# Automatic String Clustering by Editing Distance

William John Holden

May 1, 2022

## 1 Introduction

The devices used in computer networking, such as routers and switches, may generate vast quantities of semi-structured log messages. These log messages are typically delivered to a centralized log collector using the Syslog protocol (Gerhards, 2009). A challenge for network operators is to separate interesting, important, and unusual log messages from the flood of routine, useless, and boring log messages. Clustering could be used to automatically classify incoming log messages into categories that the user has never seen before, seen before and deemed important, or seen before and deemed unimportant.

This paper proposes a novel algorithm named **Automatic String Clustering by Editing Distance (ASCED)** to automatically cluster incoming log messages incrementally using editing distance as their measure of similarity.

## 2 Editing Distance

Editing distance is measure of dissimilarity between two strings $a$ and $b$ (Cormen, 2009). One form of editing distance is computed from the minimal count of insertions or deletions needed to transform one string into the other. The problem can be solved quickly in $\Theta(mn)$ time, where $m$ is the length of the $a$ and $n$ is the length of $b$, using dynamic programming.

A typical algorithm constructs an $(m+1) \times (n+1)$ matrix $D$ of distances. The value of $D_{ij}$ will represent the minimal count of insertions or deletions needed to transform the substring $a_{1:i}$ into the substring $b_{1:j}$. Some editing distance calculations allow for the replacement, transposition, or "killing" (proceeding directly to the end) of characters. These operations are not used in this algorithm; only insertion and deletion are used in ASCED.

The left column is initialized with values 0 through $m$ and the top row is initialized with values 0 through $n$. These positions indicate editing distances where the first $i$ or $j$ characters have been removed from input string $a$ or $b$.

Next, the algorithm completes the table from the bottom-up using the invariant

$$D_{ij} = \min \begin{cases} D_{i(j-1)} + 1 & \\ D_{(i-1)j} + 1 & \\ D_{(i-1)(j-1)} & \text{if } a_{i-1} = b_{j-1} \\ D_{(i-1)(j-1)} + 1 & \text{otherwise.} \end{cases}$$

When the algorithm completes, the minimum editing distance is in the bottom right position of matrix $D$.

A reference implementation in the Julia language (Bezanson et al., 2012) is shown below:

[1]:
```
function editing_distance(a, b)
    m = length(a)
    n = length(b)
    D = zeros(Int, m+1, n+1)
    D[:,1] = 0:m
    D[1,:] = 0:n
    for i in 2:m+1
        for j in 2:n+1
            c = ifelse(a[i-1] == b[j-1], 0, 1)
            D[i,j] = min(D[i-1,j] + 1, D[i,j-1] + 1, D[i-1,j-1] + c)
        end
    end
    D
end;
```

[2]:
```
editing_distance("editing", "distance")
```

[2]:
```
8×9 Matrix{Int64}:
 0  1  2  3  4  5  6  7  8
 1  1  2  3  4  5  6  7  7
 2  1  2  3  4  5  6  7  8
 3  2  1  2  3  4  5  6  7
 4  3  2  2  2  3  4  5  6
 5  4  3  3  3  3  4  5  6
 6  5  4  4  4  4  3  4  5
 7  6  5  5  5  5  4  4  5
```

## 3 String Clustering

Clustering refers to a data mining technique of grouping elements by their similarity (Kantardžić, 2020). A typical clustering approach is to choose an appropriate distance statistic, such as Euclidean or Manhattan distance, and then construct the clusters using techniques such as agglomerative hierarchical clustering, $k$-means clustering, incremental clustering, and density-based clustering (DBSCAN).

**The ASCED algorithm assigns input strings to the cluster of minimum normalized editing distance, or to a new cluster if the distance is beyond a parameterized threshold**. The cluster set, $C$, is initially empty and is enlarged for new strings that differ beyond the threshold $t$. Normalized editing distance is simply the editing distance between two strings divided by the length of the longer string. For example, if the editing distance of "editing" and "distance" is 5, then their normalized editing distance is $5/8 = 0.625$.

[3]:
```
function editing_distance_normalized(a, b)
    last(editing_distance(a, b)) / max(length(a), length(b))
end;
```

[4]:
```
editing_distance_normalized("editing", "distance")
```

[4]:
```
0.625
```

The full ASCED procedure is:

1. *Initialization.* The cluster set, $C$, is initialized as an empty set.
2. *Measurement.* Given an input string $x$, compute the editing distance $d$ for each cluster $c$ in the cluster set $C$.
3. *Assignment.* Add $x$ to the cluster of minimum editing distance. In the event of a tie, the algorithm selects only one solution at random.

The measurement step can consider only a single string in each cluster, all strings in each cluster, some fixed subset of strings in each cluster, some constant proportion of strings of each cluster, and so on. In any case, strings from existing clusters should be selected at random. This is to desensitive the algorithm to incremental changes in input. For example, the following messages should ideally be classified together:

```
*Mar 29 20:30:22.475: %LINEPROTO-5-UPDOWN: Line protocol on Interface Vlan1,
changed state to up
*Mar 31 02:05:21.824: %LINEPROTO-5-UPDOWN: Line protocol on Interface Vlan1,
changed state to down
*Apr 01 08:02:36.267: %LINEPROTO-5-UPDOWN: Line protocol on Interface Vlan1,
changed state to up
*May 14 14:34:27.728: %LINEPROTO-5-UPDOWN: Line protocol on Interface Vlan1,
changed state to down
```

Notice that the date and time changed much more than the content of the message. Additionally, messages containing both the words “up” and “down” both belong in this cluster. Randomly selecting multiple messages from each cluster allows the algorithm to overcome these small variations. This implementation selects, by default, a random sample (with replacement) of five messages from each cluster and returns the minimum normalized editing distance.

[5]:
```
using DataFrames
function initialize()
    C = DataFrame(Message = String[], Cluster = Int[])
end;
```

[6]:
```
using StatsBase
function measure(C, x, sample_size)
    D = DataFrame(Cluster = Int[], Distance = AbstractFloat[])
    for c in groupby(C, :Cluster)
        clusterId = c[1, :Cluster]
        s = c[sample(1:nrow(c), sample_size), :Message]
        d = [editing_distance_normalized(x, y) for y in s]
```

```
        push!(D, [clusterId, minimum(d)])
    end
    return D
end;
```

[7]:
```
function assign!(C, x; threshold = .5, sample_size = 5)
    if isempty(C)
        push!(C, [x, 1])
    else
        D = measure(C, x, sample_size)
        minimum_distance = minimum(D.Distance)
        if minimum_distance < threshold
            solutions = filter(:Distance => ==(minimum_distance), D)
            push!(C, [x, rand(solutions.Cluster)])
        else
            new_cluster = 1 + maximum(C.Cluster)
            push!(C, [x, new_cluster])
        end
    end
end;
```

## 4 Parameters

The ASCED algorithm accepts two parameters: threshold ($t$) and sample size. This parameter effects the algorithm's sensitivity to dissimilarity between the input string, $x$, and candidate clusters in $C$.

A *higher* value $t$ makes the algorithm *less* sensitive to dissimilarity; the threshold distance for considering $x$ a member of a cluster is large, and a value might be assigned to a cluster that looks quite different from others. This will result in fewer clusters.

A *lower* value $t$ makes the algorithm *more* sensitive to dissimilarity. The algorithm will consider an input $x$ to belong in its own cluster unless it is very similar to an existing cluster. This will result in more clusters.

The next section will show that having more clusters (large $|C|$) will slow the algorithm. However, speed is not useful if the algorithm makes poor decisions. The $t$ parameter must be manually tuned by the user to fit the application.

Sample size is the constant number of random samples used in each step of the measurement function. A smaller sample size will reduce the number editing distance calcuations needed at each measurement step, but may lead to the "misses" where the measurement step fails to identify a near match in an existing cluster. If this happens, then the assigment function may create an additional cluster when one was not necessary. A sample size of 3-5 records is suggested, but the optimal value has not been studied and is not known. This could be an area for further study.

## 5 Running Time

The running time of the `assign!` function depends entirely on running time of the `measure` function. The `measure` function computes a constant number of editing distances in the size of the cluster set, $C$.

The cost of each editing distance calculation depends on the length of the input string, $x$, and the reference string, $y$. This will depend on the application. RFC 5424, the specification for the Syslog protocol, does not specify a maximum length for messages. RFC 5424 does specify that Syslog receivers must accept messages that are 480 octets in length or longer. In practice, routers and switches generally send messages that are no longer than 200 octets in length.

Because the length of $x$ and $y$ varies, the lower bound for `measure` could be trivial. In a degenerate case, all clusters in the database could contain distinct one-character strings. In this situation, the editing distance calculation is simply

$$\texttt{editing\_distance\_normalized}(x, y) \in \Theta(|x||y|) = \Theta(|x|\Theta(1)) = \Theta(|x|)$$

and the time to `measure` is

$$\texttt{measure}(x, C) \in \texttt{editing\_distance\_normalized}(x, \Theta(1))\Theta(|C|) \in \Theta(|x|)\Theta(|C|) = \Theta(|x||C|).$$

This analysis generalizes if the length of strings is bound by some constant $|x|, |y| \in \Theta(1)$. If the length of the strings are unbounded, then we estimate the upper bound of the `measure` function is polynomial in the length $|x| \in O(m)$ and $|y| \in O(n)$ as

$$\texttt{measure}(x, C) \in O(mn|C|)$$

The number of clusters, $|C|$ will depend on the content of the inputs and the threshold value $t$ used in comparisons to decide whether or not the algorithm should form a new cluster. In the "best" case for fastest algorithm time, all input strings fall into the same cluster and $|C| = 1$. In the "worst" case, the strings differ so dramatically or the $t$-value is so sensitive to difference that all inputs end up in distinct clusters.

The *average* running time of this algorithm would require insights of the length of the alphabet, value of $t$, similarity of strings in terms of their editing distance and in relation to $t$, and even the order in which strings from each cluster are encountered. Such analysis is overly difficult, and it is simpler to simply consider $\texttt{measure}(x, C) \in O(mn|C|) = O(O(1)O(1)|C|) = O(|C|)$ and experimentally measure the constant factors $mn \in O(1)O(1)$.

## 6 Running Time Measurement

This analysis uses plain text lines from the King James Version book of Psalms. This data set is chosen for the reproduction of this analysis; the text is well-known and easily obtained. Additionally, many lines of this text are very similar to one another. Finally, these lines of text are comparable to Syslog messages found in routers and switches.

There are 2461 total lines in the book of Psalms.

[8]:
```
psalms = filter(startswith("Psa"), readlines("kjv.txt"))
length(psalms)
```

[8]:
```
2461
```

The average length is about 100 characters, which is substantially shorter than typical log messages from a network device.

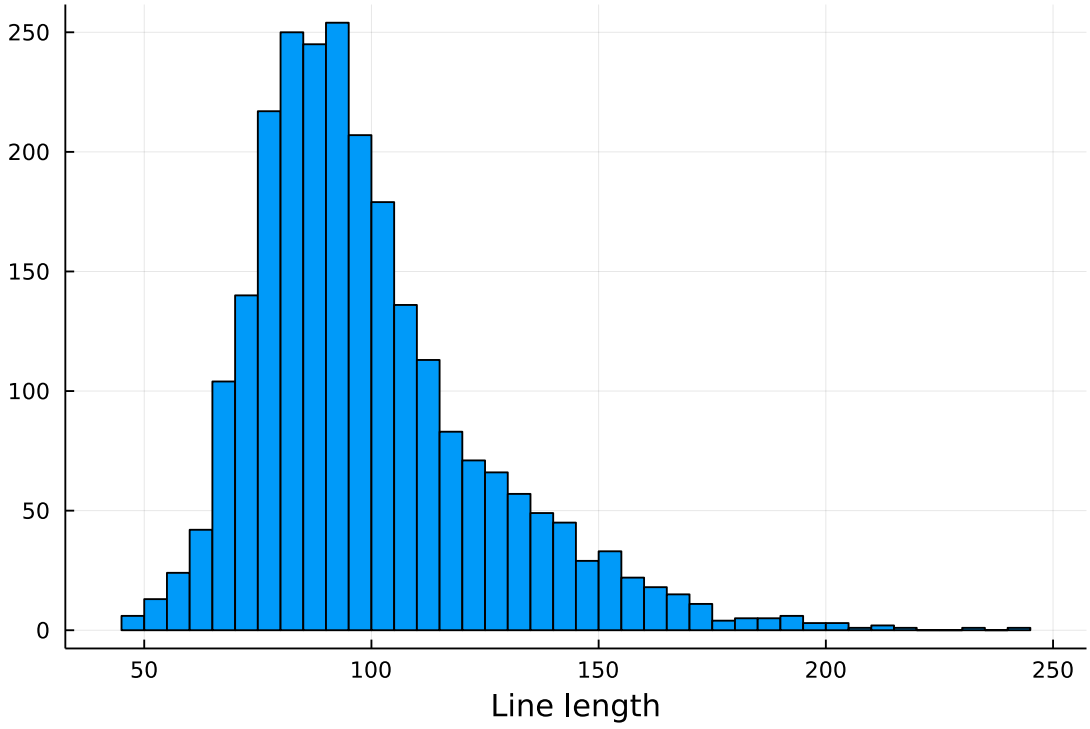
[9]:
```
using StatsBase
describe(map(length, psalms))
```

```
Summary Stats:
Length:         2461
Missing Count:  0
Mean:           98.748883
Minimum:        46.000000
1st Quartile:   81.000000
Median:         93.000000
3rd Quartile:   111.000000
Maximum:        240.000000
Type:           Int64
```

A histogram shows that the distribution of string lengths has a positive skew above the mean.

[10]:
```
using Plots
histogram(map(length, psalms), xlabel="Line length", legend=false)
```

[10]:



The following loop inserts the first 500 lines into a cluster database, $C$, and measures the time needed to do so using non-default threshold values $t = 0.6$ and sample size of 3.

[11]:
```
times = DataFrame(Time = AbstractFloat[], Size_C = Int[])
C = initialize()
for x in psalms[1:500]
    assign_time = @elapsed assign!(C, x, threshold = .6, sample_size = 3)
    push!(times, [assign_time, nrow(C)])
end
length(unique(C.Cluster))
```

[11]:
```
285
```

These parameters fail to organize the input into a small number of large clusters. This is actually not a problem with the algorithm, but with the input. The input messages themselves are quite different from one another. The largest cluster is shown below. It is not instantly obvious to the reader that these messages are similar to one another. Perhaps `measure` is fixating on the common substring "The…of the LORD".

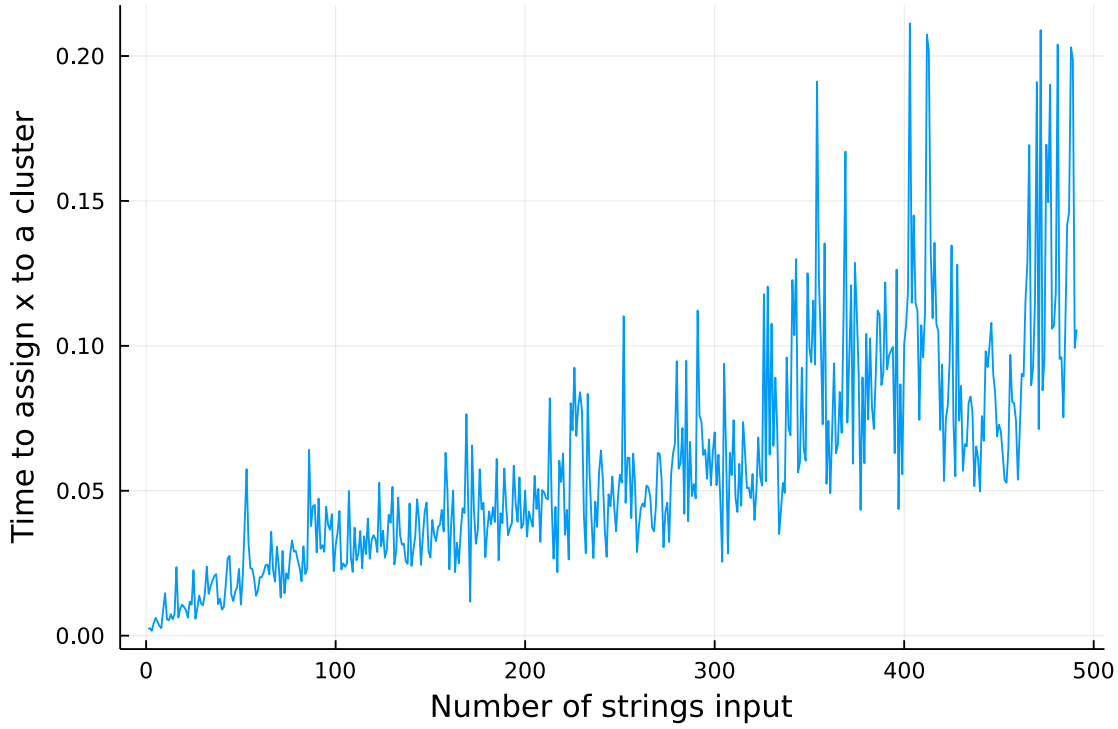
[12]:
```
occurrences, cluster = findmax(countmap(C.Cluster))
largest_cluster = C[C.Cluster .== cluster, :Message]
map(s -> s[1:60], largest_cluster)
```

[12]:
```
11-element Vector{String}:
 "Psa4:5 Offer the sacrifices of righteousness, and put your t"
 "Psa7:17 I will praise the LORD according to his righteousnes"
 "Psa9:1 I will praise thee, O LORD, with my whole heart; I wi"
 "Psa9:2 I will be glad and rejoice in thee: I will sing prais"
 "Psa9:11 Sing praises to the LORD, which dwelleth in Zion: de"
 "Psa15:2 He that walketh uprightly, and worketh righteousness"
 "Psa16:7 I will bless the LORD, who hath given me counsel: my"
 "Psa18:33 He maketh my feet like hinds' feet, and setteth me "
 "Psa20:7 Some trust in chariots, and some in horses: but we w"
 "Psa34:1 I will bless the LORD at all times: his praise shall"
 "Psa35:28 And my tongue shall speak of thy righteousness and "
```

Looking at the `assign!` times measured, we see that this function is roughly linear as $C$ grows in size. The first ten observations are omitted because of slowdowns associated with Julia's just-in-time (JIT) compilation.

[13]:
```
plot(times.Time[10:end], xlabel="Number of strings input",
    ylabel="Time to assign x to a cluster", legend=false)
```

[13]:




Repeating these linear time insertions `assign!` $\in O(|C|)$ with a growing $C$ means that the cumulative `assign!` times grow quadratically in $O(|C|^2)$.

## 7 Accuracy

ASCED trades correctness and determinism for speed. It is possible to iterate over all strings in each cluster in search of the best match, but this means that `measure` is not bounded by above by the number of clusters $|C|$, but rather bounded above and below by the number of strings $n$ that have been inserted into any cluster. With fixed-width strings but without sampling, `measure` $\in \Theta(n)$. If the input strings are similar to one another, thus forming a small number of clusters $|C| << n$, then sampling will result in significantly faster measurements.

Non-determinism is common among clustering algorithms (Kantardžić, 2020). The popular $k$-means algorithm produces different outputs that can be traced to random initial cluster assignments. Users of this algorithm should keep this in mind during parameter testing.

## 8 References

Bezanson, J., Karpinski, S., Shah, V. B., & Edelman, A. (2012). Julia: A Fast Dynamic Language for Technical Computing. arXiv:1209.5145 [Cs]. http://arxiv.org/abs/1209.5145

Cormen, T. H. (Ed.). (2009). Introduction to algorithms (3rd ed). MIT Press.

Gerhards, R. (2009). The Syslog Protocol (Request for Comments RFC 5424). Internet Engineering Task Force. https://datatracker.ietf.org/doc/rfc5424/

Kantardžić, M. (2020). Data mining: concepts, models, methods, and algorithms (3rd ed). John Wiley IEEE press.