Lecture 4:
Introducing Mobilize!

“Mobilize” is a project funded by the National Science Foundation for 2010–2013 with potential to expand through 2015. Mobilize involves a partnership between the UCLA Center for Embedded Networked Sensing (CENS), UCLA Center X in the Graduate School of Education and Information Studies, the Los Angeles Unified School District, and the Computer Science Teachers Association. This project will bring CENS Participatory Sensing systems—an innovative method of data collection and interpretation in which individuals use mobile phones to systematically collect and analyze data about their home communities—into Exploring Computer Science classrooms.

Through student-designed research projects using Participatory Sensing, Mobilize builds upon teenagers’ fascination, engagement, and involvement with technology, fostering them to be creators instead of simple users of technology. Exploring Computer Science students and teachers of Mobilize will also work in collaboration with Math, Science, and other subject classrooms in their research projects, therefore developing a multidisciplinary approach to learning.
Last time

We’ll talk a bit about what’s happening with your homework -- Specifically, we talked about the use of web access logs as a source of data for “web analytics” (as well as a popular alternative) and got a sense of the kinds of analysis the tools you have make easy and the kinds of things that seem hard.

We spent some time with “regular expressions” -- Our test cases here toggled between our web access logs from the second lecture and a new data set of recipes appearing in the New York Times in 1987.
Regular expressions

Last time, we motivated the need to think about patterns -- We noted that to define a pattern in text, we should be able to specify the following:

- white space
- word boundaries
- sets or classes of literals
- the beginning and end of a line
- alternatives ("war" or "peace")

We saw how regular expressions accomplish this through a powerful set of metacharacters.
Regular expressions

Our tour of regular expressions began with the simplest patterns, those consisting only of literals -- Characters that simply “match” a single instance of themselves.

We then introduced a collection of metacharacters, showing their application on two text data sets -- In general, the characters \[\^$.|?*+{}\] all mean something special in this pattern “language.”

\(^,\$\) specify positioning

\([\text{ and }]\) express character classes (or equivalence classes)

\((\text{ and } )\), | define subexpressions and alternatives

\(^,+,?\) indicate multiplicities
Today

We will continue our tour of regular expressions, finishing with how they (hopefully) proved helpful in parts of your homework -- We will also look at a few options to grep that make it possible to “see” what’s going on

Along the way, we’ll cover a couple more Unix commands that prove very useful in applications -- One in particular will help us deal with the mass of recipe files that were given to us from the NYTimes

We’ll end with a proposal for another way to deal with the recipe tasks -- Something that's at one instant more and less human
Options to grep

If you’ve examined the man page to `grep`, you’ve noticed that it is a pretty beefy command -- We started with very simple calls

```
grep Obama /data/text_examples/nyt_quotes.txt

grep Republican /data/text_examples/nyt_quotes.txt
```

And last time, we discussed `egrep`, a version of `grep` that provided support for an extended set of metacharacters (in the “basic” version of a regular expression, ?, +, {, |, (, and ) lose their special meanings)

```
egrep '^I (think|feel)' /data/text_examples/nyt_quotes.txt

grep -E '^I (think|feel)' /data/text_examples/nyt_quotes.txt
```
Options to `grep`

The command `egrep` is simply a synonym for `grep -E`; in the same way `fgrep` is the same as `fgrep -F` (Wait what, `fgrep`? How many of these things are there!?)

```
fgrep Republican /data/text_examples/nyt_quotes.txt
```

```
grep -F Republican /data/text_examples/nyt_quotes.txt
```

`fgrep` stands for “fixed strings” (um, and it’s fast) and interprets the pattern you specify as consisting entirely of literals -- metacharacters lose their meaning and you are simply looking for **occurrences of the exact pattern**

`fgrep` appeals to a different matching algorithm and can be much faster than plain `grep` or `egrep` when you are really looking for a “fixed string” of literals

Note: Technically, `egrep` and `fgrep` are “deprecated”, and are kept around to support older applications -- It’s recommended you use `grep -E` and `grep -F` (we thought introducing the single `egrep` command might be less confusing if you’re seeing this for the first time)
Options to `grep`

As we mentioned last time, `grep` can process several files at once -- Like most of the commands we have seen, we can specify a list of input files (notice how the output has changed)

```
grep flour /data/text_examples/1987/09/06/990187.sgml /data/text_examples/1987/09/06/989987.sgml
/data/text_examples/1987/09/06/990187.sgml:Z cup flour
/data/text_examples/1987/09/06/990187.sgml:1. Make the batter. Sift the flour into a large ...
/data/text_examples/1987/09/06/989987.sgml:Z cup flour
/data/text_examples/1987/09/06/989987.sgml:1. Make the batter. Sift the flour into a large ...
```

```
grep flour /data/text_examples/1987/09/06/*
/data/text_examples/1987/09/06/685787.sgml:Z cup flour
/data/text_examples/1987/09/06/685787.sgml:1. Make the batter. Sift the flour into a large ...
/data/text_examples/1987/09/06/989987.sgml:Z cup flour
/data/text_examples/1987/09/06/989987.sgml:1. Make the batter. Sift the flour into a large...
/data/text_examples/1987/09/06/990087.sgml:Z cup flour
/data/text_examples/1987/09/06/990087.sgml:1. Make the batter. Sift the flour into a large...
/data/text_examples/1987/09/06/990187.sgml:Z cup flour
/data/text_examples/1987/09/06/990187.sgml:1. Make the batter. Sift the flour into a large...
```

In the second expression, the star “*” is expanded by the Unix shell and is functioning like a wildcard -- Keep in mind that this “*” and the “*” we'll see in regular expressions mean two different things!

`grep` supports a number of other options for specifying which files to search through, including an option to examine whole directories or even recurse through subdirectories (more later)
Options to `grep`

Our very first regular expression was just a string of literals, the name **Obama** -- With the `--color` option, we can highlight the match exactly

I haven't met one person on the entire trip who is in favor of President **Obama** or any of his policies.

Gramps is sad - **Obama** cut $455 billion from his Medicare.

**Obama** looks to appt him COS; strange doings in the WH.

People might love or hate **Obama**.

Using the command

```
grep -F --color 'Obama' /data/text_examples/nyt_quotes.txt
```
Digging into matches

Slightly more advanced, we can look for quotes that start with “I think” or “I feel” -- the expression

\(^I (\text{think}|\text{feel})\)

will match the quotes

I feel totally normal on a set, probably because I've spent so much of my life on one. It's when I go out into the real world that I don't seem to know what to do.

I think we are bumping along the bottom.

I feel fantastic - especially as it feels like yesterday.

I think about it all the time.

I think waterfronts are great values now.

Using the command

```
grep -E --color '\(^I (\text{think}|\text{feel})\)' /data/text_examples/nyt_quotes.txt
```
Back to metacharacters: ?

Picking up where we left off last time, the question mark indicates that the indicated expression is optional

\[ \textit{George}( \ W\ .\ )\ ? \ \textit{Bush} \]

will match references to “George W. Bush” or just “George Bush”
One thing to note...

In the following

\[ \textit{George( W\.)? Bush} \]

we wanted to match a “.” as a period; Recall that to do that, we had to “escape” the metacharacter, preceding it with a backslash
More metacharacters: * and +

In addition to specifying that something is optional, we can also specify how often something should be repeated -- The * means “any number, including none, of the item” and + means “at least one of the item”

\(\text{\textbackslash (\textbullet \textbullet \textbullet)\textbackslash} \)

will match the recipe lines

1 cup California virgin olive oil (or other strong-flavored olive oil)
8 small breakfast sausages (8 ounces total)
1 dried long purple chili pepper (ancho), pulverized and seeds removed, optional
3 cups Texmati rice (see note)
1 1 1/4-pound live Maine lobster, cut into 10 pieces (cut in half length-wise down the middle, remove claws and cut each half into 4 pieces)
4 squid (about 6 ounces), cut into 1/2-inch rings

Using the command

egrep --color '\(.\*\)\' /data/text_examples/1987/09/06/380087.sgml
More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means “any number, including none, of the item” and + means “at least one of the item”

\([0-9]+ \text{ to } [0-9]+\)

will match the NYT quotes

We believe you could actually increase the corn yield in that region by anywhere from 50 to 80 percent.

You have 500 to 1,000 rogues on Wall Street.

On transcon flights, you can get 30 to 40 users.

From a scale of 1 to 10, it's a 1.

That's a waste of capital for something that might not be utilized for 20 to 30 years.

I've talked to a lot of restaurants, and probably 50 to 60 percent of them have no clue what's going on.

And the command is

```
egrep --color '[0-9]+ \text{ to } [0-9]+\]' /data/text_examples/nyt_quotes.txt
```
A machine view again...

Recall that we can think of a regular expression as a kind of machine that moves along a string, examining each character; we can now revisit that

Suppose we have the simple pattern $a(bb)^+a$, which means matching the character $a$, followed by some number of double $b$'s, followed by a final $a$

The state machine that would do this is given graphically as
A machine view again...

And here is how it would process the string *abbbba*

![Diagram](image)
More `grep` options

With the recipes we have many files scattered across a kind of filesystem database -- Years then months then days, and this structure is true of the entire archive, or at least one version of it

So far, our `grep` commands let us examine one directory (one day) at a time -- Clearly if we want to say something more broadly, we'll need to be able to recurse through the directory tree

One approach is a flavor of `grep` called `rgrep` (or, as you might guess `grep -r`) -- This will run down a directory tree and pull out matching lines from all the files it encounters
An alternative

Not all of the Unix commands we have seen are as flexible as grep -- It is not uncommon for us to have to recurse through a directory tree to operate on all the files we find.

The function `find` helps us here -- It functions as a kind of spider that will walk the directory tree for us identifying files, for example, of a particular type (a regular file or a directory, say) or with a certain name.
What's the traffic look like week after week?

cocteau@homework:~$ find /data/text_examples/1987 -type f | head -5
/data/text_examples/1987/01/21/368187.sgml
/data/text_examples/1987/01/21/608487.sgml
/data/text_examples/1987/01/14/598487.sgml
/data/text_examples/1987/01/14/517787.sgml
/data/text_examples/1987/01/14/549387.sgml

cocteau@homework:~$ find /data/text_examples/1987 -type f | wc
 368 368 15824

cocteau@homework:~$ find /data/text_examples/1987 -type d | head -5
/data/text_examples/1987
/data/text_examples/1987/01
/data/text_examples/1987/01/21
/data/text_examples/1987/01/14
/data/text_examples/1987/01/28

cocteau@homework:~$ find /data/text_examples/1987 -type d | wc
 123 123 3771

cocteau@homework:~$ find /data/text_examples/1987 -maxdepth 1 -type d | sort
/data/text_examples/1987
/data/text_examples/1987/01
/data/text_examples/1987/02
/data/text_examples/1987/03
/data/text_examples/1987/04
/data/text_examples/1987/05
/data/text_examples/1987/06
/data/text_examples/1987/07
/data/text_examples/1987/08
/data/text_examples/1987/09
/data/text_examples/1987/10
/data/text_examples/1987/11
/data/text_examples/1987/12
find /data/text_examples/1987 -type f | cut -d"/" -f5 | sort | uniq -c

24 01
28 02
34 03
31 04
43 05
26 06
32 07
36 08
29 09
16 10
34 11
35 12
An alternative

One final (for the moment) Unix command that’s helpful when used in conjunction with `find` is `xargs` -- Essentially, `xargs` builds commands from standard input

So, we can use `find` to generate a list of recipe files and then `xargs` to pass them as an arguments to `grep`, say

```
grep butter -r /data/text_examples/1987
```

```
find /data/text_examples/1987/ -type f | xargs grep butter
```

The `find/xargs` construction will work for a number of commands we’ve learned so far, whereas the `-r` option is unique to `grep`
Salt to taste if desired
Salt to taste if desired
Salt to taste
Salt to taste if desired
4. Put the vegetables in a saucepan and add cold water to cover. Add salt to taste if desired.
#6. Meanwhile, slowly melt eight tablespoons of butter over medium heat in a 12-inch skillet. Add 1/2 cup chopped
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste.
#4. When the beans are fully cooked, drain them, reserving the water.
Salt to taste if desired
Salt to taste if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
2. Peel potatoes and cut into 1-inch cubes. Combine potatoes and beans in a large saucepan and add reserved water and
Salt to taste if desired
Salt to taste if desired
Salt to taste if desired
Salt to taste if desired
Salt to taste, if desired
Salt to taste, if desired
1 3/4 pounds sea scallops Salt to taste, if desired
Salt to taste, if desired
Salt to taste if desired
Salt to taste if desired
Salt to taste, if desired
Salt to taste, if desired
Coarse salt to taste
Coarse salt to taste
Coarse salt to taste
Coarse salt to taste
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
Salt to taste, if desired
2. In a kettle, bring enough water to the boil to cover the green beans when they are added. Add salt to taste. Add the beans and cook about two minutes. Drain and set aside.
More metacharacters: \{ and \}\n
\{ and \} are referred to as interval quantifiers; the let us specify the minimum and maximum number of matches of an expression

'butter ([[:alnum:]]+ ){1,5}flour'

'butter (\w+ ){1,5}flour'

Either of these will match the recipe lines

/data/text_examples/1987/12/23/604687.sgml:2. Rub a baking sheet all over with *butter* and *sprinkle* with *flour*. Shake sheet back and forth until surface is coated with flour. Shake off excess.

/data/text_examples/1987/09/20/332387.sgml:6. Meanwhile, knead the remaining four tablespoons of *butter with the flour*.

/data/text_examples/1987/09/19/827487.sgml:5. Prepare beurre manie by blending *butter or margarine with flour* to a paste.
More metacharacters: { and }

\{m,n\} means at least m but not more than n matches

\{m\} means exactly m matches

\{m,\} means at least m matches
More metacharacters: ( and ) revisited

In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a “|”, but also can be used to “remember” text matched by the subexpression enclosed

We refer to the matched text with \1, \2, etc.
More metacharacters: ( and ) revisited

So the expression

' ([a-zA-Z]+) \1 '

will match the NYT quotes

I didn't think that what I was was so bad that I needed to hide it.

We don't use cloth diapers, we use disposable diapers. So yes, I'm guilty about that. The first kid we had, we were boasting to my parents we were going to go cloth. My parents rolled their eyes and were, like, 'Call us in six months.' It lasted about two days. We never did do an impact study of water usage for cleaning versus disposable diapers, but babies use a lot of diapers. I need to figure out how to make diapers out of Poo Poo Paper.

Call the American Soy Bean Board - you know how many gallons of fuel they'll tell you you can get out of an acre of land? Three or four gallons per bushel per year. How many gallons of gasoline do we use in a day? Twenty-two million.

Using the command

```bash
egrep --color ' ([a-zA-Z]+) \1 ' /data/text_examples/nyt_quotes.txt
```
Some words of warning

So far, we have focused mainly on the presence of at least one instance of a pattern in a string; as I mentioned at the beginning, we are often looking to extract the matches from the target strings or perhaps replace them.

Then we need to be a little more careful about how a match happens...
References

Here are lines from the NYT quotes file that match the pattern

'\'.*''

What do you notice?

/data/text_examples/1987/05/27/035887.sgml:"I don't care for the smoked," he said. "The smoked has no taste for me. But I like the pickled. I couldn't tell that from pickled herring."

/data/text_examples/1987/05/27/035887.sgml:Francine Schumback of Hastings-on-Hudson, N.Y., said she liked shad either smoked or pickled, "but not baked." And then there was Betty Porter of Ossining, N.Y., who said she adored baked shad but that her husband preferred salmon.

/data/text_examples/1987/05/27/035887.sgml:"We only have it twice a week while the shad is running," she said. "My husband won't let me serve it more than that. I clean it and bake it and serve it. It tastes like butter."
Greedy * and +

When you specify repetition with * and +, the engine performs a greedy search for your pattern; by that I mean it will continue to match arbitrary characters with . until the regular expression would fail.

This is why it combines pairs of quoted phrases; in Python and R, we can flip this behavior with *? and +? (egrep does not support this construction)

```r
> text <- readLines("/data/text_examples/r_example.txt")
> gregexpr("'.*?'",text)
[[1]]
[1]  1 43
attr("match.length")
[1]  32 102

[[2]]
[1]  95
attr("match.length")
[1]  18

[[3]]
[1]  1 71
attr("match.length")
[1]  59 112
```
Less greedy...

If you examine the output from `gregexpr()` you find the start and length of each match -- I've repeated our color scheme from previous examples here to show you what the R output indicates

```
".*?"
```

What do you notice?

"I don't care for the smoked," he said. "The smoked has no taste for me. But I like the pickled. I couldn't tell that from pickled herring."

Francine Schumback of Hastings-on-Hudson, N.Y., said she liked shad either smoked or pickled, "but not baked." And then there was Betty Porter of Ossining, N.Y., who said she adored baked shad but that her husband preferred salmon.

"We only have it twice a week while the shad is running," she said. "My husband won't let me serve it more than that. I clean it and bake it and serve it. It tastes like butter."
Less greedy

As you have already seen, regular expressions come in different flavors (POSIX v. extended, say) -- The Perl (the language we mentioned in our first lecture) had an amazingly powerful construction that R, for example, also implements, one innovation being the lazy quantifiers *? and +?

A Perl compliant regular expression engine on homework.stat202a.org is called pcregrep -- It implements Perl's regular expression

```
pcregrep "'\.*?'" /data/text_examples/1987/09/06/*
```
Resources

The presentation here is meant to give you a flavor of how regular expressions are structured; you have seen the major metacharacters and to use them to create patterns.

On our course home page we provide some good descriptions of regular expressions; in addition, the site

http://www.regular-expressions.info/

is an excellent resource.
Other resources

There are a few sites that offer translation services from text into regular expressions, allowing you to select which fields are to be generalized and which are specific (literal)

Regular expressions make a lot of hard choices (something matches or it doesn’t) -- Approximate or fuzzy regular expressions allow the engine to record a certain number of errors (inserts, deletes and substitutes)
Enter the string that you want to use a regular expression on:

1

[19/Aug/2010:00:02 -0700]

Show Matches

Select the elements that you want to extract to run regular expression generator:

What to click? extract any integer in this position
extract this integer (2006) in this position
TRE
The free and portable approximate regex matching library.

About

WP Greet Box icon Hello there! If you are new here, you might want to subscribe to the RSS feed for updates on TRE.

TRE is a lightweight, robust, and efficient POSIX compliant regex matching library with some exciting features such as approximate (fuzzy) matching.

The matching algorithm used in TRE uses linear worst-case time in the length of the text being searched, and quadratic worst-case time in the length of the used regular expression. In other words, the time complexity of the algorithm is $O(M^2N)$, where $M$ is the length of the regular expression and $N$ is the length of the text. The used space is also quadratic on the length of the regex, but does not depend on the searched string. This quadratic behaviour occurs only on pathological cases which are probably very rare in practice.

Features

TRE is not just yet another regexp matcher. TRE has some features which are not there in most free POSIX compatible implementations. Most of these features are not present in non-free implementations either, for that matter.
Learning Deterministic Regular Expressions for the Inference of Schemas from XML Data

Geert Jan Bex  Wouter Gelade*  Frank Neven  Stijn Vansummeren†

Hasselt University/Transnational University of Limburg, Belgium
firstname.lastname@uhasselt.be

ABSTRACT

Inferring an appropriate DTD or XML Schema Definition (XSD) for a given collection of XML documents essentially reduces to learning deterministic regular expressions from sets of positive example words. Unfortunately, there is no algorithm capable of learning the complete class of deterministic regular expressions from positive examples only, as we will show. The regular expressions occurring in practical DTDs and XSDs, however, are such that every alphabet symbol occurs only a small number of times. As such, in practice it suffices to learn the subclass of regular expressions in which each alphabet symbol occurs at most k times, for some small k. We refer to such expressions as k-occurrence regular expressions (k-OREs for short). Motivated by this observation, we provide a probabilistic algorithm that learns k-OREs for increasing values of k, and selects the one that best describes the sample based on a Minimum Description Length argument. The effectiveness of the method is empirically validated both on real world and synthetic data. Furthermore, the method is shown to be conservative over the simpler classes of expressions considered in previous work. The techniques we employ enable us to process large amounts of web data without relying on a pre-defined source ontology. In addition, Bex et al. [11, 38] noted that about two-thirds of XML Schema Definitions (XSDs) gathered from schema repositories and from the web at large are not valid with respect to the W3C XML Schema specification [50], rendering them essentially useless for immediate application. A similar observation was made by Sahuguet [47] concerning Document Type Definitions (DTDs). Nevertheless, the presence of a schema strongly facilitates optimization of XML processing (cf., e.g., [7, 18, 23, 28, 36, 37, 42]) and various software development tools such as Castor [1] and SUN’s JAXB [2] rely on schemas as well to perform object-relational mappings for persistence. Additionally, the existence of schemas is imperative when integrating (meta) data through schema matching [46] and in the area of generic model management [8]. Based on the above described benefits of schemas and their unavailability in practice, it is essential to devise algorithms that can infer a schema for a given collection of XML documents when no (valid) one is present. The latter problem is also acknowledged by Florescu [26] who emphasizes that in the context of data integration: “We need to extract good-quality schemas automatically from existing data and perform incremental...
Before we leave this

Below we present a (Python) regular expression for parsing our web access logs; notice the extensive use of \S (a character class of non-space characters) and the grouping construction () so that we can pull out each separate field

In a week or so you’ll be able to run this expression to not just match the lines in the log file, but to extract the data corresponding to the groups that are identified (the host, the IP address and so on)

Even with this more elaborate parsing method, some out of the millions of log lines do not exhibit this pattern...

\^\S* \S* \S* \S* ([\[\^\]\]+) \"([\^"\]\]*\.?\.[\^"\]\]*\*)\" \S* \S* \"([\^"\]\]*\.?\.[\^"\]\]*\*)\" \"([\^"\]\]*)"$
The moral

When we start programming in earnest, there’s a general principle you should follow; Saltzer and Kaashoek (Principles of Computer System Design) call it the robustness principle

Be tolerant of inputs and strict on outputs

In other words, even when you are given a file specification, things happen along the way; URLs are not properly encoded, extra quotation marks are added to the agent field and so on

Your job is to produce code that spots anomalies, documents them, and keeps going; the output of your processing should be clean and checked and any anomalies detected
Your homework
What's the traffic look like week after week?

cocsteau@homework:~$ wc /data/weblogs/acc*

1215118 24435535 306343201 /data/weblogs/access_log.1267056000
1202503 23919506 301260872 /data/weblogs/access_log.1267660800
1189816 23574071 295474587 /data/weblogs/access_log.1268265600
1092584 21397113 270137544 /data/weblogs/access_log.1268870400
1069245 21505434 265062438 /data/weblogs/access_log.1270080000
1192220 23845074 296812483 /data/weblogs/access_log.1270684800
1191896 23800538 297726668 /data/weblogs/access_log.1271289600
1186047 23514067 294013206 /data/weblogs/access_log.1271894400
1120881 22222966 282279881 /data/weblogs/access_log.1272499200
1082278 21319446 268196781 /data/weblogs/access_log.1273040000
1075591 20846190 267491156 /data/weblogs/access_log.1273708800
1065013 20576562 266100210 /data/weblogs/access_log.1274313600
1096354 20875482 269551843 /data/weblogs/access_log.1274918400
1070475 20893630 268196781 /data/weblogs/access_log.1275523200
1066577 21108954 274778895 /data/weblogs/access_log.1276128000
1118361 22475995 289652421 /data/weblogs/access_log.1276732800
1017283 20403691 263553810 /data/weblogs/access_log.1277337600
983582 19180820 250700362 /data/weblogs/access_log.1277942400
1014486 20234823 261073537 /data/weblogs/access_log.1278547200
1034419 20264332 263465553 /data/weblogs/access_log.1279152000
982086 19049910 246484043 /data/weblogs/access_log.1279756800
979824 18961241 246398653 /data/weblogs/access_log.1280361600
947963 19056256 241344051 /data/weblogs/access_log.1280966400
808150 16062070 199713370 /data/weblogs/access_log.1281571200
802234 15973793 199365412 /data/weblogs/access_log.1282176000
27641014 546404495 6942733164 total
Making sense of this

The columns of numbers are admittedly hard to look at -- Technically we haven’t been introduced to R yet, but with a couple lines of code you can make a simple line plot

On homework you can save the output of wc to a file:

    cocteau@homework:~$ wc /data/weblogs/acc* > totals.txt

And then use scp or WinSCP to bring the small file back to your desktop

    scp cocteau@homework.stat202a.org:totals.txt .

And then read it into R

    # leave off last row
    > totals <- read.table("totals.txt",nrows=26)

    # make a line plot
    > plot(totals[,1],type="l",xlab="weeks",ylab="counts")
Which hosts are most frequently visited?

About 25% of the traffic is for **www.stat.ucla.edu**, 8% to SOCR and 2% to the seminars pages, for example:

```bash
cocteau@homework:~$ cut -d" " -f1 /data/weblogs/access_log.12* | sort | \ 
  uniq -c | sort -rn | head -20
```

<table>
<thead>
<tr>
<th>Host</th>
<th>Visits</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.stat.ucla.edu">www.stat.ucla.edu</a></td>
<td>7190052</td>
</tr>
<tr>
<td>cran.stat.ucla.edu</td>
<td>6614014</td>
</tr>
<tr>
<td>limen.stat.ucla.edu</td>
<td>3189876</td>
</tr>
<tr>
<td>forums.stat.ucla.edu</td>
<td>2325370</td>
</tr>
<tr>
<td><a href="http://www.socr.ucla.edu">www.socr.ucla.edu</a></td>
<td>2102603</td>
</tr>
<tr>
<td>directory.stat.ucla.edu</td>
<td>1455642</td>
</tr>
<tr>
<td>theses.stat.ucla.edu</td>
<td>1010539</td>
</tr>
<tr>
<td>socr.ucla.edu</td>
<td>733915</td>
</tr>
<tr>
<td>seminars.stat.ucla.edu</td>
<td>450006</td>
</tr>
<tr>
<td>preprints.stat.ucla.edu</td>
<td>265563</td>
</tr>
<tr>
<td>civs.stat.ucla.edu</td>
<td>264405</td>
</tr>
<tr>
<td>socr.stat.ucla.edu</td>
<td>257182</td>
</tr>
<tr>
<td>news.stat.ucla.edu</td>
<td>224476</td>
</tr>
<tr>
<td>civs.ucla.edu</td>
<td>221163</td>
</tr>
<tr>
<td>inspire.stat.ucla.edu</td>
<td>219622</td>
</tr>
<tr>
<td>courses.stat.ucla.edu</td>
<td>155508</td>
</tr>
<tr>
<td>cts.stat.ucla.edu</td>
<td>133828</td>
</tr>
<tr>
<td>gallery.stat.ucla.edu</td>
<td>122356</td>
</tr>
<tr>
<td>scc.stat.ucla.edu</td>
<td>116969</td>
</tr>
<tr>
<td>sgsa.stat.ucla.edu</td>
<td>99528</td>
</tr>
</tbody>
</table>
Which pages are accessed?

Let's do this separately for each virtual host rather than by faculty members -- presumably the files, their types, their names, should exhibit hints that each host is performing a different function for the department.

Here's what we get for three of them...
egrep '^www\..stat' /data/weblogs/access_log.12* | cut -d" " -f8 | \
    sort | uniq -c | sort -rn | head -8
  
268238 /favicon.ico
126114 /
  55944 /graphics/GIF/paint2.gif
  51773 /robots.txt
  45490 /style_sheets/Global.css
  37823 /css/uclastat/site.css
  35718 /index.css
  33740 /rss/feed.php?unit=uclastat

egrep '^cran' /data/weblogs/access_log.12* | cut -d" " -f8 | \
    sort | uniq -c | sort -rn | head -8
  
1763991 /robots.txt
  93326 /bin/windows/base/R-2.10.1-win32.exe
  40737 /
  40068 /bin/windows/base/R-2.11.1-win32.exe
  34953 /bin/windows/base/R-2.11.0-win32.exe
  32513 /bin/linux/ubuntu/hardy/Packages.bz2
  32467 /bin/linux/ubuntu/jaunty/Release
  32455 /bin/linux/ubuntu/jaunty/Release.gpg

egrep '^seminars' /data/weblogs/access_log.12* | cut -d" " -f8 | \
    sort | uniq -c | sort -rn | head -8
  
14586 /robots.txt
12016 /rss
  6189 /upcoming_through/2010-07-29.xml
  5878 /
  4138 /upcoming_through/2010-07-07.xml
  3717 /stylesheets/site.css?1244118236
  3193 /upcoming_through/2010-08-17.xml
  2322 /archive
What do you notice?
What is this file robots.txt?

(How can we figure out what this file is about?) Let’s have a look at the IP addresses that touch this file -- There’s a good chance that humans won’t intentionally request it

We find that four IP addresses account for about 90% of the 2M downloads of this file -- What are these IP addresses? We might also want to check a few of the smaller hitters on the list

```
cut -d" " -f8 /data/weblogs/acc* | grep "robots\.txt" | wc
2140673 2140673 25735261

cut -d" " -f2,8 /data/weblogs/acc* | grep "robots\.txt" | \
   cut -d" " -f1 | sort | uniq -c | sort -rn | head -8
520843 164.67.132.222
479653 164.67.132.221
456150 164.67.132.220
437015 164.67.132.219
   8684 137.110.114.8
   7968 131.122.84.20
   6631 208.115.111.248
   4029 77.88.29.246

nslookup 164.67.132.219
Server: 172.16.0.23
Address: 172.16.0.23#53

Non-authoritative answer:
219.132.67.164.in-addr.arpa name = gsal.search.ucla.edu.
```
Another route to find robots

The User Agent field (again, how might we learn more about this?) is provided by the browser or other program making the request

http://www.useragentstring.com

This site offers a simple way to analyze what each string is all about -- Admittedly they are already pretty humanly readable

Let's look at the most popular User Agents...
<table>
<thead>
<tr>
<th>Request Count</th>
<th>User Agent Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>2811450</td>
<td>UCLA%20Google%20Search%20Appliance%20%2020%2028contact%3A%20jhuang%40ais.ucla.edu%29</td>
</tr>
<tr>
<td>2734675</td>
<td>UCLA%20Google%20Search%20Appliance%20%2020%2028contact%3A%20jhuang%40ais.ucla.edu%29</td>
</tr>
<tr>
<td>2208730</td>
<td>UCLA%20Google%20Search%20Appliance%20%2020%2028contact%3A%20jhuang%40ais.ucla.edu%29</td>
</tr>
<tr>
<td>2031090</td>
<td>UCLA%20Google%20Search%20Appliance%20%2020%2028contact%3A%20jhuang%40ais.ucla.edu%29</td>
</tr>
<tr>
<td>1565675</td>
<td>Apache/2.2.14 (Unix) mod_ssl/2.2.14 OpenSSL/0.9.71 PHP/5.2.12 Phusion_Passenger/2.2.11</td>
</tr>
<tr>
<td>1382949</td>
<td>Mozilla/5.0 (compatible; Yahoo! Slurp/3.0; <a href="http://help.yahoo.com/help/us/ysearch/slurp">http://help.yahoo.com/help/us/ysearch/slurp</a>)</td>
</tr>
<tr>
<td>724691</td>
<td>Apache/2.2.13 (Unix) mod_ssl/2.2.13 OpenSSL/0.9.71 PHP/5.2.11 Phusion_Passenger/2.0.5</td>
</tr>
<tr>
<td>627289</td>
<td>Apache/2.2.13 (Unix) mod_ssl/2.2.13 OpenSSL/0.9.71 PHP/5.2.11 Phusion_Passenger/2.2.11</td>
</tr>
<tr>
<td>435108</td>
<td>Mozilla/5.0 (compatible; Googlebot/2.1; +<a href="http://www.google.com/bot.html">http://www.google.com/bot.html</a>)</td>
</tr>
<tr>
<td>312016</td>
<td>Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0 256712</td>
</tr>
<tr>
<td>212259</td>
<td>Ubuntu APT-HTTP/1.3 (0.7.23.1ubuntu2)</td>
</tr>
<tr>
<td>205663</td>
<td>FAST Enterprise Crawler 6 / Scirus <a href="mailto:scirus-crawler@fast.no">scirus-crawler@fast.no</a>; <a href="http://www.scirus.com/srsapp">http://www.scirus.com/srsapp</a></td>
</tr>
<tr>
<td>188214</td>
<td>R (2.10.1 i386-pc-mingw32 i386 mingw32)</td>
</tr>
<tr>
<td>170748</td>
<td>NITRC/Nutch-0.9 (Neuroimaging Clearinghouse; <a href="http://www.nitrc.org">www.nitrc.org</a>; sysadmin at nitrc dot org)</td>
</tr>
<tr>
<td>150296</td>
<td>R (2.11.1 i386-pc-mingw32 i386 mingw32)</td>
</tr>
<tr>
<td>136059</td>
<td>Ubuntu APT-HTTP/1.3 (0.7.9ubuntu17.2)</td>
</tr>
<tr>
<td>123339</td>
<td>Mozilla/5.0 (compatible; DotBot/1.1; <a href="http://www.dotnetdotcom.org/">http://www.dotnetdotcom.org/</a>, crawler@dotnetdotcom</td>
</tr>
<tr>
<td>105364</td>
<td>Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.</td>
</tr>
<tr>
<td>94317</td>
<td>Mozilla/5.0 (Windows; U; Windows NT 6.1; en-US; rv:1.9.2.3) Gecko/20100401 Firefox/3.6.</td>
</tr>
<tr>
<td>83795</td>
<td>Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10_6_3; en-us) AppleWebKit/531.22.7 (KHTML, 1</td>
</tr>
<tr>
<td>79333</td>
<td>R (2.11.0 i386-pc-mingw32 i386 mingw32)</td>
</tr>
</tbody>
</table>
User Agent String explained:

Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)

Copy/paste any user agent string in this field and click 'Analyze'

### Internet Explorer 6.0

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mozilla</td>
<td>MozillaProductToken. Claims to be a Mozilla based user agent, which is only true for Gecko browsers like Firefox and Netscape. For all other user agents it means 'Mozilla-compatible'. In modern browsers, this is only used for historical reasons. It has no real meaning anymore.</td>
</tr>
<tr>
<td>4.0</td>
<td>Mozilla Version</td>
</tr>
<tr>
<td>compatible</td>
<td>Compatibility flag. Indicates that this browser is compatible with a common set of features.</td>
</tr>
<tr>
<td>MSIE 6.0</td>
<td>Name: Internet Explorer Version = 6.0</td>
</tr>
<tr>
<td>Windows NT 5.1</td>
<td>Operating System: Windows XP</td>
</tr>
<tr>
<td>SV1</td>
<td>Windows XP Service Pack 2 installed (Security Version 1)</td>
</tr>
<tr>
<td>.NET CLR 1.1.4322</td>
<td>.NET framework Version : 1.1.4322</td>
</tr>
<tr>
<td>.NET CLR 2.0.50727</td>
<td>.NET framework Version : 2.0.50727</td>
</tr>
</tbody>
</table>

All Internet Explorer user agent strings
User Agents

The first four are responsible for an incredible number of hits -- Any guess as to what IP addresses they might be associated with? How could we verify it?

The Phusion_Passenger is a module in our web server -- In this case, Jose (our system administrator) said that it is tracking the behavior of the server itself, with each call indicating that the server as added or decreased processes to deal with traffic (and, as such, most of the “requests” are logged as coming from the server itself)

```
fgrep 'Phusion_Passenger' /data/weblogs/acc* | cut -d" " -f2 | \
  sort | uniq -c | sort -rn | more
```

```
2917655 128.97.86.247
  14 128.97.60.148
  1 208.68.168.130
  1 203.186.158.150
  1 122.160.118.150
```
Downloading web pages programmatically

curl is a common Unix utility for transferring data to or from a server using HTTP, FTP, SCP and others -- R and Python also both have considerable support for performing these kinds of operations

From here, it’s not too surprising that crawlers of the kind Prof. Cho talked about could be built (or, rather, that you could build one!) -- We’ll do a lot of this in the next week or two

cocteau@homework:$ curl www.stat.ucla.edu
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="stylesheet" href="index.css"/>
<link rel="stylesheet" href="/css/uclastat/site.css"/>
<!--<link rel="alternate" type="application/rss+xml" title="RSS" href="http://www.stat.ucla.edu/rss/feed.php?unit=uclastat"/>-->
<style type="text/css">
#article0 {
    display: none;
    width: 100%;
}

#article1 {
    display: none;
    width: 100%;
}
...

Active v. slow times

We can carve up the data field to have a look at aggregate statistics, compiled across days or across hours -- For simplicity, let’s consider March

On homework you can save the output of wc to a file:

```
cut -d'[' -f2 /data/weblogs/acc* | egrep "[0-9]{2}/Mar/2010:" | \cut -d"]" -f1 | sort | uniq -c > day_totals.txt
```

And then use scp or WinSCP to bring the small file back to your desktop

```
scp cocteau@homework.stat202a.org:day_totals.txt .
```

And then read it into R

```r
# leave off last row
> totals <- read.table("day_totals.txt")

# make a line plot
> plot(totals[,2],totals[,1],type="l",xlab="days",ylab="counts")
> abline(v=(1:4)*7)
> abline(v=(1:4)*7-1)
```
cut -d'[' -f2 /data/weblogs/acc* | egrep "[0-9]{2}/Mar/2010:" | \
  cut -d"/" -f1| sort | uniq -c

181597 01
181734 02
223149 03
191299 04
171733 05
141658 06
158908 07
173449 08
176150 09
189185 10
185379 11
155660 12
148710 13
146840 14
177637 15
184991 16
176410 17
172542 18
148341 19
139422 20
135439 21
175869 22
158208 23
176411 24
161611 25
139534 26
117578 27
129174 28
165012 29
154718 30
168399 31
cut -d'[' -f2 /data/weblogs/acc* | egrep "[0-9]{2}/Mar/2010:" | \
    cut -d"":" -f 2| sort | uniq -c

298126 00
223402 01
199550 02
210029 03
226590 04
290241 05
250589 06
151206 07
157035 08
161616 09
165667 10
161113 11
164812 12
161510 13
146303 14
154057 15
137324 16
140647 17
139273 18
281146 19
272867 20
333271 21
292793 22
387580 23
counts by hour, march 2010 (complete traffic)
What happened here?
fgrep -v '164.67.132.' /data/weblogs/acc* | fgrep -v '128.97.86.247' | \
    cut -d '[' -f 2 | egrep "[0-9]{2}/Mar/2010:" | cut -d":" -f 2 | sort | uniq -c

101870  00
  93985  01
  91080  02
  98977  03
  90244  04
101300  05
108240  06
124375  07
130592  08
134758  09
138761 10
134004 11
138266 12
134696 13
121188 14
130121 15
113784 16
117310 17
115402 18
135517 19
129415 20
129079 21
114548 22
110392 23
counts by hour, march 2010 (5 ip's removed)
Other results?

What is the instantaneous load on the server?

How many different IP addresses (proxy for people) were accessing the site in a single hour? A single minute?

What are the sources of errors? Who encounters them, visitors or robots?

Do accesses come from academic institutions? Can we see people browsing our site before they apply? What do they look at?
Does anyone really do this?

During my sabbatical, I worked with the NY Times and their web access logs (technically, there are three different kinds of data collected by the Times -- The server logs, a page tagging scheme initiated by the Times, and a second tagger used by Webtrends)

In our case, access logs were paired with data we received from bit.ly, the URL shortening service -- They record information about each “encode” (shortening) and “decode” (expansion) event

From here, we used time, location (physical) and the shortened URL to connect bit.ly encodes and the appearance of tweets on Twitter -- Similarly, decodes can be tracked from bit.ly into the NY Times site

The goal is to define a set of cascades, trees of activity -- Someone reading the NY Times, posting a link can then have that link retweeted or decoded, leading to more browsing events that might, in turn, lead to new tweets and so on

In this case, our data were stored in MongoDB -- It provides somewhat amazing performance when dealing with large stores of data, and is something we’ll seen in a couple weeks in this class
Charting the Conversation

- Traditional methods of web server log analysis provide little insight into discovery methods, social referrals, or recommendations.

[Diagram showing data flow from `http_referrer` to `outbound link`]

- Adding in data from Twitter / Backtweets and from bit.ly provide additional "connection points" that can be used to plot browsing paths.

[Diagram showing data flow from `browse` to `decode` to `tweet` to `encode` to `browse`]
The week ahead

For your next assignment, you are going to consider how to extract features from the recipes we have collected from the NY Times -- Right now you have one year, but we will probably give you 1982-2000

For example, let’s consider pulling a list of ingredients and the instructions, recoding the data into a more “usable” form -- Have a look at the recipes and start to come up with strategies for pulling out this information

Next week we will also start Python, so you don’t have to consider this exercise as one of simply pulling interesting lines -- You can think about how the lines relate to each other and (perhaps) the markup in the files
Alternative strategies

One useful approach might be to consider this more of a learning problem, for which we need “tagged” data to create some kind of ingredient or instruction classifier

Where might that data come from?
Mechanical Turk is a marketplace for work.
We give businesses and developers access to an on-demand, scalable workforce.
Workers select from thousands of tasks and work whenever it’s convenient.

74,348 HITs available. View them now.

Make Money by working on HITs

HITs - Human Intelligence Tasks - are individual tasks that you work on. Find HITs now.

As a Mechanical Turk Worker you:
- Can work from home
- Choose your own work hours
- Get paid for doing good work

Find an interesting task Work Earn money

Get Results from Mechanical Turk Workers

Ask workers to complete HITs - Human Intelligence Tasks - and get results using Mechanical Turk. Register Now

As a Mechanical Turk Requester you:
- Have access to a global, on-demand, 24 x 7 workforce
- Get thousands of HITs completed in minutes
- Pay only when you’re satisfied with the results

Fund your account Load your tasks Get results

Get Started
### All HITs

1-10 of 2050 Results

<table>
<thead>
<tr>
<th>Requester:</th>
<th>HIT Expiration Date:</th>
<th>Reward:</th>
<th>Time Allotted:</th>
<th>HITs Available:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Web Search Quality (t1qa1).</strong> (WARNING: This HIT may contain adult or offensive content. Worker discretion is advised.)</td>
<td>Oct 8, 2010 (2 days 11 hours)</td>
<td>$0.03</td>
<td>5 minutes</td>
<td>18371</td>
</tr>
<tr>
<td><strong>Find a product page URL</strong></td>
<td>Oct 8, 2010 (1 day 19 hours)</td>
<td>$0.05</td>
<td>5 minutes</td>
<td>7578</td>
</tr>
<tr>
<td><strong>Find Restaurant Web Addresses</strong></td>
<td>Oct 13, 2010 (6 days 20 hours)</td>
<td>$0.07</td>
<td>60 minutes</td>
<td>5247</td>
</tr>
<tr>
<td><strong>Web Search Quality (t-cou-q1a).</strong> (WARNING: This HIT may contain adult or offensive content. Worker discretion is advised.)</td>
<td>Oct 9, 2010 (2 days 21 hours)</td>
<td>$0.03</td>
<td>5 minutes</td>
<td>3094</td>
</tr>
<tr>
<td><strong>Answer Questions in Your Own Words</strong></td>
<td>Oct 13, 2010 (6 days 21 hours)</td>
<td>$0.25</td>
<td>60 minutes</td>
<td>2598</td>
</tr>
</tbody>
</table>
In this task you’ll try to find restaurants’ official websites. You will be provided the name, address, and phone number of a restaurant. You will use our Google link to search for the official front page of the restaurant, if it exists.

- When you click on the provided google search look through the whole first page of search results.

- Ignore “directory” websites (websites with information for many businesses) like “RestaurantsInChicago” or “NYpages.” Also ignore MySpace and Facebook.

- Restaurants in different cities can have the same name, so we need to check to make sure the website belongs to the provided restaurant. All three of the following must be true:
  1. The website is food-related,
  2. The website restaurant has a similar name to the provided name
  3. The website shows a location in the provided city or with the provided phone number.

- If the site is the official site, enter the front page of the restaurant (NOT a ‘locations’ or ‘about’ page).
Sila Turkish Restaurant

Phone number: 609-835-7300
City: Beverly
State: NJ
Zip Code (Postal Code): 08010

Please find the correct website with the below search.

Click here to open Google search in new window

☐ I'm getting an error message from Google, so I can't search on Google anymore.
Toy example

During my sabbatical, we experimented using the Turk to tag data related to the length of the line at a popular burger joint on 8th avenue (visible from our office window) -- We collected a series of images and needed to train a model to estimate the number of people in line

So we set up a camera and took a picture every x seconds -- The “ground truth” for this exercise came from a HIT...
|   | U       | V       | W       | X       | Y       | Z       | AA      | AB      | AC      | AD      | AE      | AF      | AG      | Approve | Reject |
|---|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|---------|
| 1 | AutoApproval | ApprovalTime | RejectionTime | RequesterFee | WorkTimeInSite | Input.example | Input.image | Input.image | Input.image | Input.image | AnswerComm | AnswerCount | Approve | Reject |
A possibility?

If this problem excites you, we might consider generating tagged data for the recipes using the Mechanical Turk -- We’d have to come up with an appropriate HIT, architect how the data are presented and what form it will be returned in

Keep this in mind as you think about the problem...