Interactive Demo of a custom text fuzzy-matching algorithm

```
using PlutoUI    ;
```

```
include("string_similarity.jl");
```

```
heuristic_isequal (generic function with 1 method)
```

```
# naive equality, after stripping out spaces and standardizing letter-case
function heuristic_isequal(s1, s2)::Float64
    s1 = uppercase(replace(s1, " " => ""))
    s2 = uppercase(replace(s2, " " => ""))
    return isequal(s1, s2)
end
```

```
jaccard_similarity (generic function with 1 method)
```

```
# Jaccard Similarity = |intersection of characters| / |union of characters|
#  debatable whether the pre-processing of text to be standard improves the signal-to-noise, or hinders it for this algorithm
function jaccard_similarity(s1, s2)::Float64
    s1 = uppercase(replace(s1, " " => ""))
    s2 = uppercase(replace(s2, " " => ""))
    return round(length(intersect(Set(s1), Set(s2))) / length(union(Set(s1), Set(s2))), digits=3)
end
```

```
example_a =  String[
                1:  "42 - 123 HelloWorld Ave, Waterloo ON, A2C 4E6"
                2:  "Unit 42 123 Hello World Avenue Waterloo ON A2C4E6"
                ]
```

```
false
```

```
isequal(example_a[1], example_a[2])
```

```
0.0
```

```
heuristic_isequal(example_a[1], example_a[2])
```

```
0.81
```

```
jaccard_similarity(example_a[1], example_a[2])
```

```
0.906
```

```
string_compare(example_a[1], example_a[2])
```

Here is an illustrative example of the use case for this string-comparison algorithm, and some of the issues that are addressed by the design.

In just about any digital context, the two text strings from example_a are not equal. But, if asked whether or not those strings represent the same thing, or specifically in the case of the problem that motivated this algorithm 'do these two text representations refer to the same mailing address?' most human readers can say there is a high degree of confidence that they would in fact consider them as the same mailing address.

My goal was to create a general-purpose tool that could accelerate an existing manual-review process for duplicate addresses between two systems where the digital representations were often not 'equal' in the trivial sense, but where there was a high incidence of duplicates that were just slightly off

Algorithms such as taking the Jaccard Similarity (used for Power Query fuzzy-joins in Excel) offer a more granular approach to identifying similar text compared to the binary equal/not-equal operators without adding an excess of additional complexity.

My hypothesis was that a better-performing design (in terms of more closely matching human intuition) existed, if I could make use of additional information/assumptions that were part of the context that the manual reviewers were using.

Demonstration

```
run_test (generic function with 2 methods)
```

How to read the output from string_compare: An output of 0 indicates absolutely no similarity, which is incredibly unlikely in practical cases .

A similarity score in the range (0.0, 0.5) indicates very low similarity, where manual reviewers would almost certainly conclude the text is not meaningfully the same

(0.5, 0.8) is a fairly common level of similarity when comparing two valid addresses of the same format, but where a manual review would expect to find there are meaningful differences and conclude the addresses are not the same

Similarity scores above 0.8 indicates fairly high levels of similarity, and in a scenario where actual matches are expected to be abundant, these would be flagged for review (or simply confirmation)

Similarity equal to 1.0 means an exact match. Cases where this algorithm claims an exact match but a naive equality check does not is due to the basic pre-processing added to the function implementation for convenience of the intended application

A word on pre-processing: By default, string_compare strips out all spaces and ignores case/capitalization.

For the purpose of characterising the algorithm performance, all further examples will not focus on pre-processing that could be uniformly applied to any method.

(Note: choosing assumptions/business rules for preprocessing is an extremely important step, even before comparing or choosing algorithms)

```
example_b =  ["Test String", "TestString", "test string"]
```

```
# these are all the same, but the naive equality test fails to indicate that
example_b = ["Test String", "TestString", "test string"]
```

```
"similar pair: false VS differing pair: false"
```

```
run_test(example_b, isequal)
```

```
"similar pair: 1.0 VS differing pair: 1.0"
```

```
run_test(example_b, heuristic_isequal)
```

```
"similar pair: 1.0 VS differing pair: 1.0"
```

```
run_test(example_b, jaccard_similarity)
```

```
"similar pair: 1.0 VS differing pair: 1.0"
```

```
run_test(example_b, string_compare)
```

A series of examples consisting of three text strings, the first two should be rated more similar to each other (positive match), but the first and third should be less similar (negative match).

Text (strings) that share most characters should be more similar, even if there is an ommission/addition between the pair

```
example_c =  ["Thomas", "Tomas", "Data Science"]
```

```
"similar pair: 0.944 VS differing pair: 0.48"
```

```
"similar pair: 0.833 VS differing pair: 0.273"
```

Implicitly, text addresses of a similar length should end up more similar, even if there is substantial overlap in the set of characters

```
example_d =
 ["123 King St South Waterloo", "123 King Street S Waterloo", "123 Insurance Boulevard Kin
```

```
"similar pair: 0.892 VS differing pair: 0.57"
```

- `run_test(example_d, string_compare)`

```
"similar pair: 0.882 VS differing pair: 0.714"
```

- `run_test(example_d, jaccard_similarity)`

A 'bag-of-characters' approach like Jaccard without accounting for order is helpful in overlooking minor typos (false-negative matches). But that alone was not sufficiently sensitive for distinguishing actual differences (true-negative matches)

```
example_e =  ["Katherine", "Kahterine", "Katerinah"]
```

```
"similar pair: 0.963 VS differing pair: 0.852"
```

- `run_test(example_e, string_compare)`

```
"similar pair: 1.0 VS differing pair: 1.0"
```

- `run_test(example_e, jaccard_similarity)`

Some concept of order/localized matching must be implemented to better match the perspective of human readers of English.

As later discussed in more depth, while the motivation included comparing addresses between systems that did not have the same delineation/standardization/formatting of fields, it was valid to assume the parts of the addresses between systems were going to be in the same/similar order within the strings supplied to the function (ie unit, street, city, postal code)

```
example_f =  String[
                1:  "the sun life insurance company of canada"
                2:  "sun the life insurance company of canada"
                3:  "the life insurance company of canada sun"
                ]
```

```
"similar pair: 0.99 VS differing pair: 0.909"
```

- `run_test(example_f, string_compare)`

```
"similar pair: 1.0 VS differing pair: 1.0"
```

- `run_test(example_f, jaccard_similarity)`

The final implementation included a method for acknowledging matching characters within similar/localized sections between strings, but also penalizing transpositions based on how many characters were out of order, and even disregarding matching characters if they were outside that localized section

This turned out to be highly informative to the manual reviewers, and justified the effort to implement the changes to their workflow. And because there was nothing else available to offer these attributes in the format required, that justified my efforts to implement it (prototyped in Julia when it was a hobby-project, but the real-life application required VBA for use within Excel)

Try it!

The expectation is that a better-performing algorithm will rate more-similar pairs higher, and less-similar pairs lower. Further, a well-behaved similarity function will impose less of a penalty as 'less meaningful' changes are added between the compared texts, but changes that substantially change how the text address is interpreted should drastically lower the similarity score.

See if string_compare demonstrates this, based on texts that you consider to be more or less similar.

Put your example text here

Put your sample text here.

Enter something else here

```
interactive_example =   String[
                            1:  "Put your example text here"
                            2:  "Put your sample text here."
                            3:  "Enter something else here"
                            ]
```

"similar pair: 0.939 VS differing pair: 0.614"

```
run_test(interactive_example, string_compare)
```

"similar pair: 0.857 VS differing pair: 0.438"

```
run_test(interactive_example, jaccard_similarity)
```

"similar pair: 0.0 VS differing pair: 0.0"

```
run_test(interactive_example, heuristic_isequal)
```

"similar pair: false VS differing pair: false"

```
run_test(interactive_example, isequal)
```

0.954

```
# modify this code to see the influence of different parameters on your examples
string_compare(text_1, text_2,
    strip=[" ", "-", ",", "."],
    keep_case=true,
    ignore_short=5
)
```

Algorithm Design In-Depth

```
String[
    1:  "42 - 123 HelloWorld Ave, Waterloo ON, A2C 4E6"
    2:  "Unit 42 123 Hello World Avenue Waterloo ON A2C4E6"
    ]
```

0.963

```
string_compare(example_a[1], example_a[2],
    verbose=true,
    strip=[" ", "-", ",", "Unit"],
    keep_case=false,
    ignore_short=5
)
```

```
37 - 42123HELLOWORLDAVENUEWATERLOOONA2C4E6

34 - 42123HELLOWORLDAVEWATERLOOONA2C4E6

match dist - 6

  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 21, 22, 23, 24, 26, 27

  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 22, 19, 20, 21, 23, 24

matches - 34

transposes - 1
```

Try a verbose example

Try a non-concise example

Or not

"similar pair: 0.805 VS differing pair: 0.337"

0.805

```
string_compare(text_1v, text_2v,
    verbose=true,
    strip=[" "],
    keep_case=false,
    ignore_short=4
)
```

```
22 - TRYANON-CONCISEEXAMPLE

18 - TRYAVERBOSEEXAMPLE

match dist - 4

  [1, 2, 3, 4, 6, 14, 15, 16, 17, 18, 19, 20, 21, 22]

  [1, 2, 3, 4, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]

matches - 14

transposes - 0
```