PURDUE CS24200

INTRODUCTION TO DATA SCIENCE

DATA WRANGLING (CONT)

AGGREGATING DATA

PANDAS: GROUPBY

```
# drop +/- from grades
tmp = data2.Grade
tmp2 = tmp.str.replace('-','')
tmp3 = tmp2.str.replace('+','')
data2.Grade = tmp3

# find unique values
data2.Grade.unique()

# group by final grade and calculate avg Test1 score
for grade, grade_data in data2.groupby("Grade"):
    print(grade, grade_data.Test1.mean())
A 46.666666666666664
B 44.6
C 40.5
D 41.666666666666664
F 45.0
```

When you iterate over the results of a groupBy, each result is a tuple:
first element is a unique value,
second element is a DataFrame
filtered by that value

PANDAS: GROUPBY

```
# can also aggregate directly over group object
groups = data2.groupby("Grade")
groups.agg('sum')
```

Grade	Test1	Test2	Test3	Test4	Final
A	140.0	170.769231	275.0	273.0	225.0
B	223.0	116.000000	245.0	253.0	268.0
C	162.0	211.000000	183.0	196.0	175.0
D	125.0	233.384615	276.0	240.0	137.0
F	45.0	11.000000	NaN	4.0	43.0

```
# applying multiple aggregators
groups.Final.agg(['min', 'max'])
```

Grade	min	max
A	45.0	97.0
B	4.0	90.0
C	40.0	46.0
D	40.0	49.0
F	43.0	43.0

PANDAS: GROUPBY – MULTIPLE COLUMNS

```
rndGrade = lambda x: int(x / 10) * 10
data2.Test1.map(rndGrade)
data2['Rounded'] = data2.Test1.map(rndGrade)
groups = data2.groupby(['Rounded','Grade'])
groups.agg('mean')
```

		Test1	Test2	Test3	Test4	Final
Rounded	**Grade**					
30	**C**	30.000000	16.0	20.000000	30.000000	40.000000
40	**A**	46.666667	78.0	91.666667	91.000000	75.000000
	B	43.250000	26.5	38.750000	57.666667	44.500000
	C	44.000000	65.0	54.333333	55.333333	45.000000
	D	41.666667	93.5	92.000000	80.000000	45.666667
	F	45.000000	11.0	NaN	4.000000	43.000000
50	**B**	50.000000	10.0	90.000000	80.000000	90.000000

TRANSFORMING DATA

LAMBDA FUNCTIONS

- Lambda functions are functions without names, for use in situations where the function will be discarded and not used again
- Example:
  `lambda x: x>0`
- The term lambda makes the temporary function, x is the parameter name, and the code after : denotes what to do
- Often used in `apply( )` when transforming data

PANDAS: MAP & APPLY

```
# create a rounding function, apply to each value in Test1
rndGrade = lambda x: int(x / 10) * 10
data2.Test1.map(rndGrade)
```

```
0     40.0
1     41.0
2     41.0
3     42.0
4     43.0
5     44.0
6     45.0
7     46.0
8     49.0
9     48.0
10    44.0
11    47.0
12    45.0
13    50.0
14    40.0
15    30.0
Name: Test1, dtype: float64
```

```
0     40
1     40
2     40
3     40
4     40
5     40
6     40
7     40
8     40
9     40
10    40
11    40
12    40
13    50
14    40
15    30
Name: Test1, dtype: int64
```

```
# apply function to more than one column of data frame
data2[['Test1','Final']].apply('sum')
```

```
Test1    695.0
Final    848.0
dtype: float64
```

```
# standardize Final grade by subtracting mean and divding by stdev
avgFinal = data2.Final.mean()
stdFinal = data2.Final.std()
print(avgFinal,stdFinal)
data2.Final.map(lambda x: (x - avgFinal)/stdFinal)
```

```
0    -0.173596
1    -0.216995
2    -0.390591
3    -0.260394
4    -0.347192
5    -0.303793
6    -0.433990
7    -0.130197
8     1.301971
9     1.909557
10   -0.564187
11   -0.347192
12    1.041577
13    1.605764
14   -2.126552
15   -0.564187
Name: Final, dtype: float64
```

MISCELLANEOUS

```
# getting data out of pandas, output to csv
data2.to_csv('grades_mod.csv', sep='\t')
```

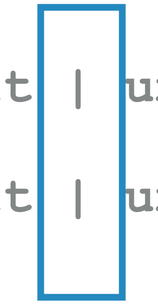
DATA MUNGING OUTSIDE OF PYTHON/PANDAS

COMMAND LINE PROCESSING AND SHELL SCRIPTING

- Some basic data science tasks are easily completed with command line functions/tools
- Note: that there are different types of shell (bash, csh, ksh, zsh, …)
  bash shell is most common (default shell on OS X and major linux distributions)
- Shell scripts allow us to program commands in chains and have the system execute them as a scripted event

USEFUL COMMANDS

```
wc -l                       # counts number of lines in a file

wc -w                       # counts number of words in a file

head -100 file1.dat         # output first 100 lines of file

tail file1.dat              # output last ten lines of file

sort file1.dat              # sort the contents of a file (default is alphabetic)

cat file1.dat file2.dat >newfile.dat     # concatenates input into new file

sort file1.dat | uniq       # sort the file and remove any duplicate lines

sort file1.dat | uniq -c    # remove duplicates, output counts for each occur.
```



Piping redirects output of previous command to input of next command

UNIX TOOLS

- Unix tools excel at manipulating strings as data, often using regular expressions

- **grep**: filters its input against a pattern
- **sed**: applies transformation rules to each line
- **awk**: manipulates tabular text files (e.g., CSV)

GREP

- **grep** searches plain text files for lines that match a pattern
- The command below prints each line of file which contains a match for pattern
  `grep pattern file`
- If you use `grep -v` instead, it prints each line of the file which does NOT contain a match for the pattern
- By default, grep uses regular expressions, e.g. the following finds all processes by user bgstm
  `ps aux | grep "^bgstm"`

GREP EXAMPLES

```
grep "this" *.txt
grep ^root /etc/passwd
grep "http.*html" index.html

grep -i "linux" index.html          # case insensitive

grep -A <N> "string" FILENAME    #print N lines after (-A) or before (-B) match

grep -c "pattern" filename       #count number of lines that match
```

http://www.grymoire.com/Unix/Grep.html

SED

- **sed** is a stream editor that can perform basic transformations on an input stream
- It reads line-by-line, conditionally applying a sequence of operations to each line and outputting the result
- By default, sed uses regular expression syntax
- Most sed programs consist a single sed command: substitute. E.g., to substitute commas with tabs, use:

```
$ sed 's/,/\t/g' <in  >out
```

SED EXAMPLES

```
sed 's/test/another test' filename
sed 's/test//g' filename

sed '/regexp/d' filename          # d=delete

sed '/^$/d' filename

sed -n `/Jones/p' filename           # p=print

sed G filename                    # double space a file
```

http://www.grymoire.com/Unix/Sed.html

AWK

- **awk** provides a more traditional programming language for text processing than sed
- The major difference between awk and sed is that awk is record-oriented rather than line-oriented
- Each line of the input to awk is treated like a delimited record. The command line parameter -F sets the field delimiter
- The essential organization of an awk program has the form:
  pattern { action }

AWK

- The basic form is:
  ```
  $ awk options program file
  ```
- Example that prints first field of each record:
  ```
  awk '{print $1}' myfile
  ```
- $0 for the whole line; $1 for the first field, …, $n for the nth field
- Can combine grep, sed, and awk with pipe on the command line
- Examples:
  ```
  ps aux | grep "^bgstm" | awk '{print $2 +  $3}'
  ```

AWK EXAMPLES

```
# add consecutive integer for each line in a csv file
awk '{print NR "," $0}' filename      #NR=number of records

awk -F "," '{print NF}' filename      #NF=number of fields

awk -F ":" '{ print "username: " $1, "uid:" $3 }' /etc/passwd
```

http://www.grymoire.com/Unix/Awk.html

TF-IDF – AT THE HIGH LEVEL

MOST COMMON WORDS IN THE ENGLISH LANGUAGE

the	it	this	or	so	me	person	than	back	even
be	for	but	will	up	make	into	then	after	new
to	not	his	an	out	can	year	now	use	want
of	on	by	my	if	like	your	look	two	because
and	with	from	one	about	time	good	only	how	any
a	he	they	all	who	no	some	come	our	these
in	as	we	would	get	just	could	its	work	give
that	you	say	there	which	him	them	over	first	day
have	do	her	their	go	know	see	think	well	most
I	at	she	what	when	take	other	also	way	us

MOST COMMON WORDS IN THE ENGLISH LANGUAGE

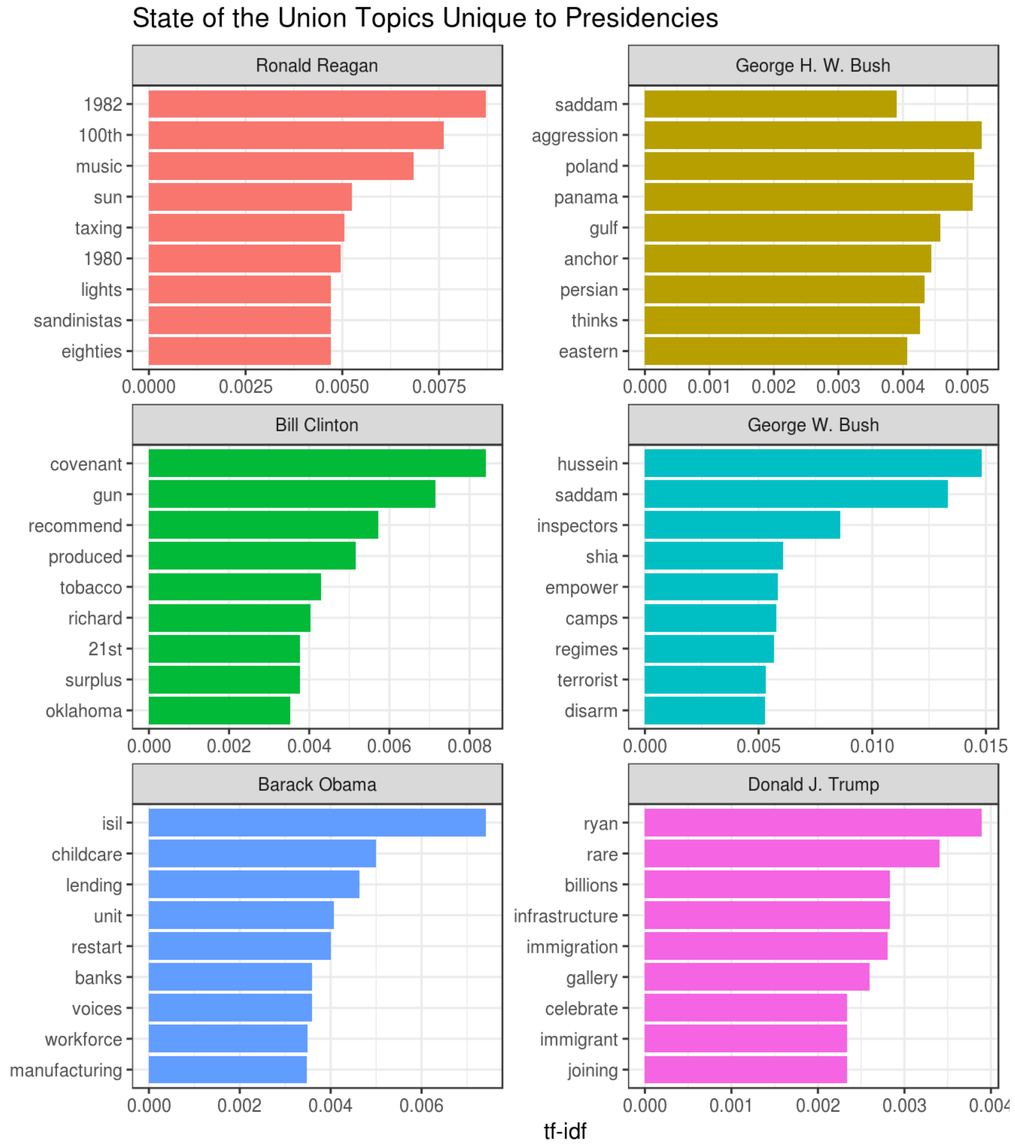
Nouns	time	part	**Adjectives**	good	right	**Verbs**	be	come
	person	child		first	big		have	think
	year	eye		new	high		do	look
	way	woman		last	different		say	want
	day	place		long	small		get	give
	thing	work		great	large		make	use
	man	week		little	next		go	find
	world	case		own	early		know	tell
	life	point		other	young		take	ask
	hand	company		old	important		see	work

TFIDF

- A simple ‘bag of words’ model of documents
- **TF:** Term Frequency
  for each term in each document, the number of times the word occurs divided by the number of the words in the document
- **IDF:** Inverse Document Frequency
  for each term across the corpus: $\ln \frac{N}{1 + n_t}$, where N is the number of documents, and
  n_t is the number of documents that contain the term
- **TFIDF:** Multiply TF by IDF to get a sense of words more unique in a document

TFIDF DATASET

- Corpus should be similar to documents you are analyzing
  - Classic Literature
  - AP News stories
  - Enron Emails
  - Webpages in Wikipedia
  - State of the Union Addresses



https://diving4data.com/projects/state-of-the-union-analysis/