Fuzzy Matching

## We need to automatically match product names (cameras, laptops, tv-s etc) that come from different sources to a canonical name in the database.

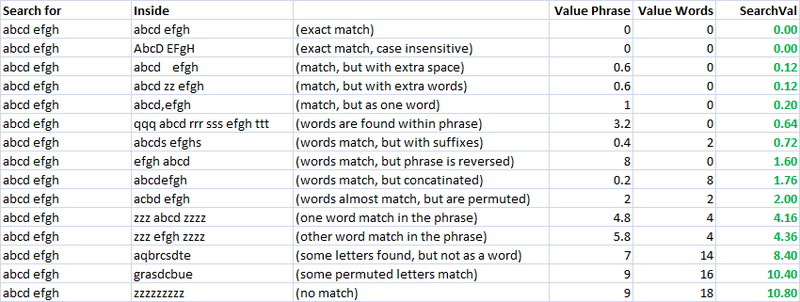
## For example **"Canon PowerShot a20IS"**, **"NEW powershot A20 IS from Canon"** and **"Digital Camera Canon PS A20IS"** should all match **"Canon PowerShot A20 IS"**. We worked with levenshtein distance with some added heuristics (removing obvious common words, assigning higher cost to number changes etc), which works to some extent, but not well enough unfortunately.

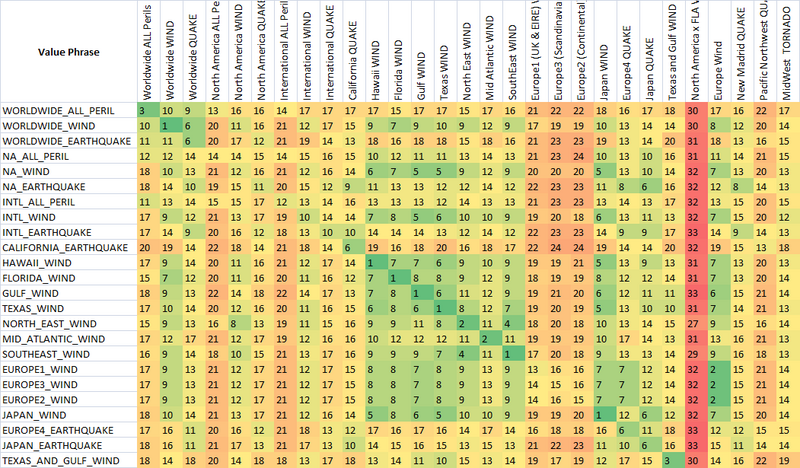
## The main problem is that even single-letter changes in relevant keywords can make a huge difference, but it's not easy to detect which are the relevant keywords. Consider for example three product names: **Lenovo T400** **Lenovo R400** **New Lenovo T-400, Core 2 Duo**

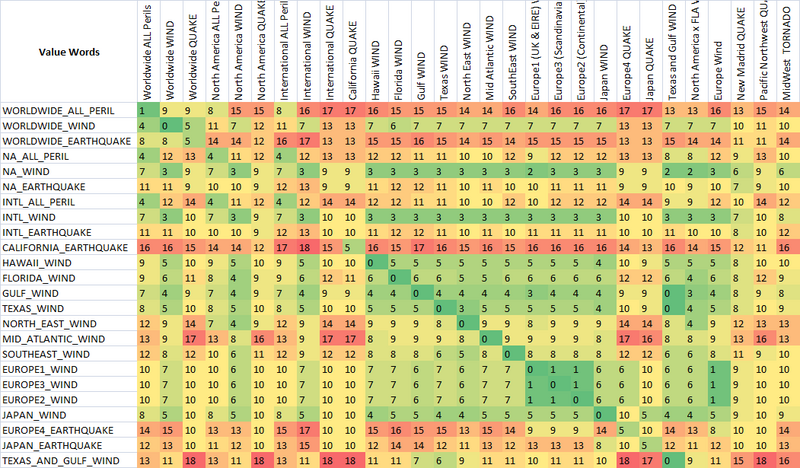
Solution: Initially implementing the [Levenshtein distance](http://en.wikipedia.org/wiki/Levenshtein_distance) algorithm, which determines how many changes must be made to a string or phrase to turn it into another string or phrase.

Fuzzy String Matching is the process of performing a human-like estimation of the similarity of two words or phrases. In many cases, it involves identifying words or phrases which are most similar to each other. This article describes an in-house solution to the fuzzy string matching problem and its usefulness in solving a variety of problems which can allow us to automate tasks which previously required tedious user involvement.

Simple, speedy, and a very useful metric. Using this, we created two separate metrics for evaluating the similarity of two strings. One I call "valuePhrase" and one I call "valueWords". valuePhrase is just the Levenshtein distance between the two phrases, and valueWords splits the string into individual words, based on delimiters such as spaces, dashes, and anything else you'd like, and compares each word to each other word, summing up the shortest Levenshtein distance connecting any two words. Essentially, it measures whether the information in one 'phrase' is really contained in another, just as a word-wise permutation. We spent a few days as a side project coming up with the most efficient way possible of splitting a string based on delimiters.







To allow the optimization of fuzzy matching, we weight each metric. As such, every application of fuzzy string match can weigh the parameters differently. The formula that defines the final score is a simply combination of the metrics and their weights:

value = Min(phraseWeight\*phraseValue, wordsWeight\*wordsValue)\*minWeight

+ Max(phraseWeight\*phraseValue, wordsWeight\*wordsValue)\*maxWeight

+ lengthWeight\*lengthValue

**The algorithm was a wonderful success, and the solution parameters say a lot about this type of problem. You'll notice the optimized score was 44, and the best possible score is 48.**

