Weighting Edit Distance to Improve Spelling Correction in Music Entity Search

AXEL SAMUELSSON
Weighting Edit Distance to Improve Spelling Correction in Music Entity Search

Axel Samuelsson
axelsam@kth.se

Master’s Thesis in Computer Science
School of Computer Science and Communication
KTH Royal Institute of Technology

Principal: Spotify AB
Supervisor at Spotify: Daði Bjarnason
Supervisor at KTH: Jens Edlund
Examiner: Olov Engwall

Stockholm – June 2017
ABSTRACT

This master's thesis project undertook investigation of whether the extant Damerau-Levenshtein edit distance measurement between two strings could be made more useful for detecting and adjusting misspellings in a search query. The idea was to use the knowledge that many users type their queries using the QWERTY keyboard layout, and weighting the edit distance in a manner that makes it cheaper to correct misspellings caused by confusion of nearer keys. Two different weighting approaches were tested, one with a linear spread from 2/9 to 2 depending on the keyboard distance, and the other had neighbors preferred over non-neighbors (either with half the cost or no cost at all). They were tested against an unweighted baseline as well as inverted versions of themselves (nearer keys more expensive to replace) against a dataset of 1,162,145 searches. No significant improvement in the retrieval of search results were observed when compared to the baseline. However, each of the weightings performed better than its corresponding inversion on a p < 0.05 significance level. This means that while the weighted edit distance did not outperform the baseline, the data still clearly points toward a correlation between the physical position of keys on the keyboard, and what spelling mistakes are made.

SAMMANFATTNING

Svensk titel: Viktat ändringsavstånd för förbättrad stavningskorrigering vid sökning i en musikdatabas.

Detta examensarbete åtog sig att undersöka om det etablerade Damerau-Levenshtein-avståndet som mäter avståndet kan anpassas för att bättre hitta och korrigera stavningsfel i sökfrågor. Tanken var att använda det faktum att många användare skriver sina sökfrågor på ett tangentbord med QWERTY-layout, och att vikta ändrings- avståndet så att det blir billigare att korrigera stavfel orsakade av hopblandning av två knappar som är närmare varandra. Två olika viktningar testades, en hade vikterna utspridda linjärt mellan 2/9 och 2, och den andra föredrog grannar över icke-grannar (antingen halva kostnaden eller ingen alls). De testades mot ett oviktat referensavstånd samt inversen av sig själva (så att närmare knappar blev dyrare att byta ut) mot ett dataset bestående av 1 162 145 sökningar. Ingen signifikant förbätting uppmättes gentemot referensen. Däremot presterade var och en av viktningarna bättre än sin inverterade motpart på konfidensnivå p < 0,05. Det innebär att trots att de viktade distansavstånden inte presterade bättre än referensen så pekar datan tydligt mot en korrelation mellan den fysiska positioneringen av knapparna på tangentbordet och vilka stavningsmisstag som begås.
# Table of Contents

**Introduction**  
1

**Background**  
2
  * Edit Distance Definitions  
  2
  * Levenshtein  
  2
  * Damerau-Levenshtein  
  2
  * Keyboard Layouts  
  3
  * Current System in use at Spotify  
  4
  * Related Work  
  4
    * Google  
    4
    * Others  
    5

**Method**  
7
  * The Algorithm  
  7
    * Deletion  
    8
    * Insertion  
    8
    * Substitution  
    9
    * Transposition  
    9
  * The Dataset  
  9
  * The Experiment  
  9
    * Statistics  
    9
    * Simulation  
    10

**Results**  
11
  * Statistical Analysis  
  11
  * Simulations  
  14

**Discussion**  
16
  * Conclusions  
  16
  * Potential Applications and Interest  
  16
  * Ethics and Sustainability  
  17
  * Issues and Limitations  
  18

**Acknowledgements**  
19

**References**  
20
Introduction

Spelling is difficult. Despite that fact, people have to spell all the time, and are expected to do so correctly. When people communicate with other people, this is generally a lesser issue, because people can intuit meaning from context. When computers need to understand people on the other hand, things get a lot more problematic.

In 1965, Vladimir Levenshtein published a mathematical paper about calculating a measure of distance between two strings – Levenshtein distance (LD) – that has held up remarkably well and is still in widespread use today [1]. LD can be used to give an indication that a given string is actually a misspelling of a known dictionary string. For example, neither backround nor fshksagd are English words, but LD can tell us that the former is probably a misspelling of background (LD=1) whereas the latter isn’t close to anything in the dictionary and is probably just nonsense [2].

However, LD is not without issues. Two problematic examples are backround and xork. The former has LD=2 to background despite having all the characters, just two swapped, arguably making it closer to correct than background. The latter has LD=1 to both cork and fork, with no distinction as to which is a better candidate. But looking at a keyboard, we could guess cork based on the fact that the c and x keys are adjacent, and we can make an educated guess that the word was typed on a standard computer keyboard. Further work based upon Levenshtein distance is needed to find such other dimensions, like keyboard distance or phonetic similarity [3].

The general idea for this thesis project is weighting edit distances – for example according to the physical distance between the keys on a keyboard, or preferring phonetically similar letters, such as cand k. Or stated as a research question:

Will a search engine using edit distance weighted by key distance more often return the desired result than a search engine using standard edit distance?

The principal of this thesis is the music streaming company Spotify. More specifically the ideas forming this thesis are of interest to attempt improvements in their search engine and its ability to retrieve the right result for search queries that contain spelling errors. With a database of 30+ million songs, and many related artists and albums, searching the whole dictionary using a straight LD matching against every single item is already infeasible. Instead, Spotify searches using a trie, and eligible branches are chosen by measuring edit distance from the typed query. The distance measurement created in this project will be integrated in the same way.
Background

For certain types of applications, searching is the only reasonable means of navigation, especially when the users know what they are looking for, but not where to find it, and the information is not organized. For example, an encyclopedia is alphabetically ordered and hence lets you find the correct page on your own with relative ease when looking up a word, e.g. *tiramisu*. A cookbook, on the other hand, is perhaps grouped into starters, mains and desserts, then types of food. So if you have never heard of *tiramisu* you need to start in the index to be able to find it.

A media database, such as Spotify’s catalog is of course different from an encyclopedia or a cookbook, but the same principles still apply. Spotify’s catalog contains more than 30 million songs spread out over several million artists and albums. On top of that there are over 100 million active users and they have created some 2 billion playlists. Browsing that number of entities is infeasible, regardless of their internal organization. One of the current trends is attempting to predict user behavior, but this art has not been perfected, and so the search feature is still key.

Edit Distance Definitions

Levenshtein
Levenshtein distance, after Vladimir Levenshtein, consists of three atomic operations: insertion, deletion and substitution. The LD between strings *a* and *b* is defined as the smallest number of these operations needed to transform *a* into *b*. E.g. the LD from *knect* to *end* is 3: substitute the *k* with an *e*, substitute one *e* with a *d*, and remove the other *e* – or if one could find another way to change one into the other in three operations, it doesn’t matter which operations are performed, just the minimum required number of them.

While this measure is common in computer science applications of natural language problem solving, and is taught extensively at KTH among other places, the original paper is mostly mathematical in nature. It makes no assertions as to the measurement’s applicability to language problems, although its usefulness in that area has been extensively tested by others dating back at least 40 years [2].

Damerau-Levenshtein
Damerau-Levenshtein distance (DLD) – named for Frederick J. Damerau in addition to Levenshtein – adds an additional operation: transposition of adjacent characters [4].

While not directly related to this thesis, it is noteworthy that Damerau actually published his paper in March of 1964, at least 8 months before Levenshtein’s paper from 1965. More to the point of this thesis, Damerau’s is a paper on natural language analysis, while Levenshtein’s is almost purely mathematical. Damerau’s paper concerns misspellings he has seemingly encountered himself, and the most feasible way to automatically correct them.

The paper is very practically written, sprung from a real need to fix issues in a coordinate indexing system with rigorous input rules, and at the time of writing, the index cards were proofread by humans. Damerau came up with his automated solution after an investigation showed that over 80% of the errors were because of a single deletion,
insertion, substitution or transposition of adjacent characters. In his words: “These are the errors one would expect as a result of misreading, hitting a key twice, or letting the eye move faster than the hand.” That very human oriented approach to spelling correction is foundational to this whole thesis and its methods.

Since the absolute cost of calculating the distance itself is less relevant to this work, and natural language is the thesis subject, Damerau’s work has been more heavily relied on. DLD is also the distance metric in use by Spotify in the current implementation of search query spelling correction, so Damerau’s theories are more directly applicable than Levenshtein's to the problem at hand.

Damerau performed several different tests on garbled text, he found that DLD correctly identified words in garbled text 95% of the time.

**Keyboard Layouts**

This thesis centers on the concept of edit distance weighted by the relative distance of the keys on the keyboard. To do that, one needs to know where on the keyboard each letter resides, which is not as trivial as it first seems to figure out.

Most countries have standardized on a keyboard layout, and most of them have selected QWERTY. However, there are differences, even within Europe, with e.g., France using AZERTY and Germany QWERTZ. Moreover, each individual user can still choose their layout freely – an American in Germany will probably do their English typing with a standard international QWERTY layout rather than the QWERTZ layout preferred by the locals. Or perhaps they belong to the small but dedicated group of keyboard layout enthusiasts that type using the Dvorak or Colemak layouts for the alleged increase in typing speed.

On the larger international scale, not all languages are alphabetic, and not all input even in alphabetic languages is done by what one would traditionally call typing, as both swipe and voice input have become increasingly popular on smartphones. The shortcomings of these methods are very different from those of typing, especially concerning a third party like Spotify receiving the query string. Automated speech recognition (ASR) will generally produce strings consisting solely of dictionary words, and without the original voice recording, judging intent from result is impossibly difficult compared to typing for the third party performing the search with the query. But all of these are outside the scope of this work.

Additionally, the search service that this thesis uses for its tests has no realistic chance to positively determine the user’s keyboard layout. However, as many countries have a standard keyboard layout that is very dominant we limit the scope of this thesis to only include the characters a-z in the weightings, and the weightings were only applied to searches originating in QWERTY-dominated countries.¹

¹ Based on Wikipedia, branah.com, goodtyping.com, starr.net/is/type/keyboard-charts.html, and terena.org/activities/multiling/ml-mua/test/kbd-all.html the dominant keyboard layout was collected for countries that together hold users that make up 99.36% of Spotify’s userbase as of November 2016. A table containing this information can be found in appendix 1. Over 85% of the users were found to be residing in QWERTY-dominated countries.
Current System in use at Spotify

The current implementation is based on DLD, and allows a certain number of misspellings for the whole search string, depending on its length, and requires them to be relatively evenly distributed between the words in the string.

The implementation is a trie that is searched very similarly to A* search. Each node in the trie is a letter, and a leaf means that the query is completely matched. Without allowing misspellings, the trie is just a straight line with no branches coming off of it, but once they are allowed, it will look more like figure 1:

![Figure 1: A simplified representation of the trie created when searching for dog. Blue nodes are possible matched queries. When any letter can be inserted or replaced in, this is exemplified by inserting/ replacing with both a and z. Note: since each node is a letter, including the root, and the root cannot be modified, the first character in a search query cannot be corrected.](image)

A three letter query is the smallest query that Spotify currently allows spelling correction on, and a single misspelling is allowed. As can be seen in Figure 1, even allowing a single misspelling in such a short query dramatically increases the scope of the search. The example in Figure 1 increases the number of possible matches from 1 to 58, and that is a literal exemplification of the smallest possible number of corrections on the shortest word allowed to be corrected.

The trie is searched using a technique called prefix search, meaning for example that typing in *fleetu* will give hits related to Fleetwood Mac. This expansion does not count against the system’s spelling correction counts.

Related Work

Google

There are a number of spelling correction systems in widespread use today. The state of the art is arguably the Did you mean?-feature of Google’s eponymous search engine. However, the details of its implementation are sparse and contradicting. One article by Peter Norvig, Google’s Director of Research and previous Director of Search Quality, describes it as a probabilistic model based on word frequencies in word lists and edit distance [5]. He also shows a 36 line working python example, and refers to a Google paper
discussing how Google uses massive amounts of crawled web data for these word lists instead of hand annotated data [6]. The other main source information is from a lecture in Prague, where Google’s then VP of Engineering and CIO Douglas Merrill claims that their spelling correction is done via a “statistical machine learning” approach and “without having any notions of morphology, without any generative grammars” [7]. Instead, he explains, it is based on patterns like this:

1. A user searches for physician, looking for things related to physician, finding none, and so doesn’t click any links.
2. Not having found what they are looking for, the user corrects their spelling to physician and searches again.

At the scale Google operates at, Merrill claims, this is enough to teach the search engine any and all common misspellings.

Both of these sources are originally from 2007 (Norvig’s article originally written in February and Merrill’s lecture from October), and from an outside perspective it is very hard to say exactly how these two approaches are combined, or if one has at this point superseded the other (looking at the industry in general, if that is the case the winner would most likely be the machine learning approach). It is problematic that this prime example is opaque and proprietary, but such are the facts, and Google cannot be ignored because of that. However, what can be observed is that neither of these sources mention anything other than standard edit distance, and so we can take from these sources that weighted edit distance seems not to be a core part of how Google does spelling correction.

Others
There have been many attempts through the years at improving spelling correction, and they generally fall into two camps. Firstly, statistical models based on ideas similar to those discussed at Google mentioned in the previous section (which still most commonly rely on Levenshtein distance at their core), and secondly, modifications to the edit distance measurement itself. Some of the seemingly more productive approaches have done so by using physical characteristics and constraints of the real world part of the software, like in one case where a statistical weighting of edit distances helped identify license plates more accurately [8].

The most closely related example is a paper titled simply “Learning String Edit Distance” by two Princeton researchers [9]. First, they created a novel stochastic model to determine the edit distance between two strings, and then they used the model on a dataset of word pronunciations to learn the spelling of a word given its pronunciation and vice versa. It is not directly applicable to the work presented in this thesis, but it is one of the clearest cut successful examples of the general approach of problem solving by consulting real world factors.

A final paper worthy of mention is a group of researchers from MITRE that used Levenshtein distance in conjunction with character equivalency classes to transliterate personal names between English and Arabic [10]. Personal names share similarity with the music in particular in that they are not in most dictionaries, and this problem in particular has an extra level of difficulty in that the correct answer is not only unknown, but undecided unless by consensus. The results were promising with a high degree of
success in determining if two names, one given in Arabic and one in English, are actually the same. The authors used Levenshtein Distance to test for matches and Perl regular expressions to perform the actual transliteration. This paper more than the others show the potential of a relatively simple approach like keyboard distance weighting.

Outside of these papers previously mentioned, there are countless others. Edit distance is used for a wide variety of tasks, not only spelling correction, and spelling correction itself is a huge problem that has existed longer than many other areas of computer science, and remains very relevant today.

---

2 Google Scholar approximates that there are 438,000 papers that mention edit distance.
Method

The hypothesis is that a weighted edit distance model, with lower costs for keys that are closer together, will more often retrieve the desired search result when querying a search engine that allows for spelling correction. Specifically, this thesis deals with search queries in a database of different music entities. The hypothesis is based on the idea that one is more likely to accidentally press a key adjacent to the one intended rather than one far away from it. Additionally, it assumes that mechanical errors (typos) are common enough that the model benefits overall from capturing them more accurately at the expense of errors of ignorance, that is cases when the user does not know the correct spelling of the entity they are searching for. Those types of errors are not differentiated by the method even if they are present in the data.

The Algorithm

To test the hypothesis, some of the operations of Damerau-Levenshtein were weighted by a distance measurement for every pair of keys. To make the distance measurement the QWERTY-layout was first turned into a graph where the nodes are keys and there are edges between any two keys that are adjacent on the keyboard (adjacent meaning that the keys would touch if there was no spacing between the keys on the keyboard – making s and e adjacent but not t and h) as seen in figure 2. After doing this, the distance between two keys was defined as the length of the shortest path between them in the graph, and the distance from any key to itself being the same as the distance to its neighbors.

![Figure 2: The adjacency map of a QWERTY keyboard.](image)

While that was a fair distance measurement, that was not quite enough to make it a good weighting. A good weighting for edit distance needs to still have the expected value of 1 when choosing a random pair of keys and a random operation, if it is to be at all comparable to standard DLD. Otherwise, the weighted edit distance could have simply yielded a shorter distance than the unweighted baseline because the average operation was cheaper even in a random testing set, like if all the weights were set to 0. That is not the hypothesis, the hypothesis is that the set is not randomly distributed, and therefore a weighting will improve the results even if it gives the same distance on random data.

To test this, all the possible operations needed to be considered. An initial distance matrix was constructed, and based on this, two different variants were created: one linear and one with a steep falloff focused on the direct neighbors.
The linear variant was constructed to have evenly distributed values from 0 to 2. On the keyboard, the shortest distance between any two keys is 1 (or 0, if you include the distance between a given key and itself), and the longest is 9, so the distance in number of keys was multiplied by 2/9 to get the desired distribution.

The average distance in this matrix was calculated, and each value was then divided by this average. This normalization produces a matrix of weights where the expected value is 1 when choosing a cell at random.

The other variant is weighted so that the neighbors are vastly preferred to the other keys, with there being no difference between replacing p or j for z, since neither neighbors it. This weighting will henceforth be referred to as neighbor xy, where x and y are the weights applied (before normalization) to neighbors and non-neighbors, respectively.

In an attempt to remove as much bias as possible from the testing, the inverse of every weighting has also been tested. In finality, this ended up meaning that every test was run unweighted, as well as with neighbor 1:2, neighbor 2:1, neighbor 0:1, neighbor 1:0, linear, and inverted linear weights.

Deletion
Originally, deletion was intended to be weighted according to whichever distance on the keyboard is shorter between the character to be deleted and the adjacent characters in the string. For example, removing the s from asmmunition to make ammunition would be weighted by the distance between a and s.

However, doing that gives a benefit to any weighted model, because of the choosing of the shorter distance. Instead, deletion is weighted by the average of the distances to the adjacent characters in the string.

Insertion
Unchanged, weight 1. Missing a key is presumed not to be affected by their placement. It seems more like to result from mental error, rather than mechanical, because otherwise something would be pressed.
Substitution
Weighted according to the distance between the character that is removed and the character that is inserted. For example, swapping the s in butter for an e to make butter would be weighted by the distance between e and s.

Transposition
Unchanged, weight 1. Pressing keys in the wrong order is presumed not to be affected by their placement. It would seem intuitive that transposition is influenced by whether or not the keys are being pressed by the same hand or not, but modelling that is beyond the scope of this thesis.

The Dataset
The dataset that all the tests are performed on consists of 1,162,145 searches logged from real Spotify users. The data is structured in a CSV file with each line like so:

<table>
<thead>
<tr>
<th>Query</th>
<th>Country</th>
<th>Spotify URI of clicked item</th>
<th>No. clicks</th>
<th>Position</th>
<th>Item name</th>
</tr>
</thead>
<tbody>
<tr>
<td>the wee</td>
<td>CA</td>
<td>spotify:artist:1Xyo4o8uXC1ZmMpsF05PJ</td>
<td>5</td>
<td>0</td>
<td>The Weeknd</td>
</tr>
</tbody>
</table>

Table 2: Sample from dataset.

Testing has been performed on both this set and a subset of 992,255 searches originating in countries previously determined to be dominated by the QWERTY layout.

The dataset contains several attributes for each search, but the only ones used in this project are the query string, originating country, and the name of the clicked entity.

The Experiment

Statistics
Without running the actual search engine, a number of tests were performed.

For each search, the unweighted DL-distance was computed between the query and the clicked item name, and also which operations were part of that shortest path. If there were multiple shortest paths, each path was saved. Then, the operations that were not considered in the weighting were filtered out. Then, for every performed operation, the distance on the physical keyboard was calculated. This was also done for a further subset of just the substitutions. The data from these tests was then binned so that the count for each bin was the number of times an edit between keys of that distance occurred. Binning like this can then be used to create a histogram comparing the distances of the edits of the dataset to the occurrences of the same distances on the keyboard. This yielded figures 3-5.

Additionally, for every item, all the different types of weighted DL-distance were computed between the query and the clicked item name. This data was binned so that the count for each integer bin was the number of query-result pairs that were that distance apart when rounded down to a whole number.
All analysis and statistics collection was done in Python 2.7, and then plotted by use of its matplotlib. The source code can be found in appendix 2.

Simulation
Spotify has previously used the dataset of old searches for testing any changes to the search engine, and there already exists a test suite for these purposes. The average edit distance with basic Damerau-Levenshtein over the entire set tells us that the average query is 8.26 edits from its desired result. Those edits were split among the different operations like so: deletions 0.12%, insertions 75.85%, substitutions 23.99%, and transpositions 0.03%. The large proportion of insertions may seem strange until one considers that the search engine does prefix search, as previously mentioned.

The pre-built test against this dataset yields a number of statistics, one of which in particular is useful for the purposes of this thesis: percent clicked item not in result set (PCINIRS). Meaning that if $n$ searches were simulated, and algorithm $x$ presented the item clicked by the user when the search was originally performed $c$ times, then $PCINIRS(x) = \frac{100 \times (1 - \frac{n}{c})}{100}$. If the hypothesis is correct, this statistic is expected to lower when testing with the weights. The definition of an item being in the result set is that it occurs among the first 15 items. This is the primary measurement used to evaluate.

There are also query times (lower times are obviously better), and a discounted cumulative gain (DCG) score that evaluates not only if the item was present, but how it was ranked (higher score is better). The DCG score was expected to stay more or less the same as the ranking is not touched, but is of course affected by whether or not the correct result was included whatsoever. The query times were expected to increase slightly, given a) that it is more expensive to compute with weights than without and b) that one idea behind the weighting is to allow more branches of higher quality rather than few branches of lower quality, but searching more branches takes more time.

DCG and search time were used as secondary measurements, observed but not directly related to the hypothesis, and as such not of particular interest unless changing dramatically. Specifically, DCG was regarded as stable if staying within 1% of the baseline, and the search times given little weight unless there is a relative increase greater than 100%.

Simulation testing was originally intended to be performed with no changes other than weights, but early on this proved to be impractical, as primarily one factor in the Spotify search engine code base severely restricts the possibility of making any sort of meaningful impact: the disallowing of adding a misspelled result if its ranking is lower than a correctly spelled result (misspelled items are downranked by a scaling factor).

A simple example is when searching for rhiann, for which Rihanna will not appear in the search results, because of the artists Rhiannon Giddens and Rhiannon Bannenberg. Extending the search query to rihanna yields a result list including Rihanna and excluding the Rhiannons. This setting was turned off for all algorithms during the testing, to achieve clearer differences between the different approaches.
Results

The manner in which this data was collected is described in detail in the method. The results as follows is all of the data collected during the testing phase, and are presented as clearly and unfiltered as possible, and with little subjective comment, which is left for the discussion.

Statistical Analysis

<table>
<thead>
<tr>
<th>Method</th>
<th>Average edit distance from query to result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>8.26</td>
</tr>
<tr>
<td>Neighbor 0:1</td>
<td>8.07</td>
</tr>
<tr>
<td>Neighbor 1:0</td>
<td>7.64</td>
</tr>
<tr>
<td>Neighbor 1:2</td>
<td>8.20</td>
</tr>
<tr>
<td>Neighbor 2:1</td>
<td>8.22</td>
</tr>
<tr>
<td>Linear</td>
<td>8.11</td>
</tr>
<tr>
<td>Inverted linear</td>
<td>8.15</td>
</tr>
</tbody>
</table>

*Table 3: Average weighted edit distance from query to result. The smaller the average edit distance from query to result, the greater the likelihood is that the actual correct result would be among the results. However, the desired result is not just a short distance from query to result, but that the distance is relatively shorter than the distance to other results. The difference of 0.5 between 1.0 and Inverted linear means that if it were regular distance, 1.0 would need one edit fewer every other search to correct the string a to b.*

Here, the importance of testing against other weightings is immediately revealed. Every single weighted approach achieved a lower average weighted edit distance compared to the unweighted. The neighbor 1:0 weighting appears to vastly outperform all of the others, but linear and neighbor 1:2 both outperform their respective inverses, if only slightly in the latter case.
Figures 3-5 are perhaps the most important part of this work. It shows, for all the edits performed when comparing queries to clicked entities, the comparative frequency of the keyboard distances of the involved characters. That data is plotted against the occurrence of the different distances for every pair of keys on the keyboard.

The 0-1 span is difficult to compare, because it is not entirely obvious whether or not, for the keyboard, the distance from each key to itself should be included. Figure 3 includes them, see figure 4 for the plot without them.

Figure 3: Histogram of Distance Occurrence in Dataset vs. on Keyboard.
The dotted line shows how common the different distances are on the keyboard (all keys have neighbors, but very few have a distance of 9 to another key). The dashed line is the same distance for every substitution key-pair for every optimal edit distance path between query and entity in the dataset.

Figure 4: Histogram of distance occurrence in dataset vs. on keyboard.
The keyboard distance data seems like it might be a Poisson distribution, and apart from a strange dip at distance 4, so does the dataset, albeit one populated slightly farther to the left and with a smaller deviation from the mean. While this is interesting to note, it does not affect the study of the subject, and so all the errors and confidence intervals have been constructed from multinomial formulae, as that requires no assumptions other than that we are dealing with integer bins with probabilities, which is known to be true.

![Histogram of distance occurrence for substitutions in dataset vs. on keyboard.](image)

*Figure 5: Histogram of distance occurrence for substitutions in dataset vs. on keyboard.*

Here in *figure 5*, we have further narrowed down by plotting just the substitutions against the keyboard, which removes the problem of the inclusion of the distance from *x* to itself in another way, as substitutions from one character to itself are never performed. *Figure 7* also shows the standard deviation, expected values, and coefficient of variation for each of the two distributions plotted in *figure 5*.

<table>
<thead>
<tr>
<th>Set</th>
<th>Expected value</th>
<th>Standard deviation</th>
<th>Coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>Substitutions</td>
<td>3.564</td>
<td>2.060</td>
<td>57.79%</td>
</tr>
<tr>
<td>Keyboard</td>
<td>3.551</td>
<td>2.046</td>
<td>57.62%</td>
</tr>
</tbody>
</table>

*Table 4: Histogram of distance occurrence for substitutions in dataset vs. on keyboard.*

The coefficient of variation measures spread and is the standard deviation divided by the expected value.
Simulations

<table>
<thead>
<tr>
<th>Method</th>
<th>PCINIRS</th>
<th>DCG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>48.73</td>
<td>0.373</td>
</tr>
<tr>
<td>Neighbor 0:1</td>
<td>48.78</td>
<td>0.373</td>
</tr>
<tr>
<td>Neighbor 1:0</td>
<td>48.96</td>
<td>0.371</td>
</tr>
<tr>
<td>Neighbor 1:2</td>
<td>48.82</td>
<td>0.373</td>
</tr>
<tr>
<td>Neighbor 2:1</td>
<td>49.09</td>
<td>0.370</td>
</tr>
<tr>
<td>Linear</td>
<td>48.73</td>
<td>0.373</td>
</tr>
<tr>
<td>Inverted linear</td>
<td>49.34</td>
<td>0.369</td>
</tr>
</tbody>
</table>

Table 5: PCINIRS scores for all implementations.

Lower PCINIRS is better and means that the correct entity was more frequently included in the results.

The original baseline for PCINIRS is 49.26%, but that was without the previously mentioned change of the Misspellings after correct®-setting. After turning that off, the new recorded baseline was 48.73%. The only method to beat this score was the linear weighting, if only ever so slightly, by 0.004%. Neighbors 0:1 and 1:2 both beat their inverted counterparts by 0.17% and 0.27% respectively, and so does linear, with an even clearer 0.60%.

The DCG scores barely move, as expected, but it is worth noting that here linear is joined by neighbor 0:1 in its slight outperforming of the baseline measurement.

Figure 6: PCINIRS from table 5 plotted with 95% confidence intervals as vertical lines.

Since PCINIRS is a binary statistic, the standard deviation and by extension confidence intervals can be calculated just from knowing the percentage score and the number of samples. Figure 4 contains this data, which helps illustrate a couple of things.

As can be seen in figure 4, the differences between unweighted, 0:1, 1:2, and linear are all within the error margin for $p < 0.05$, while 1:0, 2:1 and inverted linear are all significantly
worse than the baseline. The differences between each approach and its inverse are also significant for every pair.

![Histogram of searching times for all implementations. Log scaled since the vast majority of queries are still performed in less than 5 milliseconds. Each group of markers on the y-axis represents an increase in magnitude.](image)

In figure 7 we see at least one clear outlier. While the query times all have differing spreads, neighbor 1:0 performs noticeably worse than the others, taking more than 100 milliseconds more than 12 times as often as any of the other weightings, and more than 50 times as often as unweighted, and only coming in below 5 milliseconds 84% as often. Most of the others experience some slowdown compared to the baseline, but the worst offenders after 1:0 still came in at 92% as often below 5 milliseconds (linear) and just over 4 times as often over 100 milliseconds (inverted linear) compared to unweighted.

<table>
<thead>
<tr>
<th>Method</th>
<th>Estimated average searching time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>2.46</td>
</tr>
<tr>
<td>Neighbor 0:1</td>
<td>4.18</td>
</tr>
<tr>
<td>Neighbor 1:0</td>
<td>19.95</td>
</tr>
<tr>
<td>Neighbor 1:2</td>
<td>2.60</td>
</tr>
<tr>
<td>Neighbor 2:1</td>
<td>3.29</td>
</tr>
<tr>
<td>Linear</td>
<td>4.11</td>
</tr>
<tr>
<td>Inverted linear</td>
<td>3.45</td>
</tr>
</tbody>
</table>

*Table 5: Estimated average searching times for all implementations.*

Exact times are not known, due to the binned nature of the data gathered by the test suite, but these averages calculated based on the bins illustrate the point.
Discussion

There are two parts to making a good distance measurement for spelling correction in search. The first is minimizing the distance between the query string and the name of the desired entity. Second, and far more important, maximizing the distance between the query string and undesired entities. Setting the distance between any two strings to 0 fulfills the first requirement, but disastrously fails the second. Setting all distances to infinity does the opposite. A middle ground needs to be reached.

This is why, even though neighbor 1.0 looks like the upset winner at first glance in table 3, it can not in fact be said to be the best measurement when weighing in the simulation data from figure 6 and even more importantly, the time constraint issues that are painfully revealed in figure 7 and table 5. Figure 4 also makes it look unlikely that neighbor 1.0 should be an improvement over unweighted at all, since the dataset has a higher ratio of distance 1 to longer distances.

Conclusions

While it is difficult to come to any immediate conclusion based on the whole of the information, different pieces of data help tell a story. Figure 5 and table 4 clearly show that neighboring mistakes are overrepresented in the dataset compared to a random distribution. Table 5 and figure 4 show that while the near-weighted approaches do not outperform the baseline, they perform significantly better than their counterparts based on the opposite principle. When all of this information is considered together, it clearly points toward a correlation between the physical position of keys on the keyboard, and what spelling mistakes are made.

So, if that is the case, why do none of the weighted approaches make improvements compared to the baseline?

It may be simple issues with the method’s relation to the hypothesis. For example, to quote from its formulation in the method, the hypothesis “assumes that mechanical errors (typos) are common enough that the model benefits overall from capturing them more accurately at the expense of errors of ignorance”. If that isn’t true, the hypothesis would not be validated by an experiment like this one, even if its core concept of the correlation between key position and typos is accurate.

Another possible explanation is of course that the core of the hypothesis is incorrect, but then one would have to explain why the opposite weightings are inversely correlated. Attributing it to chance will not do when the tests were performed with multiple approaches of weighting, and a sample size of more than a million string pairs.

Potential Applications and Interest

This study has aimed to examine whether or not the physical locations of the keys of the keyboard have an effect of the typing mistakes that people make, and the potential applications are widespread, even outside the immediate field of spelling correction, although that is where the clearest implications are found. Search is of material concern with the ever growing stores of data in the modern era, and more than ever, that data is being produced by hobbyists. A knowledge of their tendency to produce typos can help
search not only from the end user perspective of helping incorrectly spelled queries find correctly spelled results, but also the opposite: helping a correctly spelled query find information related to the subject at hand from a source that has misspelled the keywords. Other potential applications include in the design of keyboards, especially virtual ones. A touchscreen press on the border between two keys could be interpreted correctly by a statistical model that compensates for that particular user’s likelihood of missing that particular key based on their typing history.

The most direct conceivable application is in text transcription, like Project Gutenberg’s attempt to digitize literature that is in the public domain. In transcription, mechanical typos ought to be the only source of spelling errors, and in whatever software they use for their work, the weighted Damerau-Levenshtein distance presented here should in theory perform better than the standard approach.

**Ethics and Sustainability**

The ethics of a work such as this may seem simple at first glance, but upon further thought become more complex and convoluted. This is mainly due to two concerns, the dataset and the approach.

The dataset contains searches from Spotify users that have consented to this type of data being collected, but that does not immediately mean that collection and study of said data is ethical. In more than one million searches, it seems likely that some of them will contain some amount of personal information, for example usernames of individual users and the names of their playlists, but potentially much more intimate information. The dataset is anonymized in that the username of the user performing the search is not stored together with each search, but that does not mean that they can not be identified. Spotify’s search is personalized, meaning that two users will not get the exact same ranking for the same set of tracks, which taken together with the user’s country, the query and the clicked entity can at the very least heavily narrow down the number of candidates.

The balance between privacy and ease of use is heavily debated in the tech field, and although this project has not gathered any additional data and the author has taken precautions to minimize manual interaction with the data, the fact remains that if a computer can read the data, so can a human.

The other aspect is that the approach is by its very definition excluding, in that it only focuses on alphabetic languages typed on a QWERTY keyboard. In the background, the technical reasons for this approach were motivated, but the ethical aspects were not. Alphabetic QWERTY users are a vast majority of users, so clearly focusing on them will be the most effective when trying to improve overall metrics, but is it ethically justifiable? It is not difficult to argue that ease of access work should be focused on the users with the lowest access, not further simplifying the lives of the users who are already being catered to.

The sustainability aspect is more straightforward. This project attempts to optimize previously existing code paths within their predefined boundaries, and as such will not affect the overall energy impact of Spotify or even this project. There is a larger discussion to be had as to whether or not such maintaining of the status quo is sufficient, or if every single project has an obligation to actively attempt to improve efficiency and cut energy usage, but while that is interesting it is too massive to be within the scope of this work.
**Issues and Limitations**

This study is very clearly limited in its scope, which was important in its design. Then, the constraining factor of Spotify's already extant search engine was applied.

In the end, this means two things. First, this thesis does take steps forward in validating the idea of the correlation between keyboard distance and misspellings (particularly figure 4). It should be viewed as a look at how that correlation is best exploited in a real world scenario where it can be at least partially presumed. Second, the work presented here must not be applied too broadly. The set of constraints is very specific, including (but not limited to) music entities being a special subset in that it is constrained, yet at the same time multilingual without labels or rules, and Spotify's particular pre-written code handling misspelled entries, their inclusion, and their ranking.

Finally, the QWERTY keyboard design is not random. For example, s, c and z are all close together, and phonetically similar. Separating all of these sources of error would go a long way in detecting what specifically caused the data seen here.

At least one clear avenue of improvement that shows promise is using machine learning to optimize the weights. This would also enable the weighting of those atomic operations where a keyboard typo model is not applicable, like transposition and insertion. Machine learning also has the advantage of allowing more easily to have different models for different people, and try different groupings, like having one set of weights for all users from the same country, or one set for all users of mobile clients. This machine learning algorithm could be given the keyboard distances as features, and the viability of the keyboard distance model can then be assessed by looking at what importance they are given by the algorithm.

Another approach that could yield interesting results is looking at phonemes rather than just letters. By doing that, one could approach the problem from a more linguistic perspective, and assign phoneme weights by the similarity of the sounds they produce, being able to swap $ph$ for $f$. This would come at some computational expense. Partially because the number of substrings in a string of length $n$ is the $n$-th triangular number, and so many more comparisons will be needed. Additionally, some sort of initial linguistic parsing would be needed since the relationship between phonemes and graphemes (letters) is not simple: $a$ sometimes sounds like $k$ as in cow, and sometimes like $s$ as in receive. However, the current approach is relatively light in operation, so this does not seem to necessarily be a roadblock.
Acknowledgements

This thesis subject was in great part inspired by Johan Boye's course DD2418 Språkteknologi and its random keyboard laboratory exercise.
References

Appendix 1: keyboard layout by country

<table>
<thead>
<tr>
<th>Country</th>
<th>ISO 3166-1 alpha-2 code</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argentina</td>
<td>AR</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Austria</td>
<td>AT</td>
<td>QWERTZ</td>
</tr>
<tr>
<td>Australia</td>
<td>AU</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Belgium</td>
<td>BE</td>
<td>AZERTY</td>
</tr>
<tr>
<td>Bolivia</td>
<td>BO</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Brazil</td>
<td>BR</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Canada</td>
<td>CA</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Switzerland</td>
<td>CH</td>
<td>QWERTZ</td>
</tr>
<tr>
<td>Chile</td>
<td>CL</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Colombia</td>
<td>CO</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Costa Rica</td>
<td>CR</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Czechia</td>
<td>CZ</td>
<td>QWERTZ</td>
</tr>
<tr>
<td>Germany</td>
<td>DE</td>
<td>QWERTZ</td>
</tr>
<tr>
<td>Denmark</td>
<td>DK</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Dominican Republic</td>
<td>DO</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Ecuador</td>
<td>EC</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Spain</td>
<td>ES</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Finland</td>
<td>FI</td>
<td>QWERTY</td>
</tr>
<tr>
<td>France</td>
<td>FR</td>
<td>AZERTY</td>
</tr>
<tr>
<td>Great Britain</td>
<td>GB</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Greece</td>
<td>GR</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Guatemala</td>
<td>GT</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Hong Kong</td>
<td>HK</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Honduras</td>
<td>HN</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Hungary</td>
<td>HU</td>
<td>QWERTZ</td>
</tr>
<tr>
<td>Indonesia</td>
<td>ID</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Ireland</td>
<td>IE</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Italy</td>
<td>IT</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Mexico</td>
<td>MX</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Country</td>
<td>Code</td>
<td>Keyboard Layout</td>
</tr>
<tr>
<td>------------------------</td>
<td>------</td>
<td>----------------</td>
</tr>
<tr>
<td>Malaysia</td>
<td>MY</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Netherlands</td>
<td>NL</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Norway</td>
<td>NO</td>
<td>QWERTY</td>
</tr>
<tr>
<td>New Zealand</td>
<td>NZ</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Panama</td>
<td>PA</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Peru</td>
<td>PE</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Philippines</td>
<td>PH</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Poland</td>
<td>PL</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Portugal</td>
<td>PT</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Paraguay</td>
<td>PY</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Sweden</td>
<td>SE</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Singapore</td>
<td>SG</td>
<td>QWERTY</td>
</tr>
<tr>
<td>El Salvador</td>
<td>SV</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Turkey</td>
<td>TR</td>
<td>Other</td>
</tr>
<tr>
<td>Taiwan</td>
<td>TW</td>
<td>QWERTY</td>
</tr>
<tr>
<td>United States of America</td>
<td>US</td>
<td>QWERTY</td>
</tr>
<tr>
<td>Uruguay</td>
<td>UY</td>
<td>QWERTY</td>
</tr>
</tbody>
</table>
#!/usr/bin/python

import sys, math, random, copy

neighbors_of = {}

#  nw ne e se sw w
neighbors_of['q'] = ['w', 'a']
neighbors_of['w'] = ['e', 's', 'a', 'q']
neighbors_of['e'] = ['r', 'd', 's', 'w']
neighbors_of['r'] = ['t', 'f', 'd', 'e']
neighbors_of['t'] = ['y', 'g', 'f', 'r']
neighbors_of['y'] = ['u', 'h', 'g', 't']
neighbors_of['u'] = ['i', 'j', 'h', 'y']
neighbors_of['i'] = ['o', 'k', 'j', 'u']
neighbors_of['o'] = ['p', 'l', 'k', 'i']
neighbors_of['p'] = ['l', 'o']

neighbors_of['a'] = ['q', 'w', 's', 'z']
neighbors_of['s'] = ['w', 'e', 'd', 'x', 'z', 'a']
neighbors_of['d'] = ['e', 'r', 'f', 'c', 'x', 's']
neighbors_of['f'] = ['r', 't', 'g', 'v', 'c', 'd']
neighbors_of['g'] = ['t', 'y', 'h', 'b', 'v', 'f']
neighbors_of['h'] = ['y', 'u', 'j', 'n', 'b', 'g']
neighbors_of['j'] = ['u', 'i', 'k', 'm', 'n', 'h']
neighbors_of['k'] = ['i', 'o', 'l', 'm', 'j']
neighbors_of['l'] = ['o', 'p', 'i', 'k']

neighbors_of['z'] = ['a', 's', 'x']
neighbors_of['x'] = ['s', 'd', 'c', 'z']
neighbors_of['c'] = ['d', 'f', 'v', 'x']
neighbors_of['v'] = ['f', 'g', 'b', 'c']
neighbors_of['b'] = ['g', 'h', 'n', 'v']
neighbors_of['n'] = ['h', 'j', 'm', 'b']
neighbors_of['m'] = ['j', 'k', 'n']

keys = sorted(neighbors_of.keys())
dists = {el:{ } for el in keys}

def distance(start, end, raw):
    if start == end:
        if raw:
            return 0
        else:
            return 1

visited = [start]
queue = []

for key in neighbors_of[start]:
    queue.append({'char': key, 'dist': 1})

while True:
    key = queue.pop(0)
    visited.append(key['char'])
    if key['char'] == end:
        return key['dist']
for neighbor in neighbors_of[key['char']]::
    if neighbor not in visited:
        queue.append({'char': neighbor, 'dist':
key['dist']+1})

def alldists(type, verbose):
    if type == "linear":
        longest_dist = 0
        avgdist = 0
        for i in range(len(keys)):
            for j in range(len(keys)):
                dist = distance(keys[i], keys[j], False)
                dists[keys[i]][keys[j]] = 2 - (2 * dist / 9.0)
                avgdist += dists[keys[i]][keys[j]]
                if dist > longest_dist:
                    longest_dist = dist
        key_dist = longest_dist
        avgdist /= len(keys) ** 2 + 0.0
        if verbose:
            print "Average distance: " + str(avgdist)

        avgdisttwo = 0

        for i in range(len(keys)):
            for j in range(len(keys)):
                dists[keys[i]][keys[j]] /= avgdist
                avgdisttwo += dists[keys[i]][keys[j]]
        avgdisttwo /= len(keys) ** 2 + 0.0
        if verbose:
            print "Average distance after normalizing: " +
str(avgdisttwo)
            print "Longest distance: " + str(key_dist)
            print "Longest logarithmed: " +
str(math.log(key_dist))
            print "Logarithmed & normalized: " +
str(math.log(key_dist) / math.log(9))
            print str(dists).replace("", ",")
    elif type == "neighbors":
        longest_dist = 0
        avgdist = 0
        for i in range(len(keys)):
            for j in range(len(keys)):
                dist = distance(keys[i], keys[j], False)
                if dist == 1:
                    dists[keys[i]][keys[j]] = 2.0
                else:
                    dists[keys[i]][keys[j]] = 1.0
                avgdist += dists[keys[i]][keys[j]]
                if dist > longest_dist:
                    longest_dist = dist
        key_dist = longest_dist
        avgdist /= len(keys) ** 2 + 0.0
        if verbose:
            print "Average distance: " + str(avgdist)

        avgdisttwo = 0

        for i in range(len(keys)):
            for j in range(len(keys)):
                dists[keys[i]][keys[j]] /= avgdist
avgdisttwo += dists[keys[i]][keys[j]]

avgdisttwo /= len(keys) ** 2 + 0.0
if verbose:
    print "Average distance after normalizing: " + str(avgdisttwo)
    print "Longest distance: " + str(key_dist)
    print "Longest logarithmed: " + str(math.log(key_dist))
    print "Logarithmed & normalized: " + str(math.log(key_dist) / math.log(0))
    print str(dists).replace('"', '')
    dists[keys[i]][keys[j]] = distance(keys[i], keys[j], True)
    avgdist += dists[keys[i]][keys[j]]
    if dists[keys[i]][keys[j]] > longest_dist:
        longest_dist = dists[keys[i]][keys[j]]
        key_dist = longest_dist
        avgdist /= len(keys) ** 2 + 0.0

buckets = [0, 0, 0, 0, 0, 0, 0, 0, 0]

for i in range(len(keys)):
    for j in range(len(keys)):
        buckets[dists[keys[i]][keys[j]]] += 1

if verbose:
    print "Average distance: " + str(avgdist)
    print "Longest distance: " + str(key_dist)
    print "Buckets: " + str(buckets)
    print str(dists).replace('"', '')
return copy.deepcopy(dists)

def main():
    if len(sys.argv) == 2:
        alldists(sys.argv[1], True)
    else:
        key_dist = distance(sys.argv[1], sys.argv[2], True)
        print "Distance from " + sys.argv[1] + " to " + sys.argv[2] + ": " + str(key_dist)

if __name__ == "__main__":
    main()