Text Processing & Data Structures for NLP

A Tutorial (CSE 562/662)

Kristy Hollingshead

Fall 2008

www.cslu.ogi.edu/~hollingk/NLP_tutorial.html

regexp Text Processing Overview

- The goal here is to make your lives easier!
- NLP is very text-intensive
- Simple tools for text-manipulation
  - `sed, awk, bash/tcsh`
  - `split`
  - `sort`
  - `head, tail`
- When & how to use each of these tools

Regular expressions crash course

- [a-z] exactly one lowercase letter
- [a-z]* zero or more lowercase letters
- [a-z]+ one or more lowercase letters
- [a-zA-Z0-9] one lowercase or uppercase letter, or a digit
- [^(] match anything that is *not* '('

sed: overview

- a stream editor
- WHEN
  - "search-and-replace"
  - great for using regular expressions to change something in the text
- HOW
  - `sed 's/regexp/replacement/g'`
    - `'s/...` = substitute
    - `.../g'` = global replace
      (otherwise will only replace first occurrence on a line!)

sed: special characters

- ^ — the start of a line…
  except at the beginning of a character set (e.g., [^a-z]), where it complements the set
- $ — the end of a line
- & — the text that matched the regexp

- We'll see all of these in examples…

sed: (simple) examples

- eg.txt =
  The cops saw the robber with the binoculars
- `sed 's/robber/thief/g' eg.txt`
  The cops saw the thief with the binoculars
- `sed 's/^/She said, "/g' eg.txt`
  She said, "The cops saw the robber with the binoculars
- `sed 's/^/She said, "/g' eg.txt | sed 's/$/"/g'`
  She said, "The cops saw the robber with the binoculars"

awk: overview

- a simple programming language specifically designed for text processing
  - somewhat similar in nature to Tcl
- WHEN
  - using simple variables (counters, arrays, etc.)
  - treating each word in a line individually
- HOW
  -
```
awk 'BEGIN      {initializations}
     /regexp1/  {actions1}
     /regexp2/  {actions2}
     END        {final actions}' file.txt
```
  (blue text indicates optional components)

awk: useful constructions & examples

- each word in a line is a 'field'
  `$1, $2, …, $NF`
  imagine every line of text as a row in a table; one word per column. `$1` will be the word in the first column, `$2` the next column, and so on up through `$NF` (the last word on the line)
- `$0` – the entire row
- eg3.txt =
  The cow jumped over the moon
- `awk '{print $2}' eg3.txt`
  cow
- ```
  cat eg3.txt | awk '{$NF=42; print $0; \
                      $1="An old brown"; print $0;}' -
  ```
  The cow jumped over the 42
  An old brown cow jumped over the 42
  ```

awk: useful constructions & examples

- eg3.txt =
  The cow jumped over the moon
- if statements
  - `awk '{if ($1 == "he") { print $0; }}' eg3.txt`

    (empty)
  - `awk '{if ($1 ~ "he") { print $0; } else { … }}' eg3.txt`

    The cow jumped over the moon
- for loops
  - `awk '{for (j=1; j <= NF; j++) { print $j }}' eg3.txt`

    The
    cow
    jumped
    over
    the
    moon
  - what if I only wanted to print every other word (each on a new line), in reverse order?

    ```
    awk '{for (j=NF; j > 0; j-=2) { print $j }}' eg3.txt
    ```

awk: useful constructions & examples

- eg4.txt =
  The cow jumped over the moon
  And the dish ran away with the spoon
- printf statements
  - `awk '{for (j=1; j <= NF; j++) { \`
    `printf("%d\t%s\n",j,$j);}}' eg4.txt`
  - what if I want continuous numbering?
  - `awk 'BEGIN {idx=0;} {for (j=1; j <= NF; j++) { \`
    `printf("%d\t%s\n",idx,$j); idx++;}}' eg4.txt`

```
1 The
2 cow
3 jumped
4 over
5 the
6 moon
1 And
2 the
...
```

awk: useful constructions & examples

- eg4.txt =
  The cow jumped over the moon
  And the dish ran away with the spoon
- substrings
  - substr(<string>, <start>, <end>)
  - ```
    awk '{for (j=1; j <= NF; j++) { \
    printf("%s ",substr($j,1,3))}; print "";}' eg4.txt
    ```
    The cow jum ove the moo
    And the dis ran awa wit the spo
- strings as arrays
  - length(<string>)
  - ```
    awk '{for (j=1; j <= NF; j++) { \
    for (c=1; c <= length($j); c++) { \
    printf("%s ",substr($j,c,1))}; \
    print "";}}' eg4.txt
    ```

```
T h e
c o w
j u m p e d
o v e r
t h e
m o o n
A n d
t h e
…
```

bash: overview

- shell script
- WHEN
  - repetitively applying the same commands to many different files
  - automate common tasks
- HOW
  - on the command line
  - in a file (type `` `which bash` '' to find your location):

```
#!/usr/bin/bash
<commands…>
```

bash: examples

- ```
  for f in *.txt; do
    echo $f;
    tail -1 $f >> txt.tails;
  done
  ```
- ```
  for (( j=0; j < 4; j++ )); do
    cat part$j.txt >> parts0-3.txt;
  done
  ```
- ```
  for f in hw1.*; do
    mv $f ${f//hw1/hw2};
  done
  ```
  ```

miscellaneous

- sort
  - `sort -u file.txt`
    for a uniquely-sorted list of each line in the file
- split
  - `cat file.txt | split -l 20 -d fold`
    divide file.txt into files of 20 lines apiece, using "fold" as the prefix and with numeric suffixes
- wc
  - a counting utility
  - `wc -[l|c|w] file.txt`
    counts number of lines, characters, or words in a file

miscellaneous

- head, tail
  - viewing a small subset of a file
  - `head -42 file.txt`
    for the first 42 lines of file.txt
  - `tail -42 file.txt`
    for the last 42 lines of file.txt
  - `tail +42 file.txt`
    for everything *except the first 42* lines of file.txt
  - `head -42 file.txt | tail -1`
    to see the 42nd line of file.txt
- tr
  - "translation" utility
  - `cat mixed.txt | tr [a-z] [A-Z] > upper.txt`

Putting it all together!

- Let's say I have a text file, and I'd like to break it up into 4 equally-sized (by number of lines) files.
- `wc -l orig.txt`
  8000
- the easy way:

```
cat orig.txt | split -d -l 2000 -a 1 - part;
for f in part*; do mv $f $f.txt; done
```

- the hard way:

```
head -2000 orig.txt > part0.txt
tail +2001 orig.txt | head -2000 > part1.txt
tail +4001 orig.txt | head -2000 > part2.txt
tail -2000 orig.txt > part3.txt
```

Putting it all together!

- Now for each of those files, I'd like to see a numbered list of all the capitalized words that occurred in each file… but I want the words all in lowercase.

```
for f in part*;
 do echo $f;
 cat $f | awk 'BEGIN {idx=0} {
  for (j=1; j <= NF; j++)
   if (substr($j,1,1) ~ "[A-Z]") {
    printf("%d\t%s\n", idx, $j);
    idx++;
   }
 }' - | tr [A-Z] [a-z] >
 ${f//part/out};
 echo ${f//part/out};
done
```

Putting it all together!

- Now I'd like to see that same list, but only see each word once (unique).
- hint: you can tell 'sort' which fields to sort on
- e.g., `sort +3 -4` will skip the first 3 fields and stop the sort at the end of field 4; this will then sort on the 4th field.
  `sort -k 4,4` will do the same thing

```
for f in out*; do
 cat $f | sort +1 -2 -u > ${f//out/unique};
done
```

- and if I wanted to re-number the unique lists?

```
for f in out*; do
 cat $f | sort -k 2,2 -u | awk 'BEGIN {idx=0}
 {$1=idx; print $0; idx++}' > ${f//out/unique};
done
```

Resources

- You can always look at the man page for help on any of these tools!
  - i.e.: `` `man sed' ``, or `` `man tail' ``
- My favorite online resources:
  - sed: www.grymoire.com/Unix/Sed.html
  - awk: www.vectorsite.net/tsawk.html
  - bash: www.tldp.org/LDP/abs/html/
    (particularly section 9.2 on string manipulation)
- Google it. ☺

Warning!

- These tools are meant for very simple text-processing applications!
- Don't abuse them by trying to implement computationally-intensive programs with them
  - like Viterbi search and chart parsing
- Use a more suitable language like C, C++, or Java ... as shown next!

Data Structures for NLP

Disclaimers

- Your coding experience
  - Tutorial intended for beginners up to experts
- C/C++/Java
  - Examples will be provided in C
  - Easily extended to C++ classes
  - Can also use Java classes, though will be slower—maybe prohibitively so
- compiling C
  - `gcc -Wall foo.c -o foo`
  - `-g` to debug with `gdb`

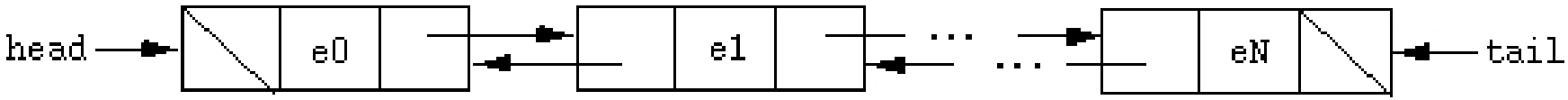
Data Structures Overview

- Storage
  - Lists
  - Trees
  - Pairs (frequency counts)
  - Memory allocation
- Search
  - Efficiency
    - Hash tables
  - Repetition
- Code
  - http://www.cslu.ogi.edu/~hollingk/code/nlp.c

Linked Lists (intro)

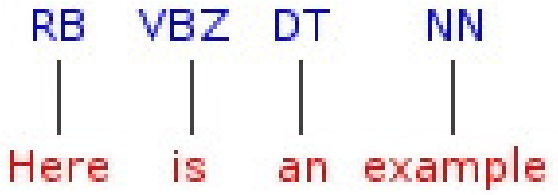
- for each list:
  - first/head node
  - last/tail node (opt)
- for each node:
  - next node
  - previous node (opt)
  - data
- vs arrays

```
struct node;
typedef struct node Node;
typedef struct list {
  Node *head;
  Node *tail;
} List;
struct node {
  char *label;
  Node *next;
  Node *prev
};
```



Linked Lists (NLP)

- example: POS sequence
  (RB Here) (VBZ is) (DT an) (NN example)



- reading in from text (pseudo-code):

```
read_nodes {
  while curr_char != '\n' {
    if (curr_char=='(') {
      prevnode=node; node=new_node();
      node->prev=prevnode;
      if (prevnode!=NULL) prevnode->next=node; }
    node->pos=read_until(curr_char,' ');
    curr_char++; // skip ' '
    node->word=read_until(curr_char,')');
    curr_char++; // skip ')'
}
```

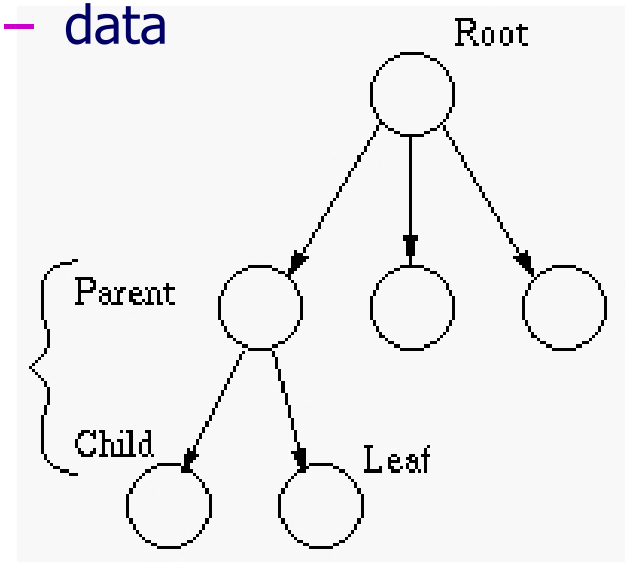
Pairs / Frequency Counts

- Examples
  - What POS tags occurred before this POS tag?
  - What POS tags occurred with this word?
  - What RHS's have occurred with this LHS?
- Lists
  - linear search—only for short lists!
- Counts
  - parallel array
  - or create a 'Pair' data structure!

```
struct pos {
  char *label;
  int numprev;
  struct pos **bitags; }
struct word {
  char *label;
  int numtags;
  struct pos **tags; }
struct rule {
  char *lhs;
  int numrhs;
  struct rhs **rhss; }
struct rhs {
  int len;
  char **labels; }
```

Trees (intro)

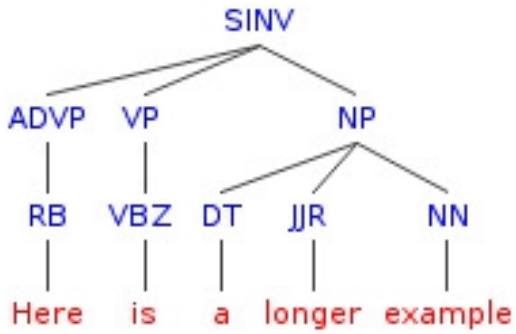
- for each tree:
  - root node
  - next tree (opt)
- for each node:
  - parent node
  - children node(s)
  - data




```
struct tree;
typedef struct tree Tree;
struct node;
typedef struct node Node;
struct tree {
  Node* root;
  Tree* next;
};
struct node {
  char* label;
  Node* parent;
  int num_children;
  Node* children[ ];
};
```

Trees (NLP)

- Examples:
  - parse trees
    (SINV (ADVP (RB Here)) (VP (VBZ is))
    (NP (DT a) (JJR longer) (NN example)) (. .))
  - grammar productions
    NP => DT JJR NN



- reading in from text (pseudo-code):

```
read_trees {
  if (curr_char=='(') {
    node=new_node(); node->lbl=read_until(curr_char,' '); }
  if (next_char!='(') node->word=read_until(curr_char,')');
  if (next_char==')') return node; // "pop"
  else node->child=read_trees();   // recurse
}
```

Manipulate (text) trees with sed

- eg2.txt =
  (TOP (NP (DT The) (NNS cops)) (VP (VBD saw) (NP (DT the) (NN robber)) (PP (IN with) (NP (DT the) (NNS binoculars)))))
- "remove the syntactic labels"
  hint!: all of (and only) the syntactic labels start with '('

```
cat eg2.txt | sed 's/([^ ]* //g' | sed 's/)//g'
```

  The cops saw the robber with the binoculars

- "now add explicit start & stop sentence symbols
  (<s> and </s>, respectively)"

```
cat eg2.txt | sed 's/([^ ]* //g' | sed 's/)//g' |
sed 's/^/<s> /g' | sed 's/$/ <\/s>/g'
```

  <s> The cops saw the robber with the binoculars </s>

Extract POS-tagged words with sed

- eg2.txt =
  (TOP (NP (DT The) (NNS cops)) (VP (VBD saw) (NP (DT the) (NN robber)) (PP (IN with) (NP (DT the) (NNS binoculars)))))
- "show just the POS-and-word pairs: e.g., (POS word)"

```
cat eg2.txt | sed 's/([^ ]* [^(]/~&/g' |
sed 's/[^)~]*~/ /g' |
sed 's/^ *//g' |
sed 's/))*/)/g'
```

(DT The) (NNS cops) (VBD saw) (DT the) (NN robber) (IN with) (DT the) (NNS binoculars)

Manipulate (text) trees with awk

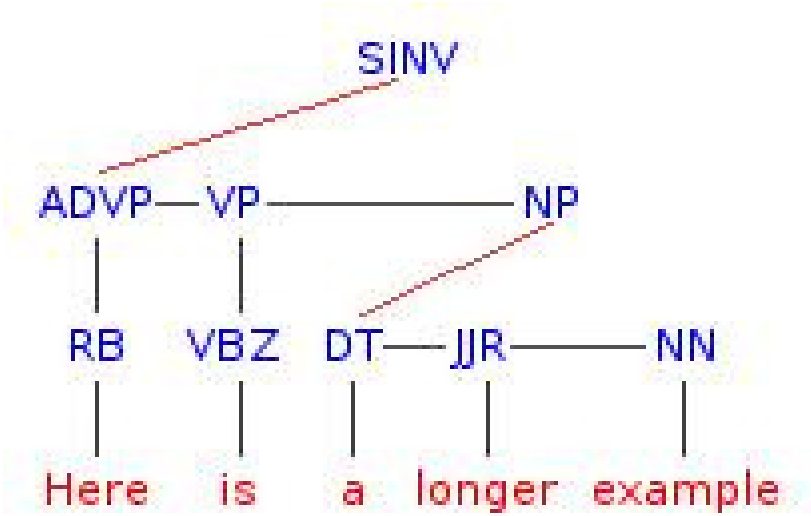
- eg2.txt =
  (TOP (NP (DT The) (NNS cops)) (VP (VBD saw) (NP (DT the) (NN robber)) (PP (IN with) (NP (DT the) (NNS binoculars)))))
- "show just the POS-and-word pairs: e.g., (POS word)"

```
cat eg2.txt | awk '{for (j=1;j<=NF;j++) {
# if $j is a word, print it (without its trailing paren's)
if (substr($j,1,1) != "(") {
 i=index($j,")"); printf("%s ",substr($j,1,i))}
# if $j is a POS label, print it
else {if (j+1<=NF &&
  substr($(j+1),1,1) != "(") printf("%s ",$j)}}
print ""}'
```

(DT The) (NNS cops) (VBD saw) (DT the) (NN robber)
(IN with) (DT the) (NNS binoculars)

Lists *in* Trees (NLP)

- navigation in trees
- convenient to link to "siblings"
  - right sibling ≈ next node
  - left sibling ≈ previous node
- convenient to "grow" children
  - children ≈ first child + right siblings



Memory allocation

- allocation
  - multi-dimensional arrays (up to 3 dim)
- initialization
  - malloc vs calloc
- re-allocation
  - realloc, re-initialize

- pointers
  - minimize wasted space given sparse data sets
- de-referencing

```
int *i;
i[0] ≈ (*i)
```

```
int **dim2;
dim2=
 malloc(10*sizeof(int));
for (i=0;i<10;i++)
 dim2[i]=
  malloc(20*sizeof(int));
dim2[1][0]=42;
```

```
int *dim1;
dim1=malloc(
  10*20*sizeof(int));
dim1[(1*20)+1]=42;
```

Overview

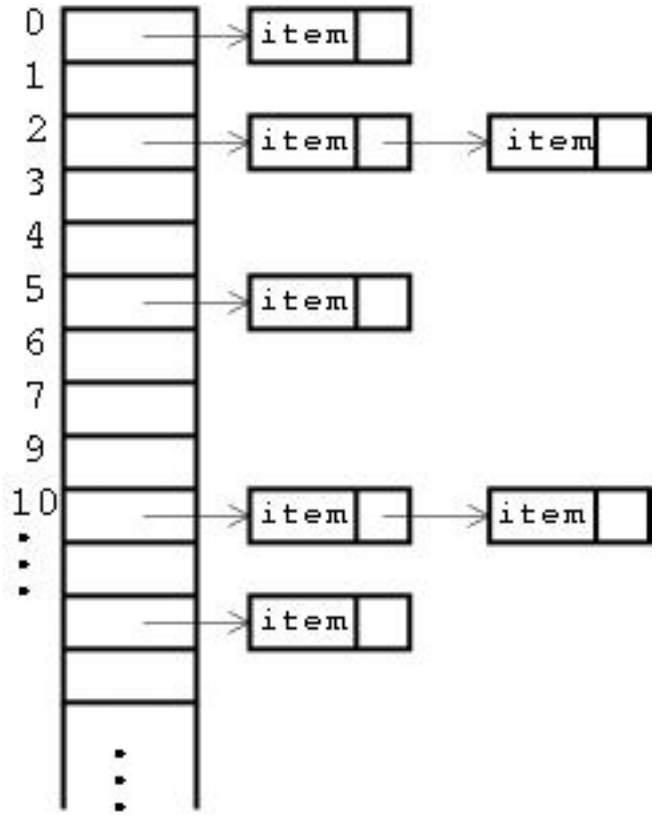
- Storage
  - Lists
  - Trees
  - Pairs (frequency counts)
  - Memory allocation
- Search
  - Efficiency
    - Hash tables
  - Repetition
- Code

Efficiency

- Huge data sets (productions, tags, features)
  - Efficient data structures
    - structs/classes (vs parallel arrays)
    - hash tables (vs binary sort, qsort, etc.)
- Repetitive, systematic searching
  - Search once, then remember
- Brute force just won't work…

Hash Tables (intro)

- Supports efficient look-up (O(1) on avg)
- Maps a key (e.g., *node label*) into a hash code
- Hash code indexes into an array, to find the "bucket" containing desired object (e.g., *node*)
- Collisions
  - Multiple keys (*labels*) mapping to the same "bucket"
  - Chained hashing
  - Open addressing



‡

Chained Hash Table (NLP)

- Data structures to be stored
  - POS data
  - dictionary entries
  - grammar productions
- look-up by label (pseudo-code):

```
typedef struct value {
  char* key;
  int idx;
} Value;

typedef struct hash {
  struct value* v;
  struct hash* next;
} Hash;
```

```
Value* get_value(char* key) {
  int code=get_hash_code(key);
  Value* entry=hash_table[code];
  while (entry && entry->v->key!=key) entry=entry->next;
  if (!entry) make_new_entry(key);
  return entry;
}
```

Repetitious search

- Very repetitive searches in NLP
- Avoid multiple look-ups for the same thing
  - Save a pointer to it
  - Store in a temporary data structure
- Look for patterns
  - Skip *as soon as* you find a (partial) mismatch
    - Make faster comparisons first
      - `(int i == int j)` before `strcmp(s1,s2)`
    - Make "more unique" comparisons first
  - Look for ways to partition the data, save a pointer to each partition
    - Left-factored grammar example

Remember…

- Use data structures (structs/classes)
- Allocate memory sparingly
- Efficiency of search is vital
  - Use hash tables
  - Store pointers
- Don't rely on brute force methods