A Machine Learning Analysis of Halting in the SKI Combinator Calculus

Euan Ong

Wolfram High School Summer Camp 2018

Much of machine learning is driven by the question: can we learn what we cannot compute? The learnability of the halting problem, the canonical undecidable problem, to an arbitrarily high accuracy for Turing machines was proven by Lathrop (Lathrop, 1996). The SKI combinator calculus can be seen as a reduced form of the untyped lambda calculus, which is Turing-complete (Turing, 1937); hence, the SKI combinator calculus forms a universal model of computation. In this vein, the growth and halting times of SKI combinator expressions is analysed and the feasibility of a machine learning approach to predicting whether a given SKI combinator expression is likely to halt is investigated.

Keywords: sample paper; formatting systems; equations and mathematical symbols; about the references

1. SK Combinators

What we will refer to as 'SK Combinators' are expressions in the SKI combinator calculus, a simple Turing-complete language introduced by Schönfinkel (1924) and Curry (1930). In the same way that NAND gates can be used to construct any expression in Boolean logic, SK combinators were posed as a way to construct any expression in predicate logic, and being a reduced form of the untyped lambda calculus, any functional programming language can be implemented by a machine that implements SK combinators. While implementations of this language exist, these serve little functional purpose - instead, this language, a simple idealisation of transformations on symbolic expressions [NKS], provides a useful tool for studying complex computational systems.

1.1 Rules and Expressions

Formally, SK combinator expressions are binary trees whose leaves are labelled either '*S*', '*K*' or '*I*': each tree *(xy)* represents a function *x* applied to an argument *y*. When the expression is evaluated (i.e. when the function is applied to the argument), the tree is transformed into another tree, the 'value'. The basic 'rules' for evaluating combinator expressions are given below:

k[x_][y_] := x

▲ The K combinator or 'constant function': when applied to *x*, returns the function *k[x]*, which when applied to some *y* will return *x*.

s[x_][y_][z_] := x[z][y[z]]

▲ The S combinator or 'fusion function': when applied to *x, y, z*, returns *x* applied to *z*, which is in turn applied to the result of *y* applied to *z*.

i[x_] := x

▲ The I combinator or 'identity function': when applied to *x*, returns *x*.

Note that the I combinator *I[x]* is equivalent to the function *S[K][a][x]*, as the latter will evaluate to the former in two steps:

S[K][a][x]
= K[x][a[x]]
= x

Thus the I combinator is redundant as it is simply 'syntactic sugar' - for the purposes of this exploration it will be ignored.

These rules can be expressed in the Wolfram Language as follows:

```
In[111]:= SKRules = {k[x_][y_] :→ x, s[x_][y_][z_] :→ x[z][y[z]]}
Out[111]= {k[x_][y_] :→ x, s[x_][y_][z_] :→ x[z][y[z]]}
```

1.2 Evaluation

The result of applying these rules to a given expression is given by the following functions:

```
In[112]:= SKNext[expr_] := expr /. SKRules;
```

▲ Returns the next ‘step’ of evaluation of the expression *expr* - evaluating all functions in *expr* according to the rules above without evaluating any ‘new’/transformed functions.

```
In[113]:= SKEvaluate[expr_, n_] := NestList[#1 /. SKRules &, expr, n];
```

▲ Returns the next *n* steps of evaluation of the expression *expr*

```
In[114]:= SKEvaluateUntilHalt[expr_, n_]  := FixedPointList[SKNext, expr, n + 1];
```

▲ Returns the steps of evaluation of *expr* until either it reaches a fixed point or it has been evaluated for n steps, whichever comes first.

Note that, due to the Church-Rosser theorem, the order in which the rules are applied does not affect the final result, as long as the combinator evaluates to a fixed point / ‘halts’. For combinators with no fixed point, which do not halt, the behaviour demonstrated as they evaluate could change based on the order of application of the rules - this is not explored here and is a topic for potential future investigation.

1.3 Examples

The functions above can be used to evaluate a number of interesting SK combinator expressions:

```
In[5]:= Column[SKEvaluateUntilHalt[s[k][a][x], 10][[1 ;; -2]]]

         s[k][a][x]
Out[5]=  k[x][a[x]]
         x
```

▲ The *I* combinator

```
In[6]:= Column[SKEvaluateUntilHalt[s[k[s[i]]][k][a][b], 10][[1 ;; -2]]]

         s[k[s[i]]][k][a][b]
         k[s[i]][a][k[a]][b]
Out[6]=  s[i][k[a]][b]
         i[b][k[a][b]]
         i[b][a]
```

▲ The reversal expression - *s[k][s[i]][k][a][b]* takes two terms, *a* and *b*, and returns *b[a]*.

2. Growth and Halting

2.1 Halting and Related Works

We will define a combinator expression to have halted if it has reached a fixed point - i.e. if no combinators in the expression can be evaluated, or if evaluating any of the combinators in the expression returns the original expression. As SK combinators are Turing-complete and so computationally universal, it is evident that the halting problem - determining whether or not a given SK combinator expression will halt - is undecidable for SK combinators. There are, however, patterns and trends in the growth of SK combinators, and it is arguably possible to speak of the probability of a given SK combinator expression halting.

Some investigations (https://pdfs.semanticscholar.org/856c/9986e9e4b1dc13bf9c1e2938c7d32d62d1ff.pdf, https://www.cs.auckland.ac.nz/~cristian/crispapers/AnytimeHP.pdf) have been made into probabilistically determining the halting time of Turing machines, with [1] proving that it is possible to compute some value K where for some arbitrary predetermined confidence *(1-δ)* and accuracy *(1-ϵ),* a program that does

A. Input a Turing machine M and program I.
B. Simulate M on I for K steps.
C. If M has halted then print 1, else print 0.
D. Halt.

has a probability greater than *(1-δ)* of having an accuracy (when predicting whether or not a program will halt) greater than *(1-ϵ)*. The key result of this is that, in some cases ‘we can learn what we cannot compute’ - ‘learning’ referring to Valiant’s formal analysis as ‘the phenomenon of knowledge acquisition in the absence of specific programming’.

2.2 Definitions and Functions

The size of a combinator expression can either be measured by its length (total number of characters including brackets) or by its leaf size (number of ‘s’ and ‘k’ characters). We use the former in most cases, and the latter when randomly generating combinator expressions.

The number of possible combinator expressions with leaf size *n* is given by

```
In[7]:= SKPossibleExpressions[n_] := (2^n) * Binomial[2 * (n - 2), n - 1] / n
```

(NKS), which grows exponentially.

2.2.1 Visualisation

We define a function to visualise the growth of a combinator, *SKRasterize*:

```
In[115]:= SKArray[expr_, n_] := Characters /@ ToString /@ SKEvaluate[expr, n];
          SKArray[expr_] := SKArray[expr, 10];
```

▲ Generates a list of the steps in the growth of a combinator, where each expression is itself a list of characters (‘s’, ‘k’, ‘[‘, ‘]’)

```
In[117]:= SKGrid[exp_, n_] := ArrayPlot[SKArray[exp, n],
             {ColorRules → {"s" → RGBColor[1, 0, 0], "k" → RGBColor[0, 1, 0],
                "[" → RGBColor[0, 0, 1], "]" → RGBColor[0, 0, 0]},
              PixelConstrained → True, Frame → False, ImageSize → 1000}];
          SKGrid[exp_] := SKGrid[exp, 10];
```

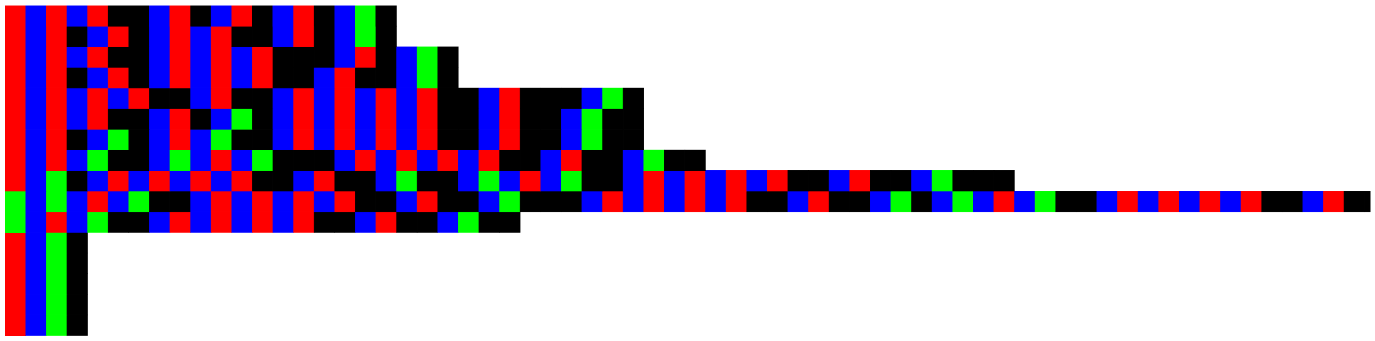
▲ Generates an ArrayPlot of a list given by SKArray, representing the growth of a combinator in a similar manner to that of cellular automata up to step n. The y axis represents time - each row is the next expression in the evaluation of an SK combinator. Red squares indicate ‘S’, blue squares indicate ‘K’, green squares indicate ‘[‘ and black squares indicate ‘]’.

```
In[119]:= SKRasterize[func_, n_] := Image[SKGrid[func, n][[1]]];
          SKRasterize[func_] := SKRasterize[func, 10];
```

▲ Generates a rasterized version of the ArrayPlot.

A visualisation of a given combinator can easily be produced, as follows:

```
In[14]:= SKRasterize[s[s[s]][s][s][s][k], 15]
```

Out[14]=


▲ The longest running halting expression with leaf size 7, halting in 12 steps (NKS http://www.wolframscience.com/nks/notes-11-12--combinator-properties/)

2.2.2 Halting graphs

We can create a table of the length (string length) of successive combinator expressions as they evaluate as follows:

```
In[15]:= SKLengths[exp_, n_] := StringLength /@ ToString /@ SKEvaluate[exp, n];
```

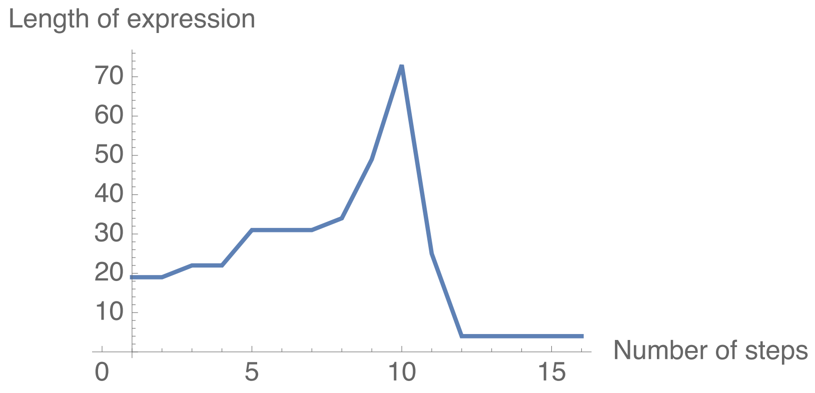
▲ Returns a list of the lengths of successive expressions until step *n*

These can be plotted as a graph (x axis :

```
In[16]:= SKPlot[expr_, limit_] := ListLinePlot[SKLengths[expr, limit],
            AxesOrigin → {1, 0},
            AxesLabel → {"Number of steps", "Length of expression"}];
```

Thus, a graph of the above combinator can be produced:

```
In[17]:= SKPlot[s[s[s]][s][s][s][k], 15]
```

Out[17]=


It is evident from the graph that this combinator halts at 12 steps.

2.2.3 Random SK combinators

To empirically study SK combinators, we need a function to randomly generate them. Two methods to do this were found:

```
In[18]:= RecursiveRandomSKExpr[0, current_] := current;
         RecursiveRandomSKExpr[depth_, current_] :=
           RecursiveRandomSKExpr[depth - 1,
            RandomChoice[{
              RandomChoice[{s, k}][current],
              current[RecursiveRandomSKExpr[depth - 1, RandomChoice[{s, k}]]]
             }]
           ];
         RecursiveRandomSKExpr[depth_Integer] :=
           RecursiveRandomSKExpr[depth, RandomChoice[{s, k}]];
```

▲ A recursive method, repeatedly appending either a combinator to the ‘head’ of the expression or a randomly generated combinator expression to the ‘tail’ of the expression. (Hennigan)

```
In[132]:= replaceWithList[expr_, pattern_, replaceWith_] :=
            ReplacePart[expr, Thread[Position[expr, pattern] → replaceWith]];
          treeToFunctions[tree_] := ReplaceRepeated[tree, {x_, y_} :→ x[y]];
          randomTree[leafCount_] :=
            Nest[ReplacePart[#, RandomChoice[Position[#, x]] → {x, x}] &,
             {x, x}, leafCount - 2];
          RandomSKExpr[leafCount_] :=
            treeToFunctions[replaceWithList[randomTree[leafCount], x,
              RandomChoice[{s, k}, leafCount]]];
```

▲ Random combinator generation based on generation of binary trees - each combinator can be expressed as a binary tree with leaves ‘S’ or ‘K’. (http://community.wolfram.com/groups/-/m/t/965400)

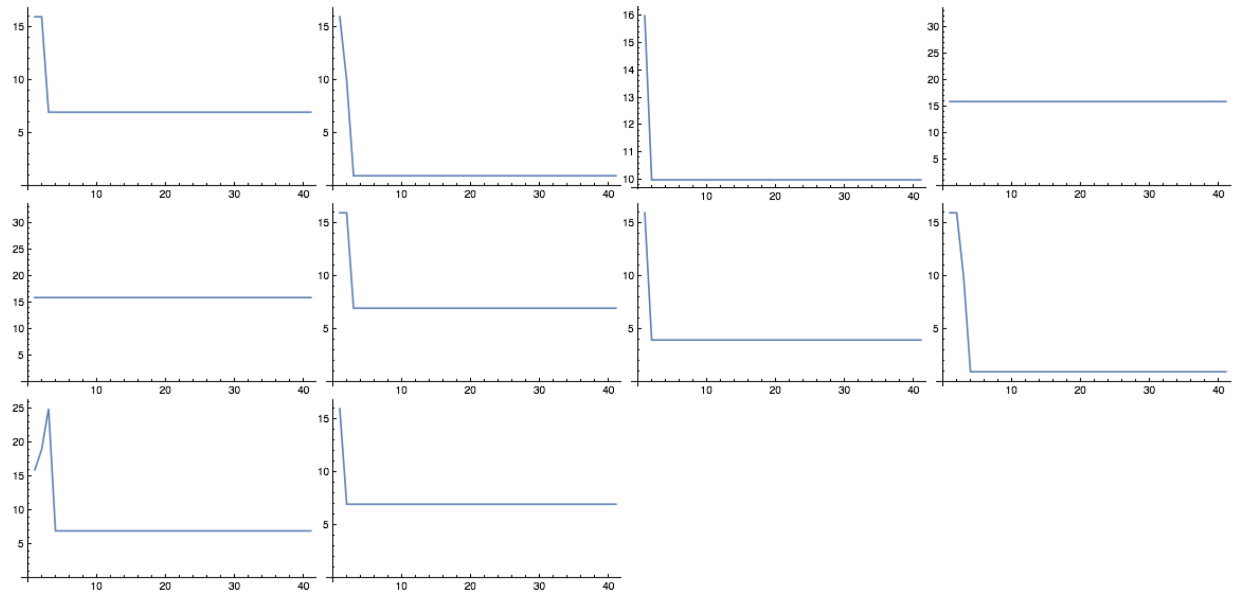
While the first method gives a large spread of combinators with a variety of lengths, and is potentially more efficient, for the purposes of this exploration the second is more useful, as it limits the combinators generated to a smaller, more controllable sample space (for a given leaf size).

2.3 Halting Graphs

All combinators of leaf sizes up to size 6 evolve to fixed points (NKS):

```
In[25]:= exprs = Table[RandomSKExpr[6], 10];
         ImageCollage[Table[ListLinePlot[SKLengths[exprs[[n]], 40]], {n, 10}],
          Background → White]
```

Out[26]=

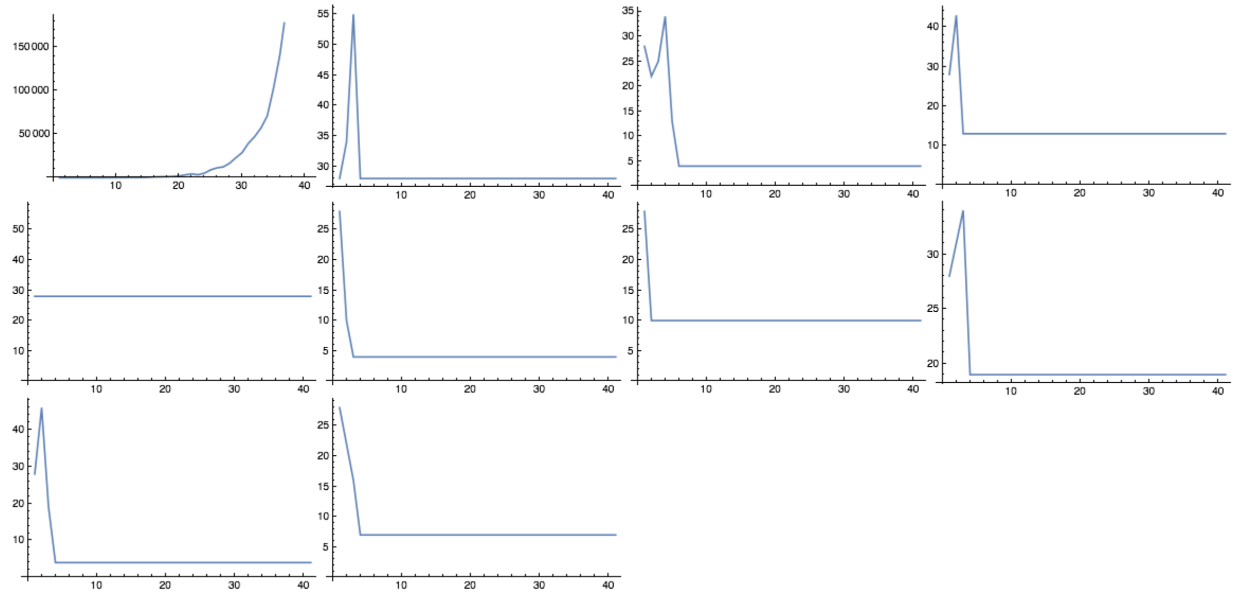


▲ 10 randomly generated combinators of size 6, with their lengths plotted until n=40.

As the leaf size increases, combinators take longer to halt, and some show exponential growth:

```
In[58]:= exprs = Table[RandomSKExpr[10], 10];
         ImageCollage[Table[ListLinePlot[SKLengths[exprs[[n]], 40]], {n, 10}],
          Background → White]
```

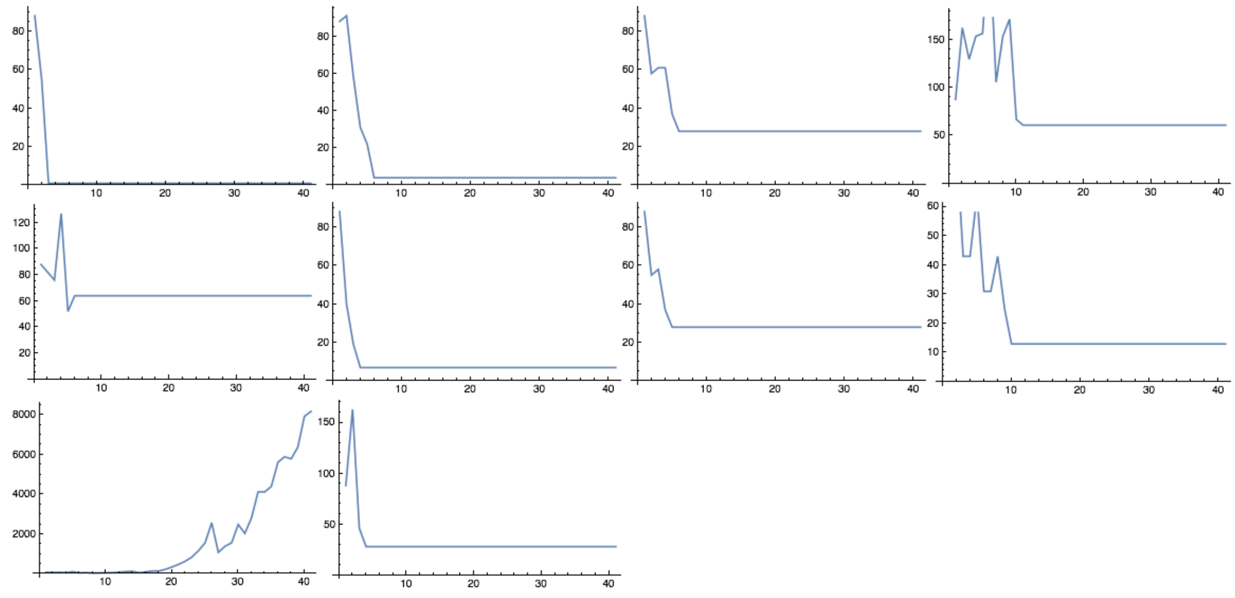
Out[59]=



▲ 10 randomly generated combinators of size 10, with their lengths plotted until n=20.

```
In[64]:= exprs = Table[RandomSKExpr[30], 10];
         ImageCollage[Table[ListLinePlot[SKLengths[exprs[[n]], 40]], {n, 10}],
          Background → White]
```

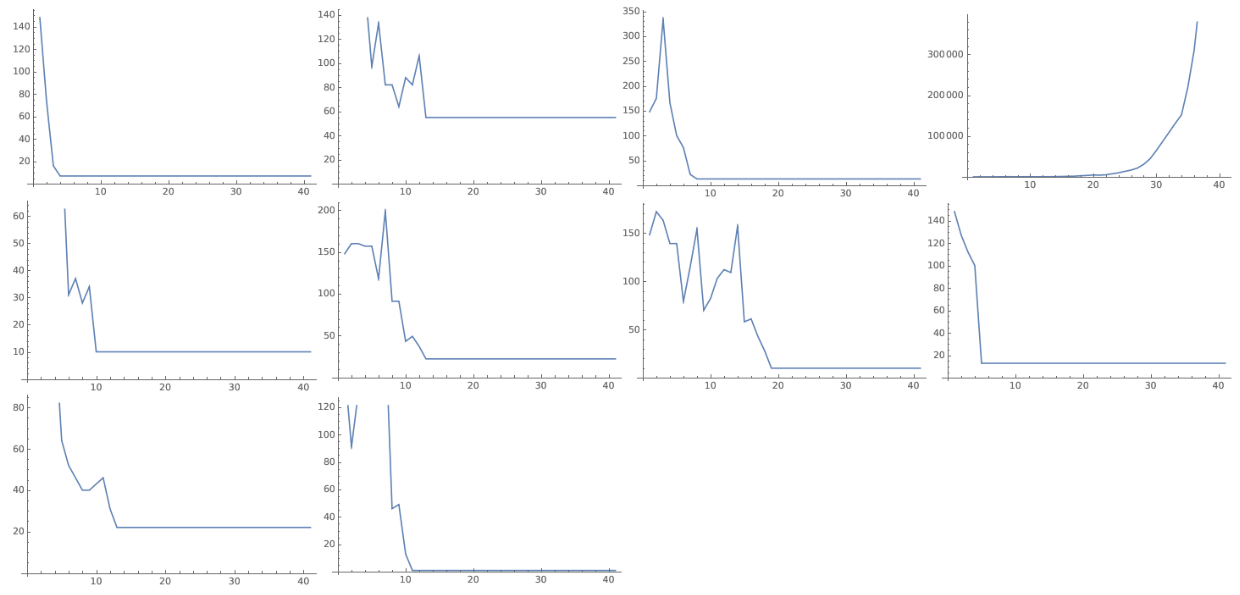
Out[65]=



▲ 10 randomly generated combinators of size 30, with their lengths plotted until n=40.

```
In[70]:= CloudEvaluate[exprs = Table[RandomSKExpr[50], 10];
          ImageCollage[Table[ListLinePlot[SKLengths[exprs[[n]], 40]], {n, 10}],
           Background → White]]
```

Out[70]=



▲ 10 randomly generated combinators of size 50, with their lengths plotted until n=40.

After evaluating a number of these combinators, it appears that they tend to either halt or grow exponentially - some sources (citation needed) reference linear growth combinators, however none of these have been encountered as yet.

2.4 Halting Times

With a random sample of combinators, we can plot a cumulative frequency graph of the number of combinators that have halted at a given number of steps:

```
In[27]:= SKHaltLength[expr_, n_] := Module[{x},
           x = Length[SKEvaluateUntilHalt[expr, n + 1]];
           If[x > n, False, x]
          ]
```

▲ Returns the number of steps it takes the combinator *expr* to halt; if *expr* does not halt within n steps, returns *False*.

```
In[28]:= GenerateHaltByTable[depth_, iterations_, number_] :=
          Module[{exprs, lengths},
           exprs = Monitor[Table[RandomSKExpr[depth], {n, number}], n];
           lengths =
            Monitor[Table[SKHaltLength[exprs[[n]], iterations], {n, number}], n];
           Return[lengths]
          ]
```

▲ Generates a table of the halt lengths of *number* random combinator expressions (*False* if they do not halt within *iterations* steps) with leaf size *depth*.

```
In[29]:= GenerateHaltData[depth_, iterations_, number_] :=
          Module[{haltbytable, vals},
           haltbytable = GenerateHaltByTable[depth, iterations, number];
           vals = BinCounts[Sort[haltbytable], {1, iterations + 1, 1}];
           Table[Total[vals[[1 ;; n]]], {n, 1, Length[vals]}]
          ]
```

▲ Generates a table of the number of *number* random combinator expressions (*False* if they do not halt within *iterations* steps) with leaf size *depth* that have halted after a given number of steps

```
In[30]:= GenerateHaltGraph[depth_, iterations_, number_] :=
          Module[{cumulative, f},
           cumulative = GenerateHaltData[depth, iterations, number];
           f = Interpolation[cumulative];
           {ListLinePlot[cumulative, PlotRange → {Automatic, {0, number}},
             GridLines → {{}, {number}},
             Epilog →
              {Red, Dashed,
               Line[{{0, cumulative[[-1]]}, {number, cumulative[[-1]]}}]},
             AxesOrigin → {1, 0},
             AxesLabel → {"Number of steps", "Number of combinators halted"}],
            cumulative[[-1]]}
          ]
```

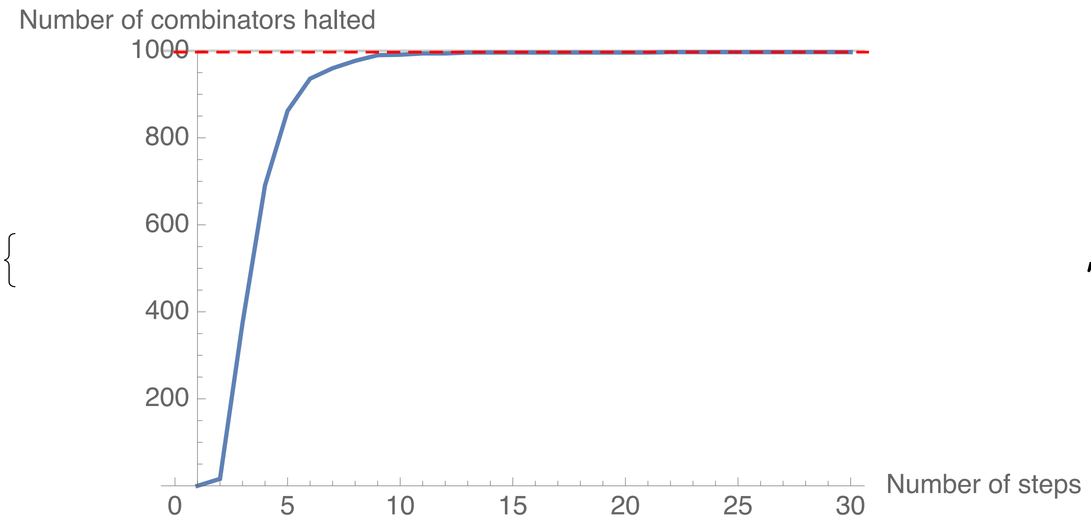
▲ Plots a graph of the above data.

2.4.1 Halting Graphs

We analyse halt graphs of random samples of 1000 combinators (to depth 30):

In[80]:= **CloudEvaluate[GenerateHaltGraph[10, 30, 1000]]**

Out[80]= {

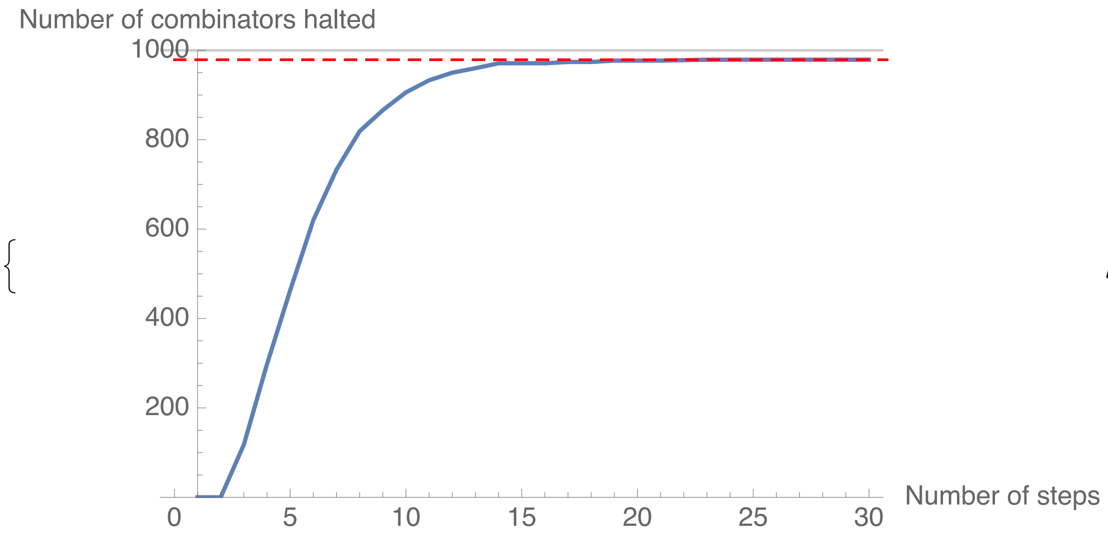


, 997}

▲ Leaf size 10: almost all combinators in the sample (997) have halted (99.7%).

In[81]:= **CloudEvaluate[GenerateHaltGraph[20, 30, 1000]]**

Out[81]= {

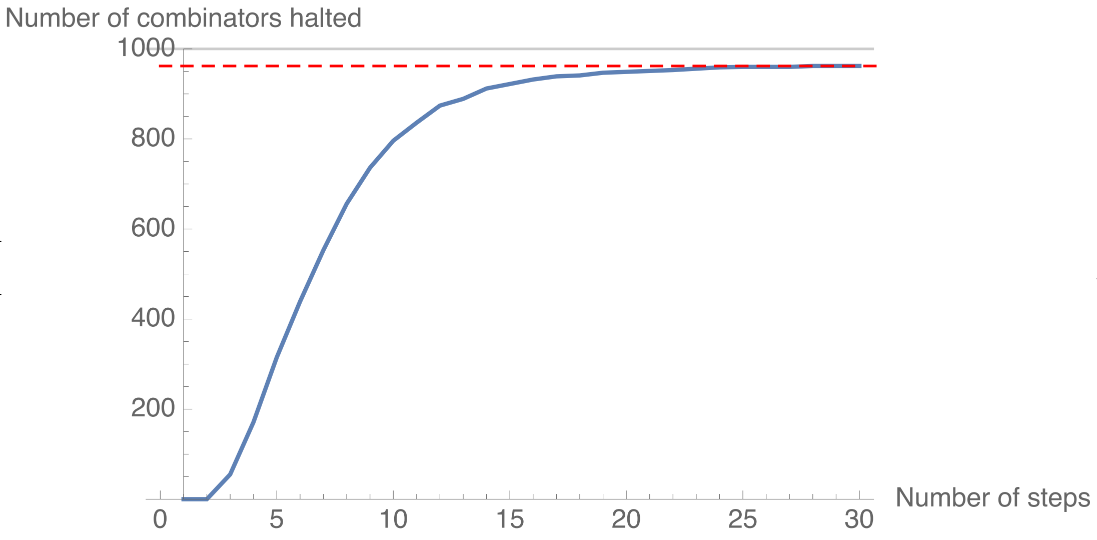


, 979}

▲ Leaf size 20: 979 combinators in the sample have halted (97.9%).

```
In[82]:= CloudEvaluate[GenerateHaltGraph[30, 30, 1000]]
```

Out[82]= {

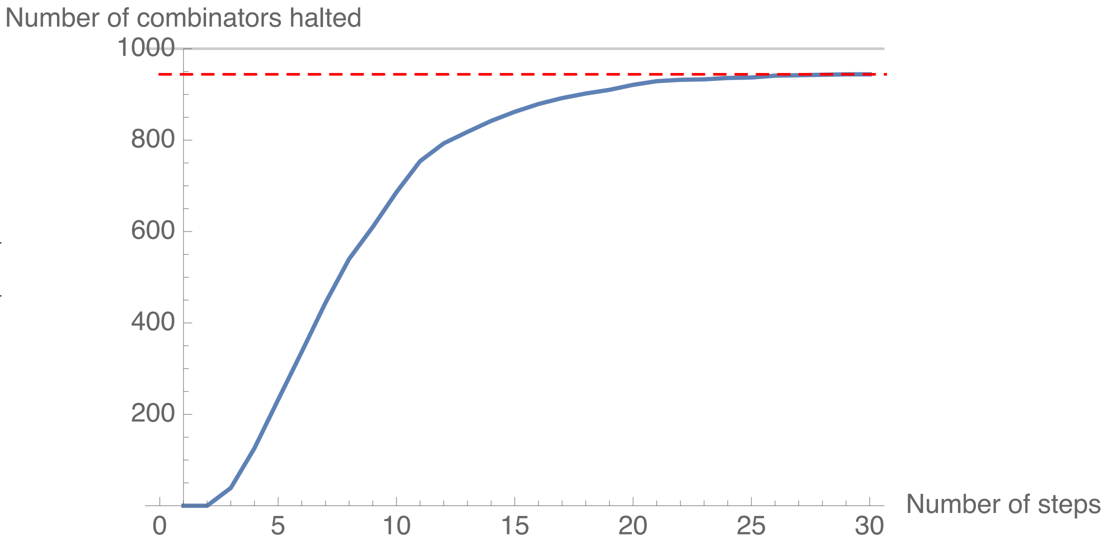


```
, 962}
```

▲ Leaf size 30: 962 combinators in the sample have halted (96.2%).

```
In[83]:= CloudEvaluate[GenerateHaltGraph[40, 30, 1000]]
```

Out[83]= {

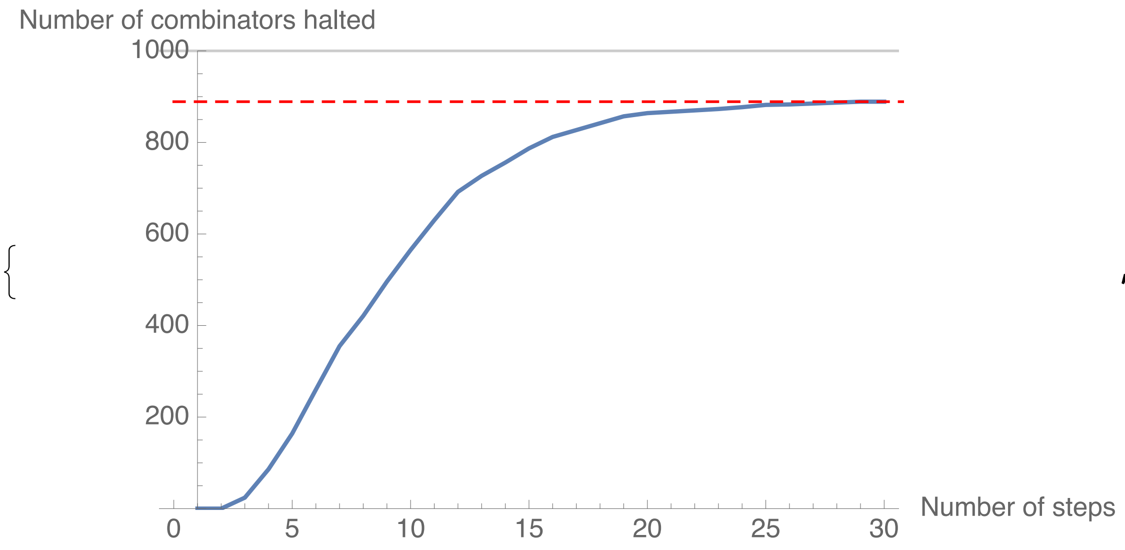


```
, 944}
```

▲ Leaf size 40: 944 combinators in the sample have halted (94.4%).

```
In[85]:= CloudEvaluate[GenerateHaltGraph[50, 30, 1000]]
```

Out[85]= {



, 889}

▲ Leaf size 50: 889 combinators in the sample have halted (88.9%).

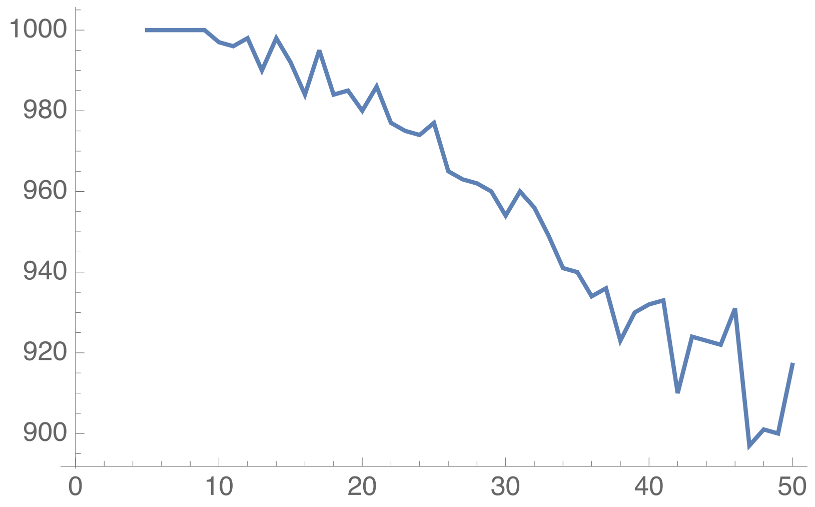
Evidently, the rate of halting of combinators in the sample decreases as number of steps increases - the gradient of the graph is decreasing. As the graph levels out at around 30 steps, we will assume that the number of halting combinators will not increase significantly beyond this point.

As the leaf size increases, fewer combinators in the sample have halted by 30 steps - however, the graph still levels out, suggesting most of the combinators which have not halted by this point will never halt.

2.4.2 Halting Times and Leaf Size

We can plot a graph of the number of halted combinators against leaf size:

```
In[87]:= CloudEvaluate[
          ListLinePlot[Table[{n, GenerateHaltGraph[n, 30, 1000][[2]]},
            {n, 5, 50, 1}]]]
```



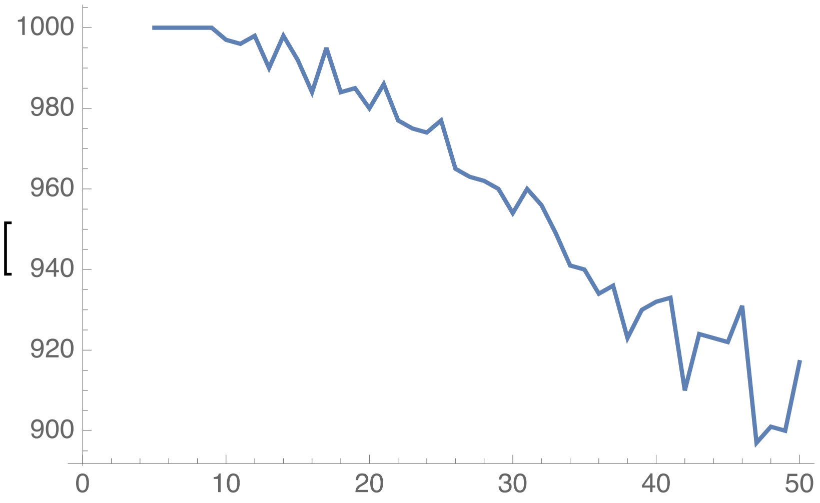
▲ A graph to show the number of combinators which halt within 30 steps in each of 45 random samples of 1000 combinators, with leaf size varying from 5 to 50.

This graph shows that, despite random variation, the number of halted combinators decreases as the leaf size increases: curve fitting suggests that this follows a negative quadratic function.

```
In[31]:= FitData[data_, func_] :=
          Module[{fitd}, fitd = {Fit[data[[1, 2, 3, 4, 1]], func, x]};
           {fitd, Show[ListPlot[data[[1, 2, 3, 4, 1]], PlotStyle -> Red],
             Plot[fitd, {x, 5, 50}]]}]
```

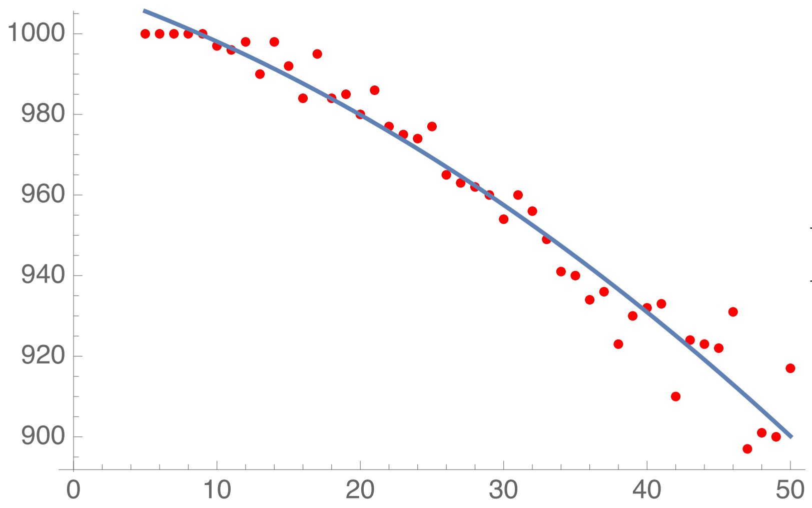
▲ A curve-fitting function: plots the curve of best fit for *data* with some combination of functions *func*.

In[134]:= **FitData**[



, {1, x, x^2}]

Out[134]= $\{\{1012.07 - 1.18915\,x - 0.0209805\,x^2\},$



$\}$

▲ Curve-fitting on the data with a quadratic function yields a reasonably accurate curve of best fit.

3. Machine Learning Analysis of SK Combinators

The graphs above suggest that the majority of halting SK combinators with leaf size <=50 will halt before ~30 steps. Thus we can state that, for a randomly chosen combinator, it is likely that if it does not halt before 40 steps, it will never halt. Unfortunately a lack of time prohibited a formal analysis of this, in the vein of Lathrop’s work - this is an area for future research.

We attempt to use modern machine learning methods to predict the likelihood of a given SK combinator expression halting before 40 steps:

3.1 Dataset Generation

We implement a function *GenerateTable* to produce tables of random SK expressions:

```
In[81]:= SKHaltLength[expr_, n_] := Module[{x},
           x = Length[SKEvaluateUntilHalt[expr, n + 1]];
           If[x > n, False, x]
          ]
```

▲ Returns the number of steps *expr* takes to halt if the given expression *expr* halts within the limit given (*limit*), otherwise returns *False*

```
In[32]:= GenerateTable[depth_, iterations_, number_] := Module[{exprs, lengths},
           exprs = Monitor[Table[RandomSKExpr[depth], {n, number}], n];
           lengths =
            Monitor[Table[exprs[[n]] → SKHaltLength[exprs[[n]], iterations],
              {n, number}], n];
           lengths = DeleteDuplicates[lengths];
           Return[lengths]
          ]
```

▲ Returns a list of *number* expressions with leaf size *depth* whose elements are associations with key *expression* and value *number of steps taken to halt* if the expression halts within *iterations* steps, otherwise *False*.

GenerateTable simply returns tables random SK expressions - as seen earlier, these tend to be heavily skewed datasets as around 90% of random expressions generated will halt. Thus we must process this dataset to create a balanced training dataset: this is done with the function *CreateTrainingData*:

```
CreateTrainingData[var_] := Module[{NoHalt, Halt, HaltTrain, Train},
   NoHalt = Select[var, #[[2]] == False &];
   Halt = Select[var, #[[2]] == True &];
   HaltTrain = RandomSample[Halt, Length[NoHalt]];
   Train = Join[HaltTrain, NoHalt];
   Return[Train]
  ];
```

▲ Counts the number of non-halting combinators in *var* (assumption is this is less than number of halting combinators), selects a random sample of halting combinators of this length and concatenates the lists.

```
ConvertSKTableToString[sktable_] :=
  Table[ToString[sktable[[n, 1]]] → sktable[[n, 2]],
   {n, 1, Length[sktable]}];
```

▲ Converts SK expressions in a table generated with *GenerateTable* to strings

We also implement a function to create rasterised training data (where instead of an individual SK combinator associated with either True or False, an image of the first 5 steps of evaluation of the combinator is associated with either True or False):

```
CreateRasterizedTrainingData[var_] :=
  Module[{NoHalt, Halt, HaltTrain, HaltTrainRaster, NoHaltTrainRaster,
    RasterTrain},
   NoHalt = Select[var, #[[2]] == False &];
   Halt = Select[var, #[[2]] == True &];
   HaltTrain = RandomSample[Halt, Length[NoHalt]];
   HaltTrainRaster =
    Monitor[Table[SKRasterize[HaltTrain[[x, 1]], 5] → HaltTrain[[x, 2]],
      {x, 1, Length[HaltTrain]}], x];
   NoHaltTrainRaster =
    Monitor[Table[SKRasterize[NoHalt[[x, 1]], 5] → NoHalt[[x, 2]],
      {x, 1, Length[NoHalt]}], x];
   RasterTrain = Join[HaltTrainRaster, NoHaltTrainRaster];
   Return[RasterTrain]
  ];
```

▲ Counts the number of non-halting combinators in *var* (assumption is this is less than number of halting combinators), selects a random sample of halting combinators of this length, evaluates and generates images of both halting and non-halting combinators and processes them into training data (image->True/False).

3.2 Markov Classification

3.2.1 Training

As a first attempt, we generate 2000 random SK expressions with depth 5, 2000 expressions with depth 10 ... 2000 expressions with depth 50, evaluated up to 40 steps:

```
lengths = Flatten[Table[GenerateTable[n, 40, 2000], {n, 5, 50, 5}]]
```

We convert all non-False halt lengths to 'True':

```
In[89]:= lengths = lengths /. (a_ → b_) /; ! (b === False) ⧴ (a → True);
```

We process the data and train a classifier using the Markov method:

```
In[90]:= TrainingData = CreateTrainingData[lengths];
         TrainingData2 = ConvertSKTableToString[TrainingData];

         HaltClassifier1 = Classify[TrainingData2, Method → "Markov"]

Out[92]= ClassifierFunction[ Input type: Text
                              Classes: False, True ]
```

3.2.2 Testing

We must now generate test data, using the same parameters for generating random combinators:

```
testlengths = Flatten[Table[GenerateTable[n, 40, 2000], {n, 5, 50, 5}]]
testlengths = testlengths /. (a_ → b_) /; ! (b === False) :→ (a → True);
TestData = CreateTrainingData[testlengths];
TestData2 = ConvertSKTableToString[TestData];
```

The classifier can now be assessed for accuracy using this data:

```
In[101]:= TestClassifier1 = ClassifierMeasurements[HaltClassifier1, TestData2]

Out[101]= ClassifierMeasurementsObject[ Classifier: Markov
                                        Number of test examples: 1454 ]
```

3.2.3 Evaluation

A machine learning solution to this problem is only useful if the accuracy is greater than 0.5 (i.e. more accurate than a random coin flip). We test the accuracy of the classifier:

```
In[102]:= TestClassifier1["Accuracy"]

Out[102]= 0.755158
```

This, while not outstanding, is passable for a first attempt. We find the training accuracy:

```
In[103]:= ClassifierInformation[HaltClassifier1]
```

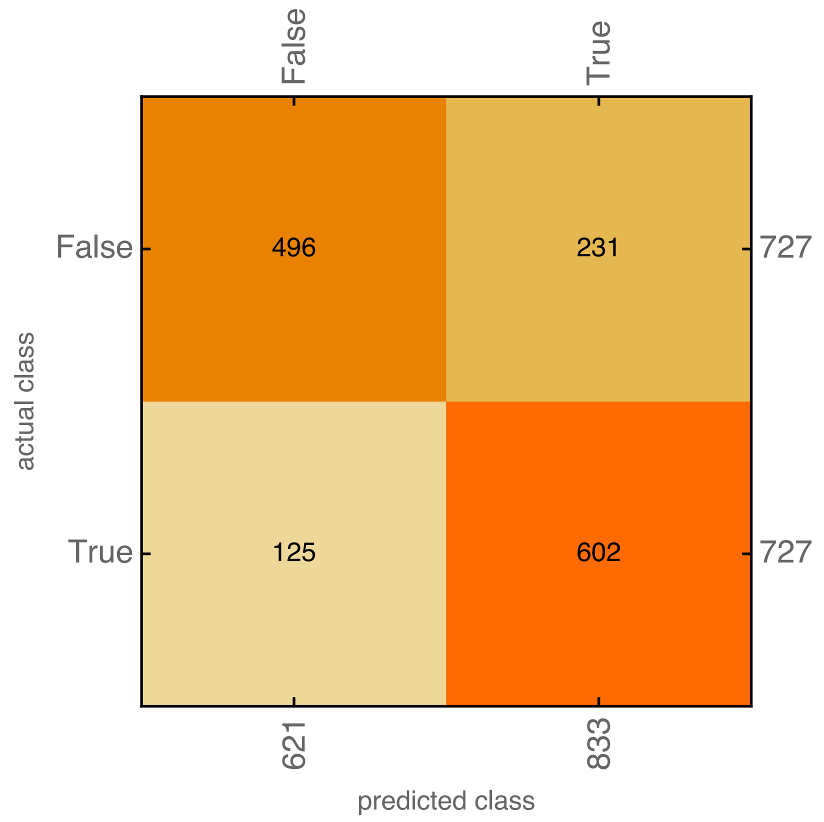
```
Classifier information

                 Input type | Text
                    Classes | False, True
                     Method | Markov
                   Accuracy | 71.3% ± 3.8%
                       Loss | 0.539 ± 0.033
     Single evaluation time | 7.15 ms/example
     Batch evaluation speed | 699. examples/s
          Classifier memory | 134. kB
    Training examples used  | 1456 examples
              Training time | 14.6 s
```

Out[103]=

The training accuracy (71.3%) is slightly lower than the testing accuracy (75.5%) - this is surprising, and is probably due to a ‘lucky’ testing dataset chosen.

We calculate some statistics from a confusion matrix plot:

```
In[104]:= TestClassifier1["ConfusionMatrixPlot"]
```

Out[104]=


Accuracy: 0.76
Misclassification rate: 0.24

Precision (halt): 0.722 (when ‘halt’ is predicted, how often is it correct?)
True Positive Rate: 0.83 (when the combinator halts, how often is it classified as halting?)
False Positive Rate: 0.32 (when the combinator doesn’t halt, how often is it classified as halting?)

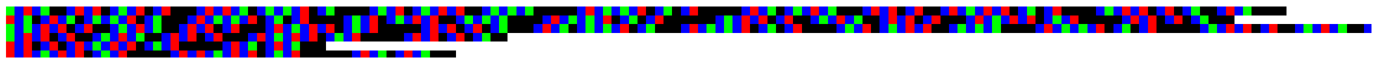
Precision (non-halt): 0.799 (when ‘non halt’ is predicted, how often is it correct?)
True Negative Rate: 0.68 (when the combinator doesn’t halt, how often is it classified as not halting?)
False Negative Rate: 0.17 (when the combinator halts, how often is it classified as not halting?)

A confusion matrix plot shows that the true positive rate is larger than the true negative rate - this would suggest that it is easier for the model to tell when an expression halts than when an expression does not halt. This could be due to the model detecting features suggesting very short run time in the initial string - for instance, a combinator k[k][<expression>] would evaluate immediately to k and halt - however, these ‘obvious’ features are very rare.

3.3 Random Forest Classification on Rasterised Expression Images

Analysing strings alone, without any information about how they are actually structured or how they might evaluate, could well be a flawed method - one might argue that, in order to predict halting, one would need more information about how the program runs. Hence, another possible method is to generate a dataset of visualisations of the first 5 steps of a combinator’s evaluation like so:

```
In[136]:= SKRasterize[RandomSKExpr[50], 5]
```

Out[136]=


and feed these into a machine learning model. Although it might seem that this method is pointless - we are already evaluating the combinators to 5 steps, and we are training a model on a database of combinators evaluated to 40 steps to predict if a combinator will halt in <=40 steps, the point of the exercise is less to create a useful resource than to investigate the feasibility of applying machine learning to this type of problem. If more computational power was available, a dataset of combinators evaluated to 100 steps (when even more combinators will have halted) could be created: in such a case a machine learning model to predict whether or not a combinator will halt in <=100 steps would be a practical approach as the time taken to evaluate a combinator to 100 steps is exponentially longer than that taken to evaluate a combinator to 5 steps.

3.3.1 Training

We generate a dataset of 2000 random SK expressions with depth 5, 2000 expressions with depth 10 ... 2000 expressions with depth 50, evaluated up to 40 steps:

```
rasterizedlengths =
  Flatten[Table[GenerateTable[n, 40, 2000], {n, 5, 50, 5}]];
```

In order to train a model on rasterised images, we must evaluate all SK expressions in the dataset to 5 steps and generate rasterised images of these:

```
RasterizedTrainingData =
  CreateRasterizedTrainingData[rasterizedlengths];
```

We then train a classifier on this data:

```
RasterizeClassifier = Classify[RasterizedTrainingData,
  Method → "RandomForest"]
```

```
Out[123]= ClassifierFunction[ Input type: Image
                              Classes: False, True ]
```

3.3.2 Testing

We must now generate test data, using the same parameters for generating random training data:

```
testrasterizedlengths =
  Flatten[Table[GenerateTable[n, 40, 2000], {n, 5, 50, 5}]];
testrasterizedlengths =
  testrasterizedlengths /. (a_ → b_) /; ! (b === False) ⧴ (a → True);
TestRasterizedData = CreateRasterizedTrainingData[
   testrasterizedlengths];
```

The classifier can now be assessed for accuracy using this data:

```
TestRasterizeClassifier =
 ClassifierMeasurements[RasterizeClassifier, TestRasterizedData]
```

```
Out[129]= ClassifierMeasurementsObject[ Classifier: RandomForest
                                        Number of test examples: 1454 ]
```

3.3.3 Evaluation

A machine learning solution to this problem is only useful if the accuracy is greater than 0.5 (i.e. more accurate than a random coin flip). We test the accuracy of the classifier:

```
In[137]:= TestRasterizeClassifier["Accuracy"]
Out[137]= 0.876891
```

This is significantly better than the Markov approach (75.5%). We find the training accuracy:

```
In[138]:= ClassifierInformation[RasterizeClassifier]
```

```
Classifier information

               Input type | Image
                  Classes | False, True
                   Method | RandomForest
                 Accuracy | 85.5% ± 3.0%
                     Loss | 0.409 ± 0.027
   Single evaluation time | 6.17 ms/example
  Batch evaluation speed  | 735. examples/s
        Classifier memory | 332. kB
  Training examples used  | 1456 examples
            Training time | 9.83 s
```

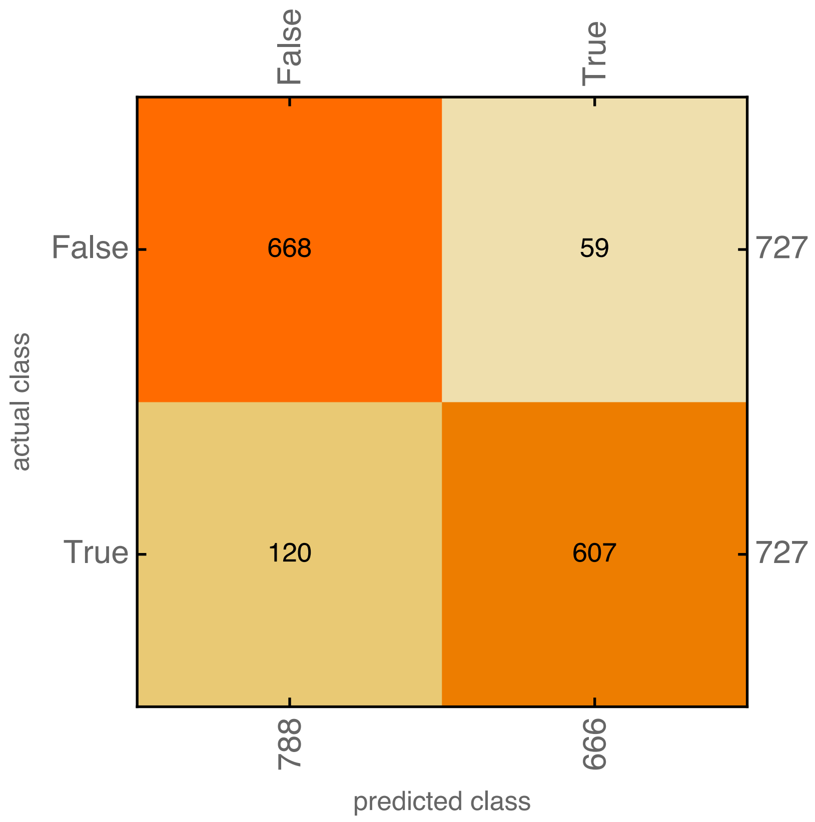
Out[138]=

Again, the training accuracy (85.5%) is slightly lower than the testing accuracy (87.7%).

We calculate some statistics from a confusion matrix plot:

```
In[139]:= TestRasterizeClassifier["ConfusionMatrixPlot"]
```

Out[139]=

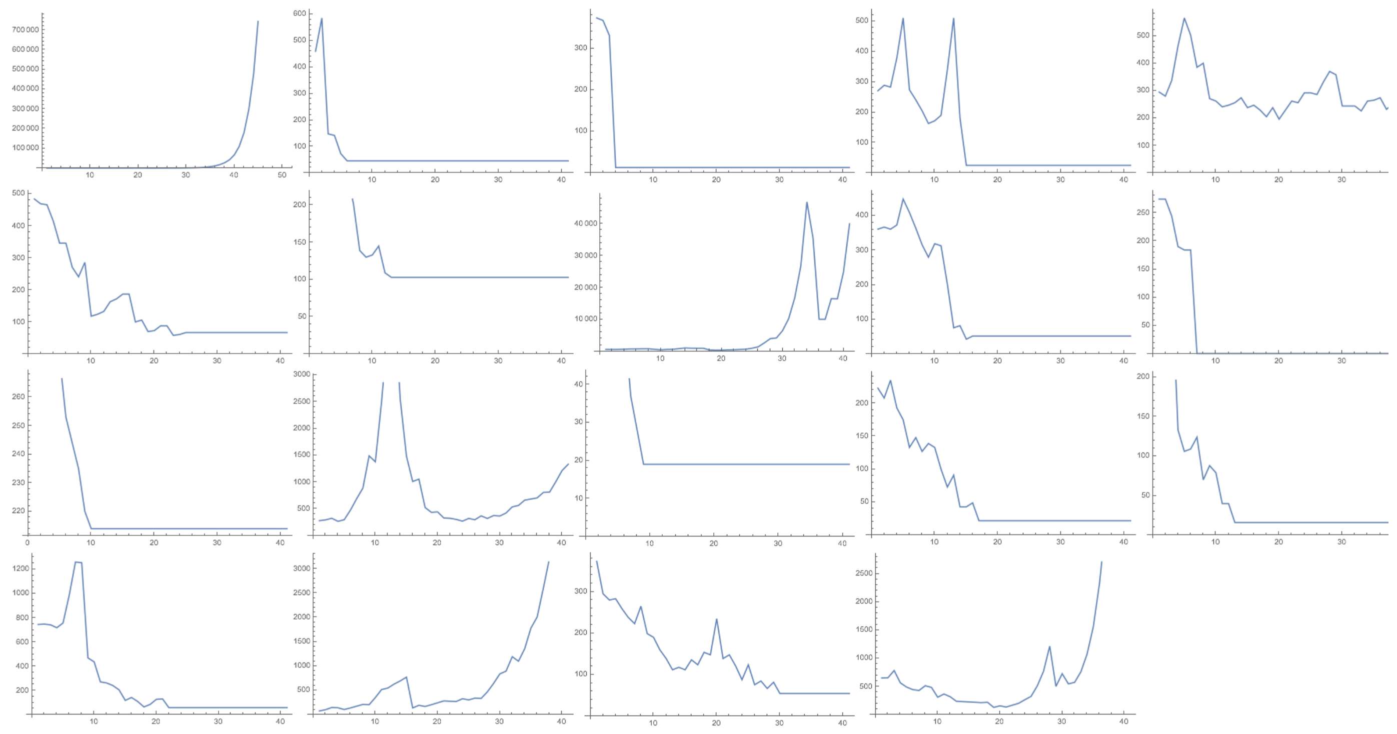


Accuracy: 0.88
Misclassification rate: 0.12

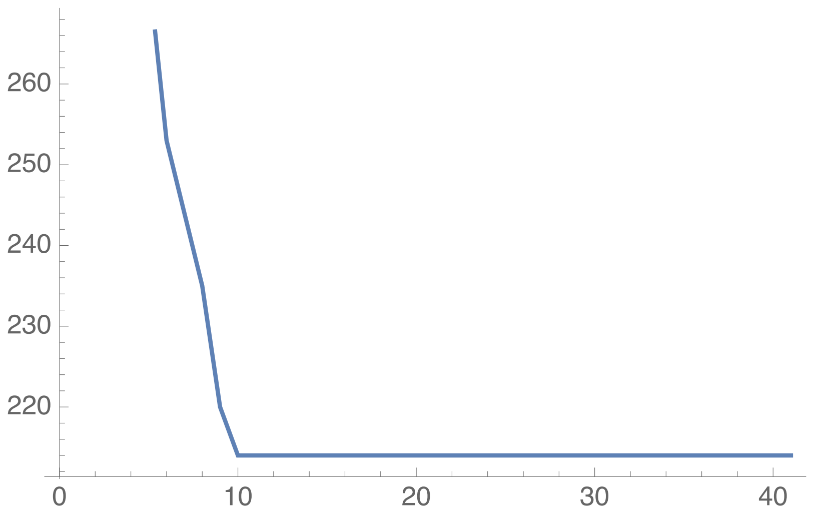
Precision (halt): 0.911 (when ‘halt’ is predicted, how often is it correct?)
True Positive Rate: 0.83 (when the combinator halts, how often is it classified as halting?)
False Positive Rate: 0.08 (when the combinator doesn’t halt, how often is it classified as halting?)

Precision (non-halt): 0.848 (when ‘non halt’ is predicted, how often is it correct?)
True Negative Rate: 0.92 (when the combinator doesn’t halt, how often is it classified as not halting?)
False Negative Rate: 0.17 (when the combinator halts, how often is it classified as not halting?)

A confusion matrix plot shows that the false negative rate is larger than the false positive rate - this would suggest that it is easier for the model to tell when an expression halts than when an expression does not halt. The precision for halting is much higher than the precision for non-halting, indicating that if the model suggests a program will halt, this is much more likely to be correct than if it suggested that the program would not halt. An (oversimplified) way to look at this intuitively is to examine some graphs of lengths of random combinators:

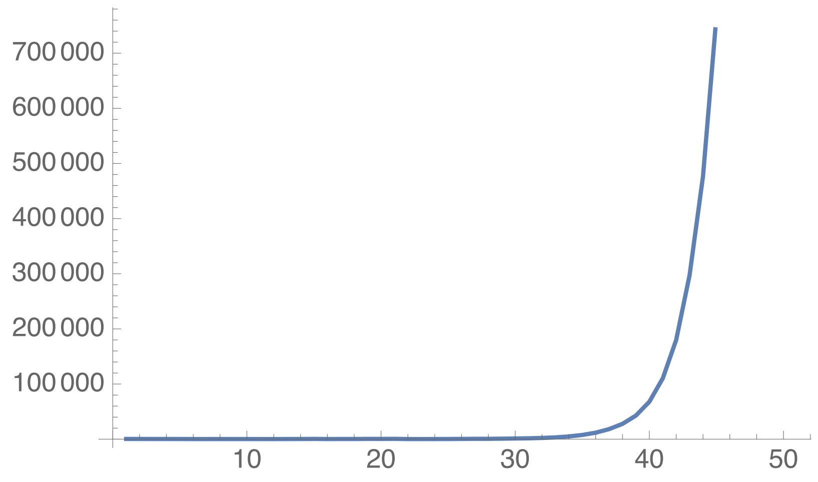


Looking at combinators that halt (combinators for which the graph flattens out), some combinators ‘definitely halt’ - their length decreases until the graph flattens out:



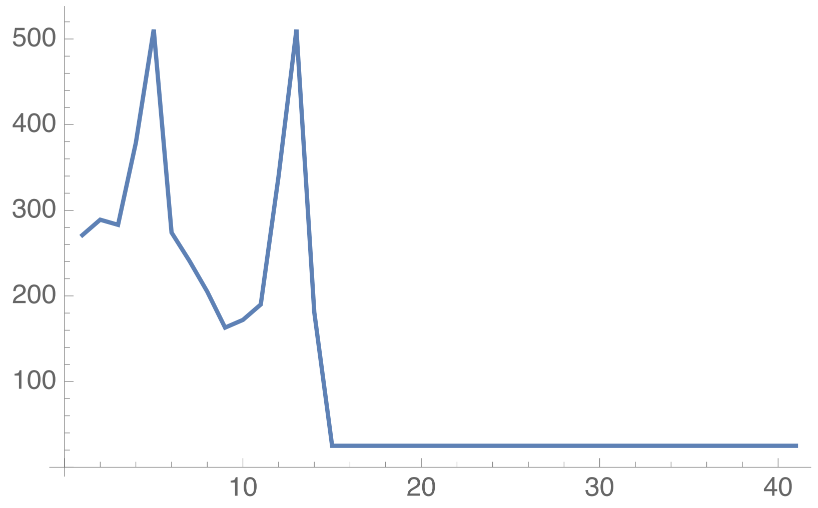
→ ‘definitely halts’ (1)

Some combinators have length that increases exponentially:



→ ‘possibly non-halting’ (2)

And some combinators appear to have increasing length but suddenly decrease:



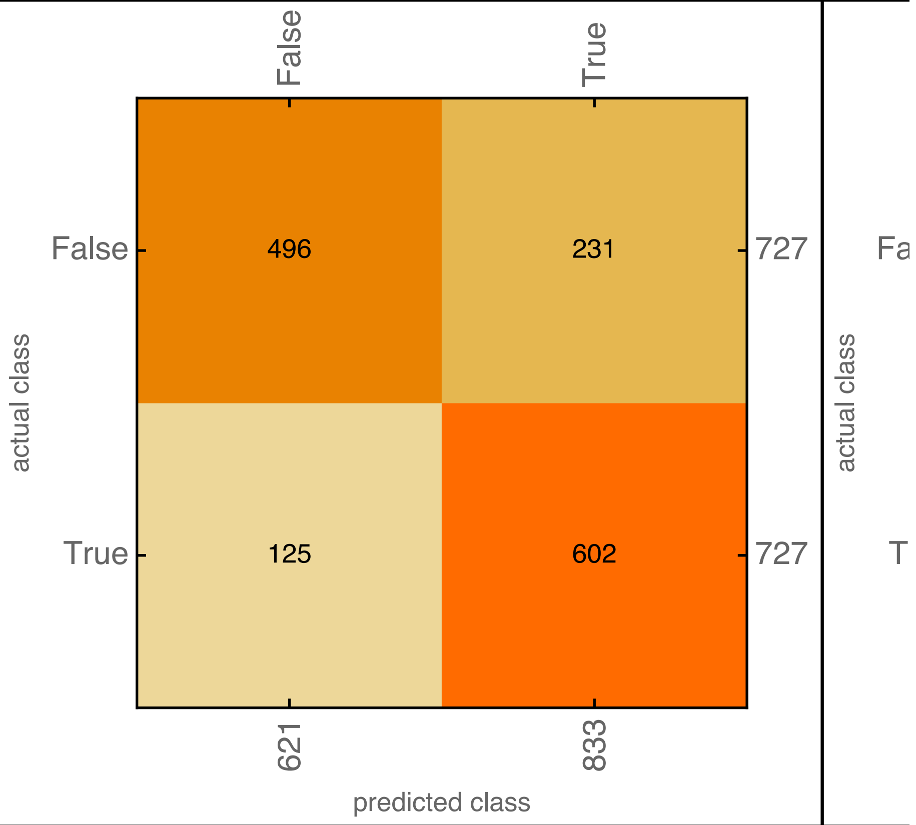
→ ‘possibly non-halting’ (3)

We do not know which features of the rasterised graphic the machine learning model extracts to make its prediction, but if, say, it was classifying based purely on length of the graphic, it would identify combinators like (1) as ‘definitely halting’, but would not necessarily be able to distinguish between combinators like (2) and combinators like (3), which both appear to be non-halting initially.

On a similar note, some functional programming languages (e.g. Agda - citation needed) have the ability to classify a function as 'definitely halting' or 'possibly non-halting', just like our classifier, whose dataset is trained on functions that either 'definitely halt' (halt in <=40 steps) or are 'possibly non-halting' (do not halt in <=40 steps - might halt later).

3.4 Table of Comparison

Out[220]=

	Markov on Strings
Confusion Matrix	
Accuracy	0.76
Misclassification rate	0.24
Precision (halt)	0.722
True Positive Rate	0.83
False Positive Rate	0.32
Precision (non-halt)	0.799
True Negative Rate	0.68
False Negative Rate	0.17

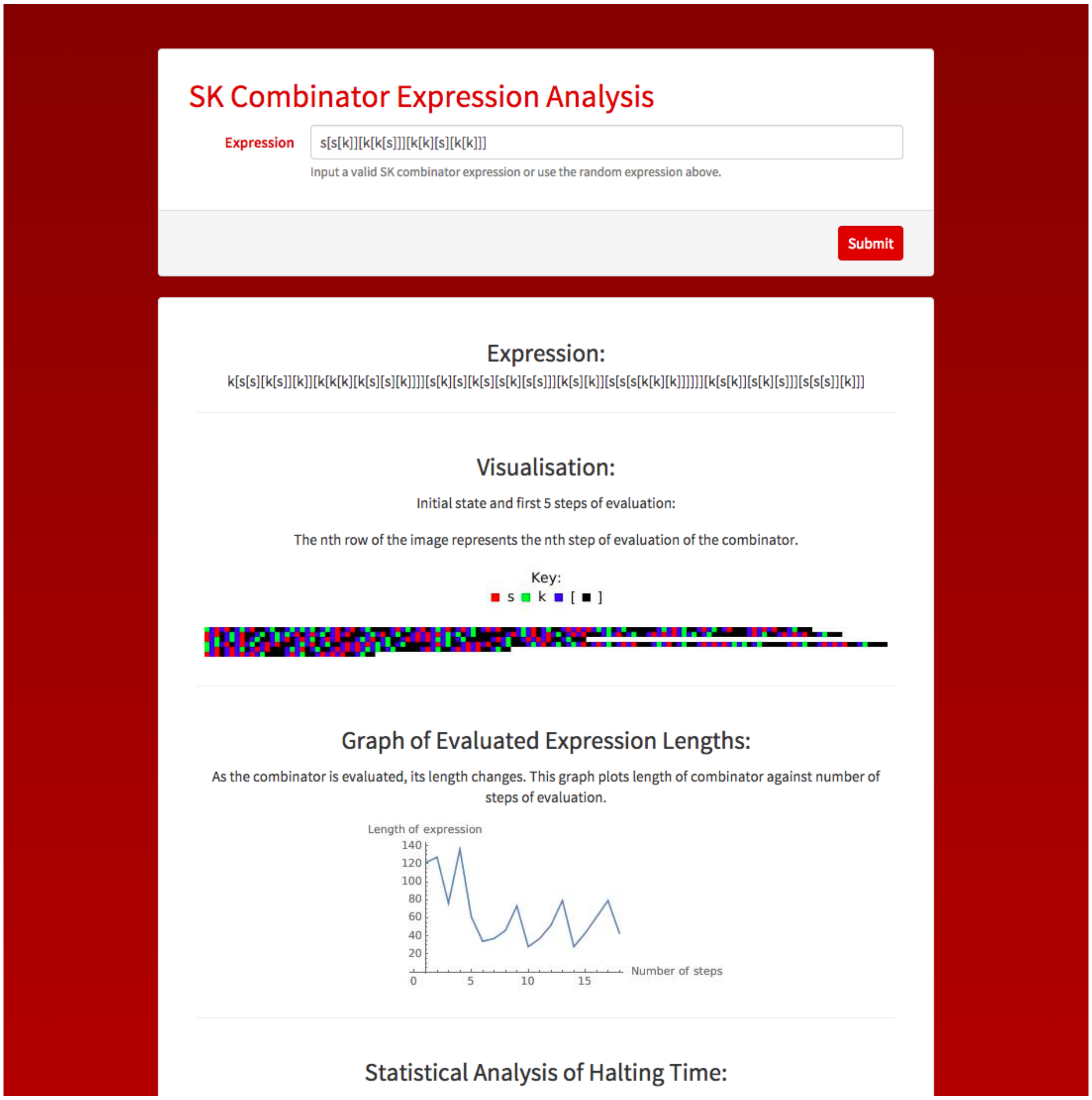
4. Conclusions and Further Work

4.1 Conclusions

The results of this exploration were somewhat surprising, in that a machine learning approach to determining whether or not a program will terminate appears to some extent viable - out of all the methods attempted, the random forest classifier applied to a rasterised image of the first five steps of the evaluation of a combinator achieved the highest accuracy of 0.88 on a test dataset of 1454 random SK combinator expressions. Note, though, that what is actually being determined here is whether or not a combinator will halt before some n steps (here, n=40) - we are classifying between combinators that ‘definitely halt’ and combinators which are ‘possibly non-halting’.

4.2 Microsite

As an extension to this project, a Wolfram microsite was created and is accessible at https://www.wolframcloud.com/objects/euan.l.y.ong/SKCombinators - within this microsite, a user can view a rasterised image of a combinator, a graph of the length of the combinator as it is evaluated, a statistical analysis of halting time relative to other combinators with the same leaf size and a machine learning prediction of whether or not the combinator will halt within 40 steps.


SK Combinator Expression Analysis

Expression: s[s[k]][k[k[s]]][k[k][s][k[k]]]

Input a valid SK combinator expression or use the random expression above.

Expression:

k[s[s][k[s]][k]][k[k[k][k[s][s][k]]]][s[k][s][k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]]][k[s[k]][s[k][s]]][s[s[s]][k]]]

Visualisation:

Initial state and first 5 steps of evaluation:

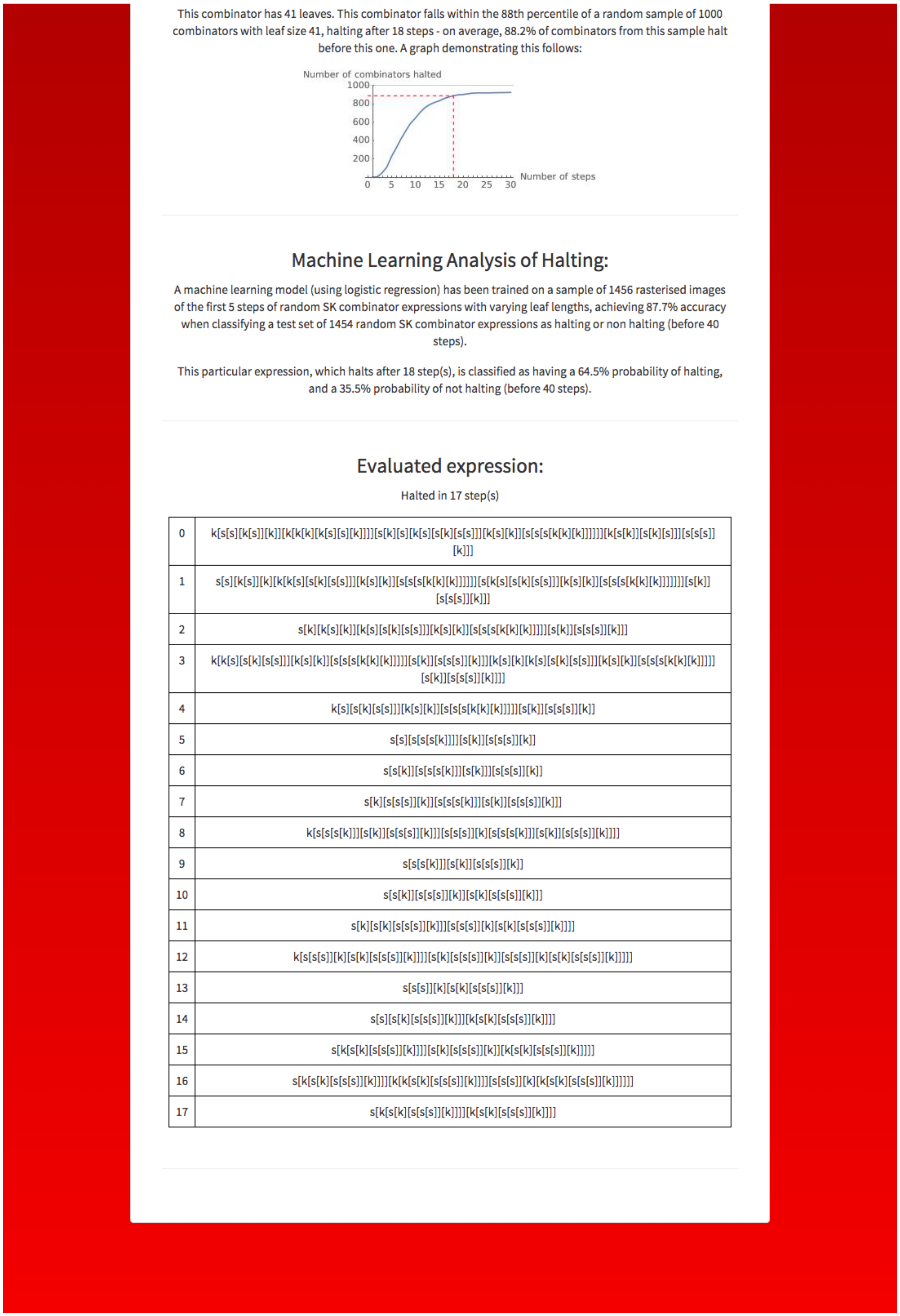
The nth row of the image represents the nth step of evaluation of the combinator.

Key:
🟥 s 🟩 k 🟦 [⬛]

Graph of Evaluated Expression Lengths:

As the combinator is evaluated, its length changes. This graph plots length of combinator against number of steps of evaluation.

Statistical Analysis of Halting Time:



This combinator has 41 leaves. This combinator falls within the 88th percentile of a random sample of 1000 combinators with leaf size 41, halting after 18 steps - on average, 88.2% of combinators from this sample halt before this one. A graph demonstrating this follows:

Machine Learning Analysis of Halting:

A machine learning model (using logistic regression) has been trained on a sample of 1456 rasterised images of the first 5 steps of random SK combinator expressions with varying leaf lengths, achieving 87.7% accuracy when classifying a test set of 1454 random SK combinator expressions as halting or non halting (before 40 steps).

This particular expression, which halts after 18 step(s), is classified as having a 64.5% probability of halting, and a 35.5% probability of not halting (before 40 steps).

Evaluated expression:

Halted in 17 step(s)

0	k[s[s][k[s]][k]][k[k[k][k[s][s][k]]]][s[k][s][k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]]][k[s[k]][s[k][s]]][s[s[s]][k]]]
1	s[s][k[s]][k][k[k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]]][s[k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]]][s[k]][s[s[s]][k]]
2	s[k][k[s][k]][k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]][s[k]][s[s[s]][k]]]
3	k[k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]][s[k]][s[s[s]][k]]][k[s][k][k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]][s[k]][s[s[s]][k]]]]
4	k[s][s[k][s[s]]][k[s][k]][s[s[s[k[k][k]]]]][s[k]][s[s[s]][k]]
5	s[s][s[s[s[k]]]][s[k]][s[s[s]][k]]
6	s[s[k]][s[s[s[k]]][s[k]]][s[s[s]][k]]
7	s[k][s[s[s]][k]][s[s[s[k]]][s[k]][s[s[s]][k]]]
8	k[s[s[s[k]]][s[k]][s[s[s]][k]]][s[s[s]][k][s[s[s[k]]][s[k]][s[s[s]][k]]]]
9	s[s[s[k]]][s[k]][s[s[s]][k]]
10	s[s[k]][s[s[s]][k]][s[k][s[s[s]][k]]]
11	s[k][s[k][s[s[s]][k]]][s[s[s]][k][s[k][s[s[s]][k]]]]
12	k[s[s[s]][k][s[k][s[s[s]][k]]]][s[k][s[s[s]][k]][s[s[s]][k][s[k][s[s[s]][k]]]]]
13	s[s[s]][k][s[k][s[s[s]][k]]]
14	s[s][s[k][s[s[s]][k]]][k[s[k][s[s[s]][k]]]]
15	s[k[s[k][s[s[s]][k]]]][s[k][s[s[s]][k]][k[s[k][s[s[s]][k]]]]]
16	s[k[s[k][s[s[s]][k]]]][k[k[s[k][s[s[s]][k]]]][s[s[s]][k][k[s[k][s[s[s]][k]]]]]]
17	s[k[s[k][s[s[s]][k]]]][k[s[k][s[s[s]][k]]]]

▲ A screenshot of the microsite evaluating a random SK combinator expression

4.3 Implications, Limitations and Further Work

Although the halting problem is undecidable, the field of termination analysis - attempting to determine whether or not a given program will eventually terminate - has a variety of applications, for instance in program verification. Machine learning approaches to this problem would not only help explore this field in new ways but could also be implemented in, for instance, software debuggers.

The principal limitations of this method are that we are only predicting whether or not a combinator will halt in a finite number k of steps - while this could be a sensible idea if k is large, at present this system is very impractical due to small datasets and a small value of k used to train the classifier ($k = 40$). Another issue with the machine learning technique used is that the visualisations have different dimensions (longer combinators will generate longer images), and when the images are preprocessed and resized before being fed into the random forest model, downsampling/upsampling can lead to loss of data decreasing the accuracy of the model.

From a machine learning perspective, attempts at analysis of the rasterised images with a neural network could well prove fruitful, as would an implementation of a vector representation of syntax trees to allow the structure of SK combinators (nesting combinators) to be accurately extracted by a machine learning model.

Future theoretical research could include a deeper exploration of Lathrop's probabilistic method of determining k, an investigation of the 'halting' features the machine learning model is looking for within the rasterised images, a more general analysis of SK combinators (proofs of halting / non-halting for certain expressions, for instance) to uncover deeper patterns, or even an extension of the analysis carried out in the microsite to lambda calculus expressions (which can be transformed to an 'equivalent' SK combinator expression).

Acknowledgments

We thank the mentors at the 2018 Wolfram High School Summer Camp - Andrea Griffin, Chip Hurst, Rick Hennigan, Michael Kaminsky, Robert Morris, Katie Orenstein, Christian Pasquel, Dariia Porechna and Douglas Smith - for their help and support writing this paper.

References

[1] F. Authorlast and S. Authorlast, "Article Title," *Full Name of Journal*, **volume**(issue number), year pp. #–#. doi:name.
R. Albert and A.-L. Barabási, "Statistical Mechanics of Complex Networks," *Reviews of Modern Physics*, **74**(1), 2002 pp. 47–97. doi:10.1103/RevModPhys.74.47.

[2] I. J. Authorlast, *Book Title*, Publisher Location: Publisher Name, year.
T. C. Schelling, *Micromotives and Macrobehavior*, New York: Norton, 1978.

[3] A. Authorlast, "Paper Title," in *A Collection* (F. Editor and S. Editor, eds.), Publisher Location: Publisher Name, year pp. #–#. doi:name.
S. Hou, J. Sterling, S. Chen and G. Doolen, "A Lattice Boltzmann Subgrid Model for High Reynolds Number Flows," in *Pattern Formation and Lattice Gas Automata* (A. T. Lawniczak and R. Kapral, eds.), Toronto: Fields Institute Communications, **6**, 1996 pp. 151–166.

[4] A. Editor, ed., *Book Title*, nth ed., Publisher Location: Publisher Name, year.
A. Law and D. Kelton, eds., *Simulation Modeling and Analysis*, 3rd ed., Boston: McGraw-Hill, 2000.

[5] A. Authorlast, "Paper Title," in *Conference Proceedings Title* (*Conference Acronym and year*), Conference Location (A. Authorlast, ed.), Publisher Location: Publisher Name, year pp. #–#.
P. Fritzson, L. Viklund, J. Herber and D. Fritzson, "Industrial Application of Object-Oriented Mathematical Modeling and Computer Algebra in Mechanical Analysis," in *Proceedings of the Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS EUROPE'92)*, Dortmund, Germany (G. Heeg,
B. Magnosson and B. Meyer, eds.), Hertfordshire, UK: Prentice Hall International (UK) Ltd., 1992 pp. 167–181.

[6] A. Authorlast, *Technical Report Title*, Classification/Number, Department, University or Organization, Location, year. URL if available.
C. Lemieux, M. Cieslak and K. Luttmer, *RandQMC User's Guide: A Package for Randomized Quasi-Monte Carlo Methods in C*, Technical report 2002-712-15, Department of Computer Science, University of Calgary, 2002. hdl.handle.net/1880/46569.

[7] A. Authorlast, *Preprint Book Title*, Publisher Location: Publisher Name, forthcoming.
J.-P. Aubin, L. Chen and O. Dordan, *Tychastic Measure of Viability Risk: A Viabilist Portfolio Performance and Insurance Approach*, Heidelberg: Springer, forthcoming. vimades.com/AUBIN/EradicationVPPI-Presentation.pdf.

[8] Company Name, *Computer Program Reference Manual* (available from Name, address).
Xerox, *InterLISP Reference Manual* (available from Xerox Palo Alto Research Center, Palo Alto, CA).

[9] A. Authorlast, "Future Paper," *Full Name of Journal*, forthcoming.
J. Riedel and H. Zenil, "Cross-Boundary Behavioural Reprogrammability Reveals Evidence of Pervasive Universality," *International Journal of Unconventional Computing*, forthcoming. arxiv.org/abs/1510.01671.

[10] A. Authorlast, "Title," presentation given at *Conference Name (Conference Acronym and year)*, Location. URL of abstract if available.
A. Banos, "Exploring Network Effects in Schelling's Segregation Model," presentation given at *S4-Modus Workshop: Multi-scale Interactions between Urban Forms and Processes*, Besançon, France, 2009.

[11] Software Name, Release Version Number, Location: Organization, year.
U. Wilensky. "NetLogo." Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL. (Oct 25, 2012) ccl.northwestern.edu/netlogo/index.shtml.

[12] A. Authorlast. "Website (or page) Title." (Month Day, Year) URL.
OnlineAtlas.us. "United States Interstate Highway Map." (May 7, 2012) www.onlineatlas.us/interstate-highways.htm.

[13] A. Authorlast. "Blog Title," Blog Series Name (blog). (Month Day, Year) URL.
B. Yorgey, "Recounting the Rationals, Part II," *The Math Less Traveled* (blog). (Apr 2, 2010) www.mathlesstraveled.com/?p=97.

[14] A. Authorlast. "Forum Post" from Forum Name. (Month Day, Year) URL.
T. Rowland. "Enumerating Strings" from The NKS Forum—A Wolfram Web Resource. (Apr 2, 2010) forum.wolframscience.com/showthread.php?s=&threadid=929.

[15] A. Authorlast. "Demonstration Title" from the Wolfram Demonstrations Project—A Wolfram Web Resource. URL.
E. Pegg Jr. "Coin Flips" from the Wolfram Demonstrations Project—A Wolfram Web Resource. www.demonstrations.wolfram.com/CoinFlips.

[16] A. Authorlast. "Wolfram Cloud Article Title." (Month Day, Year) URL.
A. A. de Laix. "Encryption with Enigma." (Jul 1, 2014) www.wolframcloud.com/objects/1f52ae4b-0686-4bde-966e-5e60d8225ae4.