Personalized Mobile Keyboard Touch Space Optimization

Problem Definition

Every user has a unique typing pattern that comes along a set of personal common typing errors. By analysing this pattern, a personalized keyboard is generated - decreasing the number of personal type errors. The approach is to optimize the touch space that each key should occupy to reduce typos.

Gathering Data

As the required data for this project are unavailable and cannot be obtained from the standard keyboard, a server and a mobile app for Android have been developed for data collection.

These work by first starting the server, which will send lines of text to the client. When the client receives these lines, one line of text is displayed at a time for the user to type. The client collects normalized x-coordinate touch inputs for that line of text, until the user presses the “next line” button. The collected x-coordinate inputs are then sent to the server, and the next line of text for the user to type is displayed in the app. When the server receives the normalized x-coordinates, each character of the typed line is paired with a coordinate and appended to a CSV file. This process repeats until there are no more lines for the user to type in the app.

The CSV file consists of two columns. The left figure below depicts an example for “hello” when the number of touch inputs matches the text length. If the number of touch inputs are below the text length, unmatched characters are paired with a question mark, as seen in the middle figure below. If there are too many touch inputs, question marks are matched with those coordinates as in the right figure below for the text: “he”.

	A	B
1	Message	hello
2	h	0.6
3	e	0.255556
4	l	0.909259
5	l	0.909259
6	o	0.840741

	A	B
1	Message	hello
2	h	0.6
3	e	0.255556
4	l	0.909259
5	l	?
6	o	?

	A	B
1	Message	he
2	h	0.6
3	e	0.255556
4	?	0.909259
5	?	0.909259
6	?	0.840741

Normalization and Search Space Reduction

Due to each mobile device having different screen dimensions, the collected touch inputs on the mobile screen are normalized. Meaning that the touch coordinates lie in the interval [0.0;1.0], where 0.0 and 1.0 are the far left and far right of the screen respectively.

Search space wise, a two-dimensional keyboard search space would be more difficult to analyse. The problem is therefore simplified to only consider the x-coordinates, and divide the keyboard in three rows. It is furthermore assumed that the user types on the correct row. Further details on search space are mentioned in the subsequent sections.

Creating a Probabilistic Keyboard Model

To train and assess the generated solutions, it is necessary to have a large amount of input data. During the training phase, being able to calculate the fitness based on different data sets helps avoiding overfitting towards the training data set used. For the assessment part, it allows us to generalize the quality of the solution created.

For our project, the data gathering has to be done with the purpose specific application described above. This activity might be boring and time consuming for the users involved. To mitigate this problem, a probabilistic model (distribution) of a user's keyboard usage was developed . The objective was to be able to draw samples from this model representing the x-coordinate where the user could have typed a letter.

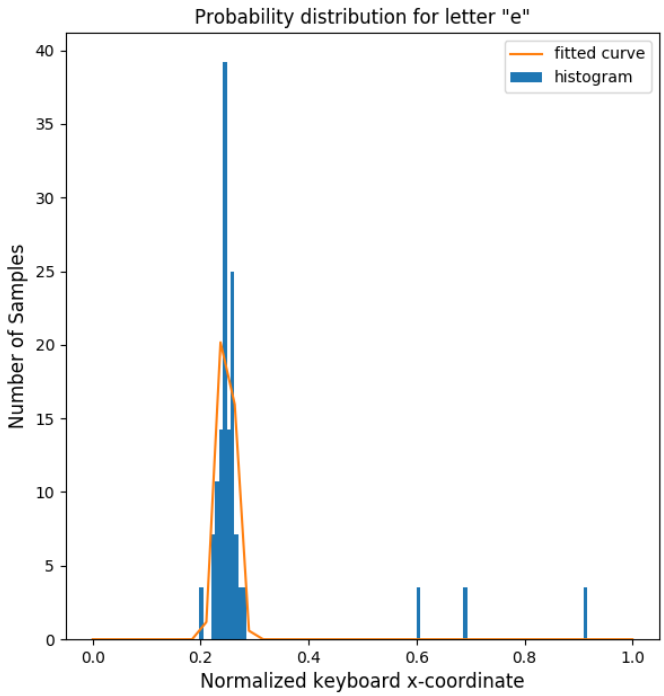
The first step to build the model is to gather samples from real users using our app. There is no real lower bound as to how much data is required, of course, the more the better. Also, there must be samples for each letter in the keyboard. We opted for approx. 1000 samples, which is roughly 40 samples per letter.

Once we have the samples, the next step is to group them by letter and build a histogram for each group of samples. The histogram tells us how often did the user type in a specific region when aiming for a specific letter. As it turned to be, when plotting the histograms, the bars resembled closely a gaussian distribution. Based on this observation a gaussian curve was fitted against the histogram to get the mean and standard deviation of the distribution for each letter.

The API was devised to be used as follows:

```
# build the model based on training data
user_model = usr_kbd_model.KBDModel("/path/to/training_data.csv")
# get one sample for the letter "e"
user_model.get_keystroke("e")
```

We can now get x-coordinates with a certain probability for a desired letter, as was our original goal.



Approach using Local Search

One of the most intuitive ways to optimize the keyboard is to use local search, which requires mainly three things: A representation of the individuals, the definition of neighborhood and a fitness function to evaluate individuals. Each individual represents a keyboard. The algorithm itself works as follows: For each keyboard, its neighbours are evaluated, and the fitness tells us how well each neighbour performs for each user. This process of finding the fittest neighbor and selecting it for the next generation will be repeated for a specified number of iterations.

First, we need the representation of an individual. We assume that the position of each key will not change, but the space between two adjacent keys will be divided into two and distributed among them. To achieve this goal, we represent each keyboard as a list of 26 weights, one weight per key, each between 0 and 10. For example, if key ‘a’

has weight 10 and key ‘s’ has weight 0, key ‘a’ will occupy all of the empty space between keys ‘a’ and ‘s’. Therefore, if a user touches the space between ‘a’ and ‘s’, it will be treated as if he typed ‘a’. Since our representation of the keyboard is meaningless if the empty spaces in between keys are too narrow, we made the space to be much larger than in a standard keyboard.

Second, we define the neighborhood of each individual. Given the representation of each individual, the list of 26 weights mentioned above, a reasonable neighborhood of an individual would be the set of keyboards where each key’s weight differs at most by one from that key’s weight in the original keyboard. For example, if an individual has weight 2 for key ‘a’, each neighbour will have a weight of 1, 2 or 3 for ‘a’. In general, we have three possibilities for the weight of each key for each neighbor, which implies a total of 326individuals in a neighborhood. Since the number of neighbors is too large to deal with, we tried to reduce the neighborhood size by just considering one layer at a time, a layer being one row in the keyboard, for example layer 1 is [q,w,e,r,t,y,u,i,o,p]. Since empty spaces will be split into two and distributed to adjacent keys, based on those keys’ weights, each layer of the keyboard is independent from the others. We can then deal with each layer separately and combine those optimized results to get the fittest neighbor. This way, the number of neighbors that we have to consider to optimize the keyboard in each iteration is reduced to 310+39+37, which is small enough to deal with, instead of the original 326.

Finally, the fitness function for evaluating each individual. Our data consists of pairs in the form (key, x-coordinate). In a normal setting, we could not know for sure which key the user intended to press, but with our application we know exactly which key had to be pressed and we can therefore determine whether a typo occurred or not. Given the typing data, we count the number of typos for each keyboard representation and that will be our fitness value, which has to be minimized. To check if a keystroke is a typo or not, we need to check if the given x-coordinate lies within the area of the key that is supposed to be pressed. Now, the only thing we need is a way to determine the space that each key occupies based on each individual’s weights. For each key, we calculate the left and right empty space that the key occupies. Since each key’s original space does not change, but only the portion of adjacent empty space that it occupies, we only need to calculate the left and right empty space distribution. Each space is linearly distributed based on the weight difference of two adjacent keys. For example the space will be divided into two equal parts if two adjacent keys’ weights are the same. If the weight difference is 10, then one key will take all of the empty space.

Approach using Genetic Algorithm

As an alternative approach, we used a genetic algorithm to optimize the keyboard. We then compared its accuracy and time complexity with the local search algorithm.

First, we simplified the representation of each individual so that the search space was reduced. Rather than thinking about the width of the key and the empty space between keys, the individual is represented as a list of floating point numbers between 0 and 1. The float numbers are the x-coordinate boundaries separating two keys.

Let's think of the representation of a normal keyboard. The keyboard has 3 layers, which we can consider as 3 independent individuals. The first layer is represented as a list of 9 float numbers between 0 and 1, each number representing a separation point, since the first layer has 10 keys (q, w, e, r, t, y, u, i, o, p). Each key has the same width in a normal keyboard, so [0.1, 0,2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9] is the representation of the first layer.

For the second and third layers of the keyboard, there are fewer keys than the first layer. This leaves a gap on the far left and right borders of those two layers, meaning that the starting x-coordinate for letter A and letter Z, as well as the ending x-coordinate for letter L and letter M, are displaced from the edges of the screen. Since the width of all keys are uniform in a keyboard, the width of the keys of layer 2 and layer 3 are also 0.1. Layer 2 can be represented as [0.15, 0.25, 0.35, 0.45, 0.55, 0.65, 0.75, 0.85]. The starting x-coordinate of the letter A is 0.05 and the ending x-coordinate of the letter L is 0.95. Although the empty space both at the left and right ends of the layer would normally not be considered to belong to any key, we give this leftover space to the first and last key respectively. In a similar way, layer 3 can be represented as [0.25, 0.35, 0.45, 0.55, 0.65, 0.75].

Second, crossover and mutation are used as genetic operators. The GA runs independently for each layer, there are therefore 3 populations for 1 keyboard. With an initial population of 40 for each layer, the GA first randomly selects n floating point numbers for each layer and creates a list of length n = 9 for the first layer, n = 8 for the second layer, and n = 6 for the third layer. We used single point crossover rate of 0.5 and Gaussian mutation rate of 0.2. We let the GA run for 100 generations.

Third, the fitness of each individual is evaluated as the number of correct inputs. From the data provided by the distribution, the GA runs toward maximizing the fitness value. In each generation we draw 5,000 samples from a previously generated keyboard distribution model. From the generated samples, the fitness of each individual is evaluated.

Evaluation

Local Search (# of typos in a 100,000 samples data set)

User	Original Keyboard	Local Search Optimized Keyboard
Sangjin	5.5%	2.3%

Final keyboard (letter / weight)

Q / 0	W / 0	E / 1	R / 1	T / 0	Y / 0	U / 1	I / 0	O / 4	P / 10
A / 0	S / 0	D / 0	F / 0	G / 3	H / 7	J / 0	K / 1	L / 10	
Z / 2	X / 5	C / 4	V / 8	B / 5	N / 10	M / 9			

GA (# of typos in a 100,000 samples data set)

User	Original keyboard	Genetic keyboard
Dio	4.38%	2.43%
Sangjin	1.82%	0.58%
Jeongmin	0.13%	0.015%
Kaz	46.15%	27.62%

Dio’s Keyboard

Q	0.09717	W	0.20136	E	0.30173	R	0.42028	T	0.48962	Y	0.48962	U	0.61203	I	0.71136	O	0.81535	P
	A	0.12490	S	0.25830	D	0.36208	F	0.45211	G	0.57474	H	0.66522	J	0.77641	K	0.86665	L	
			Z	0.24564	X	0.36830	C	0.44811	V	0.56243	B	0.63243	N	0.75435	M			

Sangjin’s Keyboard

Q	0.06 342	W	0.17 466	E	0.28 430	R	0.39 078	T	0.49 026	Y	0.58 465	U	0.69 176	I	0.79 895	O	0.88 039	P
	A	0.12 270	S	0.22 917	D	0.34 079	F	0.43 887	G	0.53 655	H	0.64 607	J	0.74 160	K	0.83 684	L	
			Z	0.22 805	X	0.34 563	C	0.43 584	V	0.56 350	B	0.64 170	N	0.76 854	M			

Jeongmin’s Keyboard

Q	0.07 243	W	0.19 897	E	0.32 314	R	0.39 115	T	0.51 477	Y	0.5 970	U	0.69 758	I	0.79 918	O	0.90 310	P
	A	0.15 296	S	0.24 192	D	0.34 972	F	0.44 549	G	0.54 392	H	0.64 535	J	0.77 641	K	0.84 836	L	
			Z	0.27 488	X	0.33 279	C	0.45 721	V	0.52 753	B	0.64 917	N	0.77 874	M			

Kaz’s Keyboard

Q	0.15 361	W	0.22 824	E	0.33 583	R	0.44 181	T	0.51 368	Y	0.59 395	U	0.67 440	I	0.76 861	O	0.82 356	P
	A	0.18 728	S	0.28 861	D	0.37 121	F	0.47 082	G	0.54 542	H	0.63 154	J	0.72 818	K	0.81 714	L	
			Z	0.33 574	X	0.43 096	C	0.47 410	V	0.53 362	B	0.92 845	N	0.99 30	M			

Results and Analysis

For the Local Search algorithm, we tried to minimize the search space, so that the local search algorithm can be applied well in a reasonable amount of time. Actually, the search radius of the local search algorithm is 10, which is reasonably small, and we expected it to work fast. We successfully optimized a keyboard using sangjin’s data, and running 10 generations, which resulted in the typo rate being reduced from 5.5% to 2.3%. However, the time that the local search took for 10 generations with 100,000

samples was 10 hours, which was too slow. Therefore, we tried to optimize the keyboard with a Genetic Algorithm.

Compared to the GA, the local search algorithm performed much worse in terms of improvement over time. This is mainly due to the local search requiring many resources. Also it is possible that the data was overfitted. In the beginning, a single dataset was generated from the distribution and used for training. For the final fitness assessment, new data was again generated from the distribution, but since only one data set was used for training, it could possibly lead to overfitting.

To overcome overfitting, for the Genetic Algorithm we decided to generate a new training set for each generation. We applied the genetic algorithm under two settings. One running for 100 generation and 5000 samples per data set for each generation and other with 100 generations, 100,000 samples per data set for each generation. From Sangjin's data, the original keyboard gave 1.82% typos, the 5000-data-set generated keyboard gave0.58% typos and the 100,000-data-set generated keyboard gave a 0.558% typo rate. The 5000-data-set-generation took roughly 30 minutes and 100000-data-set-generation took roughly 10 hours. However, the result did not improve greatly, but only a little. Therefore, we decided to adopt the setting of 5000 data set per generation with 100 generations. The results seem quite promising.

For Dio's generated model, the original keyboard gave 4.38% typos, tested with 100,000 samples. The genetic optimized keyboard gave 2.43% typos, tested under the same settings. It improved about 2% reducing the number of typos. For Sangjin's input, the original keyboard gave 1.82% typos. The genetic optimized keyboard gave 0.58% typos. For Jeongmin, the original keyboard gave 0.13% typos, and the genetic algorithm optimized keyboard gave 0.015% typos. For Kaz's, the original keyboard gave 46.15% (intentionally gathered data making lots of typos). The genetic optimized keyboard gave 27.62% typos.

As an interesting side fact, the smaller the original typo error, the larger the observed percentage improvement.

Conclusion

We developed an android application to gather keyboard usage data from real users and used it for our initial data gathering. However, this was taking a lot of time and effort, so we generalized the gathered data in the form of a distribution, which allowed us to generate large data sets in practically no time.

At first, we tried to optimize each user's keyboard using local search algorithm. For this, we came up with a keyboard representation, established what a neighborhood is, and defined the fitness function to represent the keyboard optimization as a search

problem. We also implemented it and trained it with test data. However, it took about 10 hours to reduce the typo frequency from 5.5% to 2.3%. Due to the long training time, we decided to use GA as an alternative approach. We redefined the keyboard representation, established the genetic operators and the fitness function for GA and implemented it.

GA worked really well for various typing styles. We had 4 users with different typing styles. The user with an intentionally high typo rate of 46.15% reduced to 27.62%, which is a huge improvement. For normal users with 4.38% and 1.82% typo rate, the error rate was reduced to 2.43% and 0.58%, respectively. The user with a really low typo rate of 0.13% was reduced to 0.015%, which is almost error free.

The training took about 30 minutes, and for all of 4 users, it achieved at least a 40% typo reduction. We can conclude that we could use this GA technique as a valid mobile keyboard optimization tool.