Advanced Machine Learning with Basic Excel

Vincent Granville, Ph.D.
vincentg@MLTechniques.com
www.MLTechniques.com
Version 1.0, September 2022

Abstract

The method described here illustrates the concept of ensemble methods, applied to a real life NLP problem: ranking articles published on a website to predict performance of future blog posts yet to be written, and help decide on title and other features to maximize traffic volume and quality, and thus revenue. The method, called hidden decision trees (HDT), implicitly builds a large number of small usable (possibly overlapping) decision trees. Observations that don't fit in any usable node are classified with an alternate method, typically simplified logistic regression.

This hybrid procedure offers the best of both worlds: decision tree combos and regression models. It is intuitive and simple to implement. The code is written in Python, and I also offer a light version in basic Excel. The interactive Excel version is targeted to analysts interested in learning Python or machine learning. HDT fits in the same category as bagging, boosting, stacking and adaBoost. This article encourages you to understand all the details, upgrade the technique if needed, and play with the full code or spreadsheet as if you wrote it yourself. This is in contrast with using blackbox Python functions without understanding their inner workings and limitations. Finally, I discuss how to build model-free confidence intervals for the predicted values.

Contents

The technique presented here, called hidden decision trees, blends non-standard, robust versions of decision trees and regression. Compared to adaptive boosting [Wiki], it is simpler to implement. I used it in credit card fraud detection while working at Visa, as well as for scoring Internet traffic quality and search keyword scoring and bidding at eBay. It is an ensemble method [Wiki], in the sense that it blends multiple techniques to get the best of each one, to make predictions. Here I describe an NLP (natural language processing) case study: optimizing website content. The purpose is to to predict the performance of articles published in media outlets or blogs, in particular to predict which types of articles do well.

Here the response (that is, what we are trying to predict, also called dependent variable by statisticians) is the traffic volume, measured in page views, unique page views, or number of users who read the article over some time period. Page view counts can be influenced by robots, and "unique page views" is a more robust metric. Also, older articles have accumulated more page views over time, while the most recent ones have yet to build traffic. We need to correct for this bias. Correcting for time is explained in section 2.1. A simple approach is to use articles published within the last two years but that are at least six month old, in the training set. Due to the highly skewed distribution, I use the logarithm of unique page views as the core metric.

The features, also called predictors or independent variables, are:

Feature	Comment
Title keywords	binary
Article category	blog, event, forum
Publisher website or category	
Creation date	year/month
The title contains numbers	yes/no
The title is a question	yes/no
The title contains special characters	yes/no
Length of title	
Size of article	number of words
The article contains pictures	yes/no
Body keywords	binary
Author popularity	
First few words in the body	

Table 1: List of potential features to use in the model

Each keyword is a binary feature in itself: it is set to "yes" if the keyword is found (say) in the title, and to "no" otherwise. I used a shortlist of top keywords (by volume) found in all the articles combined. Also, I used a subset, narrowed version of the features in Table 1: for instance, title keywords only, and whether the article is a blog or not.

The method takes into account at all potential key-value pair combinations, where "key" is a subset of features, and "value" is the vector of corresponding values. For instance `key=(keyword1,keyword2,category)` and `value=('Python','tutorial','Blog')`. It is important to appropriately bin the features to prevent the number of key-value pairs from exploding, using optimum binning [Python]. See recent article [2] on this topic. Another mechanism described later is also used to keep the key-value table, stored as an hash table or associate array, manageable. Finally, this can easily be implemented in a distributed environment. A key-value pair is also called a node, and plays the same role as a node in a decision tree.

1 Methodology

You want to predict p, the logarithm of unique page views for an article (over some time period), as a function of keywords found in the title, and whether the article in question is a blog or not. You start by creating a list of all one-token and two-token keywords found in all the article titles, with the article category (blog versus non-blog), after cleaning the titles and eliminating some stop word such as "that", "and" or "the". Do not eliminate all keywords made up of one or two letters: the one-letter keyword "R", corresponding to the programming language R, has a high predictive power. For each key-value pair, get the number of articles matching it, as well as the average, minimum and maximum p across these articles.

For instance, say the key-value pair `(keyword1='R',keyword2 ='Python',category='Blog')` has 6 articles and the following statistics: average p is 8.52, minimum is 7.41, and maximum is 10.45. If the average p across all articles in the training set is 6.83, then this specific key-value pair (also called node) generates $\exp(8.52 - 6.83) = 5.42$ times more traffic than an average article. It is thus a large node in terms of traffic.

Even the worst article among the 6 ones belonging to this node, with a p of 7.41, outperforms the average 6.83 across all nodes. So not only this is a large node, but a stable one. Some nodes have a higher variance $\text{Var}[p]$, for instance when one of the keywords has different meanings, such as the word "training" in "training set" and in "courses and training".

1.1 How hidden decision trees (HDT) work

The nodes are overlapping, allowing considerable flexibility. In particular, nodes with two keywords are sub-nodes of nodes with one keyword. The general idea behind this technique is to group articles into buckets that are large enough to provide good predictions, without explicitly building decision trees. The nodes are simple and easy to interpret, and unstable ones (with high variance) can be discarded. There is no splitting/pruning involved as with classical decision trees, making this methodology simple and robust, and thus fit for artificial intelligence and black-box implementation. The method is called hidden decision trees and abbreviated as HDT because you don't create decision trees, but you indirectly rely on a large number of small ones that are hidden in the algorithm.

Whether you are dealing with predicting the popularity of an article, or the risk for a client to default on a loan, the basic methodology is identical. It involves training sets, cross-validation, feature selection, binning, and populating hash tables of key-value pairs, referred to as the nodes. When you process a new observation outside the training set, you check which node(s) it belongs to. If the "ideal" nodes it belongs to are stable and not too small, you use a weighted average score computed over these nodes, as predictor. If this score (defined as p here) is significantly above the global average, and other constraints are met, then you classify the observation – in this case a potential article you want to write – as good. An ideal node has strong predictive power: its p is either very high or very low.

Also, you need to update your training set and the nodes table, including automatically discovered new nodes, every six months or so. Parameters must be calibrated to guarantee that the proportion of false positives remains small enough. Ideally, you want to end up with less than 3000 stable nodes, each with at least 10 observations (articles), covering 80% of the articles or more. I discuss the parameters of the technique, and how to fine-tune them, in section 1.3. Fine-tuning can be automated or made more robust by testing (say) 2000 sets of parameters and identify regions of stability minimizing the error rate in the parameter space. Error rate is defined as the proportion of misclassification, and false positives in particular.

A big question is what to do with observations not belonging to any usable node: they cannot be classified. A usable node is one with enough articles, with average p within the mode far away from the global mean of p computed across all nodes. One way to address this issue is to use two algorithms: the one described so far, applied to usable or ideal nodes (let's call it algorithm A) and another one called algorithm B that classifies all observations. Observations that can't be classified or scored with algorithm A are classified/scored with algorithm B. The resulting hybrid algorithm is called Hidden Decision Trees.

1.2 NLP Case study: summary and findings

If you run the Python script listed in section 2.3, besides producing the table of key-value pairs (the nodes) as a text file for further automated processing, it displays summary statistics that look like the following:

```
Average log pageview count (pv): 6.83
Avg pv, articles marked as Good: 8.09
Avg pv, articles marked as Bad : 5.95

Number of articles marked as Good: 223 (real number is 1079)
Number of articles marked as Bad : 368 (real number is 919)
Number of false positives    : 25 (Bad marked as Good)
Number of false negatives    : 123 (Good marked as Bad)
Total number of articles     : 2616

Proportion of False Positives: 11.2%
Proportion of Unclassified   : 77.4%

Aggregation factor (Good node): 29.1
Number of feature values: 16711 (marked as good: 49)

Execution time: 0.0630 seconds
```

In the code, p – the logarithm of the pageview count – is represented by `pv`. The number of nodes is the total number of key-value pairs found, including the small unstable ones, regardless as to whether they are classified as good, bad, or unclassified. An article with p above the arbitrary `pv_threshold_good=7.1` (see source code) is considered as good. This corresponds to articles having about 1.3 times more traffic than average, since I use a log scale and the average p is 6.83. Articles classified as good have an average p of 8.09, that is, about 3.3 times more traffic than average.

Two important metrics are:

- Aggregation factor: it is an indicator of the average size of a useful node, in this case classified as Good. A value above 5 is highly desirable.
- The most important error rate is measured here as the number of bad articles wrongly classified as good. The goal is to detect very good articles and find the reasons that make them popular, to be able to increase the proportion of good articles in the future. Avoiding bad articles is the second most important goal, so I am also interested in identifying what makes them bad.

Also note that the method correctly identifies a proportion of the good articles, but leaves many unclassified. I explain in section 2.3 how to improve this. Finally an article is marked as good if it meets some criteria specified in section 1.3.

Now I share some interesting findings revealed by these hidden decision tress, on the data set investigated in this study. First, articles with the following title features do well: contains a number as in "10 great deep learning articles", contains the current year, is a question (how to), is a blog post or belongs to the book category.

Then the following title keywords are a good predictor of popularity: everyone (as in "10 regression techniques everyone should know"), libraries, infographic, explained, algorithms, languages, amazing, must read, R Python, job interview questions, should know (as in "10 regression techniques everyone should know"), NoSQL databases, versus, decision trees, logistic regression, correlations, tutorials, code, free.

1.3 Parameters

Besides `pv_threshold_good` and `pv_threshold_bad`, the algorithm uses 12 parameters to identify a usable, stable node classified as good. You can see them in action in the Python code in section 2.3, in the instruction

```
if n > 3 and n < 8 and Min > 6.9 and Avg > 7.6 or \
   n >= 8 and n < 16 and Min > 6.7 and Avg > 7.4 or \
   n >= 16 and n < 200 and Min > 6.1 and Avg > 7.2:
```

Here, `n` represents the size (number of observations) of a node, while `Avg` and `Min` are the average and minimum `pv` for the node in question. I tested many combinations of values for these parameters. Increasing the required size to qualify as usable node will do the following:

- Decrease the number of good articles correctly identified as good
- Increase the error rate
- Increase the stability of the system
- Decrease the predictive power
- Increase the aggregation factor

1.4 Improving the methodology

Some two-token keywords should be treated as one-token. For instance "San Francisco" must be treated as a one-token keyword. It is easy to automatically detect this: when you analyze the text data, "San" and "Francisco" are lumped together far more frequently than dictated by pure chance. Also, I looked at nodes where the two keywords are adjacent in the text. If you allow the two keywords not to be adjacent, the number of key-value pairs (the nodes) increases significantly, but you don't get much more additional predictive power in return, and there is a risk of over-fitting.

Another improvement consists of favoring nodes containing articles spread over several years, as opposed to concentrated on a few weeks or months. The latter category may be popular articles at some time, that faded away. Finally, you cannot exclusively focus on articles with great potential. It is important to have many, less popular articles as well: they constitute the long tail. Without these less popular articles, you face excessive content concentration and readership attrition in the long term.

2 Implementation details

This section contains the Python code and details about the Excel spreadsheet. Both the decision trees and the regression part of HDT (referred to as "algorithm B" in section 1.1) are implemented in the spreadsheet. The Python version, though more comprehensive in many regards, is limited to the decision trees. But first I start by discussing a possible improvement of the methodology: bias correction.

2.1 Correcting for bias

In online rankings, the most popular books, authors, articles, restaurants, products and so on are usually those that have been around for a long time. Here I address this issue by creating adjusted scores. It allows you to make fair comparisons between new and old items.

For top time-insensitive articles, page views peak in the first three days, but popularity remains high for many years. In short, page view decay is very low over time. Finally, the most popular topics (keywords) change over time; this type of analysis helps find the trends. It is also a good idea to use two different sources of data for pageview measurements, see how they differ, understand why, and check whether the discrepancy worsens over time.

The articles scored here span over a three-year period, covering over 2600 pieces of content totaling 6 million pageviews across three websites. The summary data is on GitHub, here. It features the top 46 articles ranked according to the time-adjusted score. The number in parenthesis attached to each article is the non-adjusted (old) score. The difference between the time-adjusted score and the old one, is striking.

2.1.1 Time-adjusted scores

You measure the page view count for a specific article, and your time period is $[t_0, t_1]$. Typical models use an exponential decay of rate λ. The adjustment factor is then

$$q = \frac{1}{\lambda} \cdot \Big[\exp(-\lambda t_0) - \exp(-\lambda t_1) \Big] > 0.$$

Now define the adjusted score as p/q, where p is the observed page view count in $[t_0, t_1]$. If $\lambda = 0$ (no decay) then $q = t_1 - t_0$.

2.2 Excel spreadsheet

The interactive spreadsheet named `HDTdata4Excel.xlsx` is on my GitHub repository, here. It uses a subset of 9 binary features. The first three are respectively "published after 2014", "article is a forum discussion", and "article is a blog post". The next six ones are indicators of whether or not the title contains a specific character string. The six strings in question are "python", "r", "machine learning", "data science", "data", and "analy". The last string captures words such as "analytic" or "analyst". These strings must be surrounded by spaces, so "r" clearly represents the R programming language. True/false are encoded as 1 and 0 respectively.

For instance, node `N-001-001110` in Table 2 corresponds to blog posts published in 2014, containing the keywords "machine learning", "data science" and "data" in the title, but not "python", "r" or "analy". The column "size" tells us that the node in question has only one article.

node	size	pv	index
N-000-000000	8	7.12	1.33
N-000-000001	5	6.87	1.04
N-000-000010	8	6.86	1.02
N-000-000011	3	6.49	0.71
N-000-000110	3	7.18	1.42
N-001-000000	313	6.88	1.05
N-001-000001	75	6.71	0.88
N-001-000010	276	7.14	1.35
N-001-000011	44	7.16	1.38
N-001-000110	130	7.68	2.34
N-001-000111	5	8.05	3.37
N-001-001000	5	8.07	3.45
N-001-001010	1	7.58	2.11
N-001-001110	1	7.35	1.67

Table 2: Statistics for selected HDT nodes (Excel version)

Nodes with fewer than 10 articles are classified using the regression method via the `LINEST` Excel function, rather than the mini decision trees. Instead of standard regression, you can use a simplified logistic regression, as described in section 1.3.1 in [1]. There are 2616 observations (articles) and 74 nodes in the training set. By grouping all nodes with less than 10 observations into one node, we get down to 24 nodes. Correlations between individual features and the response p (logarithm of pageviews, denoted as `pv` in Table 2) is very low. Thus individual features have no predictive power. They must be combined together to gain predictive power. The full HDT method is superior to either the mini decision trees, or the regression model taken separately.

The index in Table 2 is the `pv` of the node in question, divided by the average `pv` across all nodes. It measures the performance of the node. Finally, an article is classified as good or bad depending on whether its index is significantly larger or lower than one. The thresholds are user-defined.

Summary stats for (blue) binary features used in model, when value = 1									
	year > 2014 *	/forum	/blog	_python_	_r_	_machine_	_data_sci	_data_	_analy
correl with pv	-0.11	-0.18	0.18	0.13	0.05	0.10	0.15	0.13	-0.09
avg pv (if val = 1)	6.71	6.35	6.95	8.23	7.27	7.53	7.33	7.00	6.56
pv index (if val = 1) *	0.88	0.62	1.13	4.04	1.54	2.01	1.65	1.18	0.76
count (val = 1)	1,499	487	2,081	39	65	85	357	1,309	424
percentage obs (val = 1)	57.3%	18.6%	79.5%	1.5%	2.5%	3.2%	13.6%	50.0%	16.2%
* average pv index is 1 (the higher the better; it is proportional to pageviews)									
** more recent articles haven't accumulated as many pageviews (thus pv index < 1 for 'year > 2014')									

Cross-correlations table								
	/forum	/blog	_python_	_r_	_machine_	_data_sci	_data_	_analy
year > 2014	-0.04	0.05	0.08	0.07	0.10	-0.04	-0.02	0.02
/forum		-0.94	0.05	0.02	-0.02	0.01	-0.05	0.05
/blog			-0.04	-0.02	0.03	0.00	0.05	-0.07
python				0.08	0.03	0.03	0.00	-0.02
_r_					0.00	-0.03	-0.08	0.00
_machine_learning_						-0.04	-0.11	-0.06
_data_science_							0.40	-0.12
data								-0.09

Figure 1: Output from the Excel version of HDT

2.3 Python code and dataset

The input dataset `HDTdata4.txt` is on my GitHub repository, here. The Python program `HDT.py` is listed below and can also be found on my GitHub repository, here. The output file `hdt-out2.txt` contains the usable key-value pairs (nodes) corresponding to popular articles, and the list of article IDs for each of these nodes. Finally, the variable `pv` represents p, the logarithm of the pageview count. The bivariate combinations (title keyword, article category) constitute the keys of the hash table `list_pv`, while the `pv` are the hash table values. Keywords are either one- or two-token. For one-token keywords, the second token is marked as N/A. In short, keyword is a bivariate entity.

As for the error rate, since the focus is on producing good articles, I am interested only in minimizing the number of bad articles flagged as good: the false positives. To reduce error rates or the proportion of unclassified nodes, use more features (for instance, more of those listed in Table 1), three-token keywords, a larger training set, a better keyword cleaning mechanism, and fine-tune the parameters.

Of course, if you choose the option `mode='perfect_fit'` in the program, your false positive rate drops to 0% on the training set, but doing may lower the performance on the validation set, and may leave many nodes unclassified. On the plus side, you have much fewer parameters to fine-tune: `pv_threshold_good`, `pv_threshold_bad`, and the minimum size of a usable node (the variable `n` in the code). The first two can be set respectively to 5% above and 10% below the global average `pv`. The minimum node size should be set above 2, and ideally above 5, though a large value results in more unclassified nodes. For `mode='robust method'`, the parameters in the conditional statements defining `good_node` and `bad_node` were set manually based on an average `pv` of 6.83. These choices can be automated.

In the end, unclassified nodes are classified via regression in the spreadsheet (see section 2.2), but this has not yet been implemented in the Python code. But for my purpose (identifying what makes an article good), I did not need to add the regression part, as the mini decision trees alone (the nodes) provide enough valuable insights.

```
from math import log
import time

start = time.time()

# This method updates the dictionaries based on given ID, pv and word
def update_pvs(word, pv, id, word_count_dict, word_pv_dict, min_pv_dict, max_pv_dict,
    ids_dict):
   if word in word_count_dict:
      word_count_dict[word] += 1
      word_pv_dict[word] += pv
      if min_pv_dict[word] > pv:
         min_pv_dict[word] = pv
      if max_pv_dict[word] < pv:
```

```
            max_pv_dict[word] = pv
         ids_dict[word].append(id)
      else:
         word_count_dict[word] = 1
         word_pv_dict[word] = pv
         min_pv_dict[word] = pv
         max_pv_dict[word] = pv
         ids_dict[word] = [id]
# dictionaries to hold count of each key words, their page views, and the ids of the
    article in which used.
List = dict()
list_pv = dict()
list_pv_max = dict()
list_pv_min = dict()
list_id = dict()
articleTitle = list() # Lists to hold article id wise title name and pv
articlepv = list()
sum_pv = 0
ID = 0
in_file = open("HDTdata4.txt", "r")

for line in in_file:
   if ID == 0: # excluding first line as it is header
      ID += 1
      continue
   line = line.lower()
   aux = line.split('\t') # Indexes will have: 0 - Title, 1 - URL, 2 - data and 3 - page
       views
   url = aux[1]
   pv = log(1 + int(aux[3]))
   if "/blogs/" in url:
      type = "BLOG"
   else:
      type = "OTHER"
#  #--- clean article titles, remove stop words
   title = aux[0]
   title = " " + title + " " # adding space at the ends to treat stop words at start,
       mid and end alike
   title = title.replace('"', ' ')
   title = title.replace('?', ' ? ')
   title = title.replace(':', ' ')
   title = title.replace('.', ' ')
   title = title.replace('(', ' ')
   title = title.replace(')', ' ')
   title = title.replace(',', ' ')
   title = title.replace(' a ', ' ')
   title = title.replace(' the ', ' ')
   title = title.replace(' for ', ' ')
   title = title.replace(' in ', ' ')
   title = title.replace(' and ', ' ')
   title = title.replace(' or ', ' ')
   title = title.replace(' is ', ' ')
   title = title.replace(' in ', ' ')
   title = title.replace(' are ', ' ')
   title = title.replace(' of ', ' ')
   title = title.strip()
   title = ' '.join(title.split()) # replacing multiple spaces with one
   #break down article title into keyword tokens
   aux2 = title.split(' ')
   num_words = len(aux2)
   for index in range(num_words):
      word = aux2[index].strip()
      word = word + '\t' + 'N/A' + '\t' + type
      update_pvs(word, pv, ID - 1, List,list_pv, list_pv_min, list_pv_max, list_id) #
          updating single words
```

```
      if (num_words - 1) > index:
         word = aux2[index] + '\t' + aux2[index+1] + '\t' + type
         update_pvs(word, pv, ID - 1, List, list_pv, list_pv_min, list_pv_max, list_id)
             # updating bigrams

   articleTitle.append(title)
   articlepv.append(pv)
   sum_pv += pv
   ID += 1
in_file.close()

nArticles = ID - 1 # -1 as the increments were done post loop
avg_pv = sum_pv/nArticles
articleFlag = ["NA" for n in range(nArticles)]
nidx = 0
nidx_Good = 0
nidx_Bad = 0
pv_threshold_good = 7.1
pv_threshold_bad = 6.2
mode = 'robust method' # options are 'perfect fit' or 'robust method'
OUT = open('hdt-out2.txt','w')
OUT2 = open('hdt-reasons.txt','w')
for idx in List:
   n = List[idx]
   Avg = list_pv[idx]/n
   Min = list_pv_min[idx]
   Max = list_pv_max[idx]
   idlist = list_id[idx]
   nidx += 1
   if mode == 'perfect fit':
     good_node = n > 2 and Min > pv_threshold_good
     bad_node = n > 2 and Max < pv_threshold_bad
   elif mode == 'robust method':
      # below values are chosen based on heuristics and experimenting
      good_node = n > 3 and n < 8 and Min > 6.9 and Avg > 7.6 or \
            n >= 8 and n < 16 and Min > 6.7 and Avg > 7.4 or \
            n >= 16 and n < 200 and Min > 6.1 and Avg > 7.2
      bad_node = n > 3 and n < 8 and Max < 6.3 and Avg < 5.4 or \
            n >= 8 and n < 16 and Max > 6.6 and Avg < 5.9 or \
            n >= 16 and n < 200 and Max > 7.2 and Avg < 6.2
   if good_node:
      OUT.write(idx + '\t' + str(n) + '\t' + str(Avg) + '\t' + str(Min) + '\t' +
          str(Max) + '\t' + str(idlist) + '\n')
      nidx_Good += 1
      for ID in idlist:
         title=articleTitle[ID]
         pv = articlepv[ID]
         OUT2.write(title + '\t' + str(pv) + '\t' + idx + '\t' + str(n) + '\t' +
             str(Avg) + '\t' + str(Min) + '\t' + str(Max) + '\n')
         articleFlag[ID] = "GOOD"
   elif bad_node:
      nidx_Bad += 1
      for ID in idlist:
         articleFlag[ID] = "BAD"
# Computing results based on Threshold values
pv1 = 0
pv2 = 0
n1 = 0
n2 = 0
m1 = 0
m2 = 0
FalsePositive = 0
FalseNegative = 0
for ID in range(nArticles):
   pv = articlepv[ID]
   if articleFlag[ID] == "GOOD":
```

```
      n1 += 1
      pv1 += pv
      if pv < pv_threshold_good:
         FalsePositive += 1
   elif articleFlag[ID] == "BAD":
      n2 += 1
      pv2 += pv
      if pv > pv_threshold_bad:
         FalseNegative += 1
   if pv > pv_threshold_good:
      m1 += 1
   elif pv < pv_threshold_bad:
      m2 += 1
#
# Printing results
avg_pv1 = pv1/n1
avg_pv2 = pv2/n2
errorRate = FalsePositive/n1
UnclassifiedRate = 1 - (n1 + n2) / nArticles
aggregationFactor = (nidx/nidx_Good)/(nArticles/n1)
print ("Average log pageview count (pv):","{0:.2f}".format(avg_pv))
print ("Avg pv, articles marked as Good:","{:.2f}".format(avg_pv1))
print ("Avg pv, articles marked as Bad :","{:.2f}".format(avg_pv2))
print()
print ("Number of articles marked as Good: ", n1, " (real number is ", m1,")", sep = "" )
print ("Number of articles marked as Bad : ", n2, " (real number is ", m2,")", sep = "")
print ("Number of false positives :",FalsePositive,"(Bad marked as Good)")
print ("Number of false negatives :", FalseNegative, "(Good marked as Bad)")
print ("Total number of articles :", nArticles)
print()
print ("Proportion of False Positives: ","{0:.1%}".format(errorRate))
print ("Proportion of Unclassified : ","{0:.1%}".format(UnclassifiedRate))
print()
print ("Aggregation factor (Good node):","{:.1f}".format(aggregationFactor))
print ("Number of feature values: ", nidx," (marked as good: ", nidx_Good,")", sep = "")
print ()
print("Execution time: ","{:.4f}".format(time.time() - start), "seconds")
```

3 Model-free confidence intervals and perfect nodes

Node `N-100-000000` in the spreadsheet has an average `pv` of 5.85. It consists of 10 articles with the following `pv`: 5.10, 5.10, 5.56, 5.56, 5.66, 5.69, 6.01, 6.19, 6.80, 6.80. The 15th and 85th percentiles are 5.26 and 6.68 respectively, when computed with the `Percentile` function in Excel. Thus, [5.26, 6.68] is a 70% confidence interval (CI) for `pv`, for the node in question.

The whole CI including its upper bound is below the average `pv` of 6.83. In fact this node corresponds to articles posted after 2014, not a blog or forum question (it could be a video or event announcement), and with a title containing none of the keyword features in the spreadsheet (columns `K:P` in the `data` tab). This node has a maximum predictive power, in the sense that 100% of the articles that it contains are bad, and 0% are good. This would also be true if it was the other way around, with Good swapped with Bad. Such a node is called a perfect node. When selecting the option `mode='Perfect fit'` in the Python code, the method looks at perfect nodes only. The concept of predictive power is further discussed in section 11.4.2 in [1].

3.1 Interesting asymptotic properties of confidence intervals

I focus here on traditional model-free confidence intervals, as computed in the above paragraphs. The reader should be aware that there are other ways to define them, for instance credibility intervals in the context of Bayesian inference [Wiki], or Bayesian-like dual confidence intervals as in section 11.3 in [1].

In almost all cases, as the number n of observations becomes large within a node, the length of the confidence interval, in this case for the expected `pv`, is asymptotically $L_n \sim \alpha n^{\beta}$. I discuss in an upcoming paper how to estimate α and β. The order of magnitude of the range $R_n = \max(\text{pv}) - \min(\text{pv})$ computed on a node with n observations, depends on the distribution of `pv`, and more specifically, on the type of this distribution.

pv distribution	Type	E$[R_n]$	Stdev$[R_n]$
Uniform	short tail	1	$1/n$
Gaussian	medium tail	$\sqrt{\log n}$	$1/\sqrt{n}$
Exponential	fat tail	$\log n$	1

Table 3: Order of magnitude for the expectation and standard deviation of the range R_n

The result is summarized in Table 3, and discussed in the same upcoming article. In practice, `pv` may have a mixture distribution.

References

[1] Vincent Granville. *Intuitive Machine Learning*. MLTechniques.com, 2022. To be published in October 2022 [Link]. 5, 9

[2] Guillermo Navas-Palencia. Optimal binning: mathematical programming formulation. *Preprint*, pages 1–21, 2020. arXiv:2001.08025 [Link]. 2