When Are Names Similar Or the Same? Introducing the Code Names Matcher Library

Moshe Munk Dror G. Feitelson
department of Computer Science
The Hebrew University of Jerusalem

Abstract

Program code contains functions, variables, and data structures that are represented by names. To promote human understanding, these names should describe the role and use of the code elements they represent. But the names given by developers show high variability, reflecting the tastes of each developer, with different words used for the same meaning or the same words used for different meanings. This makes comparing names hard. A precise comparison should be based on matching identical words, but also take into account possible variations on the words (including spelling and typing errors), reordering of the words, matching between synonyms, and so on. To facilitate this we developed a library of comparison functions specifically targeted to comparing names in code. The different functions calculate the similarity between names in different ways, so a researcher can choose the one appropriate for his specific needs. All of them share an attempt to reflect human perceptions of similarity, at the possible expense of lexical matching.

Download: https://pypi.org/project/namecompare/
Code: https://github.com/AutoPurchase/name_compare

1 Introduction

One of the powerful tools for code comprehension research is investigating the names that are given to functions, variables, and data structures. Names bear witness to what the developer thought about the role of each part of code in the whole program [26]. There has therefore been substantial research on names and their meanings [3, 4, 9, 10, 11, 13, 16, 18, 21, 23, 27].

As part of this body of work, it has been shown that different developers often given different names to the same objects [17]. And renaming, where a maintainer changes a previously given name, is a common form of refactoring [7, 24]. Additional research has considered how easy it is to remember names [12, 15], and whether similar names may cause confusion [28, 5]. In all these contexts it is important to be able to compare names to each other. However, so far this has been done using general string-matching approaches, such as the edit distance. Even worse, using Python's default Sequence Matcher is susceptible to giving results that depend on the order of the inputs.

To advance the research on variable naming we suggest a library of comparison functions that are specifically designed for comparing names in code. These functions include features like focusing on matching words rather than disconnected letters, dealing with reordering, and considering semantic similarities. Importantly, they attempt to promote a metric of similarity that matches human perception, at the possible expense of lexical matching. Thus they are not as mathematically crisp as the longest common subsequence or edit distance metrics, but may be better tuned for research involving humans and names.

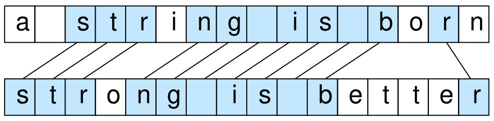
This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18)

2 Existing Algorithms for String Comparison

The most common algorithms for comparing two strings (or variable names) are the Longest Common Subsequence algorithm, the Edit Distance calculation, and Python's Sequence Matcher.

2.1 Longest Common Subsequence

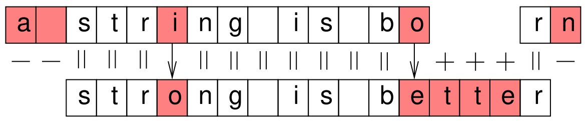
The Longest Common Subsequence algorithm (LCS) finds the longest sequence of characters that appear in the same order in both strings. The length of this sequence divided by the length of the shorter (or longer) input string can then be used as a metric for the degree of similarity between the strings. For example, if the input strings are "a string is born" and "strong is better", the longest common subsequence is "strng is br":



Thus 11 of 16 characters match, implying that the strings are quite similar to each other. Note that the letters in the common subsequence need not be consecutive to each other. When the *are* consecutive we call it a sub*string*.

2.2 Edit Distance

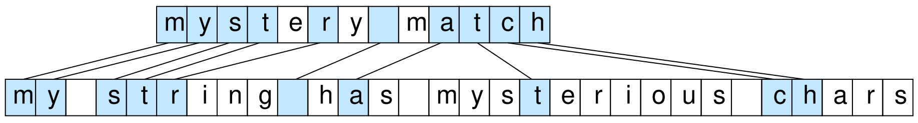
The Edit Distance (also known as the Levenshtein Distance [22]) counts the minimum number of edit operations required to transform one string into the other. This has been used in naming research by Tashima et al. [28]. There are several variants depending on which operations are allowed. The most commonly allowed operations are the insertion, deletion, or substitution of a single character; a possible fourth is the transposition of adjacent characters [14]. Using the above two strings as an example again, turning the first into the second requires the following edit operations: delete 2 characters 'a' and ' '; substitute 'i' with 'o'; substitute 'o' with 'e'; add 3 characters 't', 't', and 'e'; and delete the character 'n':



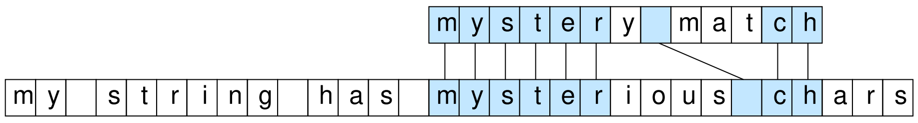
The total number of character operations which divide the strings is therefore 8. This can again be turned into a score by dividing by the length of longer input string [5]. (It is important to normalize relative to the longer string, because the edit distance can be larger than the length of the shorter one but is limited by the length of the longer one.) Note, however, that this score reflects the divergence between the strings, not their similarity. To turn it into a similarity score, subtract it from 1.

2.3 Sequence Matcher

The above algorithms sometimes produce counter-intuitive results. For example, if we use LCS to compare "my string has mysterious chars" with "mystery match", the longest common subsequence has 10 letters, but does not include a match of "mysterious" with "mystery". This is counter-intuitive to humans, who see this as the most prominent match between these two strings.



The Python Sequence Matcher algorithm, which is part of the difflib library, is intended to better match human intuitions, and has been used in naming research by Etgar et al. [15]. Interestingly, it has also been claimed to be advantageous for comparing sketch-based passwords [6]. It is based on the Ratcliff-Obershelp algorithm, which was designed in the context of educational software to be "forgiving and understanding of simple typing mistakes, and allow intelligent responses to erroneous input" [25]. The idea is to prefer matching *contiguous substrings* over matching subsequences of disjoint letters. Specifically, the algorithm starts by finding the longest contiguous substring of the two input strings, and then continues recursively on both sides of this substring. Using the above example again, the longest common substring is "myster", and the recursive calls will find nothing on the left but a space and "ch" on the right, for a total of 9 matching letters.



The final score is the relative length of the resulting subsequence of identical letters, in this case 9/21.5 (the divisor is the average length of the two input strings).

2.4 Shortcomings of Existing Algorithms

While all the above algorithms provide reasonable measures for the similarity or difference between strings, they also have shortcomings. Some of these are especially troublesome in the context of code comprehension studies involving variable names.

One problem, shared by the LCS and Edit Distance algorithms, is the possible fragmentation of the matched subsequences. For example, if we compare the input string "stop rampant germs at church" with the other string "strange match", we will find that the shorter string is perfectly matched in the longer one! But human beings reading these two strings will find this match artificial, because the two strings are composed of words that have no relation to each other.

Another problem is that these algorithms do not deal with changes in order. As humans we can see that "first second" is closely related to "second first". But LCS will just identify the longer repeated part and ignore the other, while calculating the edit distance will double-charge by suggesting a deletion from one place and an insertion in another.

The Python Sequence Matcher algorithm was designed to solve the problem of fragmentation. However, it too suffers from some drawbacks. The major one is that it is not symmetric, meaning that comparing string a with string b may lead to a different result than comparing b to a. The reason for this unfortunate "feature" is that there can be more than one substring with the same maximal length. The result then depends on which is found first and used as the root of the recursion, and this may depend on the order of the input strings[1]. For example, if our input strings are "FirstLightAFire" and "LightTheFireFirst" in this order, we will find that "First" is the longest common substring. But using this as the basis for matching the input strings leads to an empty remainder to the left in the first one, and an empty remainder to the right in the second. Therefore the recursive calls will not find any additional matches. However, if we reverse the order of the

[1] See https://stackoverflow.com/questions/35517353/how-does-pythons-sequencematcher-work.

inputs, “Light” becomes the first longest common substring. In this case the recursive call with the remainders to the right will find an additional match, “Fire”.

In the context of variable names (and actually also any other text), another issue is concerned with structure. Humans do not consider names and strings as sequences of letters — they parse them into words. So it may be beneficial to actually compare the words in strings and names, rather than just looking for uninterrupted sequences of letters as done by Python’s Sequence Matcher. In addition, acknowledging the role of words opens the door to also consider the use of synonyms, which may be composed of completely different letters, but convey the same meaning.

3 The New Library

Our name matching library is based on the following decisions:

1. To prefer long *contiguous substrings* over long disjoint subsequences, as in Python’s Sequence Matcher. As the example above indicates (matching 10 letters with LCS but only 9 with Python’s Sequence Matcher), this may lead to *sub-optimal* matchings. But it is supposed to be more in line with how humans look at strings.

2. To be *symmetric*, thus avoiding the main problem with Python’s Sequence Matcher. This is achieved by basing the match on the longest substring that maximizes the score rather than on the first one that is found.

3. To support a focus on *words* rather than letters. This is achieved by creating two versions of all the functions, one based on comparing letters and the other based on comparing words. It gives users the choice of whether to work at the level of letters or words. The difference is that respecting word boundaries better reflects the semantics of the names, and avoids situations such as that shown above where “mystery” was matched to “my string”. In addition, when comparing words long and short words receive the same weight.

4. To take possible *reorderings* into consideration. This is achieved by creating functions that allow reordering in addition to the original functions which only match in the same order. It is important in the case of variable names to acknowledge that variable names like countItems and itemsCount are actually the same. Users can use the difference between the score of the ordered and unordered functions to see the effect of word reordering.

5. To support the option of *semantic matching* based on synonyms. This is achieved by adding functions that use a dictionary of synonyms when comparing words. It is again important for variable name research, for example to recognize that itemsNumber and itemsCount are actually more or less the same. In addition, words forming singular-plural pairs or abbreviation-expansion pairs are also matched.

3.1 Provided Functionality

Based on the above, we wrote a set of functions for finding matches between strings (or variable names). Almost all of these functions use the concepts of Python’s Sequence Matcher — first finding the longest matching substring, and then matching both sides of the original match. The difference is that they seek the optimal matches, so the algorithm is necessarily symmetrical. Specific functions also enable matching between synonyms and singular/plural words, with a configurable “cost” for these cases vs. identical words. In addition, we developed another set of functions that also perform "cross matches", i.e. matches between the left side of the match in one string to the right side of the other string. For these functions we had to build a new scoring formula, because using the Sequence Matcher formula will give the same score to matches in any order, while clearly the preservation of order should lead to a higher score.

With all these variations, our library provides six main functions for comparing names:

1. **Ordered Match**: A function that operates on *letters*, which improves upon Python's Sequence Matcher algorithm by being symmetric and finding the matches that maximize the score.
2. **Ordered Words Match**: A similar function that operates on *words*. As a result, each element match is not binary (match or mismatch) as in letters, but can also have intermediate values when words are similar but not identical. This naturally affects the scoring.
3. **Ordered Semantic Match**: A function like the previous one, that uses a database of English synonyms and plural words to score semantically similar words.
4. **Unordered Match**: A function that operates on *letters*, which enables cross matches (from different sides of the previous match). Note that this is different from just taking the maximum of running the original functions on both orders of the input [15], because this needs to be done at all levels of the recursion.
5. **Unordered Words Match**: A function that operates on *words*, that enables cross matches.
6. **Unordered Semantic Match**: A function like the previous one, that scores on semantically similar words.

Some of these functions accept parameters which define nuances and variations in the matching, for example the threshold value for considering similar words to be a match. Full details are given below.

All of these functions return detailed information about the found matches, and specifically:

1. The "normalized" version of the input names (with all letters converted to lowercase and with word separators removed), and the list of words extracted from each one (when the function works on words).
2. The locations of all matches that were found, and, when the function works on words, the score of each individual word match.
3. The final score for the similarity between the two names, in the range $[0, 1]$.

In addition to the above functions we also have a function that divides a multi-word name to its constituent words, either based on underscores (or some other separator character provided by the user) or on camelCase notation. This is used to normalize names and allow for valid comparisons that reflect content and not style.

3.2 The Scoring formula

3.2.1 Scoring Rules

Before committing to a scoring formula, we developed ten desirable rules that the scoring formula should respect. These rules can then be used as guidelines when we weigh alternative formulas. The first three are preliminary requirements reflecting basic principles:

1. **Style neutrality**: the result of comparing names should not depend on style. Therefore changes in capitalization should be ignored, and the _ should be ignored in snake_case names.
2. **Non-optimality**: giving a higher score to the biggest match is not a requirement. Of course we want to promote good matches, but counting matching letters is not the only and most important yardstick.
3. **Normalization**: Matches are not absolute but relative to the length of the names. If the compared names have different lengths, this should be reflected by lowering the match score.

The next two rules are defined by equalities:

4. **Symmetry**: the result should be the same regardless of the order of the inputs. Thus for any matching function
$$\text{func}(str1, str2) = \text{func}(str2, str1)$$
5. **Consistency**: different functions should produce the same result in simple cases. The definition of "simple" cases is cases that are not handled by special versions of the functions. As we have special functions to handle multiple words, reordering, and semantic matching, the simple cases are names composed of a single word, with no reordering, and no semantic equivalences. For such cases and all matching functions
$$\text{func1}(str1, str2) = \text{func2}(str1, str2)$$

The other five rules are inequalities, which reflect different possible levels of matching:

6. **Focus**: adding extra baggage reduces the matching score. Thus for any matching function
$$\text{func}(\text{"match"}, \text{"match"}) > \text{func}(\text{"match"}, \text{"match.and.more"})$$
7. **Continuity**: preserving the continuity of adjacent items should always receive a higher score than when they are separated. Thus for all functions
$$\text{func}(\text{"begin.end.aaaaaa"}, \text{"begin.end.zzzzzz"}) > \text{func}(\text{"begin.end.aaaaaa"}, \text{"begin.middle.end"})$$
8. **Order preservation**: matching in the same order should always receive a higher score than when words or letters are reordered. And for functions that support reordering, matching with reordering should score higher than matching of unrelated strings. Thus for all functions
$$\text{func}(\text{"begin.end"}, \text{"begin.end"}) > \text{func}(\text{"begin.end"}, \text{"end.begin"})$$
and for reordering functions
$$\text{unordered}(\text{"begin.end"}, \text{"end.begin"}) > \text{unordered}(\text{"begin.xyz"}, \text{"wuv.begin"})$$
9. **Words weight equality**: when names are divided into words, long and short words have the same weight. Thus letter-matching achieves a higher score than word-matching on long words, and a lower score on short words:
$$\text{words}(\text{"gargantuan\_small"}, \text{"gargantuan\_little"}) < \text{letters}(\text{"gargantuan\_small"}, \text{"gargantuan\_little"})$$
and
$$\text{words}(\text{"gargantuan\_small"}, \text{"humongous\_small"}) > \text{letters}(\text{"gargantuan\_small"}, \text{"humongous\_small"})$$

10. **Semantics count**: exact matches should always score higher than those based on semantics. But for functions that support semantic matching, this should score higher than matching of unrelated words. Thus for all functions

$$\text{func("little", "little")} > \text{func("little", "small")}$$

and for semantic functions

$$\text{semantic("little", "small")} > \text{semantic("little", "smell")}$$

3.2.2 Crafting a Formula

It is impossible to reduce all the above rules to practice in a single formula applicable to all the special cases. We therefore made some compromises.

We start with the scoring formula used in Python's Sequence Matcher, called the ratio, and use it for scoring the similarity between sequences of letters. This formula works as follows: given two strings a and b, and letting m stand for all the matches between them, the similarity score between those two strings is

$$ratio = \frac{2|\mathsf{m}|}{|\mathsf{a}| + |\mathsf{b}|}$$

(where the | | refers to the number of letters in each string). This score is just the ratio between all the matching letters and the average length of both strings. It provides normalization, symmetry, and focus from the list of desired rules.

In functions which handle words we depart from the above formula. On one hand, we want to distinguish close-but-not-identical words. For example, the score when comparing "similar" to itself should be higher than when comparing it to "similarity". But at some point we need to recognize that the words are probably just different, e.g. when comparing "similar" with "smaller". Our compromise is to use the same ratio formula from above also when comparing words, but to define a minimal required threshold. If the threshold is not reached the similarity score between the words is set to 0. Note that for dissimilar words this breaks the requirement for consistency. The default threshold value is 2/3 (but it can be changed by a parameter). Using $ratio$ to denote the basic similarity between two words, the word similarity is then

$$word\_ratio = \begin{cases} ratio & \text{if } ratio \geq 2/3 \\ 0 & \text{otherwise} \end{cases}$$

when the compared names are composed of multiple words, the scores for the different words need to be combined. This is done using a simple generalization of the basic formula. Denoting the two inputs by a and b, and using m to denote the set of matching words (that is, words that score above the threshold), the formula is

$$multi\_word\_ratio = \frac{2 \sum_{w \in \mathsf{m}} word\_ratio(w)}{||\mathsf{a}|| + ||\mathsf{b}||}$$

where || || denotes the number of words in each name. This gives the same weight to each word.

When using semantic search, we need to decide on the value to assign to two dissimilar words which are found to be synonyms. We decided to use the threshold value used to identify matching words. In other words, in the functions that support the matching of synonyms and the such, these matches will be given a score of 2/3 — at the bottom of the range of scores for matching words. If the threshold is changed, so is the score for semantic matches.

To score for continuity we consider a string of n elements (letters or words) as if it was composed of $2n - 1$ elements. These elements are the original n elements, and an additional $n-1$ "glue" elements binding neighboring letters or words. For example, the string "ab" will be considered as being composed of "a", "a&b", and "b". When comparing it with another string "cab" all three will match. But if the other string is "acb", only the "a" and "b" elements will match, and the "a&b" glue element will not, leading to a lower total score.

The appropriate relative weight of glue elements is debatable. It could be any arbitrary configurable value, but this would be hard to justify. We therefore chose to use the highest possible value that respects a preference for longer matching over continuity. This means that we want n matching elements with no continuity to score higher than $n-1$ matching elements that are all continuous. To achieve this, the default score for all the glue elements together should be equal to the score for matching one basic element. So, when comparing names with n words each, each matching word adds $\frac{1}{n+1}$ to the score, and each matching glue adds $\frac{1}{(n-1)(n+1)}$ to the score. Importantly, this leads to a perfect score of 1 when the inputs are indeed identical. If the inputs have different numbers of elements, n is set to the average number, as in the basic matching formula above.

Finally, we need to consider the distinction between "straight matches" and "cross matches". As noted in the rules above, it is desirable to give a somewhat lower score to matches that require re-ordering. Note, however, that scoring for continuity already provides some score for order, because if words match after re-ordering their glue elements will not. We therefore decided to avoid the overhead of measuring any additional deviations in order, and make do with scoring continuity as detailed above. The price is that a comparison of "one_and_two_and_three_and_four" with "one_or_two_or_three_or_four" will achieve the same score (when reordering is allowed) as a comparison with "four_or_three_or_two_or_one": in both cases 4 of 7 words are matched, and no glue elements.

3.3 Algorithmic Effects

It should be noted that the scoring depends not only on the formula, but also on algorithmic decisions. We explicitly decided not to strive for the optimal match, meaning the one that would lead to the highest score. The reason was a desire to better match human intuitions.

This decision led to the selection of the following algorithms for matching. When matching letters, we start with the longest consecutive substring, and use it as an anchor. This may lead to a suboptimal score as shown above in Section 2.3. When matching words, we likewise search for the longest sequence of matching words. Again, this can lead to "missed" matches for other words, or not matching the longest words, and a reduced score.

For example, consider the matching of "multi_multiplayer" with "multiplayers_layer". Matching at the letters level finds "multiplayer" as the first match, and then does not find any additional matches, for a final score of 0.665. But with word matching, "multi" is matched with "multiplayers", and "multiplayer" is matched with "layer", which together with the glue connecting them leads to a score of 0.734. Beyond demonstrating possible differences in matching, this example also shows that it is probably impossible to find a single formulation that will be optimal in some sense for all possible cases.

3.4 Examples

We start with a simple example, comparing the multi-word strings “FirstLightAFire” and “LightTheFireFirst”. As noted above, the Python Sequence Matcher function gives different results depending of the otrder of the inputs. Our ordered matcher finds the optimal match regardless of input order.

difflib_match_ratio():		
FirstLightAFire LightTheFireFirst	0.312	matches 5 of 16 letters
LightTheFireFirst FirstLightAFire	0.562	switch inputs, now 9 of 16 letters match
ordered_match():		
FirstLightAFire LightTheFireFirst	0.557	finds optimal matching (the slightly lower score is due to partial continuity)

When we match words instead of letters, the score is slightly reduced, because matching a long word match does not add more to the score than matching a short word. And when stop words are ignored, the score is increased because the denominator is smaller.

ordered_words_match():		
FirstLightAFire LightTheFireFirst	0.400	matches 2 of 4 words and no glue
ordered_words_match(ignore_stop_words=True):		
FirstLightAFire LightTheFireFirst	0.625	matches 2 of 3 words (ignoring “a” and “the”) and 1 glue

If we allow reordering, more letters or words can be matched.

unordered_match():		
FirstLightAFire LightTheFireFirst	0.867	14 of 16 letters + 11 of 15 glue
unordered_words_match():		
FirstLightAFire LightTheFireFirst	0.600	3 of 4 words + 0 of 3 glue

In words matching, each word is a unit. This is exemplifid by comparing “multiword_name” with “multiple_words_name”. Using the default threshold of 2/3, only “name” matches. Reducing the threshold to 0.57 allows “multiword” to match “multiple”, but the “word” part is left unmatched. Reducing the threshold even further, to 0.5, allows “multiword” to match “words”, which adds the benefit of matching adjacent words. Finally, matching based on letters does even better, as it does not respect word boundaries.

ordered_words_match() [using the default min_word_match_degree=2/3]:		
multiword_name multiple_words_name	0.286	matches 1 word out of 2.5 and no glue
ordered_words_match(min_word_match_degree=0.57):		
multiword_name multiple_words_name	0.452	matches 2 words out of 2.5, with scores of 0.588 and 1, and no glue
ordered_words_match(min_word_match_degree=0.5):		
multiword_name multiple_words_name	0.637	lower threshold allows matching 2 words and glue for a higher score
ordered_match():		
multiword_name multiple_words_name	0.857	matches 13 of 15 letters and 10 glues

As another example consider the names “MultiplyDigitExponent” and “DigitsPowerMultiplying”. the Python Sequence Matcher function matches “multiply” and two additional letters. Our ordered match does the same if the minimal match length is set to 1. With the default minimal required match of 2 it only matches “multiply”.

difflib_match_ratio():		
MultiplyDigitExponent DigitsPowerMultiplying	0.465	10 letters out of 21.5
ordered_match(min_len=1):		
MultiplyDigitExponent DigitsPowerMultiplying	0.460	lower score due to partial continuity
ordered_match() [using the default min_len=2]:		
MultiplyDigitExponent DigitsPowerMultiplying	0.371	8 letters of 21.5 and partial continuity

When matching words, if we require exact matches then none are found. But “digit” matches “digits” with a score of 0.909, and “multiply” matches “multiplying” with a score of 0.842. Due to the order constraint, only one of these can be used, and the higher one is selected.

ordered_words_match(min_word_match_degree=1):		
MultiplyDigitExponent DigitsPowerMultiplying	0.000	no perfect matches
ordered_words_match() [using the default min_word_match_degree=2/3]:		
MultiplyDigitExponent DigitsPowerMultiplying	0.227	one out of 3 words matches with a score of 0.909, and no glue

if semantic matching is used, “exponent” is found to be related to “power”. If we require perfect matching, by setting a matching threshold of 1, this is applied to this semantic match, and there are no additional perfect matches. With the default matching threshold the semantic match is combined with a regular partial match.

ordered_semantic_match(min_word_match_degree=1):		
MultiplyDigitExponent DigitsPowerMultiplying	0.250	one semantic match considered a perfect match, and no glue
ordered_semantic_match() [using the default min_word_match_degree=2/3]:		
MultiplyDigitExponent DigitsPowerMultiplying	0.519	regular match scoring 0.909, semantic match with default 2/3, and one glue

And all the above options can also be combined with reordering.

unordered_match(min_len=1):		
MultiplyDigitExponent DigitsPowerMultiplying	0.782	17 matching letters and 12 glues
unordered_match() [using the default min_len=2]:		
MultiplyDigitExponent DigitsPowerMultiplying	0.693	2 is the default to avoid matching anagrams
unordered_words_match(min_word_match_degree=1):		
MultiplyDigitExponent DigitsPowerMultiplying	0.000	no perfect matches
unordered_words_match() [using the default min_word_match_degree=2/3]:		
MultiplyDigitExponent DigitsPowerMultiplying	0.438	two matches with scores of 0.909 and 0.842, and no glue matches
unordered_semantic_match() [using the default min_word_match_degree=2/3]:		
MultiplyDigitExponent DigitsPowerMultiplying	0.729	the above two matches, a semantic match scoring 2/3, and one glue

4 Code Design and implementation

class NamesMatcher

The main class that includes all the functionality is NamesMatcher. In its typical usecase it accepts two names (strings) to compare, and a set of parameters that control nuances of the matching. It then provides a set of functions that return the score for different types of matches, e.g. based on letters or words, and with or without reordering.

The constructor for NamesMatcher is:

```
NamesMatcher(
    name_1=NONE,                  # first name
    name_2=NONE,                  # second name
    case_sensitivity=False,       # True to retain case differences
    word_separators='_ \t\n',     # to separate words for word matching
    support_camel_case=True,      # to separate words for word matching
    numbers_behavior=NUMBERS_SEPARATE_WORD, # how to treat digits
    stop_words_list=⟨see below⟩   # words to ignore when matching
)
```

The parameters are:

case_sensitivity: whether to retain letter case in the comparisons. The default is to fold all letters to lowercase, so for example cRaZYcaP is the same as crazyCap. If you are analyzing code in a language that distinguishes between upper and lower case you might want to set this to True. Also if you are just comparing two strings. But if you are interested in the *meanings* of names, it is better to ignore case differences.

word_separators: letters that signify a word break. This is only relevant if you are using matching functions that operate on words. A set of such letters can be provided, and the appearance any one of them in a name indicates a word break. It is impossible to break only on a sequence of several letters.

support_camel_case: This too is only relevant if you are using matching functions that operate on words. The default is to recognize camelCase as an indication for word breaks.

Note: camelCase and word separators may be combined. Thus "FileMenu_saveAsOption" will be separated into 5 words.

numbers_behavior: how to treat digits that appear in names. There are three options:

1. NUMBERS_SEPARATE_WORD = a number is treated as a separate word, namely break words on both sides of the number
2. NUMBERS_IGNORE = match only on letters and ignore numbers as if they didn't exist
3. NUMBERS_LEAVE = digits are considered to be lowercase letters and conjoined with adjacent letters as one word

stop_words_list: a list of words to ignore when matching words. The default list includes the words a, are, as, at, be, but, by, for, if, of, on, so, the, there, was, where, were. This is derived from lists commonly used in information retrieval. However, some words are intentionally not included, such as "is" (which is often used in Boolean variable names like isUpper) and logical operators ("and", "or", and "not").

In place of the constructor, you can also use setters later. The provided setters are:

set_name_1(n)

set_name_2(n)

set_names(n_1, n_2)

set_case_sensitivity(cs)

set_word_separators(s)

set_support_camel_case(b)

set_numbers_behavior(nb)

set_stop_words(sw)

In addition there are getters for all the parameters, and also a couple of special ones for internal representations of the names:

get_norm_names(): return both names after normalization. If a name is undefined, None is returned. Normalization means three things:

- Convert the name to all lowercase (if case_sensitivity was not set)
- Erase word separators
- If numbers_behavior was set to NUMBERS_IGNORE, erase numbers

So the normalized version of "defaultIs_FOObar" is "defaultisfoobar".

get_words(): return an array of the words in name. The procedure for breaking a name into words is:

1. Break on the letters identified as word separators, and erase those letters.
2. Break on both sides of numbers if numbers_behavior was set to NUMBERS_SEPARATE_WORD.
3. Handle camelCase notation, unless support_camel_case was set to False. This is done in two steps:
   (a) If an uppercase letter is followed by lowercase letters, break before the uppercase one. This handles cases like “camelCase” ⇒ camel+Case, and also “USAToday” ⇒ USA+Today.
   (b) Break on an uppercase letter that appears after a lowercase letter. This handles cases like “theUSA” ⇒ the+USA.

   If numbers_behavior was set to NUMBERS_LEAVE, numbers are treated as lowercase letters.

The individual words are normalized as described above.

The main API methods provided by NamesMatcher are variants of the ratio() method of Python’s Sequence Matcher. They each perform a different type of matching and compute its score. Scores are in the range [0,1], with 0 representing no relation whatsoever and 1 representing complete identity. In addition to the score, these functions also provide full details of the matches that were found. This is conveyed by an object of class MatchingBlocks, which is described below.

The API methods are:

ordered_match()

This method finds the matches that maximize the ratio between the names. It is a basic string matching routine, working on individual letters, rather similar to Python’s Sequence Matcher. Logically it starts the matching with the longest consecutive match, and then continues recursively on both sides, thereby maintaining order. In practice, however, the implementation uses dynamic programming rather than recursion.

The method signature is

```
ordered_match(
    min_len=2,
    continuity_heavy_weight=False
)
```

The parameters are:

min_len: the minimum number of consecutive letters that will be considered as a match. The default is 2 to avoid the impact of matching individual unrelated letters.

continuity_heavy_weight: the weight given to continuity in the match. Setting this to True sets the weight of each continuity element to 1, like the weight of each matching letter, rather than the default where all of them together have a weight of 1.

In addition the method, as all others, is modulated by the parameters of the NamesMatcher object. For this method the only relevant parameter is case_sensitivity. To perform simple string matching on the strings as they appear set case_sensitivity to True and min_len to 1.

As this method operates on letters and not on words, it does not respect the word_separators parameter of NamesMatcher. In other words, the word separator letters will be matched like any other letter.

Note: this method uses dynamic programming for its calculation. As a result its running time is about m^2n^2, where m and n are number of letters in the first and second names, respectively.

unordered_match()

This method also finds the matches that maximize the ratio between the names, but it allows cross-matching, that is matches that do no maintain the original order in the two names being matched.

Logically it starts the matching with the longest consecutive match, and then continues recursively to match the concatenations of the leftovers from both sides. As a result the found matches need not respect the original order.

The implementation, however, is iterative rather than recursive. Each iteration is composed of two steps. First, we find the longest consecutive match, and record it. Then we overwrite the matching letters to block them out. In each name, the overwriting is done with some letter that does not appear in the other name, to avoid spurious additional matches. The iterations continue in this manner until no additional matches are found. Blocking out is used rather than erasing the matching letters because this way the indices of the letters in subsequent matches remain as in the original names.

The method signature and parameters are similar to the previous method:

```
unordered_match(
    min_len=2,
    continuity_heavy_weight=False
)
```

Note that if min_len is set to 1 this method will identify anagrams as perfect matches.

ordered_words_match()

This method finds the word matches that maximize the ratio between the names. Like the other methods based on words it respects word boundaries, and will not match one word with parts of multiple other words.

Logically the method starts by finding the matching sequence of words with the highest total ratio, and continues recursively on both sides, thereby maintaining order. The implementation, however, uses dynamic programming rather than recursion. If multiple highest matching sequences are found with exactly the same score, all of them are checked to find the one that leads to the maximal total score.

Note that this method does not guarantee that the found match will include the individual maximal matching word, because some other sequence may lead to a higher total score. Likewise, it does not guarantee that the final match includes the maximal number of matching words, because other shorted matches may include words with higher individual scores.

The method signature is

```
ordered_words_match(
    min_word_match_degree=2/3,
    prefer_num_of_letters=False,
    ignore_stop_words=False
)
```

The parameters are:

min_word_match_degree: a float in the range $(0, 1]$, which defines the minimum score for two words to be considered as a match. Setting this to 1 means a perfect match is required.

prefer_num_of_letters: When there are two or more matches with the same maximal ratio, setting this to True will cause the best match to be chosen based on the number of letters rather then the number of words.

ignore_stop_words: if True, ignore any word appearing in the list of stop words and do not count them when scoring the match. This is only done if there is a perfect match with the word.

Note: this method uses dynamic programming for its calculation. As a result its running time is about m^2n^2, where m and n are number of words in the first and second names, respectively.

unordered_words_match()

This method also finds the word matches that maximize the ratio between the names, but it allows cross-matching, that is matches that do no maintain the original order in the two names being matched. Logically the method starts by finding the matching sequence of words with the highest total ratio, and continues recursively to match the concatenations of the leftovers from both sides. As a result the found matches need not respect the original order. The implementation works by erasing the found match and repeating as long as additional matches are found, as described above for unordered_match().

The method signature and parameters are similar to the previous method:

```
unordered_words_match(
        min_word_match_degree=2/3,
        prefer_num_of_letters=False,
        ignore_stop_words=False
)
```

ordered_semantic_match()

This and the next method are the same as the previous two, except for allowing semantic matches between individual words. This aims to identify semantically similar names even if different words are used. The implementation is based on the following considerations regarding what words are considered to be semantically equivalent:

- Synonyms are considered to be semantically equivalent and therefore match. Synonyms are identified based on a an open thesaurus[2].
- Words that differ only in number, that is singular and plural, are considered to be semantically equivalent and therefore match. This includes ending with “s” or “es”, and a list of special

[2]https://raw.githubusercontent.com/zaibacu/thesaurus/master/en_thesaurus.jsonl

cases (e.g. “half” and “halves”)[3].

- Abbreviations and their extensions are considered to be semantically equivalent and therefore match. This is identified by one word being a prefix of the other, coupled with the requirement that the shorter one be at least 3 letters long.

The method signature is

```
ordered_semantic_match(
    min_word_match_degree=2/3,
    prefer_num_of_letters=False,
    ignore_stop_words=False
)
```

The first parameters has an added role:

min_word_match_degree: a float in the range $(0, 1]$, which defines the minimum score for two words to be considered as a match. This is also used as the score for semantically matching words, when the calculated score is lower.

And the last one may have special significance:

ignore_stop_words: stop words (in, if, of, the, that, etc.) are considered semantically meaningless. To avoid diluting the score with such meaningless matches, these words can be removed from consideration by setting this parameter to True. The list of stop words is set by a parameter to the NamesMatcher constructor.

unordered_semantic_match()

Same as the previous method, except that reordering of words is allowed. The reordering is done as in unordered_words_match().

The method signature and parameters are similar to the previous method:

```
unordered_semantic_match(
    min_word_match_degree=2/3,
    prefer_num_of_letters=False,
    ignore_stop_words=False
)
```

unedit_match()

This method is a variant on unordered_match(), in which found matches are indeed erased rather than being blocked out. This enables subsequent matches to span both sides of previous matches, which may be useful in some cases. Note that this only makes a difference for short matches, where either or both of the matching parts individually are shorter than min_len.

[3]https://github.com/tagucci/pythonrouge/blob/master/pythonrouge/RELEASE-1.5.5/data/WordNet-2.0-Exceptions/noun.exc

As the found matches may be discontinuous, this requires a complicated recording of matches locations.

The method signature is

```
unedit_match(
    min_len=2,
    continuity_heavy_weight=False
)
```

In addition to the above, NamesMatcher also includes 3 methods implementing traditional string matching algorithms. These are actually wrappers to implementations in other libraries. They return the score of the found match, and not a full MatchingBlocks object like the previous methods. These methods are:

edit_distance(enable_transposition=False)

An implementation of edit distance calculation, based on insert, delete, and substitute operations. This has two versions, which come from the strsimpy library. The default is to use the Levenshtein() method. The alternative, used is enable_transposition is set to True, is to use the damerau() method. This adds a fourth basic operation, which is to transpose adjacent letters.

normalized_edit_distance(enable_transposition=False)

This is an implementation of the edit distance calculation, like the previous method. The difference is that the calculated edit distance is normalized relative to the length of the longer input.

difflib_sequence_matcher()

This is a wrapper for Python's Sequence Matcher ratio() function.

class MatchingBlocks

As noted above, most of the matching methods return all the data about their results in a MatchingBlocks object. Such objects contain the following members:

name_1: the first name as input to the matching (could be normalized or array of words)

name_2: the second name as input to the matching (could be normalized or array of words)

matching_type: LETTERS_MATCH or WORDS_MATCH

ratio: the calculated ratio between the two names

matches: array of matches (OneMatch objects, describing the locations and lengths of all matches)

cont_type: a Boolean indicating if the match is necessarily continuous or not (discontinuous only for unedit_match())

continuity_heavy_weight: how glue elements were weighted

5 Limitations

We believe our matching functions provide significant advantages over previous ones, especially for measuring the similarity of names in code. For example, they avoid the asymmetry of Python's Sequence Matcher. However, they have their limitations.

In our implementation we preferred breadth over depth: we wanted to incorporate many ideas, and made do with a basic implementation of each one. Thus the results may be improved by using better dictionaries or word-splitting procedures [19, 20]. In addition, our implementation assumes that it will only be called to compare reasonably short names, and not long texts. In some of the functions we use dynamic programming or other algorithms that do not scale very well.

One of the perennial problems with splitting is to identify conjoined words like “multiword” or “schoolbus”. The common approach is to try all possible splittings and compare to a dictionary. This might be reasonable for 2 words, but not for more. In our case, if both names include the same conjuncts the problem is moot. And if one contains the individual base words, or even just some of them, comparing the results of letter-matching with word-matching can provide the necessary information on how to split.

Also, we argued above in Section 3.3 that matching words should start with the highest single match, even if this comes at the expense of not achieving the highest total score. A possible alternative may be to check all possible matches, and find the combination that leads to the highest score. Our choice was based on anecdotal evidence that this leads to matches that seem better. But it may be that better compromises can be found.

Our scoring function also completely ignores reordering, except as detected by lack of continuity. And indeed, it seems hard to create a formula that acknowledges both continuity and order preservation. When names with only two words are considered, continuity and order are actually the same thing, so this would lead to double counting. This may also happen when there are more words, e.g. when the last two are switched. In other cases we need to divide the extra score between the two attributes in some way. And if we want to retain the preference for higher scores for more matches, the total attributed to continuity and order is reduced as the number of words grows. Future work may better resolve these issues.

Finally, our work like the vast majority of work on comparing names places a focus on the matching letters which compose words. However, the mapping of words to meanings is not one-to-one. We acknowledge this by providing functions that attempt to take semantics into account by looking for synonyms. However, it is also important to try to avoid homonyms — words with the same spelling that mean different things [8]. A possible approach is to use natural language approaches based on big data to identify words that appear in similar contexts. However, such approaches suffer from possible confounding of synonyms and antonyms [1]. For example, Table 1 in the groundbreaking paper by Alon et al. introducing code2vec includes “notEqual” as one of the tokens that appear in the same contexts as “equal” [2]. Dealing with this would require a whole new level of analysis, which we leave to future work.

References

[1] M. A. Ali, Y. Sun, X. Zhou, W. Wang, and X. Zhao, “*Antonym-synonym classification based on new sub-space embeddings*”. In *Conf. Artificial Intelligence*, vol. 33, pp. 6204–6211, Jan 2019, DOI: 10.1609/aaai.v33i01.33016204.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “*code2vec: Learning distributed representations of code*”. *Proc. ACM Programming Languages* **3(POPL)**, art. 40, Jan 2019, DOI: 10.1145/3290353.

[3] R. S. Alsuhaibani, C. D. Newman, M. J. Decker, M. L. Collard, and J. I. Maletic, “*On the naming of methods: A survey of professional developers*”. In 43rd *Intl. Conf. Softw. Eng.*, pp. 587–599, May 2021, DOI: 10.1109/ICSE43902.2021.00061.

[4] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "*A large-scale investigation of local variable names in Java programs: Is longer name better for broader scope variable?*" In 14th *Quality of Inf. & Commun. Tech.*, pp. 489–500, Sep 2021, DOI: 10.1007/978-3-030-85347-1_35.

[5] H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "*An investigation of compound variable names toward automated detection of confusing variable pairs*". In 36th *Intl. Conf. Automated softw. Eng. Workshops*, pp. 133–137, Nov 2021, DOI: 10.1109/ASEW52652.2021.00036.

[6] S. Amarnadh, P. V. G. D. Prasad Reddy, and N. V. E. S. Murthy, "*Freehand sketch-based authenticated security system – a comparative study on convolutional neural network, Levenshtein distance and sequence matcher procedure*". *Materials Today: Proc.* 2021, DOI: 10.1016/j.matpr.2021.01.478.

[7] V. Arnaoudova, L. M. Eshkevari, M. Di Penta, R. Oliveto, G. Antoniol, and Y.-G. Guéhéneuc, "*REPENT: Analyzing the nature of identifier renamings*". *IEEE Trans. Softw. Eng.* **40(5)**, pp. 502–532, May 2014, DOI: 10.1109/TSE.2014.2312942.

[8] V. Arnaoudova, L. Eshkevarti, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol, "*Physical and conceptual identifier dispersion: Measures and relation to fault proneness*". In 26th *Intl. Conf. Softw. Maintenance*, Sep 2010, DOI: 10.1109/ICSM.2010.5609748.

[9] E. Avidan and D. G. Feitelson, "*Effects of variable names on comprehension: An empirical study*". In 25th *Intl. Conf. Program Comprehension*, pp. 55–65, May 2017, DOI: 10.1109/ICPC.2017.27.

[10] G. Beniamini, S. Gingichashvili, A. Klein Orbach, and D. G. Feitelson, "*Meaningful identifier names: The case of single-letter variables*". In 25th *Intl. Conf. Program Comprehension*, pp. 45–54, May 2017, DOI: 10.1109/ICPC.2017.18.

[11] D. Binkley and D. Lawrie, "*The impact of vocabulary normalization*". *J. Softw.: Evolution & Process* **27(4)**, pp. 255–273, Apr 2015, DOI: 10.1002/smr.1710.

[12] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "*Identifier length and limited programmer memory*". *Sci. Comput. Programming* **74(7)**, pp. 430–445, May 2009, DOI: 10.1016/j.scico.2009.02.006.

[13] S. Butler, M. Wermelinger, and Y. Yu, "*A survey of the forms of Java reference names*". In 23rd *Intl. Conf. Program Comprehension*, pp. 196–206, May 2015, DOI: 10.1109/ICPC.2015.30.

[14] F. J. Damerau, "*A technique for computer detection and correction of spelling errors*". *Comm. ACM* **7(3)**, pp. 171–176, Mar 1964, DOI: 10.1145/363958.363994.

[15] A. Etgar, R. Friedman, S. Haiman, D. Perez, and D. G. Feitelson, "*The effect of information content and length on name recollection*". In 30th *Intl. Conf. Program Comprehension*, pp. 141–151, May 2022, DOI: 10.1145/3524610.3529159.

[16] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope, "*Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization*". *Empirical Softw. Eng.* **25(3)**, pp. 2140–2178, May 2020, DOI: 10.1007/s10664-019-09751-4.

[17] D. G. Feitelson, A. Mizrahi, N. Noy, A. Ben Shabat, O. Eliyahu, and R. Sheffer, "*How developers choose names*". *IEEE Trans. Softw. Eng.* **48(1)**, pp. 37–52, Jan 2022, DOI: 10.1109/TSE.2020.2976920.

[18] E. M. Gellenbeck and C. R. Cook, "*An investigation of procedure and variable names as beacons during program comprehension*". In *Empirical Studies of Programmers: Fourth Workshop*, J. Koenemann-Belliveau, T. G. Moher, and S. P. Robertson (eds.), pp. 65–81, Intellect Books, 1991.

[19] E. Hill, D. Binkley, D. Lawrie, L. Pollock, and K. Vijay-Shanker, "*An empirical study of identifier splitting techniques*". *Empirical Softw. Eng.* **19(6)**, pp. 1754–1780, Dec 2014, DOI: 10.1007/s10664-013-9261-0.

[20] M. Hucka, "*Spiral: Splitters for identifiers in source code files*". *J. Open Source Softw.* **3(24)**, art. 653, 2018, DOI: 10.21105/joss.00653.

[21] D. Lawrie, C. Morrell, H. Field, and D. Binkley, "*Effective identifier names for comprehension and memory*". *Innovations in Syst. & Softw. Eng.* **3(4)**, pp. 303–318, Dec 2007, DOI: 10.1007/s11334-007-0031-2.

[22] V. I. Levenshtein, "*Binary codes capable of correcting deletions, insertions, and reversals*". *Soviet Physics – Doklady* **10(8)**, pp. 707–710, Feb 1966.

[23] C. D. Newman, R. S. AlSuhaibani, M. J. Decker, A. Peruma, D. Kaushik, M. W. Mkaouer, and E. Hill, "*On the generation, structure, and semantics of grammer patterns in source code identifiers*". *J. Syst. & Softw.* **170**, art. 110740, Dec 2020, DOI: 10.1016/j.jss.2020.110740.

[24] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman, "*An empirical investigation of how and why developers rename idnetifiers*". In 2nd *Intl. Workshop on Refactoring*, pp. 26–33, Sep 2018, DOI: 10.1145/3242163.3242169.

[25] J. W. Ratcliff and D. E. Metzener, "*Pattern matching: The gestalt approach*". *Dr. Dobbs J.* Jul 1988. Https://www.drdobbs.com/database/pattern-matching-the-gestalt-approach/184407970?pgno=5.

[26] F. Salviulo and G. Scanniello, "*Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals*". In 18th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. 48, May 2014, DOI: 10.1145/2601248.2601251.

[27] A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "*Descriptive compound identifier names improve source code comprehension*". In 26th *Intl. Conf. Program Comprehension*, pp. 31–40, May 2018, DOI: 10.1145/3196321.3196332.

[28] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara, "*Fault-prone Java method analysis focusing on pair of local variables with confusing names*". In 44th *Euromicro Conf. Softw. Eng. & Advanced Apps.*, pp. 154–158, Aug 2018, DOI: 10.1109/SEAA.2018.00033.