of the induction task increase, so the technique can be applied to large tasks.

5. Noise

So far, the information supplied in the training set has been assumed to be entirely accurate. Sadly, induction tasks based on real-world data are unlikely to find this

correct decision tree of similar size, training sets of 20% of these 551 objects gave decision trees of almost identical accuracy. In a simpler domain (1,987 objects with a correct decision tree of 48 nodes), randomly-selected training sets containing 20% of the objects gave decision trees that correctly classified 98% of the unseen objects. In all three cases, it is clear that the decision trees reflect useful (as opposed to random) relationships present in the data.

This discussion of ID3 is rounded off by looking at the computational requirements of the procedure. At each non-leaf node of the decision tree, the gain of each untested attribute A must be determined. This gain in turn depends on the values p_i

Outline of ID3 algorithm:

1. Create a root decision tree node for the whole dataset.
2. Calculate the information gain for every possible split of every dimension of the dataset.
3. Split the dataset into two subsets along the dimension for which information gain (after splitting) is maximized.
4. Create a new decision tree node for each subset and recurse to step 2.

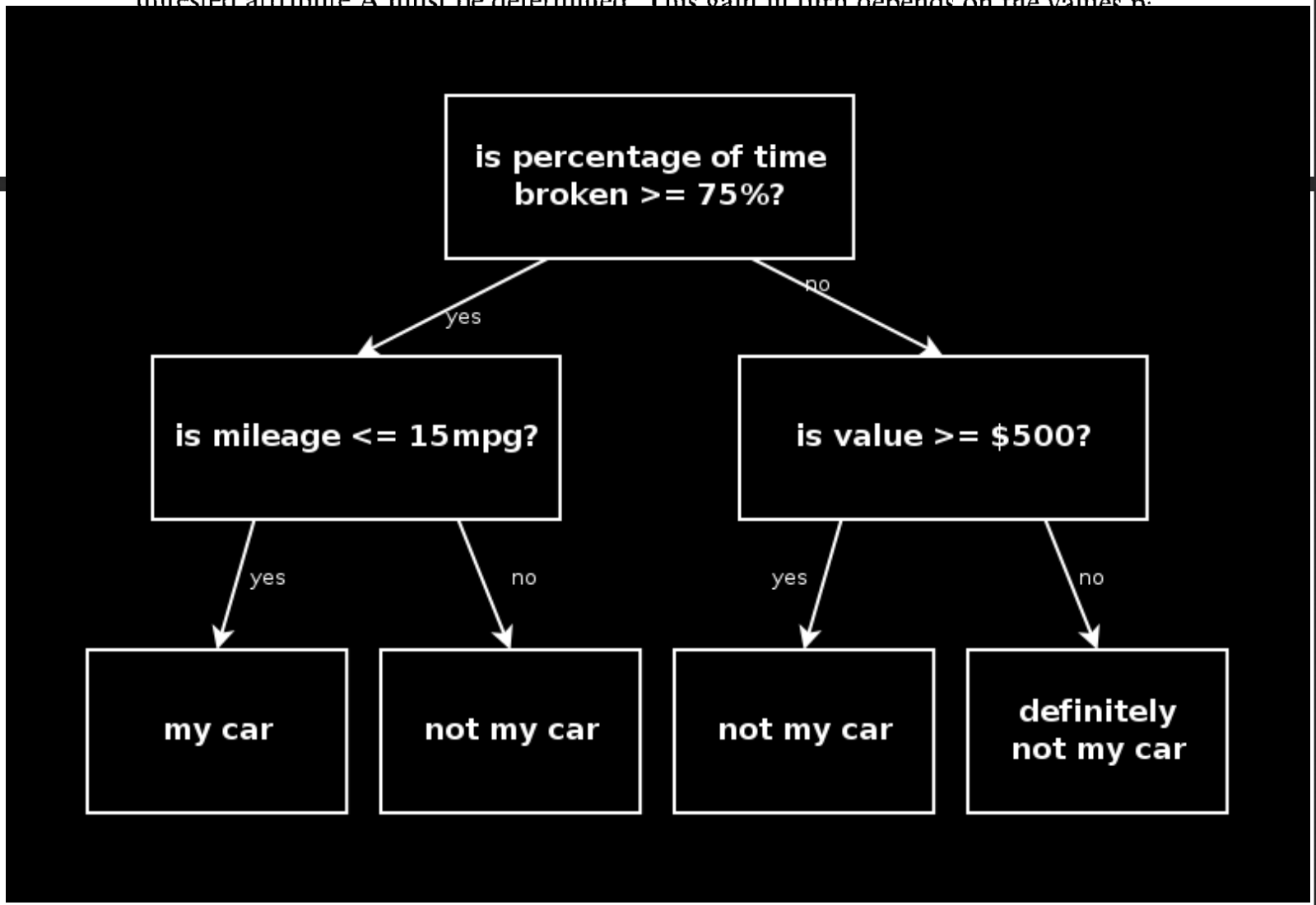
of non-leaf nodes in the decision tree. The same relationship appears to extend to the entire induction process, even when several iterations are performed. No exponential growth in time or space has been observed as the dimensions of the induction task increase, so the technique can be applied to large tasks.

5. Noise

So far, the information supplied in the training set has been assumed to be entirely accurate. Sadly, induction tasks based on real-world data are unlikely to find this

correct decision tree of similar size, training sets of 20% of these 551 objects gave decision trees of almost identical accuracy. In a simpler domain (1,987 objects with a correct decision tree of 48 nodes), randomly-selected training sets containing 20% of the objects gave decision trees that correctly classified 98% of the unseen objects. In all three cases, it is clear that the decision trees reflect useful (as opposed to random) relationships present in the data.

This discussion of ID3 is rounded off by looking at the computational requirements of the procedure. At each non-leaf node of the decision tree, the gain of each untested attribute A must be determined. This gain in turn depends on the values p:



So far, the information supplied in the training set has been assumed to be entirely accurate. Sadly, induction tasks based on real-world data are unlikely to find this

What design choices will you make for your algorithm?

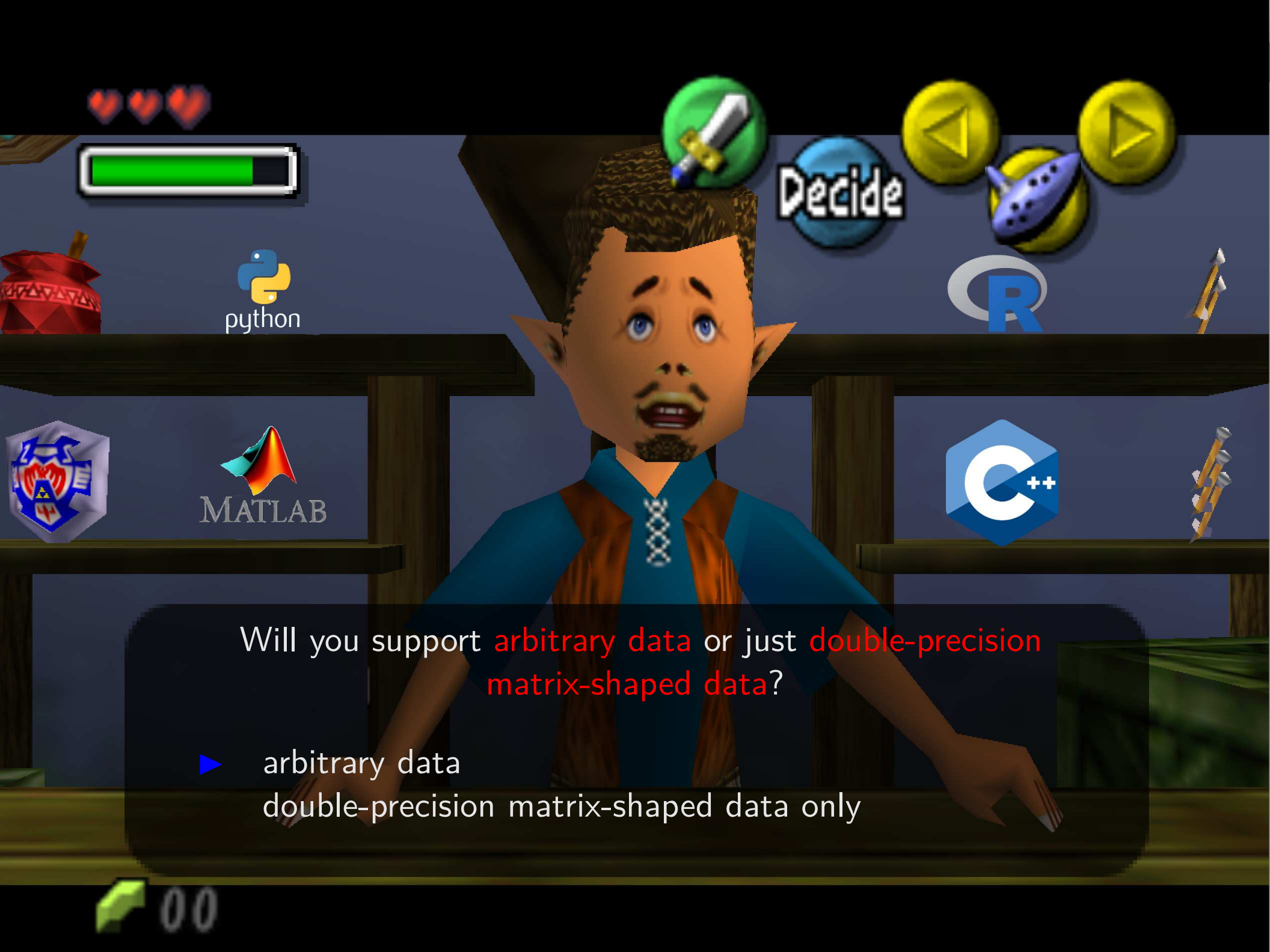

Decide
python
R
MATLAB
C++
What language will you use?
MATLAB
Python
R
C++
00


Decide
python
MATLAB
R
C++
What language will you use?
MATLAB
Python
R
C++
00


Decide
python
R
MATLAB
C++
What language will you use?
MATLAB
Python
R
C++

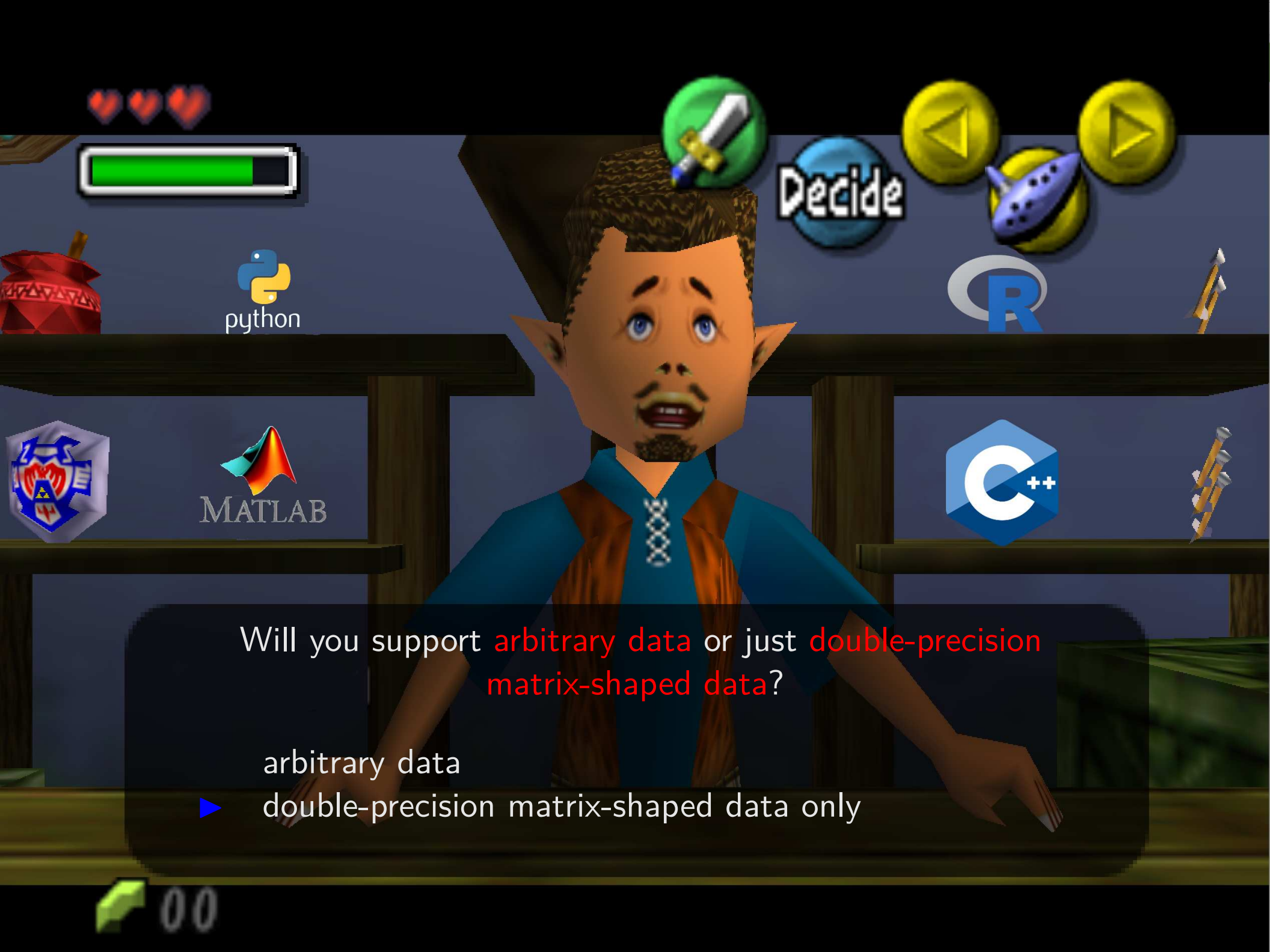

Decide
python
R
MATLAB
C++
What language will you use?
MATLAB
Python
R
C++
00


Decide
python
MATLAB
C++
Okay, you will write the algorithm in C++.



Will you support arbitrary data or just double-precision matrix-shaped data?

- arbitrary data
- double-precision matrix-shaped data only


Decide
python
MATLAB
C++
Will you support arbitrary data or just double-precision matrix-shaped data?
arbitrary data
double-precision matrix-shaped data only

Okay, your implementation will support only double-precision matrix-shaped data.

Do you expect user data to be in reasonable ranges (i.e. no `NaN`, `inf`, or similar)?

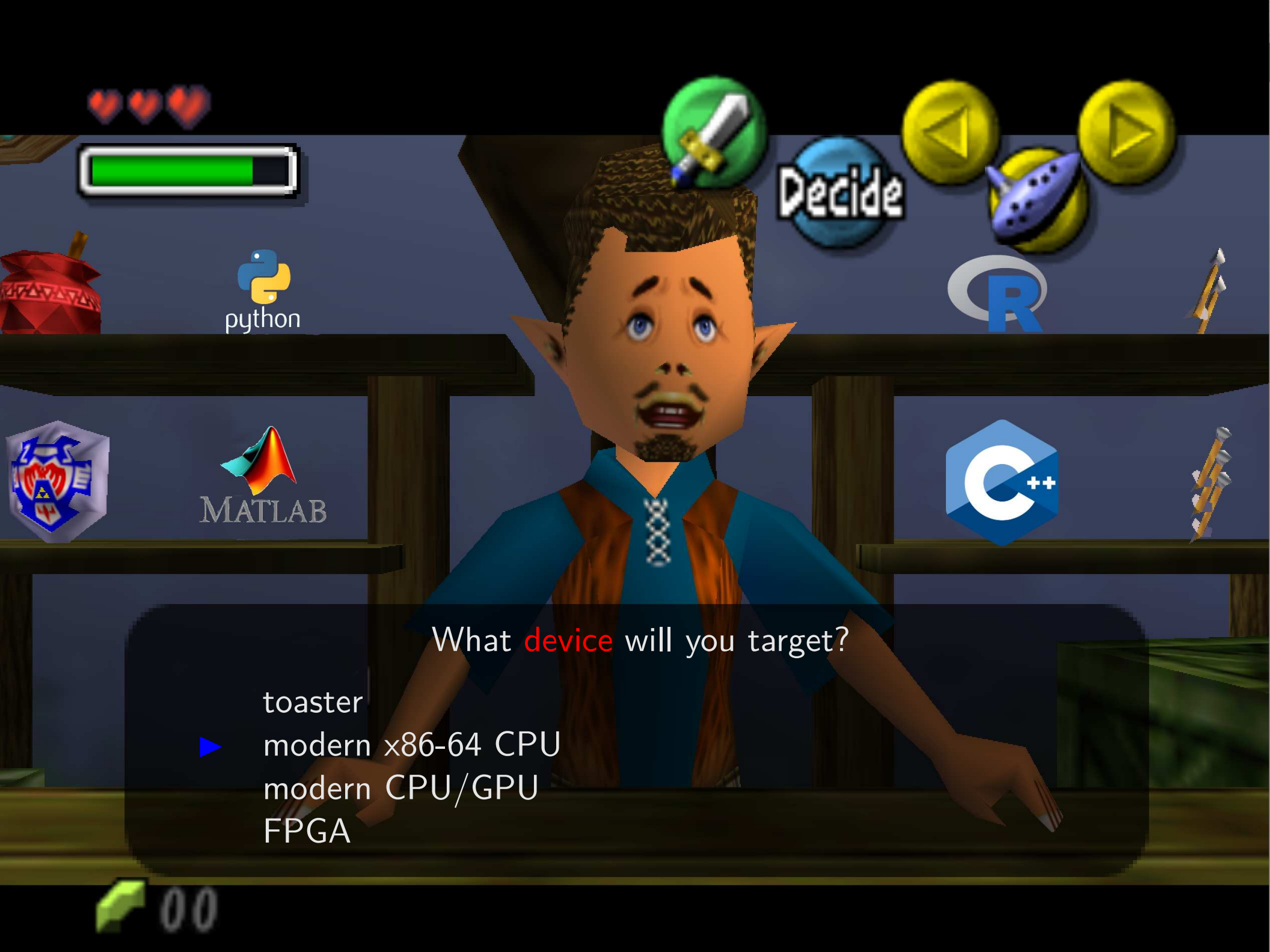
- no
- yes

Do you expect user data to be in reasonable ranges (i.e. no `NaN`, `inf`, or similar)?

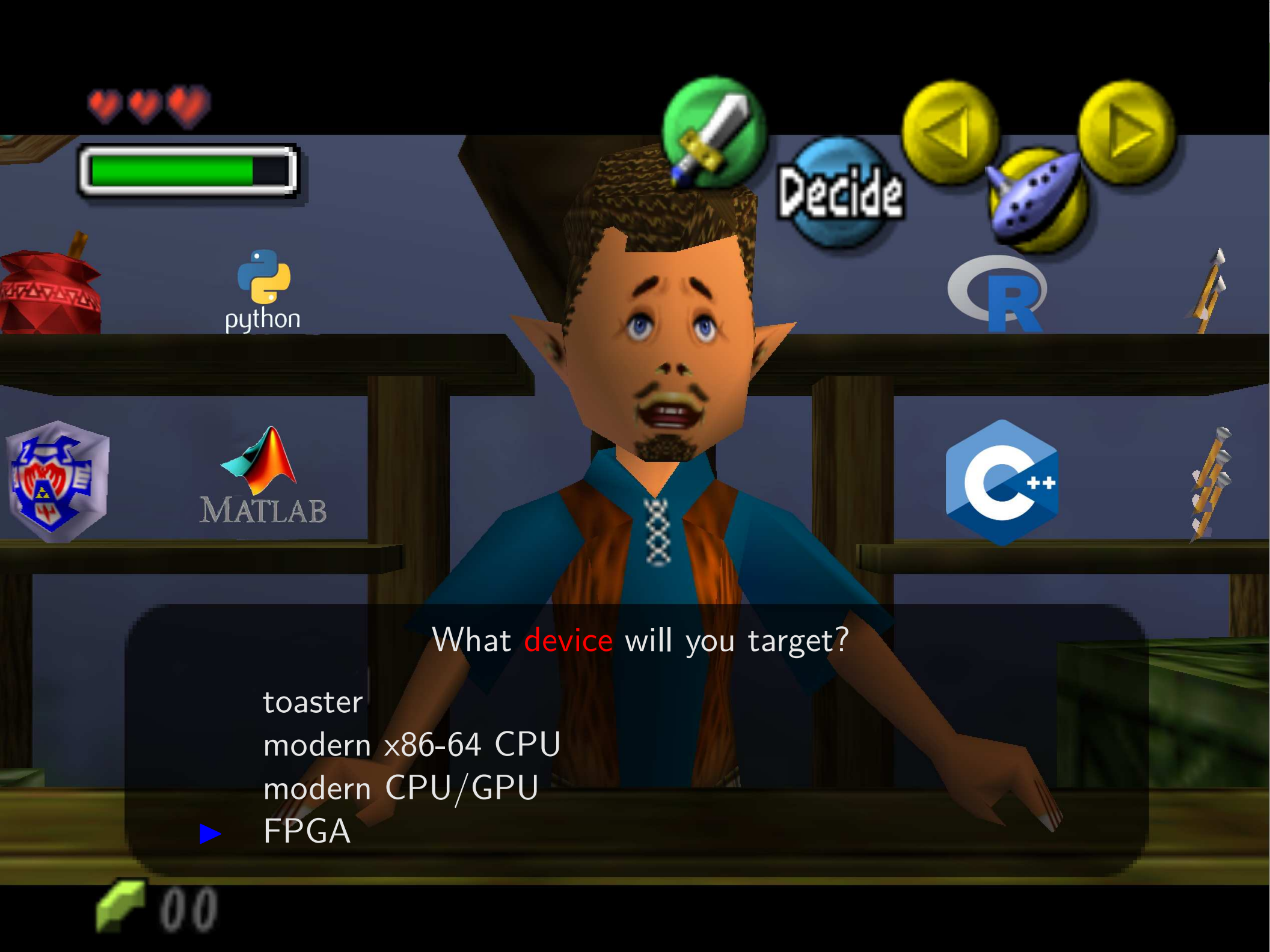
- no
- ▶ yes

Okay, your implementation will assume that users are not passing in extreme-valued or `NaN` data.

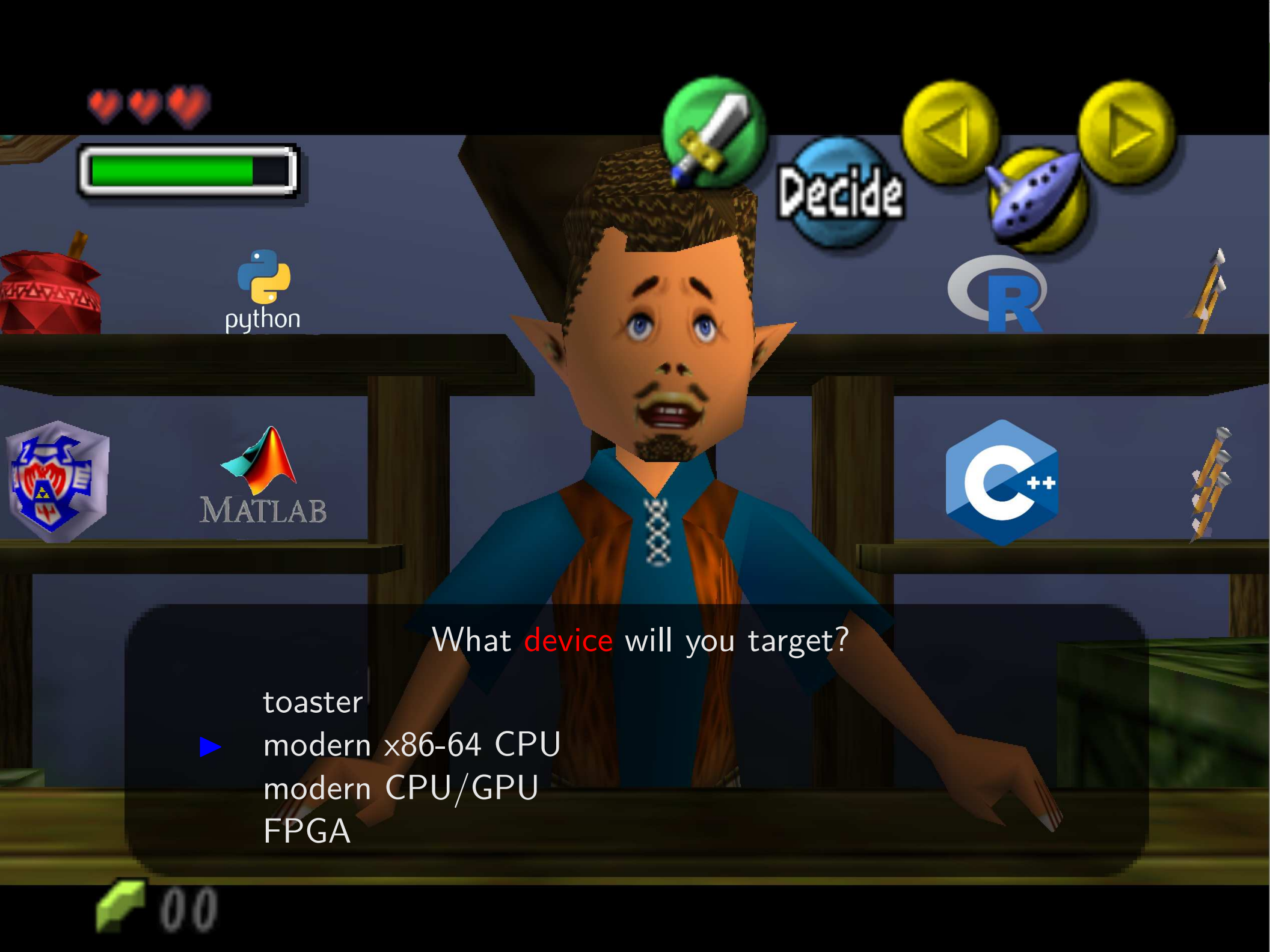

Decide
python
MATLAB
C++
What device will you target?
toaster
modern x86-64 CPU
modern CPU/GPU
FPGA

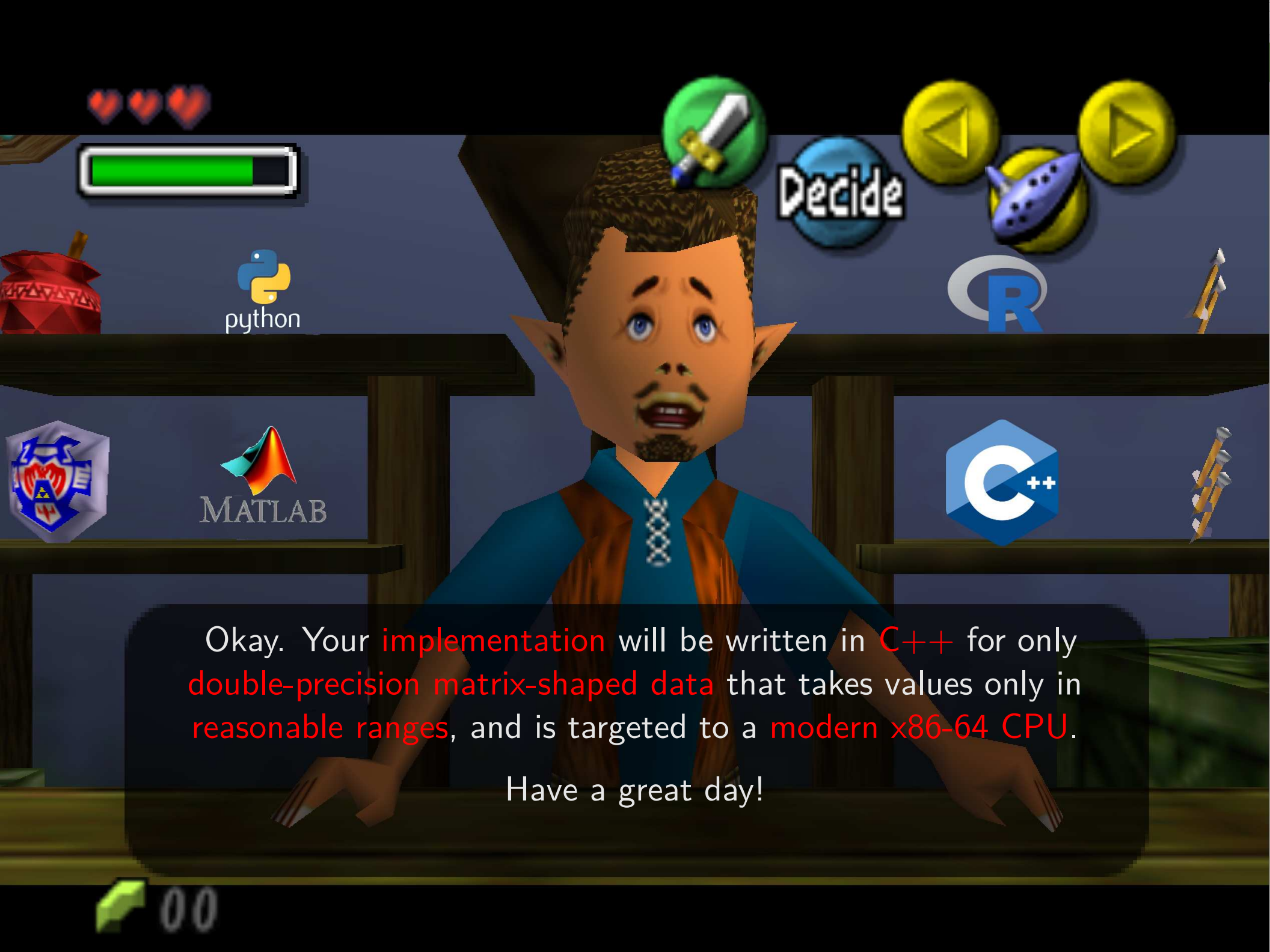

Decide
python
MATLAB
C++
R
What device will you target?
toaster
modern x86-64 CPU
modern CPU/GPU
FPGA
00


Decide
python
MATLAB
C++
What device will you target?
toaster
modern x86-64 CPU
modern CPU/GPU
FPGA


Decide
python
MATLAB
C++
What device will you target?
toaster
modern x86-64 CPU
modern CPU/GPU
FPGA


Decide
python
MATLAB
C++
What device will you target?
toaster
modern x86-64 CPU
modern CPU/GPU
FPGA


Decide
python
MATLAB
C++
R
What device will you target?
toaster
modern x86-64 CPU
modern CPU/GPU
FPGA
00


Decide
python
MATLAB
C++
R
Okay. Your implementation will be written in C++ for only double-precision matrix-shaped data that takes values only in reasonable ranges, and is targeted to a modern x86-64 CPU.
Have a great day!

MINOR CIRCUIT

"JUST MAKE IT CONVERGE !!"

0- 0 0KO

RANKED: #3

LITTLE MAC

VS.

PUSH START!

RANKED: #2

GLASS JOE

1-99 1KO

"MAKE IT QUICK... I WANT TO RETIRE!"

First let's design our data structure...

First let's design our data structure...

We want it to be as small as possible. We absolutely must hold:

- A link to the left child
- A link to the right child
- The dimension we are splitting on
- The split value to determine if a point goes left or right

First let's design our data structure...

We want it to be as small as possible. We absolutely must hold:

- A link to the left child
- A link to the right child
- The dimension we are splitting on
- The split value to determine if a point goes left or right

- *(if the node is a leaf)* Class probabilities for prediction

```
struct DecisionTree
{
  size_t splitDim;
  double splitValue;
  DecisionTree* left;
  DecisionTree* right;

  arma::vec classProbs;
};
```

Next, we want to use the information gain:

$$\sum_{c \in \mathcal{C}} P(c) \log_2 P(c)$$

where $\mathcal{C}$ is the set of possible class labels. This quantity is maximized when $P(c) = 1$ (i.e. when all labels are of one class).

Next, we want to use the information gain:

$$\sum_{c \in \mathcal{C}} P(c) \log_2 P(c)$$

where $\mathcal{C}$ is the set of possible class labels. This quantity is maximized when $P(c) = 1$ (i.e. when all labels are of one class).

Let's write an implementation (using the Armadillo C++ matrix library)...

```
// Labels should be in [0, numClasses).
template<typename LabelsType>
double InfoGain(const LabelsType& labels, const size_t numClasses)
{
```

```
// Labels should be in [0, numClasses).
template<typename LabelsType>
double InfoGain(const LabelsType& labels, const size_t numClasses)
{
  // Count the number of elements in each class (calculate P(C)).
  arma::uvec counts(numClasses, arma::fill::zeros);
  for (size_t i = 0; i < labels.n_elem; ++i)
    counts[labels[i]]++;
```

```
// Labels should be in [0, numClasses).
template<typename LabelsType>
double InfoGain(const LabelsType& labels, const size_t numClasses)
{
  // Count the number of elements in each class (calculate P(C)).
  arma::uvec counts(numClasses, arma::fill::zeros);
  for (size_t i = 0; i < labels.n_elem; ++i)
    counts[labels[i]]++;

  double gain = 0.0;
  for (size_t i = 0; i < numClasses; ++i)
  {
    const double f = ((double) counts[i] / (double) labels.n_elem);
    if (f > 0.0) // Work around divergence!!
      gain += f * std::log2(f);
  }
```

```
// Labels should be in [0, numClasses).
template<typename LabelsType>
double InfoGain(const LabelsType& labels, const size_t numClasses)
{
  // Count the number of elements in each class (calculate P(C)).
  arma::uvec counts(numClasses, arma::fill::zeros);
  for (size_t i = 0; i < labels.n_elem; ++i)
    counts[labels[i]]++;

  double gain = 0.0;
  for (size_t i = 0; i < numClasses; ++i)
  {
    const double f = ((double) counts[i] / (double) labels.n_elem);
    if (f > 0.0) // Work around divergence!!
      gain += f * std::log2(f);
  }

  return gain;
}
```

How do we split a node? *The paper doesn't consider continuous data!*

How do we split a node? *The paper doesn't consider continuous data!*

Let's borrow from the CART strategy...

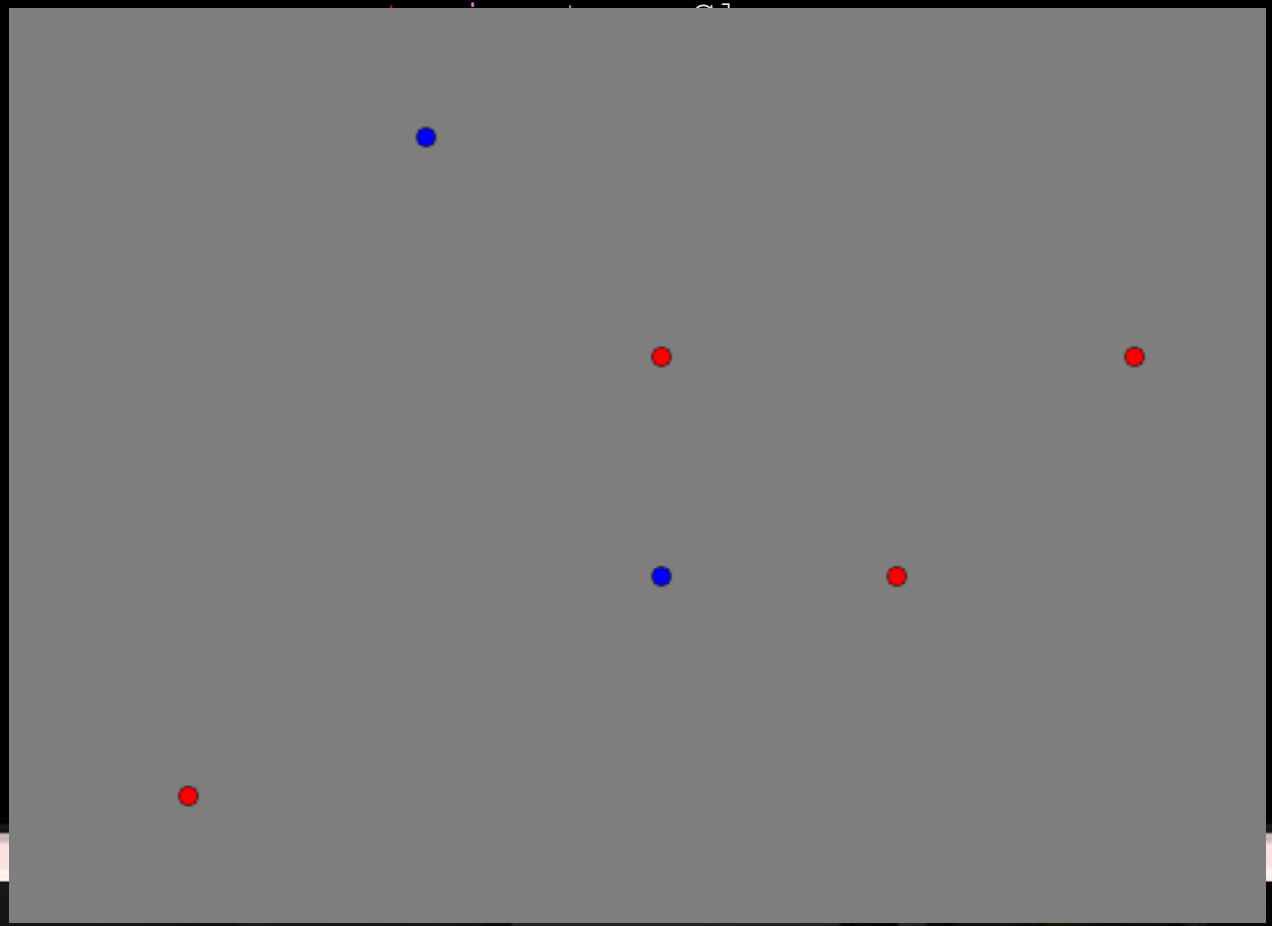
How do we split a node? *The paper doesn't consider continuous data!*

Let's borrow from the CART strategy...

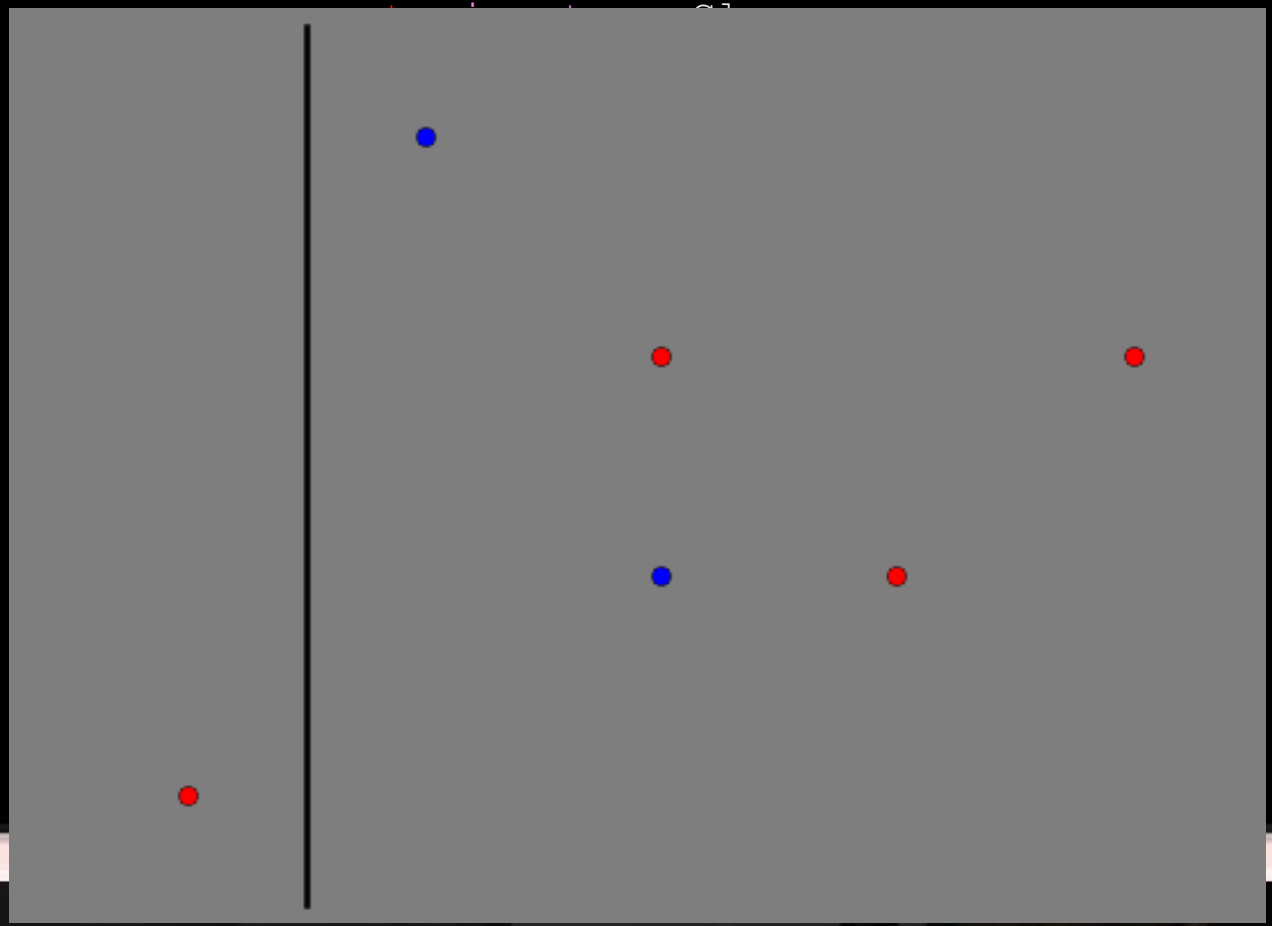
- Find the information gain of the unsplit node.
- For each dimension...
  - For each possible binary split in that dimension...
    - See if this split provides a new best information gain.
- If the best found split's information gain is better than the information gain of the unsplit node, then split!

```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,
                     const size_t numClasses,
                     double& bestGain,
                     double& bestSplitValue)
{
```

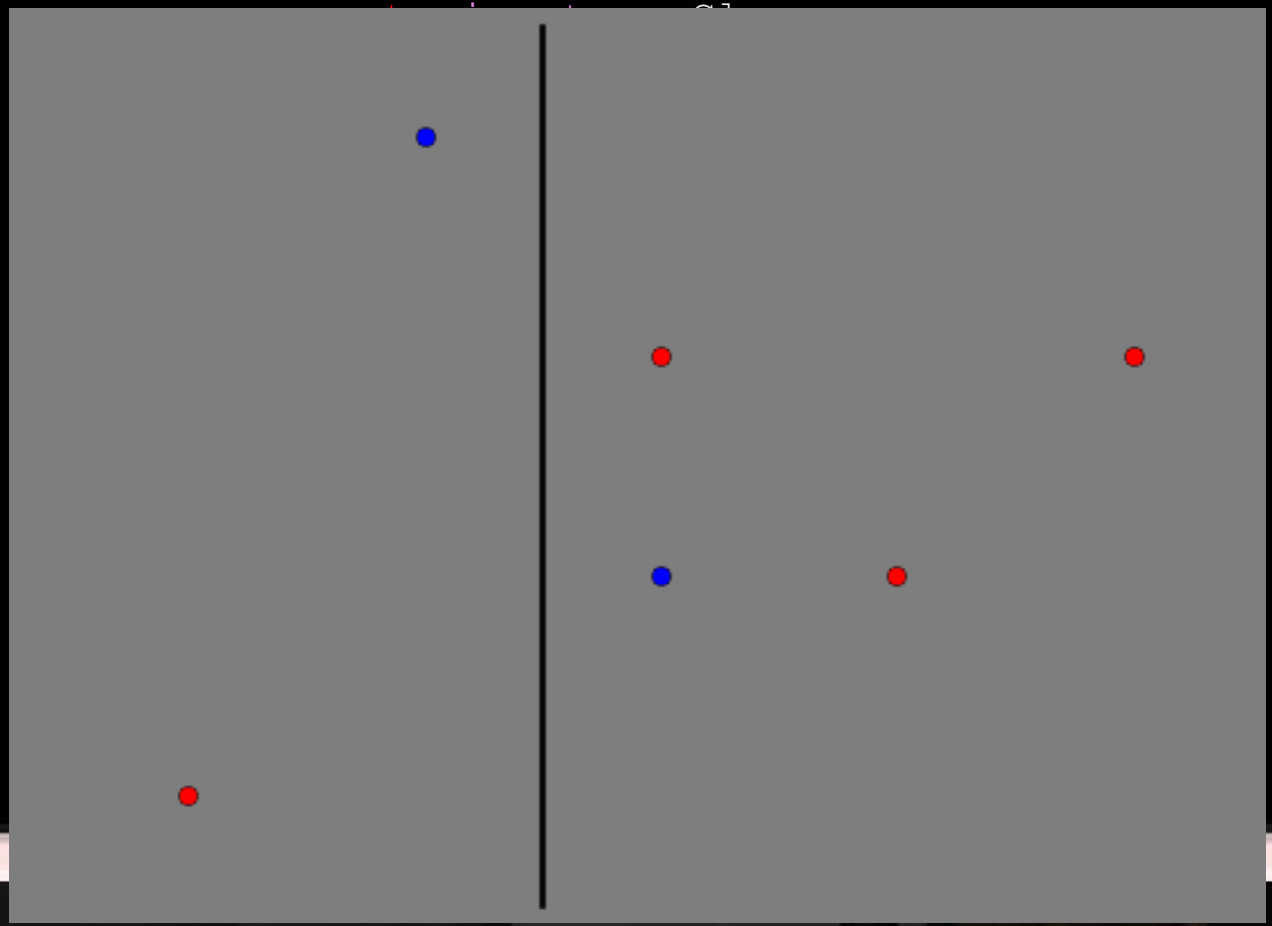
```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,

{
```



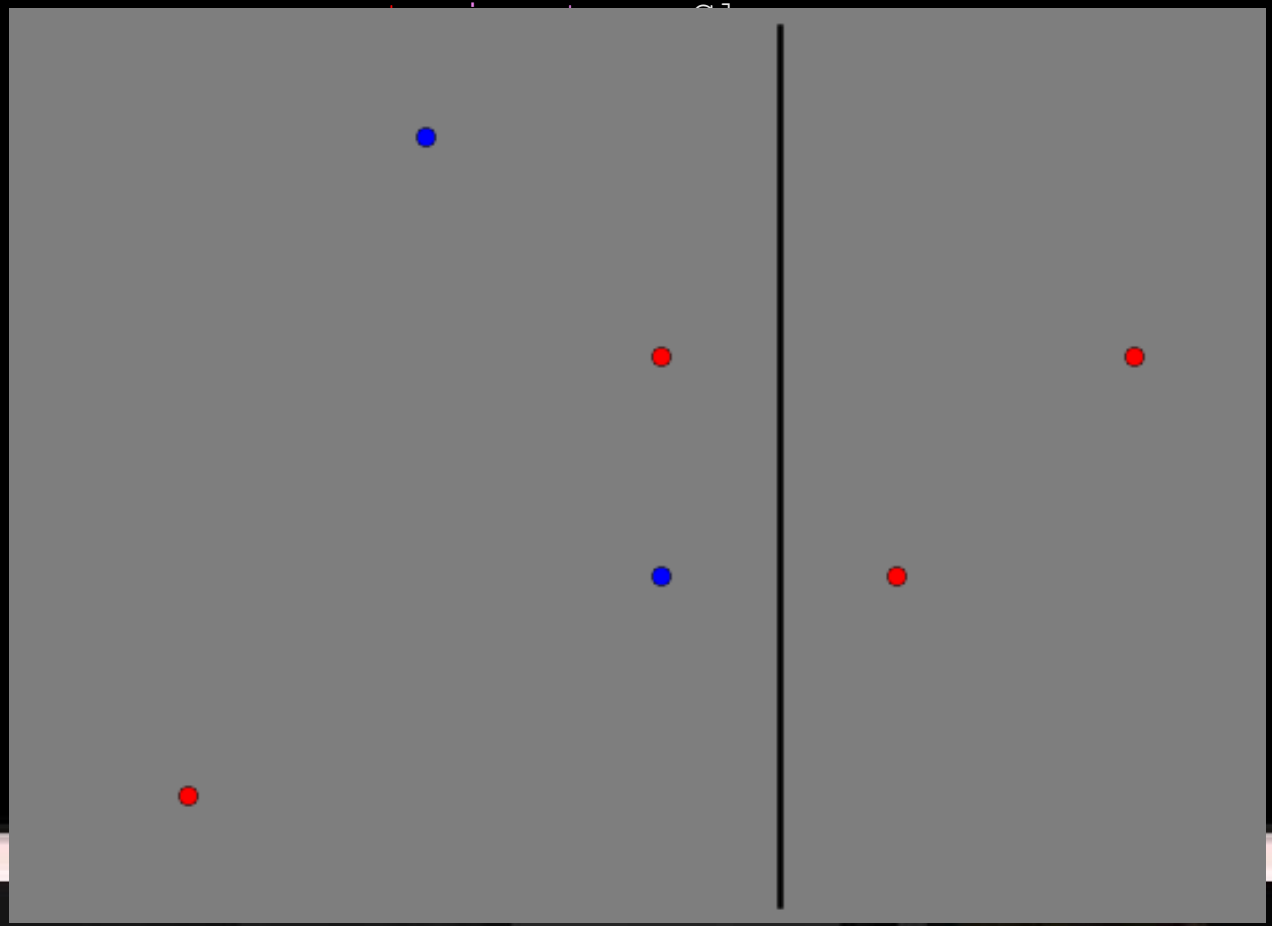
```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,
{
```



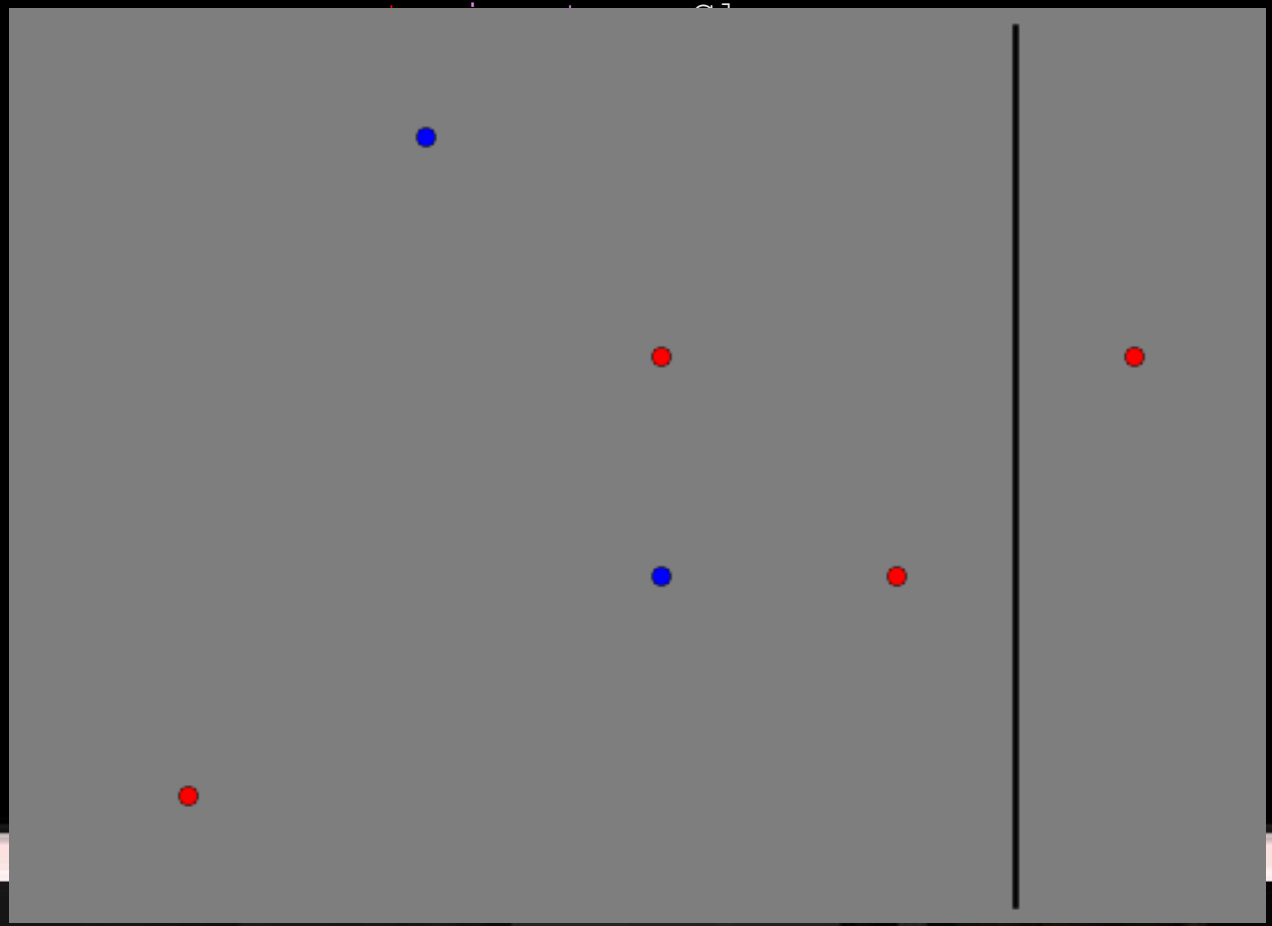
```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,
{
```



```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,

{
```



```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,
{
```

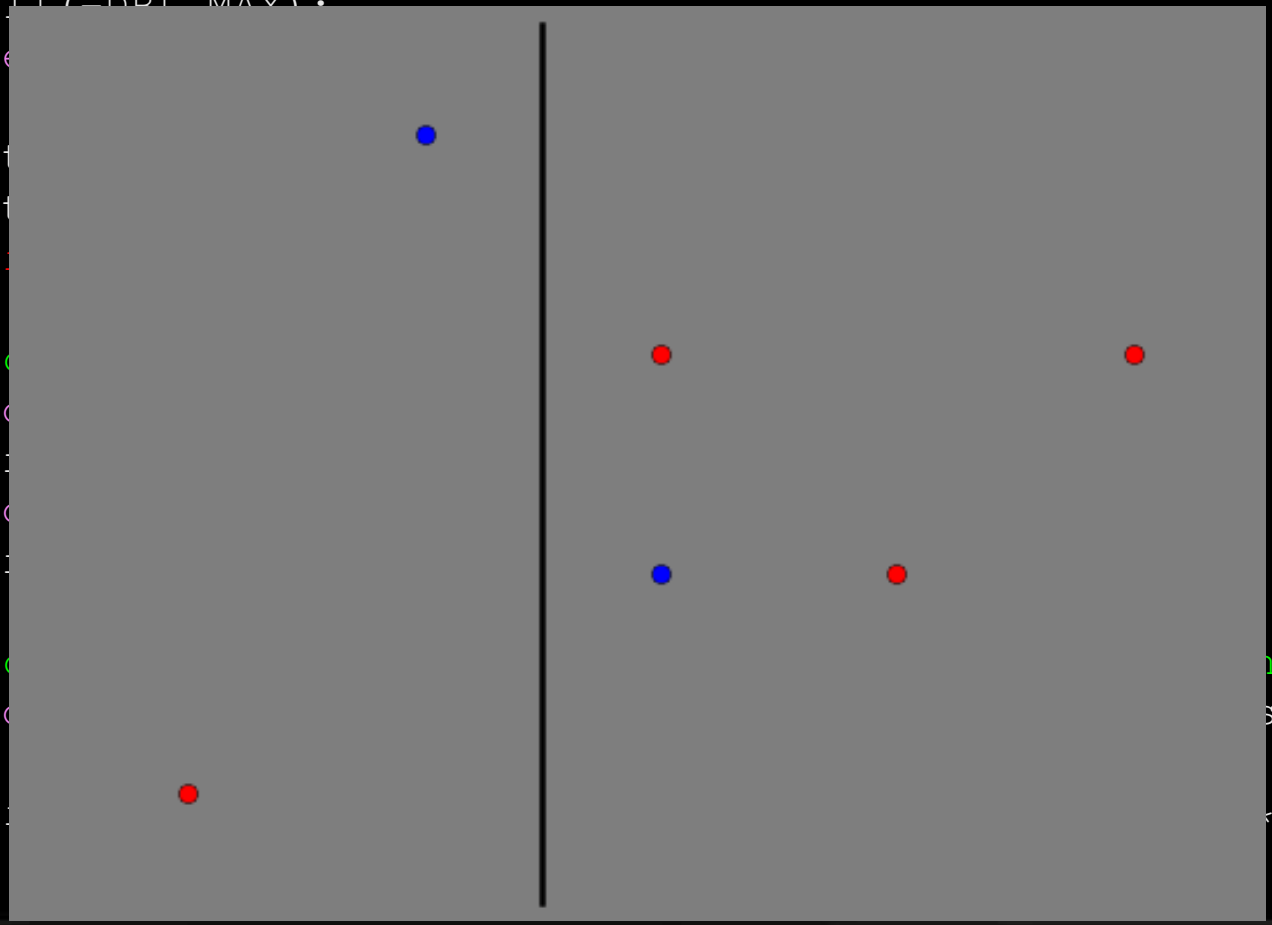


```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,
                     const size_t numClasses,
                     double& bestGain,
                     double& bestSplitValue)
{
```

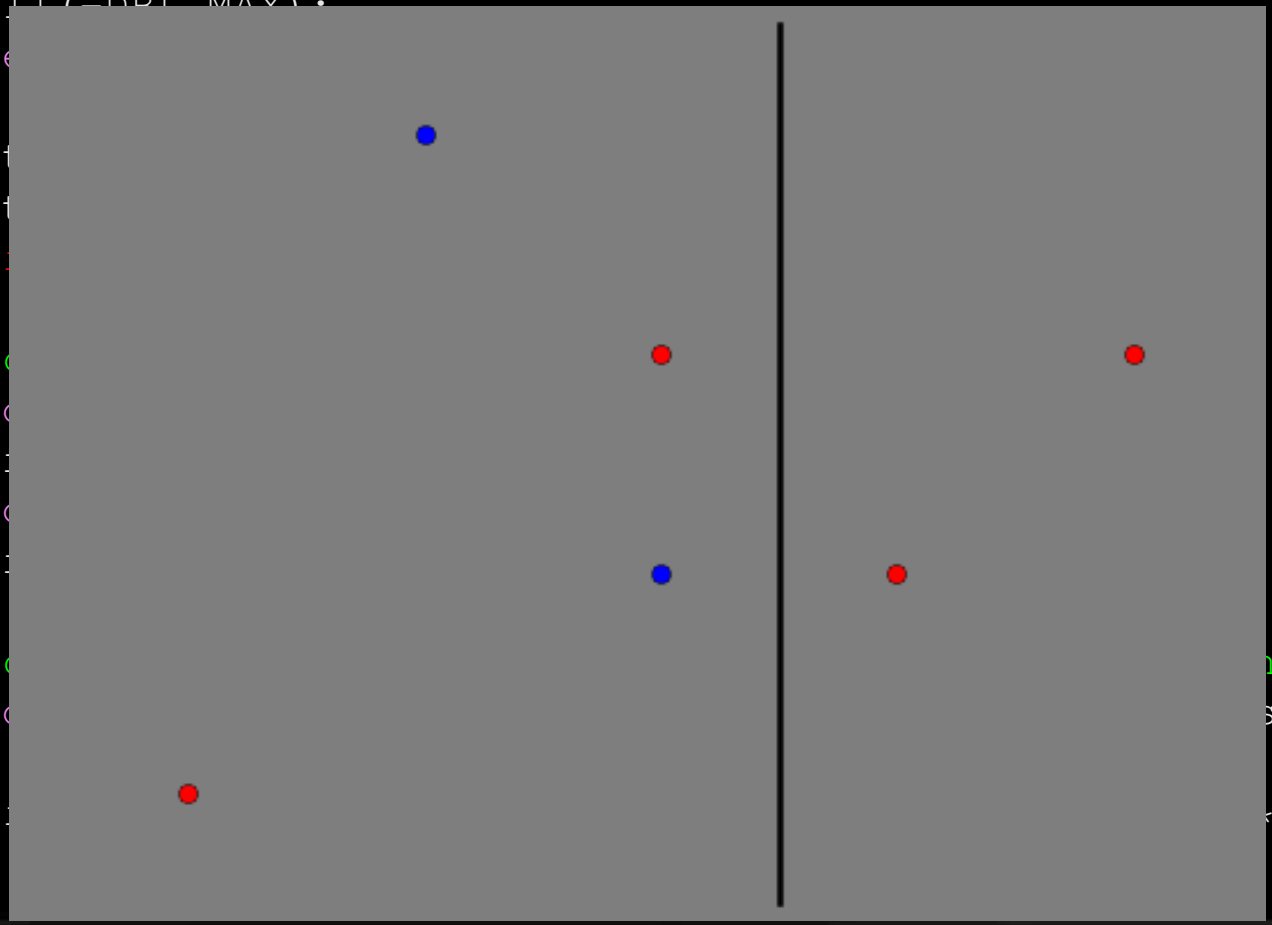
```
void BestBinarySplit(const size_t dimension,
                     arma::mat& data,
                     arma::uvec& labels,
                     const size_t numClasses,
                     double& bestGain,
                     double& bestSplitValue)
{
  // Sort the labels.
  arma::uvec sortedIndices = arma::sort_index(data.row(dimension));
  arma::uvec sortedLabels(labels.n_elem);
  for (size_t i = 0; i < sortedLabels.n_elem; ++i)
    sortedLabels[i] = labels[sortedIndices[i]];
```

```
arma::vec gains(data.n_cols - 1);
gains.fill(-DBL_MAX);
for (size_t i = 1; i <= gains.n_elem; ++i)
{
  if (data(dimension, sortedIndices[i]) ==
      data(dimension, sortedIndices[i - 1]))
    continue;

  // Calculate the gain for the left and right child.
  const double leftGain = InfoGain(
      sortedLabels.subvec(0, i - 1), numClasses);
  const double rightGain = InfoGain(
      sortedLabels.subvec(i, data.n_cols - 1), numClasses);

  // Calculate the fraction of points in the left and right children.
  const double leftRatio = double(i) / double(sortedLabels.n_elem);

  gains[i - 1] = leftRatio * leftGain + (1. - leftRatio) * rightGain;
}
```

```
arma::vec gains(data.n_cols - 1);
gains.fi
for (size
{
  if (dat
      dat
    cont

  // Calc
  const c
      so
  const c
      so                                          ;

  // Calc                                         nt children.
  const c                                         s.n_elem);

  gains[                                        * rightGain;
}
```



```
arma::vec gains(data.n_cols - 1);
gains.fi
for (size
{
  if (dat
      dat
    cont

  // Calc
  const c
      so
  const c
      so                                          ;

  // Calc                                         nt children.
  const c                                         s.n_elem);

  gains[                                        * rightGain;
}
```



```
arma::vec gains(data.n_cols - 1);
gains.fill(-DBL_MAX);
for (size_t i = 1; i <= gains.n_elem; ++i)
{
  if (data(dimension, sortedIndices[i]) ==
      data(dimension, sortedIndices[i - 1]))
    continue;

  // Calculate the gain for the left and right child.
  const double leftGain = InfoGain(
      sortedLabels.subvec(0, i - 1), numClasses);
  const double rightGain = InfoGain(
      sortedLabels.subvec(i, data.n_cols - 1), numClasses);

  // Calculate the fraction of points in the left and right children.
  const double leftRatio = double(i) / double(sortedLabels.n_elem);

  gains[i - 1] = leftRatio * leftGain + (1. - leftRatio) * rightGain;
}
```

```
  // (after filling gains vector with possible gains...)

  // These were passed by reference, so we just have to set them.
  bestGain = gains.max();
  bestSplitValue = data(dimension,
      sortedIndices[gains.index_max() + 1]);
}
```

How do we split a node? *The paper doesn't consider continuous data!*

Let's borrow from the CART strategy...

- Find the information gain of the unsplit node.
- For each dimension...
  - For each possible binary split in that dimension...
    - See if this split provides a new best information gain.
- If the best found split's information gain is better than the information gain of the unsplit node, then split!

How do we split a node? *The paper doesn't consider continuous data!*

Let's borrow from the CART strategy...

- Find the information gain of the unsplit node.
- For each dimension...
  - For each possible binary split in that dimension...
    - See if this split provides a new best information gain.
- If the best found split's information gain is better than the information gain of the unsplit node, then split!

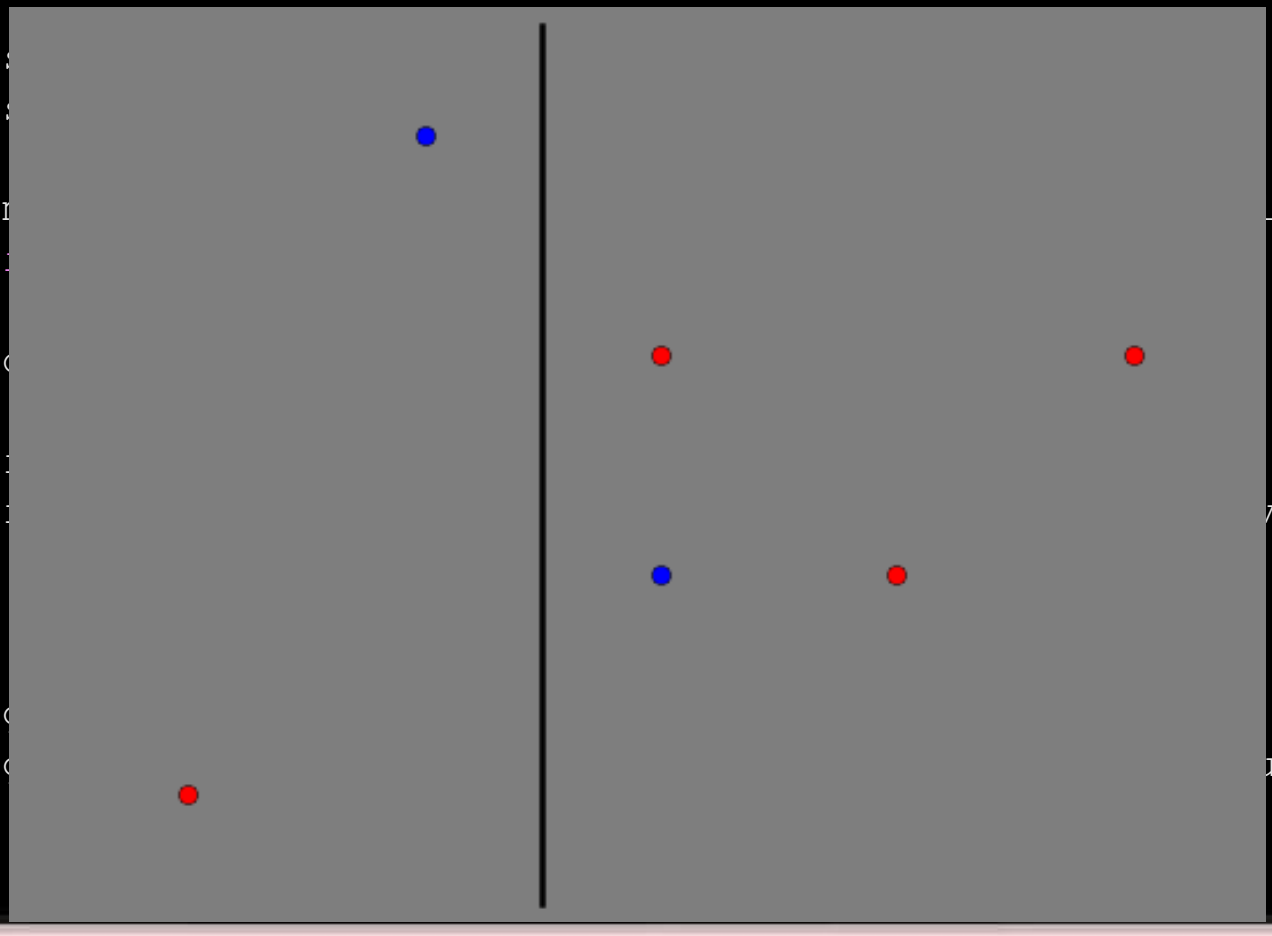
```
void SplitNode(DecisionTree* node,
               arma::mat& data,
               arma::uvec& labels,
               const size_t numClasses)
{
  if (data.n_cols == 1)
    return; // no split
```

```
void SplitNode(DecisionTree* node,
               arma::mat& data,
               arma::uvec& labels,
               const size_t numClasses)
{
  if (data.n_cols == 1)
    return; // no split

  // Get baseline gain.
  double bestGain = InfoGain(labels, numClasses);
  size_t bestDim = data.n_rows;
  double bestSplitValue;
```

```
// Find the best possible split.
for (size_t dim = 0; dim < data.n_rows; ++dim)
{
  double dimGain, splitVal;
  BestBinarySplit(dim, data, labels, numClasses, dimGain, splitVal);

  if (dimGain > bestGain)
  {
    bestGain = dimGain;
    bestDim = dim;
    bestSplitValue = splitVal;
  }
}
```

```
if (bestDim != data.n_rows)
{
  node->splitDim = bestDim;
  node->splitValue = bestSplitValue;

  arma::mat left, right; arma::uvec leftLabels, rightLabels;
  for (size_t i = 0; i < data.n_cols; ++i)
  {
    if (data(bestDim, i) < bestSplitValue) // Point is on the left.
    {
      left = arma::join_rows(left, data.col(i));
      leftLabels = arma::join_cols(leftLabels, labels.subvec(i, i));
    }
    else // Point is on the right.
    {
      right = arma::join_rows(right, data.col(i));
      rightLabels = arma::join_cols(rightLabels, labels.subvec(i,i));
    }
  }
```

```
if (bestDim != data.n_rows)
{
  node->
  node->

  arma::
  for (s
  {
    if (
    {
      le
      le
    }
    else
    {
      ri
      ri
    }
  }
```



```
                                                    s;
                                                    the left.
                                                    ec(i, i));
                                                    bvec(i,i));
```

```
if (bestDim != data.n_rows)
{
  node->splitDim = bestDim;
  node->splitValue = bestSplitValue;

  arma::mat left, right; arma::uvec leftLabels, rightLabels;
  for (size_t i = 0; i < data.n_cols; ++i)
  {
    if (data(bestDim, i) < bestSplitValue) // Point is on the left.
    {
      left = arma::join_rows(left, data.col(i));
      leftLabels = arma::join_cols(leftLabels, labels.subvec(i, i));
    }
    else // Point is on the right.
    {
      right = arma::join_rows(right, data.col(i));
      rightLabels = arma::join_cols(rightLabels, labels.subvec(i,i));
    }
  }
```

```
  // Recurse and build the children.
  node->left = new DecisionTree();
  SplitNode(node->left, left, leftLabels, numClasses);
  node->right = new DecisionTree();
  SplitNode(node->right, right, rightLabels, numClasses);
}
```

```
  else // We are not going to split the node.
  {
    // Calculate leaf probabilities.
    arma::uvec counts(numClasses);
    for (size_t i = 0; i < labels.n_elem; ++i)
      ++counts[labels[i]];

    node->classProbs.set_size(numClasses);
    for (size_t i = 0; i < numClasses; ++i)
      node->classProbs[i] = double(counts[i]) / double(numClasses);
  }
}
```

- `InfoGain()`: calculate information gain

- `InfoGain()`: calculate information gain
- `BestBinarySplit()`: find best binary split in a dimension

- `InfoGain()`: calculate information gain
- `BestBinarySplit()`: find best binary split in a dimension
- `SplitNode()`: split a decision tree node

- `InfoGain()`: calculate information gain
- `BestBinarySplit()`: find best binary split in a dimension
- `SplitNode()`: split a decision tree node

That's all we need!

- `InfoGain()`: calculate information gain
- `BestBinarySplit()`: find best binary split in a dimension
- `SplitNode()`: split a decision tree node

That's all we need!

```
// Create and build the tree on our data!
DecisionTree* tree = new DecisionTree();
SplitNode(tree, data, labels, numClasses);
```

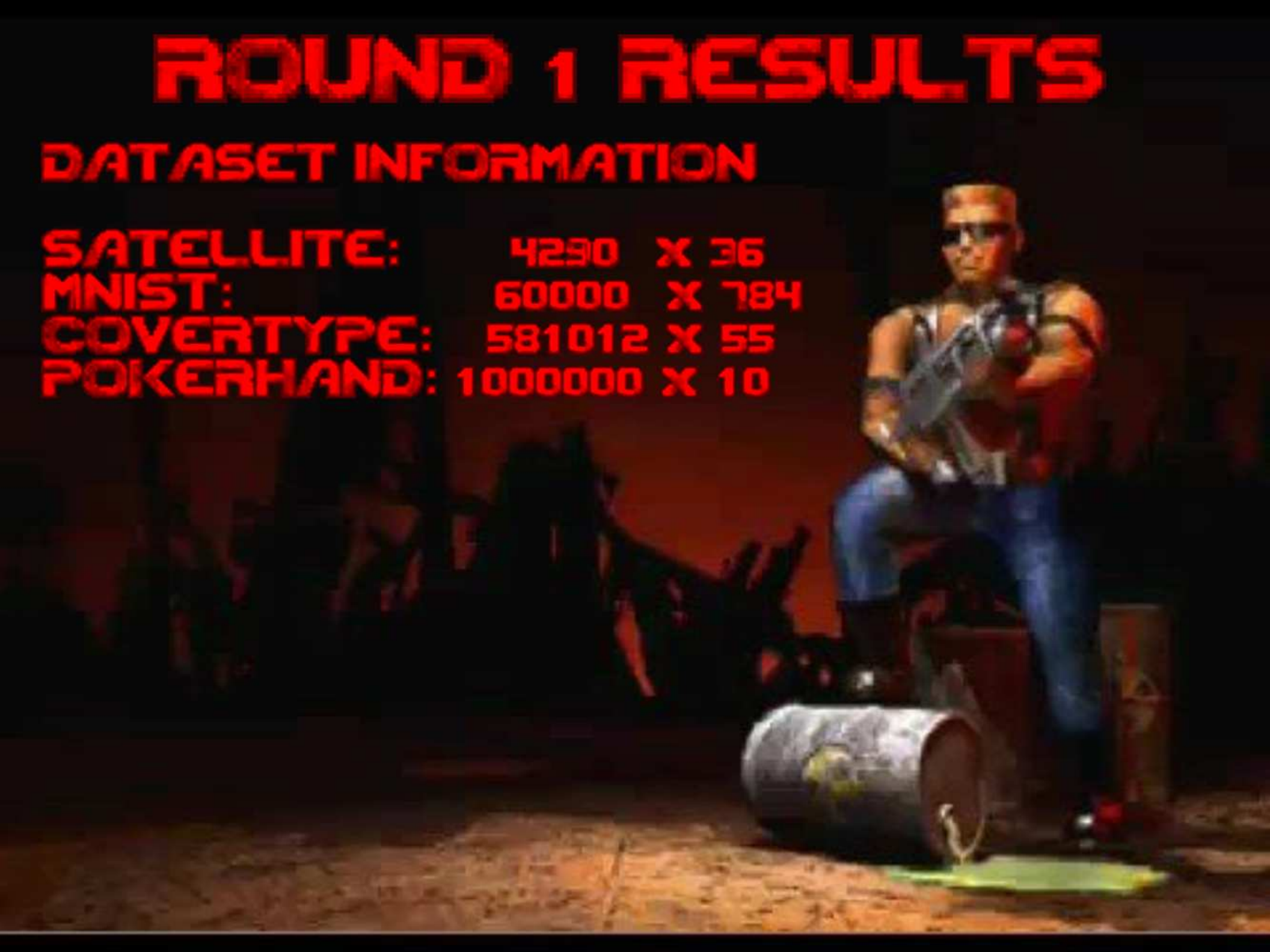
```
$ g++ -o tree tree.cpp -O3 -march=native -DARMA_NO_DEBUG -larmadillo
```

ROUND 1 RESULTS

ROUND 1 RESULTS

DATASET INFORMATION

SATELLITE:	4290 X 36
MNIST:	60000 X 784
COVERTYPE:	581012 X 55
POKERHAND:	1000000 X 10



ROUND 1 RESULTS

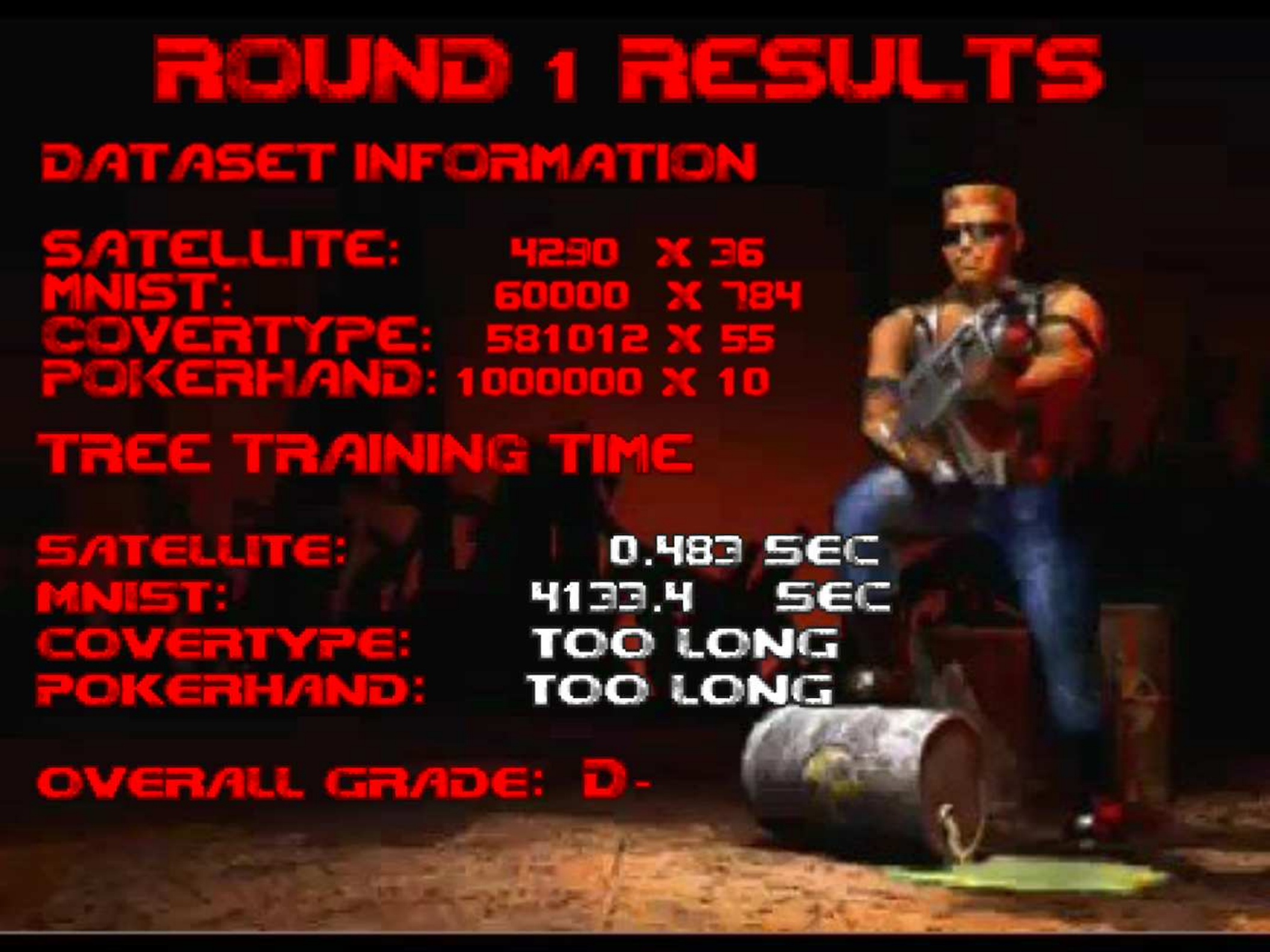
DATASET INFORMATION

SATELLITE:	4290 X 36
MNIST:	60000 X 784
COVERTYPE:	581012 X 55
POKERHAND:	1000000 X 10

TREE TRAINING TIME

SATELLITE:	0.483 SEC
MNIST:	4133.4 SEC
COVERTYPE:	TOO LONG
POKERHAND:	TOO LONG

OVERALL GRADE: D-



MAJOR CIRCUIT
TITLE BOUT

"TRY AVOIDING
UNNECESSARY
COPIES.!!"

6- 0 6KO

RANKED: #1

LITTLE MAC

VS.

CHAMPION

BALD BULL

34- 4 29KO

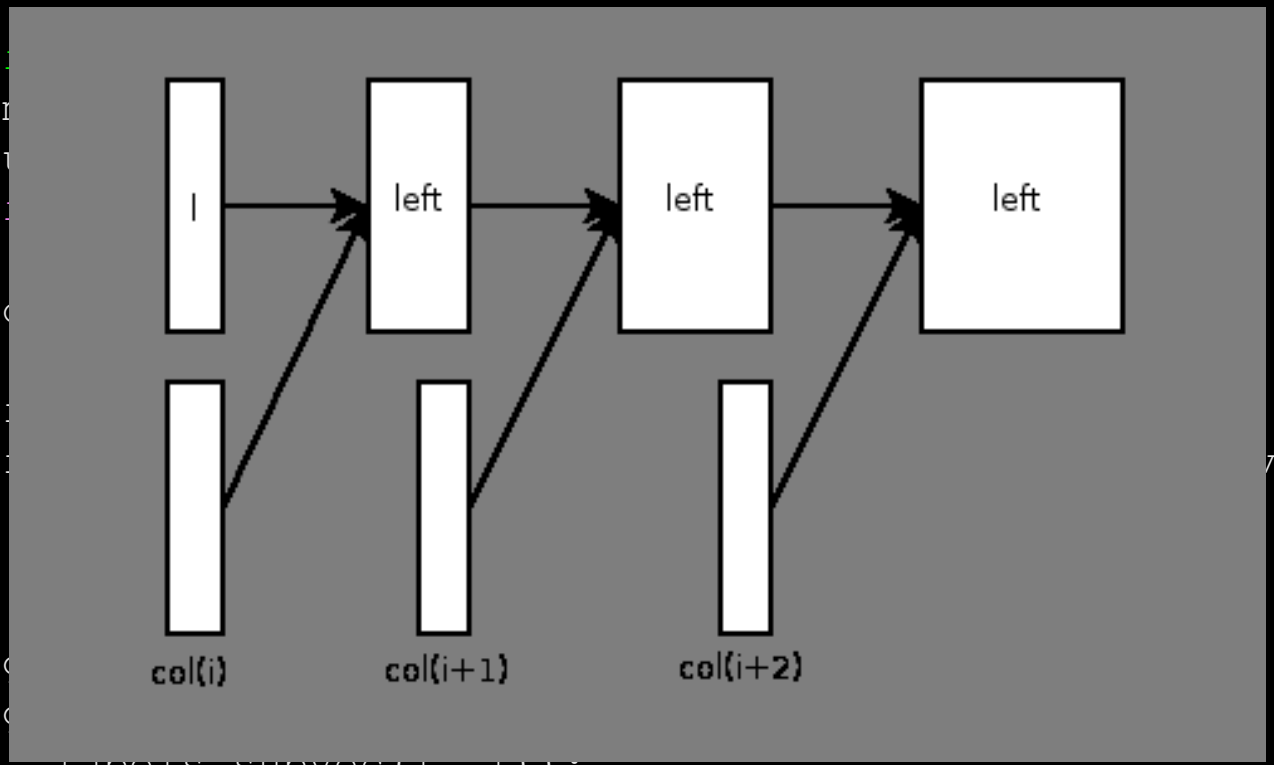
"YOUR EXTRA
MEMORY USAGE
IS YOUR
WEAKNESS.!!"

Every time we recurse we are copying data in `SplitNode()`:

```
// Split the data.
arma::mat left, right;
arma::uvec leftLabels, rightLabels;
for (size_t i = 0; i < data.n_cols; ++i)
{
  if (data(bestDim, i) < bestSplitValue)
  {
    left = arma::join_rows(left, data.col(i));
    leftLabels = arma::join_cols(leftLabels, labels.subvec(i, i));
  }
  else
  {
    right = arma::join_rows(right, data.col(i));
    rightLabels = arma::join_cols(rightLabels,
        labels.subvec(i, i));
  }
}
```

Every time we recurse we are copying data in `SplitNode()`:

```
// Spli
arma::r
arma::u
for (si
{
  if (c
  {
    le
    le                                              vec(i, i));
  }
  else
  {
    ri
    ri
        labels.subvec(i, i));
  }
}
```



Every time we recurse we are copying data in `SplitNode()`:

```
// Split the data.
arma::mat left, right;
arma::uvec leftLabels, rightLabels;
for (size_t i = 0; i < data.n_cols; ++i)
{
  if (data(bestDim, i) < bestSplitValue)
  {
    left = arma::join_rows(left, data.col(i));
    leftLabels = arma::join_cols(leftLabels, labels.subvec(i, i));
  }
  else
  {
    right = arma::join_rows(right, data.col(i));
    rightLabels = arma::join_cols(rightLabels,
        labels.subvec(i, i));
  }
}
```

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  while (data(bestDim, left) < bestSplitValue && left < right)
    ++left;
  while (data(bestDim, right) >= bestSplitValue && left < right)
    --right;

  if (left >= right)
    break;

  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  v
  v
  i
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

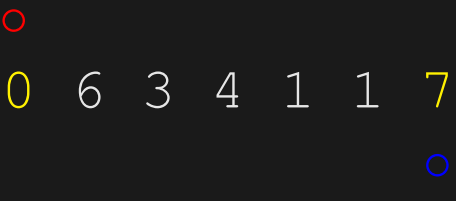
split value: 3.5

```
[0 6 3 4 1 1 7]
```

Instead use an in-place quick-sort-like approach:

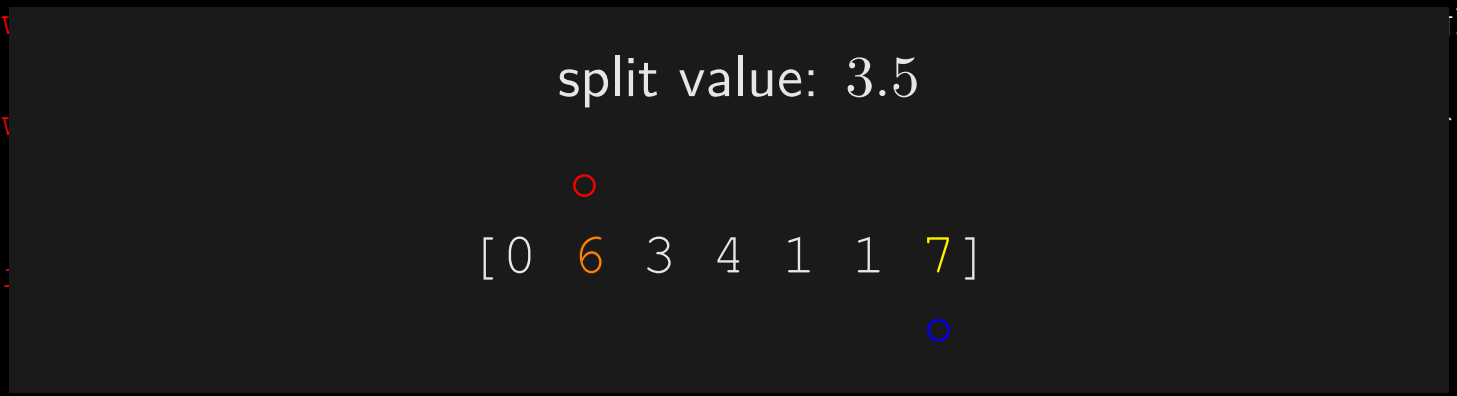
```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5



Instead use an in-place quick-sort-like approach:

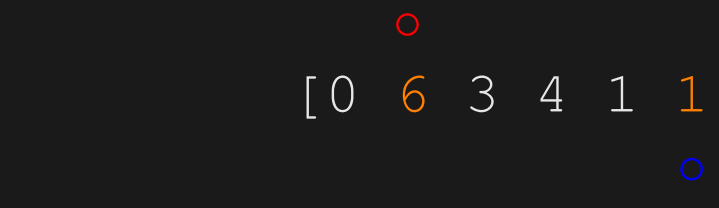
```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```



Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5



Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  v                                                    ht)
  v                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5

Instead use an in-place quick-sort-like approach:

```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  w                                                    ht)
  w                                                    ight)
  f
  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

split value: 3.5

Instead use an in-place quick-sort-like approach:

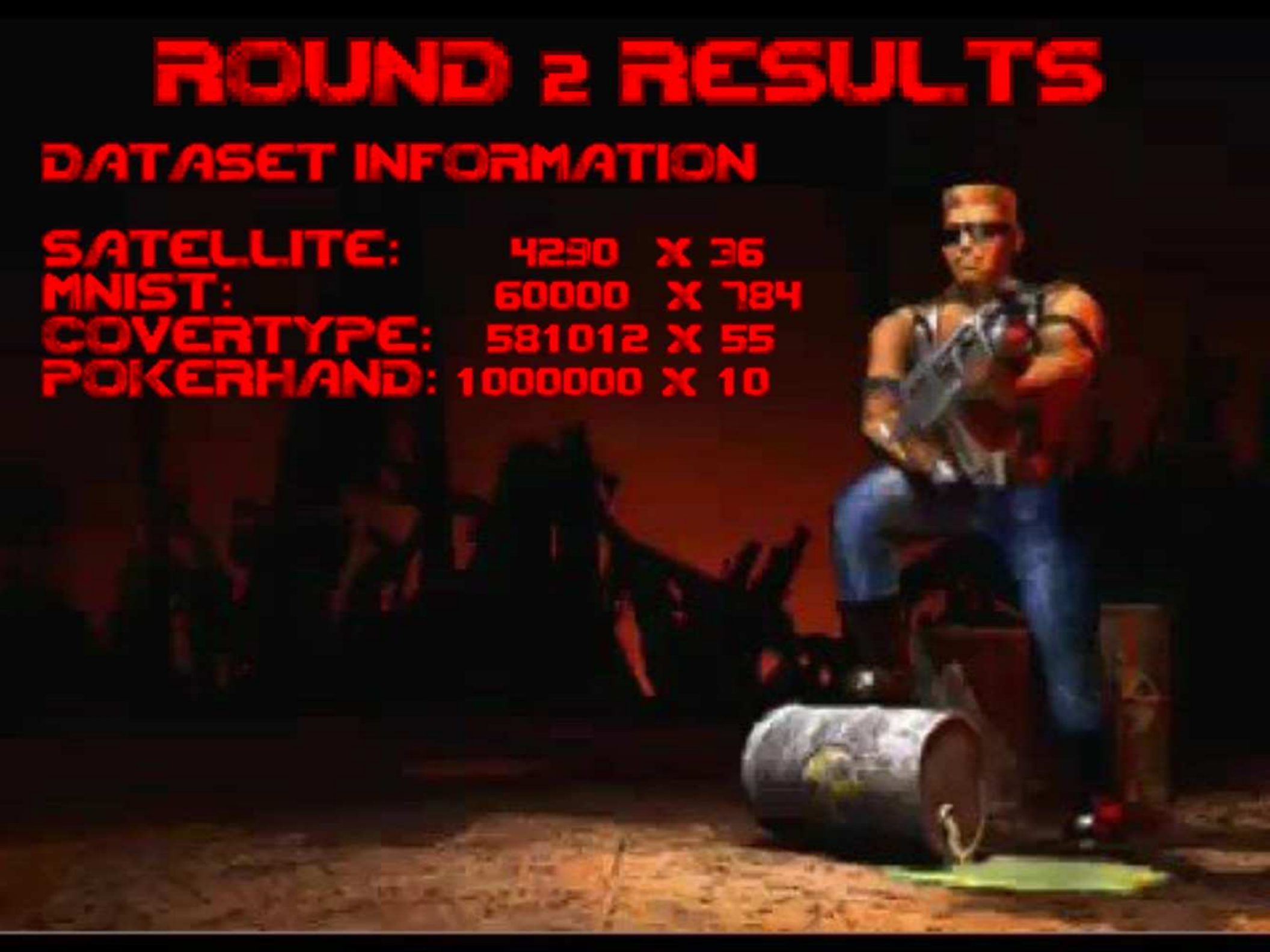
```
// Split the data, but in-place.  Just like quicksort!
size_t left = 0, right = data.n_cols - 1;
while (left < right)
{
  while (data(bestDim, left) < bestSplitValue && left < right)
    ++left;
  while (data(bestDim, right) >= bestSplitValue && left < right)
    --right;

  if (left >= right)
    break;

  data.swap_cols(left, right);
  labels.swap_rows(left, right);
}

// ...
```

```
// Build children recursively using Armadillo subviews.
// (The definition of SplitNode() requires very slight
// modification.)
node->left = new DecisionTree();
SplitNode(node->left, data.cols(0, left - 1),
    labels.subvec(0, left - 1), numClasses);
node->right = new DecisionTree();
SplitNode(node->right, data.cols(left, data.n_cols - 1),
    labels.subvec(left, data.n_cols - 1), numClasses);
```

ROUND 2 RESULTS

DATASET INFORMATION

SATELLITE:	4290 X 36
MNIST:	60000 X 784
COVERTYPE:	581012 X 55
POKERHAND:	1000000 X 10



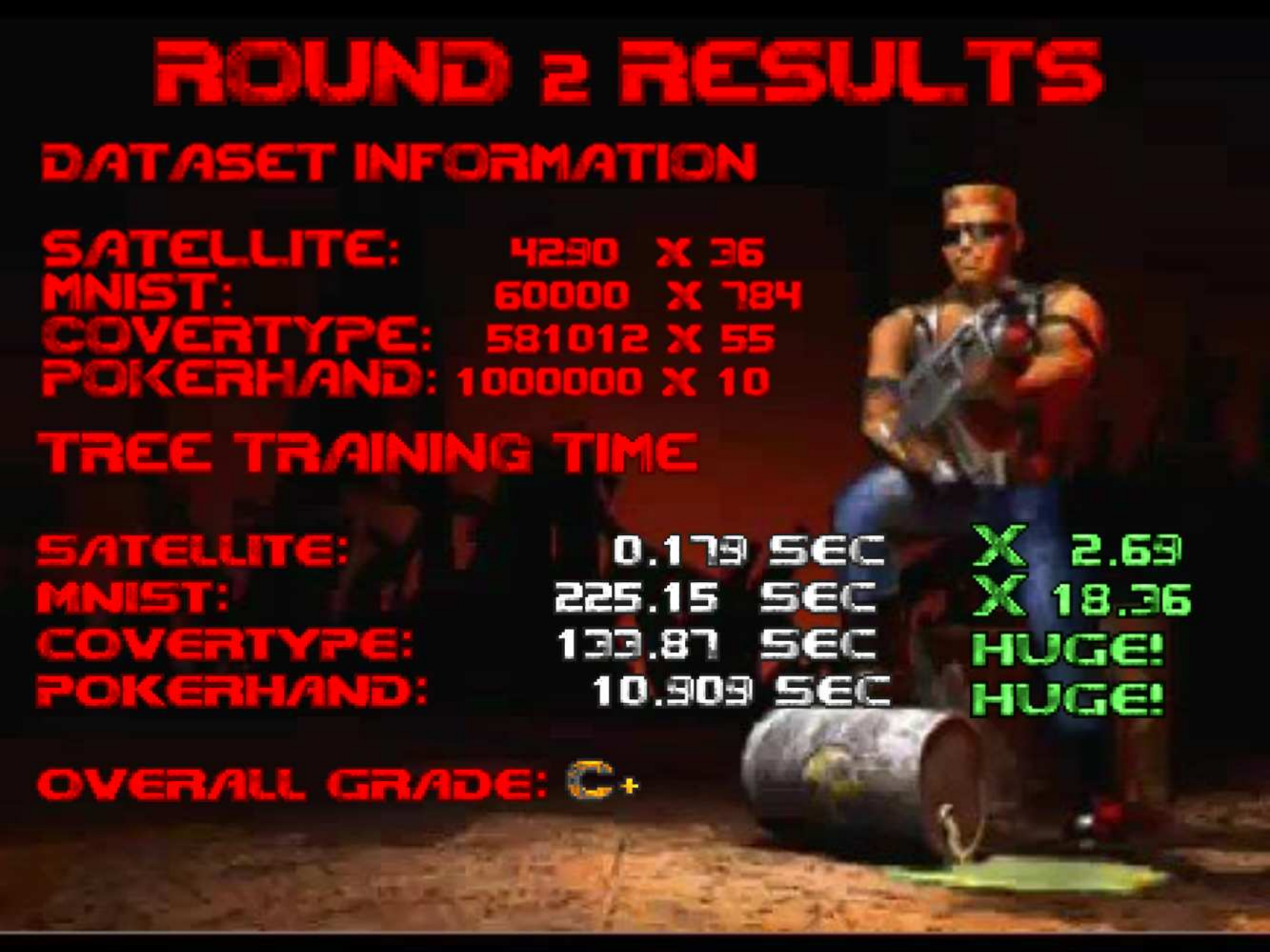
ROUND 2 RESULTS

DATASET INFORMATION

SATELLITE:	4290	X 36
MNIST:	60000	X 784
COVERTYPE:	581012	X 55
POKERHAND:	1000000	X 10

TREE TRAINING TIME

SATELLITE:	0.179 SEC	X 2.69
MNIST:	225.15 SEC	X 18.36
COVERTYPE:	133.87 SEC	HUGE!
POKERHAND:	10.909 SEC	HUGE!



OVERALL GRADE: C+

WORLD CIRCUIT

"CAN YOU TERMINATE THE LOOPS EARLY??"

8- 1 8KO

RANKED: #5

LITTLE MAC

VS.

PUSH START!

RANKED: #4

SODA POPINSKI

33- 2 24KO

"WORK SMART. NOT HARD.!!"

Information gain is 0 when we have a perfect split:

$$\sum_{c \in \mathcal{C}} P(c) \log_2 P(c)$$

Information gain is 0 when we have a perfect split:

$$\sum_{c \in \mathcal{C}} P(c) \log_2 P(c)$$

So if we see a gain of 0, we can't do any better. *This will often happen for decision tree nodes with few points!*

```
for (size_t i = 1; i <= gains.n_elem; ++i)
{
  if (data(dimension, sortedIndices[i]) ==
      data(dimension, sortedIndices[i - 1]))
    continue;

  // Calculate the gain for the left and right child.
  const double leftGain = InfoGain(
      sortedLabels.subvec(0, i - 1), numClasses);
  const double rightGain = InfoGain(
      sortedLabels.subvec(i, data.n_cols - 1), numClasses);

  // Calculate the fraction of points in the left and right children.
  const double leftRatio = double(i) / double(sortedLabels.n_elem);

  gains[i - 1] = leftRatio * leftGain + (1. - leftRatio) * rightGain;
}
```

```
for (size_t i = 1; i <= gains.n_elem; ++i)
{
  if (data(dimension, sortedIndices[i]) ==
      data(dimension, sortedIndices[i - 1]))
    continue;

  // Calculate the gain for the left and right child.
  const double leftGain = InfoGain(
      sortedLabels.subvec(0, i - 1), numClasses);
  const double rightGain = InfoGain(
      sortedLabels.subvec(i, data.n_cols - 1), numClasses);

  // Calculate the fraction of points in the left and right children.
  const double leftRatio = double(i) / double(sortedLabels.n_elem);

  gains[i - 1] = leftRatio * leftGain + (1. - leftRatio) * rightGain;
  if (gains[i - 1] == 0.0)
    break; // The split is optimal---stop searching.
}
```

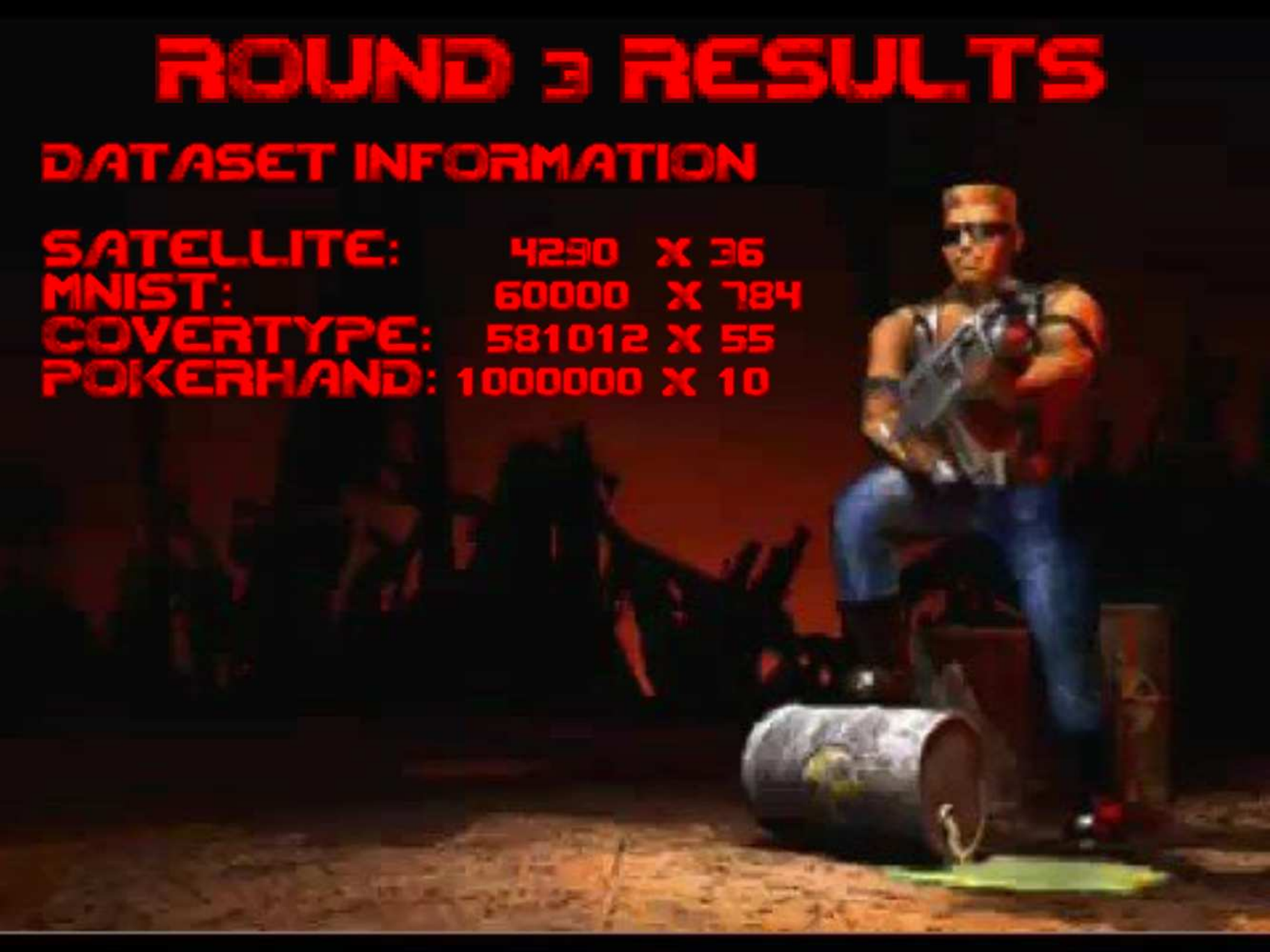
```
for (size_t dim = 0; dim < data.n_rows; ++dim)
{
  double dimGain, splitVal;
  BestBinarySplit(dim, data, labels, numClasses, dimGain, splitVal);

  if (dimGain > bestGain)
  {
    bestGain = dimGain;
    bestDim = dim;
    bestSplitValue = splitVal;
  }
}
```

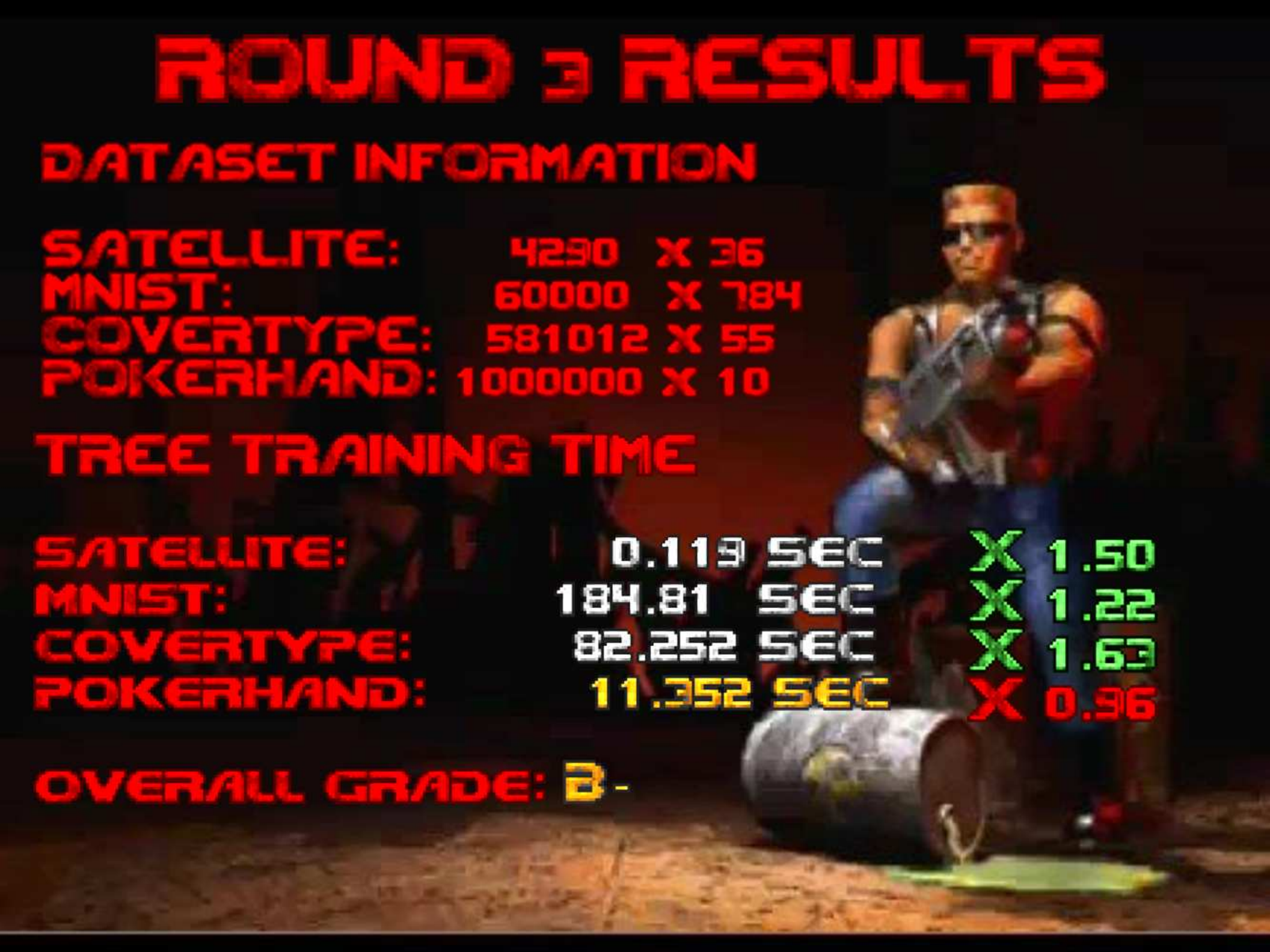
```
for (size_t dim = 0; dim < data.n_rows; ++dim)
{
  double dimGain, splitVal;
  BestBinarySplit(dim, data, labels, numClasses, dimGain, splitVal);

  if (dimGain > bestGain)
  {
    bestGain = dimGain;
    bestDim = dim;
    bestSplitValue = splitVal;

    if (bestGain == 0.0)
      break; // Split is optimal---stop searching.
  }
}
```

ROUND 3 RESULTS

DATASET INFORMATION

SATELLITE:	4290 X 36
MNIST:	60000 X 784
COVERTYPE:	581012 X 55
POKERHAND:	1000000 X 10




ROUND 3 RESULTS
DATASET INFORMATION
SATELLITE: 4290 X 36
MNIST: 60000 X 784
COVERTYPE: 581012 X 55
POKERHAND: 1000000 X 10
TREE TRAINING TIME
SATELLITE: 0.119 SEC X 1.50
MNIST: 184.81 SEC X 1.22
COVERTYPE: 82.252 SEC X 1.63
POKERHAND: 11.352 SEC X 0.96
OVERALL GRADE: B-

WORLD CIRCUIT

"REMEMBER THAT YOUR PROCESSOR HAS MULTIPLE CORES.!.!"

8 - 1 8KO

RANKED: #5

LITTLE MAC

VS.

PUSH START!

RANKED: #3

DON FLAMENCO

22 - 3 9KO

"SINGLETHREADED PROGRAMS ARE FOR LOSERS.!.!"

Modern processors have far more than one core...

An easy way to exploit this is with OpenMP.

OpenMP allows us to define individual tasks with the

```
#pragma omp task
```

directive. We can use one of these each time we recurse...

```
// Recurse and build children.
node->left = new DecisionTree();
SplitNode(node->left, data.cols(0, left - 1),
    labels.subvec(0, left - 1), numClasses);
node->right = new DecisionTree();
SplitNode(node->right, data.cols(left, data.n_cols - 1),
    labels.subvec(left, data.n_cols - 1), numClasses);
```

```
// Recurse and build children.
if (data.n_cols > 100)
{
  // Build children in parallel.
  // ...
}
else
{
  // Build children serially.
  node->left = new DecisionTree();
  SplitNode(node->left, data.cols(0, left - 1),
      labels.subvec(0, left - 1), numClasses);
  node->right = new DecisionTree();
  SplitNode(node->right, data.cols(left, data.n_cols - 1),
      labels.subvec(left, data.n_cols - 1), numClasses);
}
```

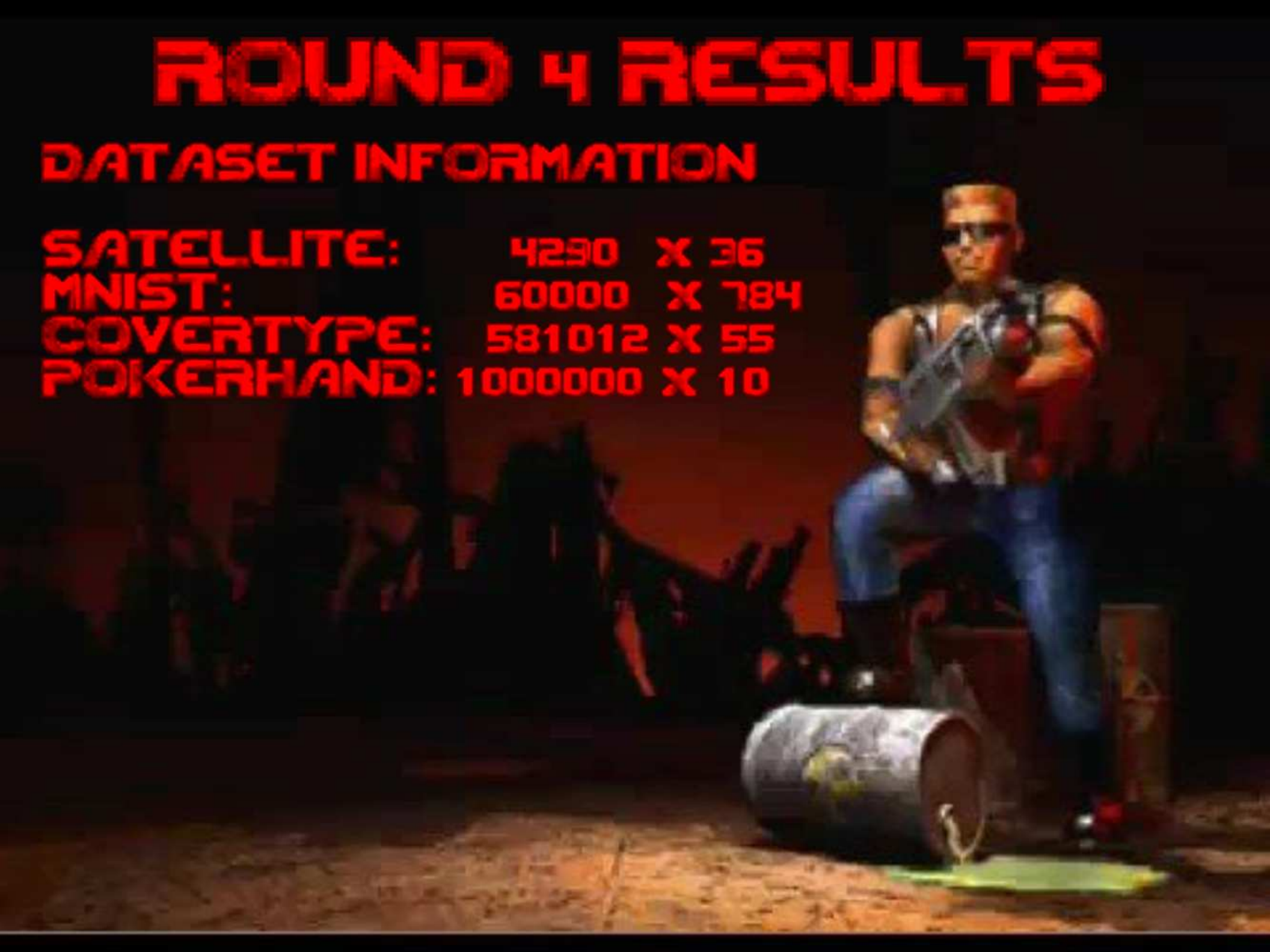
```
if (data.n_cols > 100)
{
  // Build children in parallel.
  #pragma omp task
  {
    node->left = new DecisionTree();
    SplitNode(node->left, data.cols(0, left - 1),
        labels.subvec(0, left - 1), numClasses);
  }

  #pragma omp task
  {
    node->right = new DecisionTree();
    SplitNode(node->right, data.cols(left, data.n_cols - 1),
        labels.subvec(left, data.n_cols - 1), numClasses);
  }

  #pragma omp taskwait
}
else // Build serially ...
```

Now we have to compile with the `-fopenmp` option!

ROUND 4 RESULTS

DATASET INFORMATION

SATELLITE:	4290 X 36
MNIST:	60000 X 784
COVERTYPE:	581012 X 55
POKERHAND:	1000000 X 10



ROUND 4 RESULTS

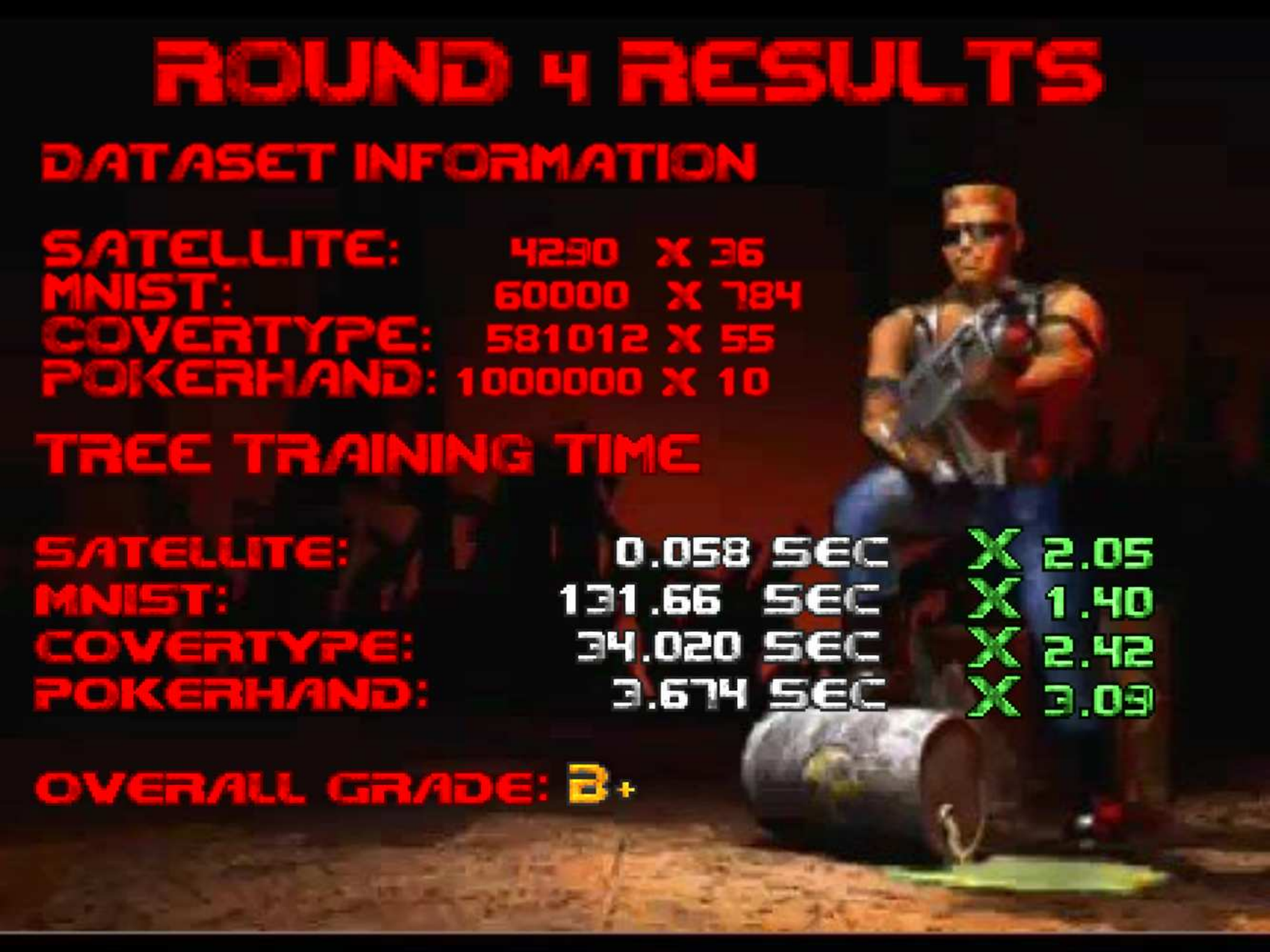
DATASET INFORMATION

SATELLITE:	4290	X 36
MNIST:	60000	X 784
COVERTYPE:	581012	X 55
POKERHAND:	1000000	X 10

TREE TRAINING TIME

SATELLITE:	0.058 SEC	X 2.05
MNIST:	131.66 SEC	X 1.40
COVERTYPE:	34.020 SEC	X 2.42
POKERHAND:	3.674 SEC	X 3.09

OVERALL GRADE: B+



WORLD CIRCUIT

"FOR EVEN MORE SPEED TRY USING SOME SIMD INSTRUCTIONS.!!"

8- 1 8KO

RANKED: #5

LITTLE MAC

VS.

PUSH START!

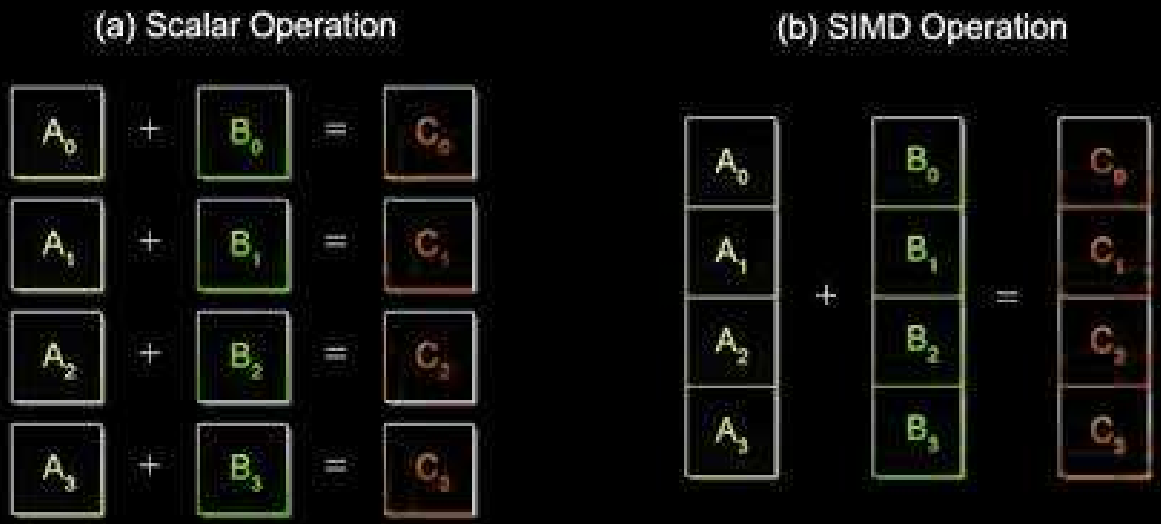
KID DYNAMITE

MIKE TYSON

31- 0 27KO

"MOST PEOPLE JUST ARE NOT AWARE OF WHAT THEIR PROCESSOR CAN DO.!!"

Most modern processors support SIMD (single instruction multiple data) instructions.



Most modern processors support SIMD (single instruction multiple data) instructions.

`gcc` (and `clang`) can and may (depending on options) auto-vectorize simple loops:

```
for (size_t i = 0; i < numClasses; ++i)
{
  const double f = ((double) counts[i] / (double) labels.n_elem);
  if (f > 0.0)
    gain += f * std::log2(f);
}
```

But what about more complicated indirect-access-type loops?

```
// Count the number of elements in each class.
arma::uvec counts(numClasses, arma::fill::zeros);
for (size_t i = 0; i < labels.n_elem; ++i)
  counts[labels[i]]++;
```

But what about more complicated indirect-access-type loops?

```
// Count the number of elements in each class.
arma::uvec counts(numClasses, arma::fill::zeros);
for (size_t i = 0; i < labels.n_elem; ++i)
  counts[labels[i]]++;
```

This turns out to be a really hard problem. We want something like this:

```
counts[0] += a[0];
counts[1] += a[1];
counts[2] += a[2];
counts[3] += a[3];
```

Can we get there?

```
counts[0] += a[0];
counts[1] += a[1];
counts[2] += a[2];
counts[3] += a[3];
```

```
counts[0] += a[0];
counts[1] += a[1];
counts[2] += a[2];
counts[3] += a[3];
```

Then `a` must be some kind of *partial count* of the number of labels in each class.

(seems like a non sequitor)

Most modern processors also support the POPCNT (population count) instruction. In a single instruction, we can count the number of `1` bits in a 64-bit vector.

`0xFF00FF00` $\rightarrow$ `16` in a single instruction!

(seems like a non sequitor)

Most modern processors also support the POPCNT (population count) instruction. In a single instruction, we can count the number of `1` bits in a 64-bit vector.

`0xFF00FF00` $\rightarrow$ `16` in a single instruction!

- We want a partial count.
- Population counts can count 64 one-bit values **extremely** quickly.

(seems like a non sequitor)

Most modern processors also support the POPCNT (population count) instruction. In a single instruction, we can count the number of `1` bits in a 64-bit vector.

`0xFF00FF00` $\rightarrow$ `16` in a single instruction!

- We want a partial count.
- Population counts can count 64 one-bit values **extremely** quickly.

Encode our labels as bitsets!

(seems like a non sequitor)

Most modern processors also support the POPCNT (population count) instruction. In a single instruction, we can count the number of `1` bits in a 64-bit vector.

`0xFF00FF00` $\rightarrow$ `16` in a single instruction!

- We want a partial count.
- Population counts can count 64 one-bit values **extremely** quickly.

Encode our labels as bitsets!

One bitset for whether or not the label is class 0, one bitset for whether or not the label is class 1, and so forth...

So given the labels

```
[0, 2, 0, 1, 1, 3, 0, 0],
```

So given the labels

```
[0, 2, 0, 1, 1, 3, 0, 0],
```

we want to construct the bitsets

```
[1, 0, 1, 0, 0, 0, 1, 1],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 1, 0, 0]
```

So given the labels

```
[0, 2, 0, 1, 1, 3, 0, 0],
```

we want to construct the bitsets

```
[1, 0, 1, 0, 0, 0, 1, 1],
[0, 0, 0, 1, 1, 0, 0, 0],
[0, 1, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 1, 0, 0]
```

4 POPCNTs to evaluate 8 labels vs. 8 iterations of a loop! (and with more labels it gets significantly faster...)

So the overall strategy is going to be this:

So the overall strategy is going to be this:

- In `BestBinarySplit()`, when we sort the labels, instead assemble the sorted bitsets.

So the overall strategy is going to be this:

- In `BestBinarySplit()`, when we sort the labels, instead assemble the sorted bitsets.
- In `InfoGain()`, use the sorted bitsets to do something like below:

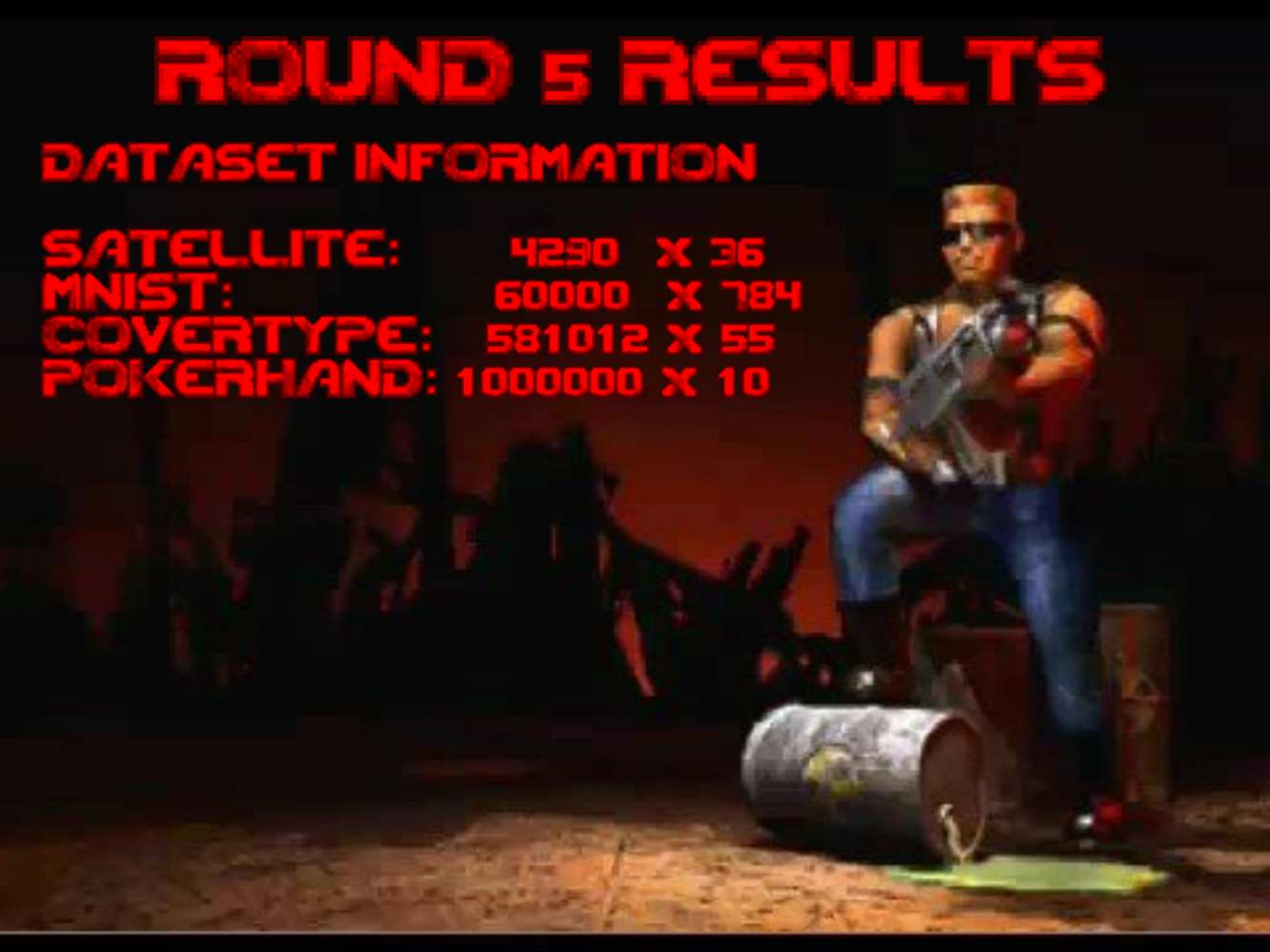
```
// Four POPCNT instructions!  (or go buy an AVX-512 processor!)
a[0] = popcnt64(bitset[0][i]);
a[1] = popcnt64(bitset[1][i]);
a[2] = popcnt64(bitset[2][i]);
a[3] = popcnt64(bitset[3][i]);

// One AVX2 vector add instruction!
counts[0] += a[0];
counts[1] += a[1];
counts[2] += a[2];
counts[3] += a[3];
```

Let's talk about memory alignment issues.

ROUND 5 RESULTS

DATASET INFORMATION

SATELLITE:	4290 X 36
MNIST:	60000 X 784
COVERTYPE:	581012 X 55
POKERHAND:	1000000 X 10



ROUND 5 RESULTS

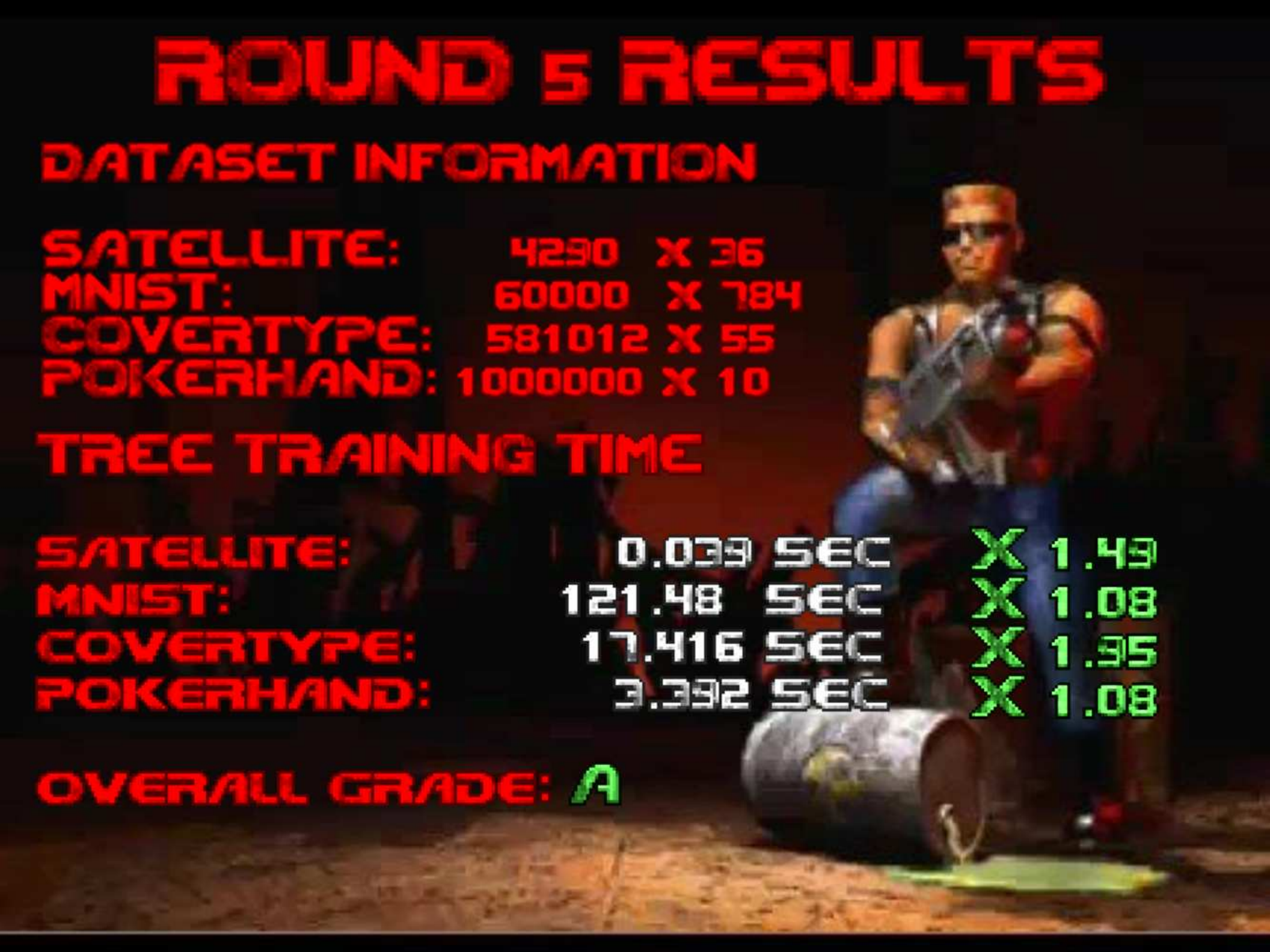
DATASET INFORMATION

SATELLITE:	4290	X 36
MNIST:	60000	X 784
COVERTYPE:	581012	X 55
POKERHAND:	1000000	X 10

TREE TRAINING TIME

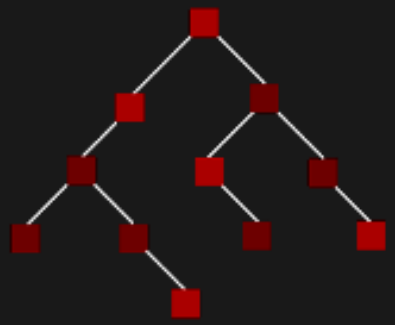
SATELLITE:	0.039 SEC	X 1.49
MNIST:	121.48 SEC	X 1.08
COVERTYPE:	17.416 SEC	X 1.95
POKERHAND:	3.392 SEC	X 1.08

OVERALL GRADE: A




MARIO
008200
x15
WORLD
1-1
TIME
370
5000

More decision trees and fast implementations?



mlpack

A fast C++ machine learning library with emphasis on flexibility and efficiency. Now with Python bindings! More implementation tricks there than what's been shown here.

`http://www.mlpack.org`
`https://github.com/mlpack/mlpack/`

Your BOSS is impressed with your IMPLEMENTATION. He says the great job you did has inspired him to give you much more WORK. It seems for now that your JOB is secure.

? _

Your BOSS is impressed with your IMPLEMENTATION. He says the great job you did has inspired him to give you much more WORK. It seems for now that your JOB is secure.

?

Your BOSS is impressed with your IMPLEMENTATION. He says the great job you did has inspired him to give you much more WORK. It seems for now that your JOB is secure.

? inventory

You possess the following TRICKS:

- Avoid unnecessary copies and memory allocations.
- Terminate the algorithm early if possible.
- Use parallelism when possible.
- Exploit SIMD or other processor-specific instructions.

There are many more TRICKS. Search the WORLD to find more.

? _

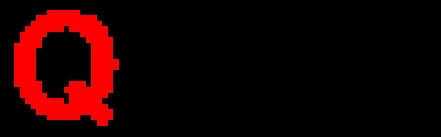
CONGRATULATIONS !

YOUR SCORE IS IN THE TOP FIVE !

INPUT YOUR INITIALS : _ _ _

CONGRATULATIONS !

YOUR SCORE IS IN THE TOP FIVE !

INPUT YOUR INITIALS : 

CONGRATULATIONS !

YOUR SCORE IS IN THE TOP FIVE !

INPUT YOUR INITIALS : Q N _

CONGRATULATIONS !

YOUR SCORE IS IN THE TOP FIVE !

INPUT YOUR INITIALS : Q N A

CONGRATULATIONS !

YOUR SCORE IS IN THE TOP FIVE !

INPUT YOUR INITIALS : Q N A

Ryan Curtin
ryan@ratml.org

http://www.mlpack.org/