Overview

You use auto-correct everyday. When you send your friend a text message, or when you make a mistake in a query, there is an autocorrect behind the scenes that corrects the sentence for you. This week you are also going to learn about minimum edit distance, which tells you the minimum amount of edits to change one word into another. In doing that, you will learn about dynamic programming which is an important programming concept which frequently comes up in interviews and could be used to solve a lot of optimization problems.

- What is autocorrect?
- Building the model
- Minimum edit distance
- Minimum edit distance algorithm

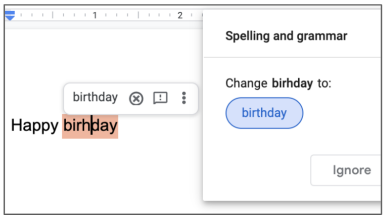
deah → dear ✅
yeah
dear
dean
... etc

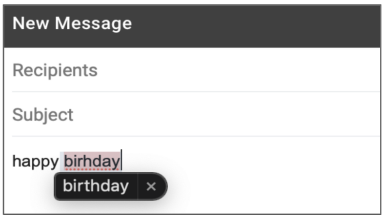
	#	s	t	a	y
#	0	1	2	3	4
p	1	2	3	4	5
l	2	3	4	5	6
a	3	4	5	4	5
y	4	5	6	5	4

Autocorrect

Autocorrects are used everywhere. You use them in your phones, tablets, and computers.

- Phones
- Tablets
- Computers





To implement autocorrect in this week's assignment, you have to follow these steps:

- Identify a misspelled word
- Find strings **n** edit distance away: (these could be random strings)

- Filter candidates: (keep only the real words from the previous steps)
- Calculate word probabilities: (choose the word that is most likely to occur in that context)

Building the model

1. Identify the misspelled word

When identifying the misspelled word, you can check whether it is in the vocabulary. If you don't find it, then it is probably a typo.

2. Find strings n edit distance away

- Edit: an operation performed on a string to change it

- Insert (add a letter) 'to': 'top', 'two' ...
- Delete (remove a letter) 'hat': 'ha', 'at', 'ht'
- Switch (swap 2 adjacent letters) 'eta': 'eat', 'tea'
- Replace (change 1 letter to another) 'jaw': 'jar', 'paw', ...

3. Filter candidates

In this step, you want to take all the words generated above and then only keep the actual words that make sense and that you can find in your vocabulary.

deah		deah
_eah		yeah
d_ar	→	dear
de_r		dean
... *etc*		... *etc*

Building the model II

4.0 Calculating word probabilities

Calculate word probabilities

Example: “I am happy because I am learning”

Word	Count
I	2
am	2
happy	1
because	1
learning	1

Total : 7

$$P(w) = \frac{C(w)}{V} \qquad P(\text{am}) = \frac{C(\text{am})}{V} = \frac{2}{7}$$

$P(w)$ Probability of a word

$C(w)$ Number of times the word appears

V Total size of the corpus

Note that you are storing the count of words and then you can use that to generate the probabilities. For this week, you will be counting the probabilities of words occurring. If you want to build a slightly more sophisticated auto-correct you can keep track of two words occurring next to each other instead. You can then use the previous word to decide. For example which combo is more likely, *there friend* or *their friend?* For this week however you will be implementing the probabilities by just using the word frequencies. Here is a summary of everything you have seen before in the previous two videos.

1. Identify a misspelled word
2. Find strings n edit distance away
   - Insert
   - Delete
   - Switch
   - Replace
3. Filter candidates
4. Calculate word probabilities

$$P(w) = \frac{C(w)}{V}$$

deah → dear ✅
yeah
dear
dean
... *etc*

Minimum edit distance

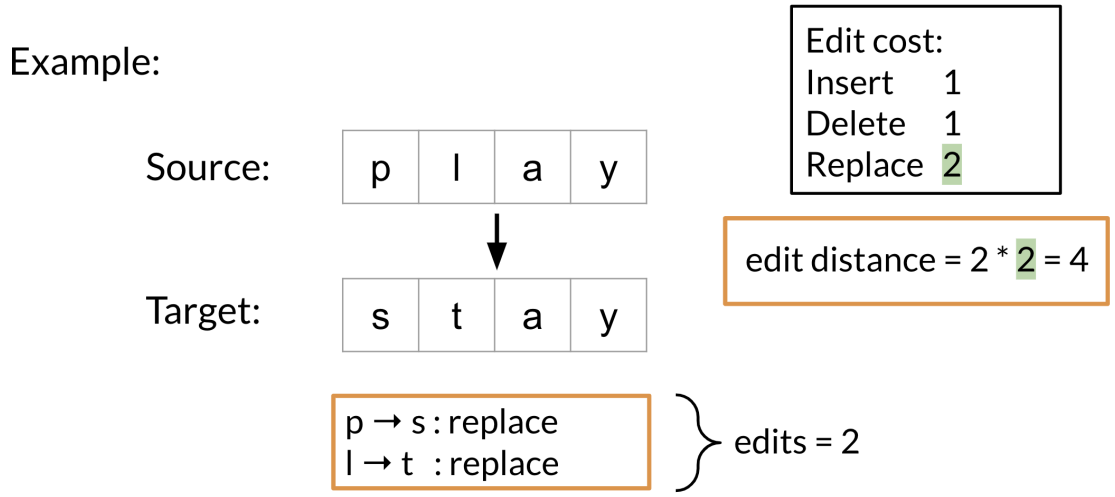
Minimum edit distance allows you to:

- Evaluate similarity between two strings
- Find the minimum number of edits between two strings
- Implement spelling correction, document similarity, machine translation, DNA sequencing, and more

Remember that the edits include:

- Insert (add a letter) 'to': 'top', 'two' …
- Delete (remove a letter) 'hat': 'ha', 'at', 'ht'
- Replace (change 1 letter to another) 'jaw': 'jar', 'paw', …

Here is a concrete example where we calculate the cost (i.e. edit distance) between two strings.



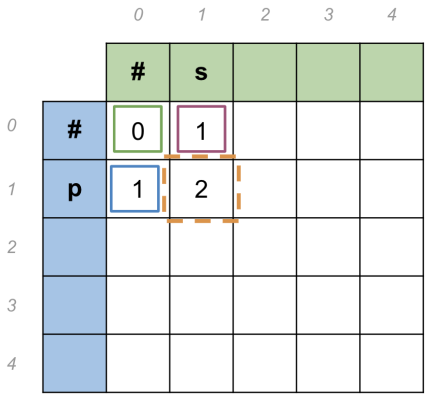
Note that as your strings get larger it gets much harder to calculate the minimum edit distance. Hence you will now learn about the minimum edit distance algorithm!

Minimum edit distance algorithm

When computing the minimum edit distance, you would start with a *source word* and transform it into the *target word*. Let's look at the following example:

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

p → s
insert + *delete*: p → ps → s: 2
delete + *insert*: p → # → s: 2
replace: p → s: 2



To go from #→# you need a cost of 0. From $p \rightarrow \#$ you get 1, because that is the cost of a delete. $p \rightarrow s$ is 2 because that is the minimum cost one could use to get from **p** to **s**. You can keep going this way by populating one element at a time, but it turns out there is a faster way to do this. You will learn about it next.

Minimum edit distance algorithm II

To populate the following table:

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

p → s

$$D[i,j] = \min \begin{cases} D[i-1,j] + del\_cost \\ D[i,j-1] + ins\_cost \\ D[i-1,j-1] + \begin{cases} rep\_cost; & \text{if } src[i] \neq tar[j] \\ 0; & \text{if } src[i] = tar[j] \end{cases} \end{cases}$$

		#	s	t	a	y
0	#	0	1	2	3	4
1	p	1	2			
2	l	2				
3	a	3				
4	y	4				

There are three equations:

- **D[i,j] = D[i-1, j] + del_cost:** this indicates you want to populate the current cell (i,j) by using the cost in the cell found directly above.
- **D[i,j] = D[i, j-1] + ins_cost:** this indicates you want to populate the current cell (i,j) by using the cost in the cell found directly to its left.
- **D[i,j] = D[i-1, j-1] + rep_cost:** the rep cost can be 2 or 0 depending if you are going to actually replace it or not.

At every time step you check the three possible paths where you can come from and you select the least expensive one. Once you are done, you get the following:

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

		#	s	t	a	y
0	#	0	1	2	3	4
1	p	1	2	3	4	5
2	l	2	3	4	5	6
3	a	3	4	5	4	5
4	y	4	5	6	5	4

Minimum edit distance III

Source: play → Target: stay
Cost: insert: 1, delete: 1, replace: 2

- Levenshtein distance
- Backtrace
- Dynamic programming

		0	1	2	3	4
		#	s	t	a	y
0	#	0	1	2	3	4
1	p	1	2	3	4	5
2	l	2	3	4	5	6
3	a	3	4	5	4	5
4	y	4	5	6	5	4

To summarize, you have seen the levenshtein distance which specifies the cost per operation. If you need to reconstruct the path of how you got from one string to the other, you can use a backtrace. You should keep a simple pointer in each cell letting you know where you came from to get there. So you know the path taken across the table from the top left corner, to the bottom right corner. You can then reconstruct it.

This method for computation instead of brute force is a technique known as dynamic programming. You first solve the smallest subproblem first and then reusing that result you solve the next biggest subproblem, saving that result, reusing it again, and so on. This is exactly what you did by populating each cell from the top right to the bottom left. It's a well-known technique in computer science!