Natural Language Processing

Session 3: **Regular Expressions**

Instructor: Behrooz Mansouri
Spring 2023, University of Southern Maine

Previous Session

In previous session we learned about:

- ✓ Coding in python
- ✓ Using Google Colab
- ✓ Revisited programming concepts in Python

Regular Expressions

Regular Expressions and their Usage

Regular expression (RE): a language for specifying text search strings

- They are particularly useful for searching in texts, when we have a pattern to search for and a corpus of texts to search through
  - In an information retrieval (IR) system such as a Web search engine, the texts might be entire documents or Web pages
  - In a word-processor, the texts might be individual words, or lines of a document
- grep command in Linux
  - grep ‘nlp’ /path/file

Basic Regular Expressions

The simplest kind of regular expression is a sequence of simple characters

- For example, to search for language, we type */language/*
- The search string can consist of a single character (like /!/) or a sequence of characters (like /urgl/)

Regular expressions are case-sensitive; lower case /s/ is distinct from uppercase /S/

Basic Regular Expressions

The simplest kind of regular expression is a sequence of simple characters

- For example, to search for language, we type */language/*
- The search string can consist of a single character (like /!/) or a sequence of characters (like /urgl/)

Regular expressions are case-sensitive; lower case /s/ is distinct from uppercase /S/

Can use of the square braces

- The string of characters inside the braces specifies a **disjunction** of characters to match
- /[lL]anguage/
- The regular expression */[1234567890]/* specified any single digit

Ranges in []: If there is a well-defined sequence associated with a set of characters, dash (-) in brackets can specify any one character in a range

- /[A-Z]/ an upper case letter "we should call it 'Drenched Blossoms' "

Basic Regular Expressions

Negations in []:

- The square braces can also be used to specify what a single character cannot be, using the caret ^
- If the caret ^ is the first symbol after the open square brace [, the resulting pattern is negated
- *[^A-Z]* Not an upper case letter
- *[^a-z]* Not a lower case letter
- *[^Ss]* Neither ‘S’ nor ‘s’

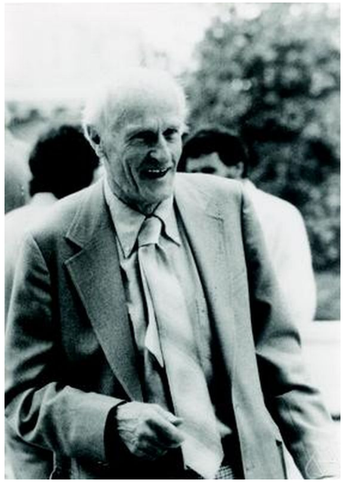
If E_1 and E_2 are regular expressions, then $E_1 \mid E_2$ is a regular expression

- *woodchuck|groundhog* → woodchuck or groundhog
- *a|b|c* → a, b or c

Closure Operators: Kleene * and Kleene +

- Kleene * (closure) operator: The Kleene star means “zero or more occurrences of the immediately previous regular expression
- Kleene + (positive closure) operator: The Kleene plus means “one or more occurrences of the immediately preceding regular expression

Regular Expression	Matches
ba*	b, ba, baa, baaa, ...
ba+	ba, baa, baaa, ...
(ba)*	∅, ba, baba, bababa, ...
(ba)+	ba, baba, bababa, ...
(b|a)+	b, a, bb, ba, aa, ab, ...


Stephen Kleene, 1909 - 1994

Wildcard, Question Mark, and Curly Bracelet

- A wildcard expression (dot) . matches any single character (except a carriage return)
  - beg.n → begin, begun, begxn, …
  - a.*b → any string starts with a and ends with b

- The question mark ? marks optionality of the previous expression
  - woodchucks? → woodchuck or woodchucks
  - colou?r → color or colour
  - (a|b)?c → ac, bc, c

- {m,n} causes the resulting RE to match from m to n repetitions of the preceding RE
- {m} specifies that exactly m copies of the previous RE should be matched
  - (ba){2,3} → baba, bababa

Precedence of Operators

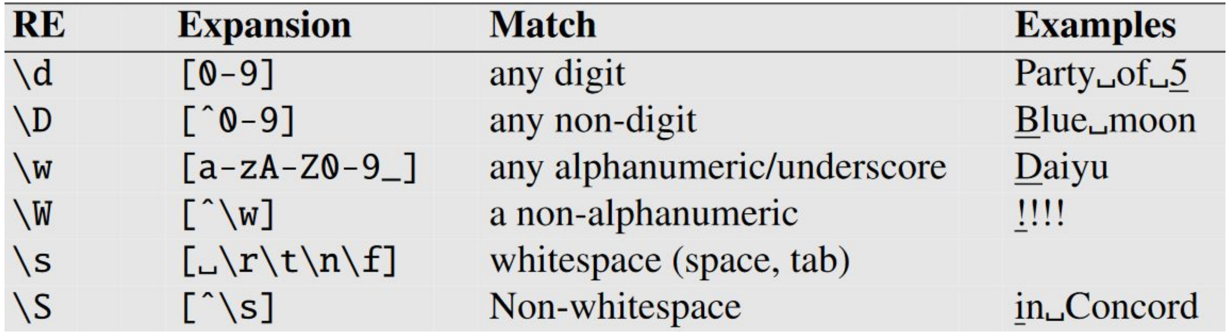
The order precedence of RE operator precedence, from highest precedence to lowest precedence is as follows

- Parenthesis ()
- Counters * + ? {}
- Sequences and anchors ^ $
- Disjunction |

- The regular expression the* matches theeeee but not thethe
- The regular expression (the)* matches thethe but not theeeee

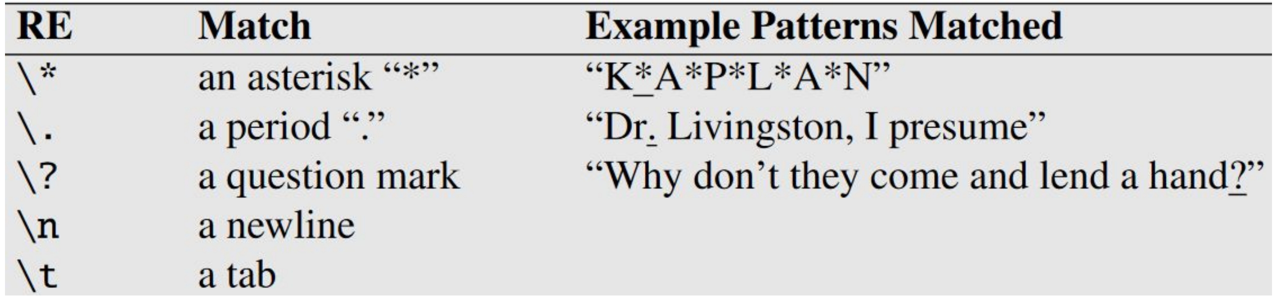
Advanced Operators

Aliases for common sets of characters



RE	Expansion	Match	Examples
`\d`	`[0-9]`	any digit	Party␣of␣5
`\D`	`[^0-9]`	any non-digit	Blue␣moon
`\w`	`[a-zA-Z0-9_]`	any alphanumeric/underscore	Daiyu
`\W`	`[^\w]`	a non-alphanumeric	!!!!
`\s`	`[␣\r\t\n\f]`	whitespace (space, tab)	
`\S`	`[^\s]`	Non-whitespace	in␣Concord

Special characters need to be backslashed



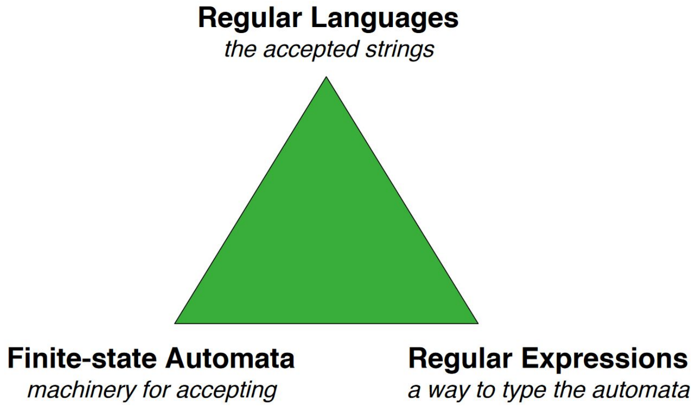
RE	Match	Example Patterns Matched
`\*`	an asterisk "*"	"K*A*P*L*A*N"
`\.`	a period "."	"Dr. Livingston, I presume"
`\?`	a question mark	"Why don't they come and lend a hand?"
`\n`	a newline	
`\t`	a tab	

Finite State Automaton

Any regular expression can be realized as a finite state automaton (FSA)

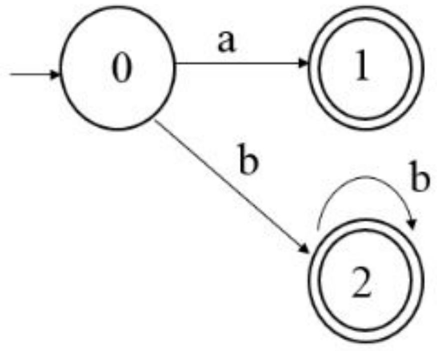
There are two kinds of FSAs

- Deterministic Finite state Automatons (DFAs)
- Non-deterministic Finite state Automatons (NFAs)

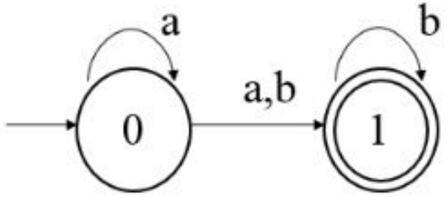

Regular Languages
the accepted strings
Finite-state Automata
machinery for accepting
Regular Expressions
a way to type the automata

A FSA (a regular expression) represents a regular language

Regular Expressions: A DFA and A NFA



A DFA : $a \mid b^+$



A NFA: $a^*(a|b)b^*$

Any NFA can be converted into a corresponding DFA

Python Regular Expressions

The re library in Python is a built-in library that provides support for regular expressions

The re library provides several functions for searching and manipulating strings, including:

- search(): searches for a match anywhere in the string
- findall(): returns a list of all non-overlapping matches in the string
- finditer(): returns an iterator yielding match objects
- sub(): replaces all occurrences of the pattern in the string
- split(): splits the string by the occurrences of the pattern
- compile(): compiles a regular expression pattern into a regular expression object, which can be used for efficient reuse

To use the re library, you first need to import it at the beginning of your Python script by using the following line: import re

And then use the functions and constants provided by the re library to perform regular expression operations on strings

https://www.w3schools.com/python/python_regex.asp

Python Regular Expressions

Extract all the digits from a string:

```
import re
string = "There are 12 monkeys in the zoo"
print(re.findall(r'\d+', string))
# Output: ['12']
```

\d is a metacharacter that matches any digit (0-9)

Extract all the words starting with a specific letter:

```
import re
string = "The quick brown fox jumps over the lazy dog"
print(re.findall(r'\b[Tt]\w+', string))
# Output: ['The', 'the']
```

\b is a metacharacter that matches only at the beginning or end of a word

Python Regular Expressions (Class Activity)

1. Extract all email addresses from a string
   string = "The email addresses are be-mansouri@maine and behrooz.mansouri@cs.maine.edu"

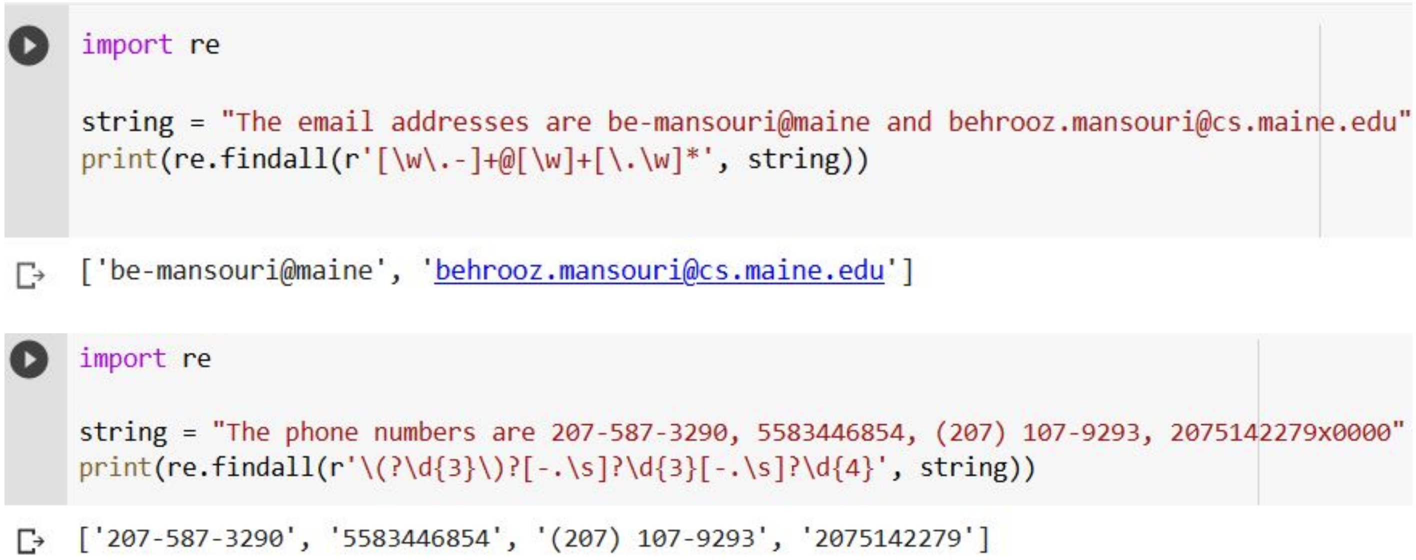
2. Extract all phone numbers from a string
   "The phone numbers are 207-587-3290, 5583446854, (207) 107-9293, 2075142279x0000"

\w Matches Unicode word characters
\s Matches Unicode whitespace characters (which includes [\t\n\r\f\v], and also many other characters, for example the non-breaking spaces mandated by typography rules in many languages)
[-.\s] Matches any of the characters inside the square bracket, in this case -, . and space

Python Regular Expressions (Class Activity)

1. Extract all email addresses from a string
   string = "The email addresses are be-mansouri@maine and behrooz.mansouri@cs.maine.edu"

2. Extract all phone numbers from a string
   "The phone numbers are 207-587-3290, 5583446854, (207) 107-9293, 2075142279x0000"



```python
import re

string = "The email addresses are be-mansouri@maine and behrooz.mansouri@cs.maine.edu"
print(re.findall(r'[\w\.-]+@[\w]+[\.\w]*', string))
```

```
['be-mansouri@maine', 'behrooz.mansouri@cs.maine.edu']
```

```python
import re

string = "The phone numbers are 207-587-3290, 5583446854, (207) 107-9293, 2075142279x0000"
print(re.findall(r'\(?\d{3}\)?[-.\s]?\d{3}[-.\s]?\d{4}', string))
```

```
['207-587-3290', '5583446854', '(207) 107-9293', '2075142279']
```

Words and Corpora

Corpus

Corpus (plural corpora), a computer-readable collection of text or speech

- For example the Brown corpus is a million-word collection of samples from 500 written texts from different genres (newspaper, fiction, non-fiction, academic, etc.), assembled at Brown University in 1963–64

Punctuation is critical for finding boundaries of things (commas, periods, colons) and for identifying some aspects of meaning (question marks, exclamation marks, quotation marks)

- For some tasks, like part-of-speech tagging or parsing or speech synthesis, we sometimes treat punctuation marks as if they were separate words

Utterance

An utterance is the spoken correlate of a sentence

- I do *uh main*- mainly business data processing

This utterance has two kinds of disfluencies: fragments and fillers

(1) The broken-off word main- is called a fragment

(e.g., taking pause)

(2) Words like *uh and um* are called fillers or filled pauses

Another Example; A speech disfluency, also spelled speech dysfluency;

- is any of various breaks, irregularities in particular language.

Word Type

“How many words are there in English?”

Word Type

“How many words are there in English?”

To answer this question, we need to distinguish two ways of talking about words: (1) WORD TYPE; (2) WORD TOKEN

Word Type is the number of distinct words in a corpus

- If the set of words in the vocabulary is V, the number of types is the vocabulary size |V|

Word Token are the total number of N running words

e.g. They picnicked by the pool, then lay back on the grass and looked at the stars

- Word Types in this sentence are 14
- Word Tokens in this sentence are 16

Text Normalization

Text Preprocessing

Before process any natural language processing of a text, the text has to be normalized

At least three tasks are commonly applied as part of any normalization process

(a) Segmenting/tokenizing words from running text

(b) Normalizing word formats

(c) Segmenting sentences in running text

Segmenting / Tokenizing

Separating out words from sentences

Example: “It is sunny today.” → “It” “is” “sunny” “today”

Tokenization algorithms may also tokenize multiword expressions like New York or rock 'n' roll as a single token

Tokenization can be more challenging for other languages:

- German noun compounds are not segmented:
    - Lebensversicherungsgesellschaftsangestellter "Life Insurance Company Employee"
- Chinese and Japanese no spaces between words
    - 莎拉波娃现在居住在美国东南部的佛罗里达。

Text Normalization

Tokens can also be normalized, in which a single normalized form is chosen for words with multiple forms like USA and US

- This standardization may be valuable, despite the spelling information that is lost in the normalization process
  - "$200" would be pronounced as "two hundred dollars" in English
- For information retrieval, we want a query for US to match a document that has USA

Case folding is another kind of normalization: Reduce all letters to lower case. (US versus us are important)

- For most applications (information retrieval), case folding is helpful
- For some NLP applications (MT, information extraction) cases can be helpful

Lemmatization

Lemmatization is the task of determining that two words have the same root, despite their surface differences

● am, are, is	→ be
● car, cars, car's, cars'	→ car

Lemmatization: have to find correct dictionary headword form of the Word

Lemmatization algorithms can be complex. For this reason, we sometimes make use of a simpler but cruder method, which mainly consists of chopping off word-final affixes

- This naïve version of morphological analysis is called stemming

Sentence Segmentation

Separate out sentences from a paragraph/text

Example: Shelby is a student. She is sick today. She will not go to school.

After segmenting:

- “Shelby is a student”
- “She is sick today”
- “She will not go to school”

Question marks and exclamation points are relatively unambiguous markers of sentence boundaries

Periods, on the other hand, are more ambiguous

- Abbreviations like Inc. or Dr.
- Numbers like .02% or 4.3

Minimum Edit Distance

String Edit Distance

Given two strings (sequences) return the “distance” between the two strings as measured by the minimum number of “character edit operations” needed to turn one sequence into the other

Natural → Nature

1.	Substitution	a → e
2.	Deletion	l

Distance = 2

Damerau-Levenshtein distance

Counts the minimum number of insertions, deletions, substitutions, or transpositions of single characters required

- e.g., Damerau-Levenshtein distance 1 (80% of errors caused by a single char)

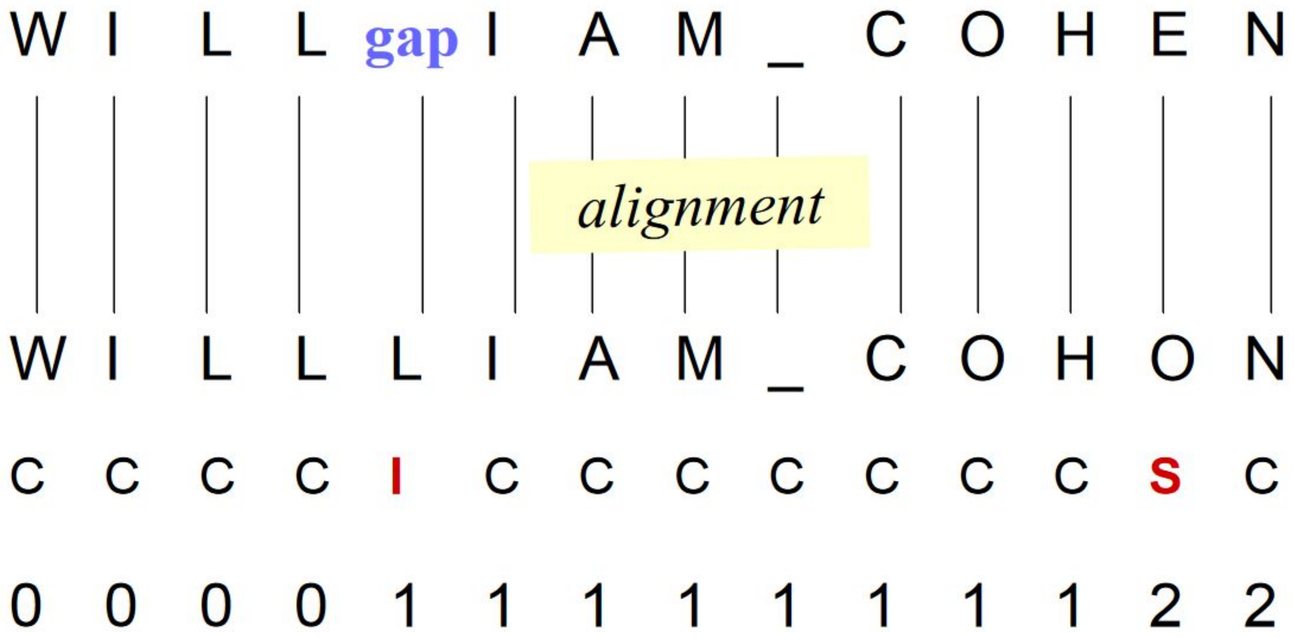
$$\text{extenssions} \rightarrow \text{extensions}(\text{insertion error})$$
$$\text{poiner} \rightarrow \text{pointer (deletion error)}$$
$$\text{marshmellow} \rightarrow \text{marshmallow (substitution error)}$$
$$\text{brimingham} \rightarrow \text{birmingham (transposition error)}$$

- distance 2

$$\text{doceration} \rightarrow \text{deceration}$$
$$\text{deceration} \rightarrow \text{decoration}$$

Alignment

distance("William Cohen", "Willliam Cohon")


W I L L gap I A M _ C O H E N
alignment
W I L L L I A M _ C O H O N
C C C C I C C C C C C C S C
0 0 0 0 1 1 1 1 1 1 1 1 2 2

Given two sequences, an alignment is a correspondence between substrings of the two sequences

Dynamic Program Table for String Edit

Measure distance between strings PARK and SPAKE

		P	**A**	**R**	**K**
S					
P					
A			$\mathbf{C_{ij}}$		
K					
E					

c_{ij} = the number of edit operations needed to align PA with SPA

Dynamic Program Table for String Edit

Measure distance between strings PARK and SPAKE

		P	A	R	K
	0	1	2	3	4
S	1				
P	2				
A	3		C_{ij}		
K	4				
E	5				

D(i,j) = score of best alignment from s1..si to t1..tj

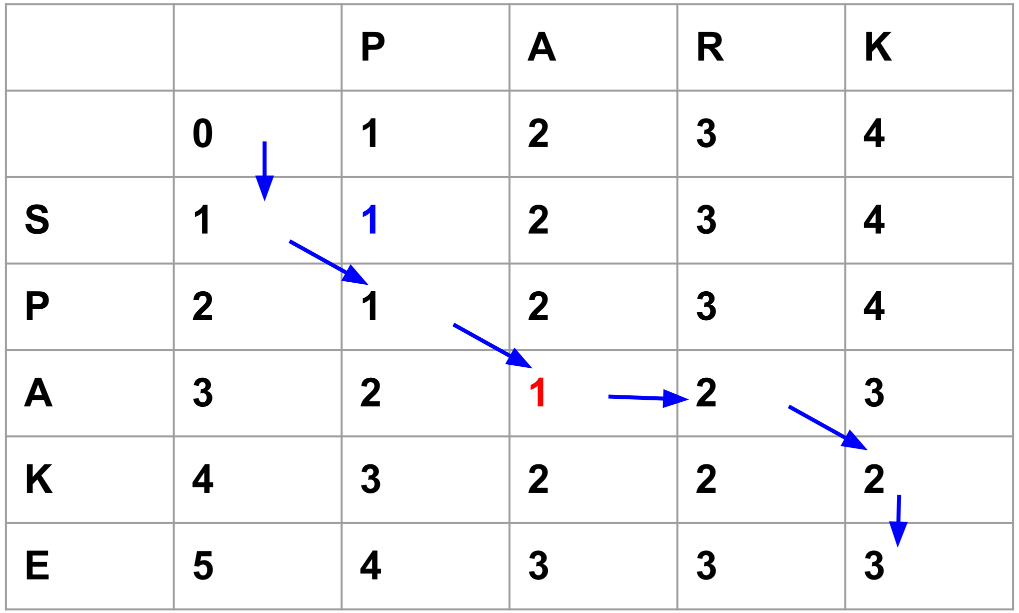
$$= \min \begin{cases} D(i-1,j-1)+\boxed{d(si,tj)} & \textit{//substitute} \\ D(i-1,j)+1 & \textit{//insert} \\ D(i,j-1)+1 & \textit{//delete} \end{cases}$$

d(c,d) is an arbitrary distance function on characters

c_{ij} = the number of edit operations needed to align PA with SPA

Dynamic Program Table for String Edit

Measure distance between strings PARK and SPAKE



		P	A	R	K
	0	1	2	3	4
S	1	1	2	3	4
P	2	1	2	3	4
A	3	2	1	2	3
K	4	3	2	2	2
E	5	4	3	3	3

c_{ij} = the number of edit operations needed to align PA with SPA

WHAT HAVE YOU LEARNED?

Summary

Today we learned about:

- ✓ Regular Expressions
- ✓ Word and Corpora
- ✓ Text Normalization
- ✓ String Edit Distance

Next Session

Tokenization and Stemming

In the next session, we will explore tokenization and stemming

To do before next session:

- Chapter 2 of Jurafsky's book (link)
- Assignment 1 is ready
- We will have the first Quiz