Lecture Notes: AI at a Webscale

Dr. Maurits Kaptein

May 19, 2015

Contents

Chapter 1

Lecture 1: Introduction

1.1 Structure of the course / admin

Welcome to the course "AI at a web scale". In this course we will try to address AI and machine learning challenges on the web. The course consists of four main parts:

1. *Classic online algorithms:* In the first part of the course we will cover "classic" algorithms that currently drive the web. We will cover basic search, spam filtering, collaborative filtering, etc.
2. *Technical necessities:* In the second part of the course we will cover a number of technical details regarding web technologies. We will cover data structures, databases, and (briefly) Map/Reduce.
3. *Modern challenges:* In this part of the course we will cover more modern ideas regarding AI on the web. We will cover (contextual) bandit problems and streaming (or online) analysis
4. *Practicals:* In the second half of the course we will try to address a number of practical, and industry relevant, problems.

The aim of the course is to get you up to speed with the developments on the web, and inspire you to develop novel (AI) solutions for web-related problems.

For this course some background in basic probability theory and statistics is assumed, as well as exposure to programming. We will try to use Python as our programming language of choice throughout, so make sure you read up on Python basics. If you bump into troubles with background / prerequisites please let me know and we will try to find a solution.

1.1.1 Overview of the course, planning

The following table gives a brief overview of the planned meetings in this course and the topics I intend to address.

Date	Nr	Type	Topic
03/02/15	1	HC	Introduction
10/02/15	2	HC	Spam filtering
10/02/15	1	WC	
		No lectures	
24/02/15	3	HC	Collaborative Filtering
24/02/15	2	WC	
03/03/15	4	HC	Networks and network statistics
03/03/15	3	WC	
10/03/15	5	HC	Search
10/03/15	4	WC	
17/03/15	6	HC	Hierarchical data structures
17/03/15	5	WC	
		No lectures	
31/03/15	7	HC	SQL and noSQL databases
31/03/15	6	WC	
07/04/15	8	HC	Map Reduce & Rest API's
07/04/15	7	WC	
14/04/15		**Midterm exam**	
21/04/15	9	HC	Streaming analysis
28/04/15	10	HC	Stochastic Gradient Descent
		No lectures	
12/05/15	11	HC	Bandit problems
19/05/15	12	HC	Contextual bandits
26/05/15	13	HC	Questions and Answers
27/05/15	1	Practical	Streaming contextual bandit challenge
03/06/15	2	Practical	
10/06/15	3	Practical	
16/06/15		**Final Exam**	
17/06/15	4	Practical	
24/06/15	5	Practical	
01/07/15	6	Practical	Team presentations

1.1.2 Lecture types

The course consists of three lecture types: Lectures, Tutorials, and Practicals. During the lectures I will talk, and you will primarily listen and ask questions: these lectures are more theoretical. During the tutorials you will be asked to (individually) work on assignments to implement the things we talked about during the lectures. Finally, in the practicals you will work (in groups) on real-live challenges, and in here you are free to implement everything you have learned in a way you think fits.

1.1.3 Grade

The grade for this course is composed of several parts:

- *Midterm exam:* In April we will have a midterm exam, this is a written exam during the regular time of the lecture. This will count for 25% towards your final grade.
- *Exam:* Half June there is the final written exam (also during the lecture time). This will count for 50% of the final grade and will cover all topics up to that point in the course.
- *Practicals:* You will have to present, in the final practical session, your (group) achievements in the practical sessions. These presentations will be graded and will count for 25% towards your final grade.

There will be a resit for the midterm and final exam (in one go). There is no opportunity to resit the practicals (outside of taking them again next year). The midterm and exam grade will carry over one year, as will the practical point. After a year the only way to retake the course is by doing all anew.

1.1.4 Contact

If you have any questions regarding the course, its structure, the grades, etc. please do not hesitate to ask them. Try saving up your questions for the lectures so we can deal with them in front of the whole group. If things are urgent, then send me an email: (m.kaptein [at] donders.ru.nl).

1.2 Probability review

Some basic probability notation and facts you should know regarding probability:

Probability of event A (definition):

$$P(A) \ : \ P(A) \geq 0, \ \sum_{A} P(A) = 1$$

Joint probability of events A and B =

$$P(A, B)$$

Conditional probability of A given B =

$$P(A|B) = \frac{P(A, B)}{P(B)}$$

Product rule:

$$P(A, B) = P(A \mid B)P(B) = P(B \mid A)P(A)$$

Chain rule (example for 4 events):

$$P(A_4, A_3, A_2, A_1) = P(A_4|A_3, A_2, A_1)P(A_3|A_2, A_1)P(A_2|A_1)P(A_1)$$

Marginal probability of A given all possible values of B =

$$P(A) = \sum_{B} P(A, B)$$

(which logically becomes an integral if B is continuous).

Independence of A and B:

$$P(A, B) = P(A)P(B)$$

(remember: independence means multiply)

Bayes' Rule:

$$P(B|A) = \frac{P(A|B)P(B)}{P(A)} = \frac{P(A|B)P(B)}{\sum_B P(A \mid B)P(B)}$$

Combinations:

$$\left(\begin{array}{c} n \\ r \end{array} \right) = \frac{n!}{(n-r)!r!}$$

1.2.1 Distributions (small selection)

A random variable is a function that maps events to the real number line. A probability distribution assigns a probability to each of the possible outcomes of a random variable.

Discrete distributions

The probability mass function (PMF) of a *bernoulli* distribution with parameter p is

$$P(X = x) = p^x(1-p)^{1-x} \tag{1.1}$$

where $x \in \{0, 1\}$.

The probability mass function (PMF) of a *binomial* distribution with parameters p and n is

$$\begin{aligned} P(X = x) &= \left(\begin{array}{c} n \\ x \end{array} \right) p^x q^{n-x} \\ &= \frac{n!}{x!(n-x)!} p^x (1-p)^{n-x} \end{aligned} \tag{1.2}$$

where x is the number of "successes".

The binomial distribution is the expected distribution of outcomes in random samples of size n, with probability p of success. Mean and variance of binomial distribution:

$$\begin{aligned} \mu &= np \\ \sigma &= \sqrt{npq} \\ \sigma^2 &= npq \end{aligned}$$

Or, more generally: the expected value of $\mathbb{E}(X) = \sum xP(x)$ (or $\mathbb{E}(X) = \int xf(x)dx$) and $VAR(X) = \mathbb{E}[(X - \mathbb{E}(X))^2]$

The *Poisson* distribution can be used to approximate the Binomial distribution when one event is rare ($p < 0.1$), and the sample size is large ($np > 5$).

A Poisson variable Y must be

1. **Rare:** Small mean relative to the number of possible events per sample
2. **Random:** Independent of previous occurrences in the sample

This distribution can model the number of times that a rare event occurs, and test whether rare events are independent of each other. The parameter λ is the expected number of

successes. If X is binomial with large n and small p, the number of success is approximately a Poisson random variable with $\lambda = np$.

The probability mass function of a Poisson distribution is

$$P(X = x) = e^{-\lambda}\frac{\lambda^x}{x!}. \tag{1.3}$$

λ is the only parameter needed to describe a Poisson distribution. It is equal to both the variance and the mean:

$$\lambda = \mu = \sigma^2. \tag{1.4}$$

Continuous distributions

The probability density function (PDF) for a *uniform* random variable X on interval (α, β) is

$$f(X) = \begin{cases} \dfrac{1}{\beta - \alpha} & if \ \ \alpha < x < \beta \\ 0 & otherwise \end{cases} \tag{1.5}$$

and the cumulative density function (CDF) is $\frac{x-\alpha}{\beta-\alpha}$ for $x \in [\alpha, \beta]$.

The mean and variance of X:

$$\begin{aligned} \mu &= \frac{\beta + \alpha}{2} \\ \sigma^2 &= \frac{(\beta = \alpha)^2}{12} \end{aligned}$$

The *normal* probability density function is

$$f(X) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(X-\mu)^2/2\sigma^2} \tag{1.6}$$

with parameters μ and σ^2 (which are also the mean and variance of the distribution). The curve is symmetric about the mean.

$\mu \pm \sigma$ contains 68.3% of the items
$\mu \pm 2\sigma$ contains 95.5% of the items
$\mu \pm 3\sigma$ contains 99.7% of the items

We will encounter different distributions. You should know what PMF's, PDF's, and CDF's are.

1.2.2 Central Limit Theorem

The Central Limit Theorem states that as sample size increases, the means of samples drawn from a population having any distribution will approach the normal distribution.

As the number of samples increases, the Central Limit Theorem states that the:

- distribution of sample means will approximate normal, regardless of the original distribution
- mean value of sample means will equal the population mean
- standard deviation of the sample means (standard error of the mean) depends on population standard deviation and sample size.

There are many central limit theorems at various levels of abstraction. Central limit theorems are still an active area of research in probability theory.

1.2.3 Maximum Likelihood: The basics

Given observations $x_1, \ldots, x_n$ we often want to estimate the parameters of the data-generating distribution. A very common approach is maximum likelihood estimation: MLE.

The basic idea is simple. Suppose our data are presumed to be I.I.D. (independent and identically distributed) R.V.'s from a Bernoulli(p). We consider the likelihood function, which is the density function regarded as a function of its parameters:

$$L(\theta|\vec{x}) = L(p|x_1, \ldots, x_n) = \prod_{i=1}^{n} p^{x_i}(1-p)^{1-x_i} \tag{1.7}$$

(remember: independence means multiply).

We now look for the value of p that maximizes the above function. Obviously we set the derivative to zero. However, this is tricky due to the product term. Thus, it is a custom to work with the log likelihood function:

$$\begin{aligned} l(p|x_1, \ldots, x_n) &= \ln(L(p|x_1, \ldots, x_n)) && (1.8)\\ &= \ln p(\sum x_i) + \ln(1-p)(n - \sum x_i) && (1.9)\\ &= n\bar{x} \ln p + n(1-\bar{x}) \ln(1-p) && (1.10) \end{aligned}$$

where $\bar{x}$ is the sample mean.

Computing the derivative gives:

$$\frac{\delta}{\delta p} l(p|\vec{x}) = n\left(\frac{\bar{x}}{p} - \frac{1-\bar{x}}{1-p}\right) \tag{1.11}$$

which is zero when $p = \bar{x}$ indicating that the sample mean is the MLE of the parameter of the Bernoulli. Thus $\hat{\theta}(\vec{x}) = \bar{x}$.

We will encounter MLE estimation (and other types of parameter estimation) throughout the course. However, you should at least follow the structure of the above argument. Note that this is *not* an algebra course.

For a thorough introduction to probability theory please make sure to buy and read (Williams and Williams, 2001).

1.3 Basic Python

Here are some very basic things about Python you should know (but really, you should know quite bit more...). A few examples of Python code which do exactly what you would expect:

```
print('Hello world.')

print('Hello world.')
print(4 + 5)
```

Data **types** are the building blocks from which everything else is built. In Python, some core data types are:

- Simple Types: numbers and strings
  - numbers: 3, 12.443, 89, ...
  - strings: "hello", 'manny', "34", ...
- Complex Types: lists and dictionaries (& sets & tuples)
  - lists: [1,2,3], [1,2,"a"], ["john", "george", "paul", "ringo"], ...
  - dictionaries: {"a":1, "b":16}, ...

All of these can be stored into variables. Python is **dynamically typed**: you do not have to declare what type each variable is. It is good practice, however, to not switch between types for a given variable

Python can work just like a calculator:

```
>>> 2+2
4
>>> 3/2
1
>>> 3/2.
1.5
```

```
x = 2 + 2   # x = 4
x = 3/2     # x = 1
x = 3/2.    # x = 1.5
```

Python has integers and floating point numbers (& complex numbers), and operations to convert between them:

```
>>> float(3)
3.0
>>> int(4.123)
4
```

```
x = 3
y = 4.123
x1 = float(x)  # x1 = 3.0
y1 = int(y)    # y1 = 4
```

You can print floating point numbers with different levels of precision, but we won't cover that here.

What is a variable? A variable is a name that refers to some value (could be a number, a string, a list etc.)

1. Store the value 42 in a variable named *foo*
   `foo = 42`
2. Store the value of foo+10 in a variable named bar
   `bar = foo + 10`

What is the difference between an expression and a statement? An expression *is* something, and a statement *does* something.

Ask the user to input a name, and store it in the variable:

```
name = raw_input('enter your name:  ')
greet = 'hello ' + name
```

1. Ask the user to input a number, and store it in the variable *foo*
   `foo = int(raw_input('enter a number:  '))`
2. Add foo and bar together
   `foo + bar`
3. Calculate the average of *foo* and *bar*, and save it in a variable named *avg*
   `avg = (foo + bar)/2`

What is a function? A function is a mini-program. It can take several *arguments*, an *returns* a value.

What is a module? Python is easily *extensible*. Users can easily write programs that extend the basic functionality, and these programs can be used by other programs, by loading them as a *module*:

1. load the math module
   `import math`
2. Round 35.4 to the nearest integer
   `math.round(35.4)`
3. Round the quotient of foo and bar down to the nearest integer
   `math.floor(foo/bar)`

Saving and executing programs: Save the following in file "hello.py":

```
# this script prints 'hello, world', to stdout
print("hello, world")
```

And then run the program:
`python hello.py`

String Basics:

- Strings must be enclosed in quotes (double or single)
- Strings can be concatenated using the + operator
- Many ways to write a string:
  - single quotes: `'string'`
  - double quotes: `"string"`
  - can also use `"""` to write strings over multiple lines:

```
>>> """<html>
... <body>
... something
... </body>
... </html>
... """
'<html>\n<body>\nsomething\n</body>\n</html>\n'
```

- There are string characters with special meaning: e.g., `\n` (newline) and `\t` (tab)
- Get the length of a string by the `len` function

String indices & slices: You can use slices to get a part of a string

```
>>> s = "happy"
>>> len(s)  # use the len function
5
>>> s[3]    # indexed from 0, so 4th character
'p'
>>> s[1:3]  # characters 1 and 2
'ap'
>>> s[:3]   # first 3 characters
'hap'
>>> s[3:]   # everything except first 3 characters
'py'
>>> s[-4]   # 4th character from the back
'a'
```

Obviously, there is lots more to know about Python, but you should, prior to the second lecture, at the bare minimum check out these functionalities.

1.3.1 Packages used in the course:

In this course we will be using a number of modules frequently:

- pandas
- numpy
- scipy
- matplotlib

Note that on blackboard I will add a number of useful Python resources.

I want to recommend the use of `pandas` to handle data. Please check http://pandas.pydata.org/pandas-docs/stable/10min.html and work through the examples presented there to get some feel for the whole thing.

You can run python scripts directly from the terminal, and you can use and kind of text-editor you would like. However, we recommend using Spyder: https://pythonhosted.org/spyder/. Please use Python 3.x, as we will be using this in the tutorials.

Note that a part of this quick introduction into python is inspired by the materials that can be found here: http://cl.indiana.edu/~md7/12/555/slides/

Chapter 2

Lecture 2: Naive Bayes

This second lecture we will ease into the topic by looking at a very classical AI (or machine learning, I will use those words interchangeable throughout) algorithm that has proven very useful on the web for a very specific task: spam classification. It is conservatively estimated that about 85% of the worlds emails are spam, hence picking them out automatically makes sense and is used everywhere. We will cover two variations of the Naive Bayes (NB) algorithm (which is not really *naive*, nor is it really *Bayesian*) with the following goals:

- You will learn how to represent a piece of text (e.g., email) as a feature vector
- You will learn how to get from a mathematical model to an actual spam classifier
- You will understand Laplace smoothing
- You will understand different ways in which to setup a NB classifier
- You will make one yourself (in the tutorials).

For additional info on this topic both (Hastie et al., 2013) and (Bishop et al., 2006) provide excellent introductions (and extensions beyond what we cover here).

2.1 Standard Naive Bayes for spam classification

Our aim is to predict whether or not an email is spam or not spam (or rather, the probability that an email is spam). Lets denote $y = 1$ for a spam email and $\vec{x}$ for the feature vector (e.g., a representation of the email text). Our aim is to estimate $P(y = 1|x)$. We can now choose to represent the feature vector $\vec{x}$ (and I will drop the $\vec{x}$ notation and use x)

as depicted in Table 2.1. Note that we are presenting the presence of each word in our dictionary, and we are ignoring the frequency of occurrence or the order of words.

a	0
aardvark	1
...	
buy	1
...	
zz	0

In this setup the feature vector x denotes for $x_1, \dots x_n$, where n is the number of words in a dictionary, whether or not the word appears in an email. Thus $x \in \{0,1\}^n$. From here on I will use $x_j^{(i)}$ to refer to the i-th training example (e.g., the i-th email) and the j-th word. Note that this is only one possible representation, but its the one we will work with for now.[1]

As stated above, our aim is to estimate $P(y = 1|x)$, the probability of spam given a specific email. However we cannot estimate this directly. What we can estimate however – given a training set – is the probability of spam in general $P(y = 1)$, and the probability of our feature vector given that an email is spam $P(x|y = 1)$. We can thus use Bayes theorem (which is where the Bayes part in the name of the algorithm comes from, but technically we are using a frequentist approach for estimation, see below):

$$P(y|x) = \frac{P(x|y)P(y)}{P(x)} \tag{2.1}$$

$$= \frac{P(x|y)P(y)}{P(x|y=1)P(y=1) + P(x|y=0)P(y=0)} \tag{2.2}$$

The latter formulation requires us to estimate $p(y = 1)$, the probability of a spam email, and $P(x|y = 1)$, the probability of the feature vector given a spam email. The latter requires estimation of the joint probability $P(x_1, x_2, \dots, x_n|y)$ which, given the *chain rule of probability* is equal to:

$$P(x_1, x_2, \dots, x_n|y) = P(x_1|y)P(x_2|y, x_1)P(x_3|y, x_1, x_2)\dots \tag{2.3}$$

which looks cumbersome given the conditioning on both y and the other x's. This is where the second part of the name Naive Bayes comes from, because we will make the so-called

[1] Note that this representation gives us, if n is large (which it likely is, since n is the number of words in a dictionary) 2^n possible feature vectors. We could also choose to model $P(x)$ using a multinomial distribution with $2^n - 1$ params, but that is hard.

Naive Bayes assumption that the x_j's are *conditionally independent*[2]:

$$P(x_1, x_2, \ldots, x_n|y) = P(x_1|y)P(x_2|y)P(x_3|y)\ldots \tag{2.4}$$

$$= \prod_{j=1}^{n} P(x_j|y) \tag{2.5}$$

We thus obtain a model – if we treat, e.g., $P(y)$ as a Bernoulli(ϕ), with the following parameters;

$$\phi_{j|y=1} = P(x_j|y=1) \tag{2.6}$$

$$\phi_{j|y=0} = P(x_j|y=0) \tag{2.7}$$

$$\phi_y = P(y) \tag{2.8}$$

which we can estimate by maximising the likelihood

$$\mathcal{L}(\phi_y, \phi_{j|y=1}, \phi_{j|y=1}) = \prod_{i=1}^{M} P(x^{(i)}, y^{(i)}) \tag{2.9}$$

where $P(x^{(i)}, y^{(i)})$ denotes the joint probability of the i-th training example.

Note that, as described in Lecture 1, we can now work out all the algebra. We can use the product rule to specify $P(x, y)$. Then, we can derive the log likelihood function $l(\theta|x, y)$, take its derivative, and set to zero.[3] Suffices to say, that if you work things out, you get to the following estimates for the parameters:

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{M} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 1\}}{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 1\}} \tag{2.10}$$

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{M} \mathbb{1}\{x_j^{(i)} = 1, y^{(i)} = 0\}}{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 0\}} \tag{2.11}$$

$$\phi_y = \frac{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 1\}}{M} \tag{2.12}$$

where M denotes the number of training examples and $\mathbb{1}\{\}$ is the indicator function ("returns" 1 when the expression inside is TRUE, 0 otherwise). Note that to use Bayes Theorem that $P(x|y=1) = \prod_{i=1}^{n} P(x_i|y=1)$

That sums up NB.

[2]Note that this implies that learning that word A is in a spam email tells you nothing about learning that word B is in a spam email. Which obviously is not true, but works well in practice and is an often used trick to deal with large dimensional problems.

[3]See http://www.cs.columbia.edu/~mcollins/em.pdf if you are interested in the algebra.

2.1.1 Laplace smoothing

In practice, we often encounter a slight problem: suppose we ad a new word to our dictionary (e.g., $n+1$), or that some words that are in our dictionary are not in our training set, we get the following problem:

$$P(x_{n+1}|y=1) = 0 \tag{2.13}$$

and similarly for $P(x_{n+1}|y=0)$. That means that our NB will encounter $\frac{0}{0}$ at some point, which is undefined.

This is often "solved" by using Laplace smoothing which is defined as follows: *Given* $P(y) \sim \mathit{Multinomial}(1,\ldots,k)$ *we estimate*:

$$P(y=j) = \frac{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)}=1\}+1}{M+k} \tag{2.14}$$

which in our NB case gets us to

$$\phi_{j|y=1} = \frac{\sum_{i=1}^{M} \mathbb{1}\{x_j^{(i)}=1, y^{(i)}=1\}+1}{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)}=1\}+2} \tag{2.15}$$

$$\phi_{j|y=0} = \frac{\sum_{i=1}^{M} \mathbb{1}\{x_j^{(i)}=1, y^{(i)}=0\}+1}{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)}=0\}+2} \tag{2.16}$$

$$\phi_y = \frac{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)}=1\}+1}{M+2} \tag{2.17}$$

Note that with Laplace smoothing NB actually becomes "proper" Bayesian since from a Bayesian point of view, it corresponds to using a symmetric Dirichlet distribution with parameter 1 as a prior.

Since this is not a thorough course on Bayesian statistics, we will not dig into the nitty gritty of the Dirichlet distribution. However, it is good to consider that we could approach the problem of estimating the parameter p of a Bernoulli distribution not using MLE, but rather using a Bayesian framework. Here, we would put a prior distribution on p (which for generality I will now denote θ), and then compute the posterior distribution:

$$p(\theta \mid \mathbf{X}) = \frac{p(\mathbf{X} \mid \theta)p(\theta)}{p(\mathbf{X})} \propto p(\mathbf{X} \mid \theta)p(\theta) \tag{2.18}$$

If we use a $Beta()$ prior, which is *conjugate*, we obtain a $Beta()$ posterior. Laplace smoothing in this case corresponds to using a $Beta(1,1)$ — which is a uniform — prior for θ. The Dirichlet is merely the multivariate generalization of the Beta. I will talk a bit more about this in class.

2.2 Variation on standard Naive Bayes

In the previous implementation of NB – of which many versions exist – we did not consider the length of the email, nor the number of times a word occurs. The model we discussed is called the "Multivariate Bernoulli" model.

We can change these assumptions and derive the so-called "Multinomial Event" model. Here we create a slightly different feature vector x where we use $x = x_1, \ldots, n_i$, where n_i denotes the number of words in email i and $x_j \in \{1, \ldots, w\}$, where w is the number of words in the dictionary (sorry for the change in notation). Thus, the feature vector indexes for each word, in order of appearance, which word of the dictionary it is.

We can now specify the joint probability as

$$P(x, y) = (\prod_{j=1}^{n_i} P(x_j|y))P(y) \tag{2.19}$$

with parameters:

$$\phi_{k|y=1} = P(x_j = k|y = 1) \tag{2.20}$$
$$\phi_{k|y=0} = P(x_j = k|y = 0) \tag{2.21}$$
$$\phi_y = P(y = 1) \tag{2.22}$$

where k is the k-th word in the dictionary. Given a training set we can work out the ML estimate of the parameters:

$$\phi_{k|y=1} = \frac{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 1\} \sum_{j=1}^{n_i} \mathbb{1}\{x_j^{(i)} = k\}}{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 1\} n_i} \tag{2.23}$$
$$\phi_{k|y=0} = \frac{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 0\} \sum_{j=1}^{n_i} \mathbb{1}\{x_j^{(i)} = k\}}{\sum_{i=1}^{M} \mathbb{1}\{y^{(i)} = 0\} n_i} \tag{2.24}$$
$$\phi_y = P(y = 1) \tag{2.25}$$

so, the numerator of $\phi_{k|y=1}$ says: "sum over all the spam emails and count the number of times you observed the word k in spam emails", and the denominator says: "sum over all spam emails, and sum the length of that email". So the last one is the total length of your spam emails, and the ratio is the fraction of words that are word k out of the total length of your spam emails. That is the estimate of the next spam email generating the word k. For Laplace smoothing we add 1 to the numerator, and w to the denominator.

Note that some of the material presented in the lecture was inspired by the assignment presented here: http://www.cs.cmu.edu/~wcohen/10-605/assignments/hashtable-nb.pdf. For another explanation of NB see https://www.youtube.com/watch?v=qRJ3GKMOFrE.

2.3 Assignment

For this assignment we will be working with the ham vs. spam dataset which can be found on http://www.aueb.gr/users/ion/data/enron-spam/. Do the following in python:

1. Open the data in "Enron 1", and create a useable dataset using a dictionary found on http://www.manythings.org/vocabulary/lists/l/
2. Estimate your NB parameters ϕ for the multivariate Bernoulli model.
3. Classify "Enron 2" into spam or not spam and see how good you are doing. What measures will you use to show how good your classifier is?
4. Discuss how you would deal with "new" words (e.g., words that are in the email but not in your dictionary)?
5. If you feel courageous, also implement the Multinomial Event model.

Chapter 3

Lecture 3: Collaborative Filtering

In this lecture we will cover (a few versions of) very often used methods on the web that are collectively known as *collaborative filtering*. Collaborative filtering is often used in so-called recommender systems: They are used by Amazon.com to recommend you new products, and by Netflix.com to find the movies that you might like.[1] The basic aim of collaborative filtering methods is to match items i with users u. Since the items could be anything (movies, friends, news articles, etc.) the methods are very generally applicable.

The goals of this lecture are the following:

- You will learn to represent user ratings in a useful way for recommendation
- You will learn about (basic) neighborhood based collaborative filtering
- You will learn about distance measures and how to use these to provide recommendations.
- You will be able to implement a basic recommender system.

This lecture is—I guess—relatively easy. However, we are also still getting a hang of Python, so I don't want to force too many novel theoretical constructs. Also, note that recommender systems and collaborative filtering methods are a large (sub)-field of computer science, and hence you will be able to find many articles detailing implementations for specific context. This lecture is mainly intended to give a broad overview of the reasoning behind the methods, and to give you a feel for the pro's and con's.

[1] Note that not all recommender systems use collaborative filtering.

3.1 Formulation of the problem

The problem of recommending new product, news articles, friends, etc. can be formalized using a triplet (u, y, r), where u is a vector describing the user (in its simples case just a user identifier), y is a vector describing the item (again, in its simplest case just an identifier, and r is a rating. Often $r \in \{1, \ldots, k, \perp\}$ where $\perp$ indicates an empty rating.[2] If we treat the u and y vector merely as indexes, then a simple dataset containing the data used to compute recommendations is given in table 3.1. The table describes users in columns (lets index these running from 1 to n, and the rows describe the items (lets take these to run from 1 to m).

Table 3.1: Example of dataset used for (simple) collaborative filtering.

item id.	Angelica	Bill	Chan	Dan	Hailey	Jordyn	Sam	Veronica
1	3.5	2	5	3			5	3
2	2	3.5	1	4	4	4.5	2	
3		4	1	4.5	1	4		
4	4.5		3		4	5	3	5
5	5	2	5	3		5	5	4
6	1.5	3.5	1	4.5		4.5	4	2.5
7	2.5			4	4	4	5	3
8	2	3		2	1	4		

Note that you could use a python dictionary to store the data, or (e.g.,) a pandas data frame. This is what we will be using in the assignments. However, also note that in reality, if you are Amazon.com or the like, the number of users and items might both be in the millions (hence large), and storing the data in memory is not feasible. Data would be stored in a database (either SQL or no-SQL, depending on the structure and the use of the data). We will discuss different methods of storing the data in later lectures.

In the literature a distinction is made between *explicit* ratings – e.g., ratings filled out on a scale by users – and *implicit* ratings: ratings derived from observations of the behavior of users. Often, on the web, in the second case $r \in \{0, 1, \perp\}$ where 1 denotes a click on the item.

[2] Don't ask me why $\perp$ is used for "missing". I think its confusing since it is used for other things as well, such as (statistical) independence...

3.2 Neighborhood based collaborative filtering

One of the simplest collaborative filtering algorithms is based on the simple idea that to recommend an item y to a user u we find a user u^* that is similar to the current user, and we recommend whether he / she rates high. To do so, we need to compute a distance $d(u, u^*)$ for every user pair. The simplest of such distance measures would be

$$d(u, u^*) = \sum_{i=1}^{n} |r_i^u - r_i^{u^*}| \tag{3.1}$$

where n denotes the total number of items that both users have rated. Note that we would in practice not only sum over items, but also sum over all possible user to find the closest user. This can be computationally complex (depending on the implementation it is of order $\mathcal{O}(nm)$). Also note that users might differ in the number of "jointly" rated items, and thus we might need to correct $d(y, u^*)$ for this.

After computing the distance *for each* user pair we can select the "closest" user (or set of close users) and subsequently select items that the closest users rate high.

3.2.1 Distance measures

The closeness measure defined above is called the manhattan distance. You might be more familiar with Euclidean distance:

$$d(u, u^*) = \sqrt{\sum_{i=1}^{n} (r_i^u - r_i^{u^*})^2} \tag{3.2}$$

Note that these are two special cases of what is called the Minkowski Distance Metric:

$$d(u, u^*) = (\sum_{i=1}^{n} |r_i^u - r_i^{u^*}|^r)^{\frac{1}{r}} \tag{3.3}$$

For higher values of r more distant observations are given more and more weight. Dividing by the total number of jointly rated items n is often used to correct for differences in the number of rated items.

Note that the algorithm described above is an instance of a k-nearest neighbor algorithm (which you might have heard of).

3.3 Predictions

After finding the k closest users to the user of interest, we can obviously select the highest rated items by the close users as recommendations for the current user. We then implicitly use the rating by a "close" user as the predicted rating of the user of interest, and we search for the highest predicted values. Many other methods of getting to predicted values exist (see also the section on Model based approaches below), but one that is relatively common (and coined Resnick's algorithm) is the following:

To predict the score of a user u for a item i we can use:

$$p_{u,i} = \bar{r}_u + \frac{\sum d(u,v)(r_{v,i} - \bar{r}_v)}{\sum |d(u,v)|} \tag{3.4}$$

where $\bar{r}_u$ is the average rating of user u, and the sum is over $v \in V$ nearest neighbors (the number of which is selected by you). Note the here often as a distance measure the *Pearson correlation between users* (see below for the formula for items) is used. Contrary to the previous measures we discussed, for correlations higher scores mean a closer distance (inspect the formula to see that this makes sense). So, if we want to use (e.g.,) Minkowski distances we have to invert them in some way.

Looking at the formula we see that the prediction is based on the average score of the user herself $\bar{r}_u$ (which is controlling for the overall "positivity" of the user), and subsequently we are adding the ratings of neighbors: here we are also controlling for the average tendency to rate items of the neighbor, $(r_{v,i} - \bar{r}_v)$, and we are weighting the contribution of each neighbors rating's by their distance: (informally:) $\frac{d(u,v)}{\sum |d(u,v)|}$.

3.4 Item similarity based filtering

Next to user similarity based filtering, people often use *item* similarity based filtering. This is easily motivated by the example that a user clicks on a product, and next to that product we would like to present other products that the user might also like.

We are in this case looking for *items* that are similar to other items. Obviously, we could use a nearest neighbor computation on the items $d(i, i^*) = \sqrt{\sum_{u=1}^{m}(r_u^i - r_u^{i^*})^2}$, but often other measures of similarity are used in item based filtering.[3] Let's consider items i and j (instead of i^*, since I think the i^* can look confusing) and define the following similarity measures between items:

[3] Yes, off course we can use anything that is here introduced for items also for users and vice-versa.

Correlation based similarity (just pearson r_{xy} really):

$$sim(i,j) = \frac{\sum_{u \in U}(r_u^i - \bar{r}^i)(r_u^j - \bar{r}^j)}{\sqrt{\sum_{u \in U}(r_u^i - \bar{r}^i)^2}\sqrt{\sum_{u \in U}(r_u^j - \bar{r}^j)^2}} \tag{3.5}$$

where $\bar{r}^i$ denotes the average rating of item i over all users $u \in U$.

Equation 3.5 can also be expressed as $r_{i,j} = \frac{COV(i,j)}{SD(i)SD(j)}$. Since correlations are a building block of so many Machine Learning and Statistical methods we will briefly catch up on the interpretation of correlations and their relationship to linear models during the lecture. In any case: $0 \leq r_{ij} \leq 1$.

Cosine-based similarity:

$$sim(i,j) = cos(\vec{i}, \vec{j}) = \frac{\vec{i} \cdot \vec{j}}{||\vec{i}||_2 ||\vec{j}||_2} \tag{3.6}$$

where $\vec{i}$ denotes the vector of all ratings i and j have in common, $\cdot$ denotes the dot product (sum over the product of elements) between vectors i and j, and finally $||\vec{i}||_2$ denotes the Eucidean norm (square root of the sum of squared elements) of vector i. For non-negative ratings $0 \leq sim(i,j) \leq 1$

Adjusted cosine based similarity:

$$sim(i,j) = \frac{\sum_{u \in U}(r_u^i - \bar{r}_u)(r_u^j - \bar{r}_u)}{\sqrt{\sum_{u \in U}(r_u^i - \bar{r}_u)^2}\sqrt{\sum_{u \in U}(r_u^j - \bar{r}_u)^2}} \tag{3.7}$$

which is very similar to the correlation based similarity but the average rating of (e.g.,) item i, $\bar{r}^i$ is replaced by the average rating of the current user, $\bar{r}_u$.

Note that basically all the measures presented above (Correlation, cosine similarity, and adjusted cosine similarity) are variations on the dot product of two vectors.

3.5 User, Item, or mixed filtering

Which distance measure, or which approach (user vs. item based) you should use depends on your own problem. Thus, make sure you try and test. There are a few general things to note however:

- If the data is grade-inflated (thus, different users use the scales differently) then make sure to correct for this using (e.g,) adjusted cosine based similarity.

- If the data is very dense (thus, you have lost of observations for all pairs) then Minkowski distance based algorithms often work well.
- If the data is very sparse, cosine similarity is easy to use and often works well.

Also, there is no need to pick either user based or item based filtering: it is pretty obvious that the two can be combined in numerous ways. One obvious way to combine them is to – once a user arrives lands at a product page and you have 6 items to recommend – to use item based recommendations for half of your items and user based recommendations for the other half. Obviously, you can also weight the recommendations in more intricate manners. One thing that is often used is to select the k nearest users to user u, and subsequently weight the ratings of these near users using the correlations between ratings to derive a final rating for a new product (as we have seen above using Resnick's algorithm).

3.5.1 Problems

Collaborative filtering methods as described above are definitely not without problems. Let's name a few (that we did not cover earlier):

- *Sparsity:* Often, there are millions+ users, and there might be only a very small – or zero – number of joint ratings between u and u^*.
  - This might be solved using *imputation*: (e.g.,) using a model to predict the missing values. However, this is often not a good idea when the number of missing values is large (which it often is in the case of recommender systems), so you might look for other ways.
  - Sparsity is a problem for the predictions, but can be useful to cut down on computational complexity: the sparsity can often be explicitly used to design fast algorithms.
- *Scalability:* User based collaborative filtering grows, in computational burden, both with the number of users as well as with the number of items. These quickly become too large.
  - Here smart storage of the data, utilization of the sparsity, offline computations, etc. can all help to solve the problem.
- *Comparability / grade inflation:* Some users might give, on average, higher ratings then others. Thus, two users who's "pattern" of ratings (e.g., $r_1 > r_2 > r_3$) completely correspond might still have very different distance measures.
  - Standardization of scores might help.

  - We could also compute distance measures based on rank scores instead of actual scores.

- *Cold start:* What do we do with items for which we do not have any ratings? How do we obtain ratings if we never recommend them?
  - Impute? This is actually a hard problem to solve.
  - Use properties of the items to get a head-start.

- *New users:* What do we do with new users?
  - Same as above.
  - Use properties of the users to get a head-start.

- *Overfitting:* For example in user based filtering, whomever is closest might be pure luck. How do we prevent capitalizing on random agreements?
  - You probably do not want k (in a method based on k nearest neighbors) to be very small, and you want to average over ratings.

- *Curse of dimensionality:* If the number of dimensions is extremely large, then the closest or most similar items or users might still be very dissimilar in practice.
  - You might want to introduce minimal similarity bounds to make sure that what is "most similar" is indeed similar, and not least dissimilar.
  - Also note that in very high dimensions, basically every distance is large, and the differences (in the limit) vanish. This can be a problem that needs to be solved by mapping to lower dimensions.

- *Changes over time:* The algorithms above do not at all consider the possibility that ratings and preferences might change over time.
  - Hard problem ...

- *Cost to update:* Next to general computational complexity, the above algorithms are costly to update when the data changes.

- *Effect of the recommendations:* In practice, recommendations will hardly happen using a static data-set (which is what we have been considering up to now. Rather, recommendations will actually be used in (web) applications, and the recommendation will lead to new data. In this dynamic setting the recommendation itself influences the data that you collect. Hence, you have to take into account the effect of your recommendation on the observed data.

3.6 A brief introduction to different approaches

Recommender systems (of which collaborative filtering approaches are an instance) are an active area of research (see, e.g., Sarwar et al., 2001; Marlin, 2004) .[4] Collaborative filtering approaches are only one possible way to approach the problem. Here I highlight another avenue of attack that is also often used.

We could approach the problem of recommending an item i for user j by setting up a simple linear regression model (which you should understand thoroughly, if not, look it up). For example:

$$\begin{aligned} r_j^{i=c} &= \beta_0 + \beta_1 u_{1j} + \beta_2 u_{2j} + \cdots + \beta_{k-1} r_j^{i!=c} + \beta_k r_j^{i!=c} + \epsilon \\ &= X\beta + \epsilon \end{aligned}$$

where, with admittedly slight abuse of notation, I am trying to denote that we can try to model $r_j^{i=c}$, the rating of a specific item $i = c$ by user j using a (linear) combination of properties of the user (here denoted by u_{1j}, u_{2j}) such as her age and gender, and / or earlier ratings of that user of other items $i^{i!=c}$. With a flexibel representation of the design matrix X we can actually setup pretty complex models to predict the ratings on items and we can setup a regression model for each i to predict the user ratings. Note that the regression model "automatically" solves the comparability problem by weighting the importance of features.

The regression model can be fitted using Maximum Likelihood (discussed for these types of models in more detail in Lecture 6). This gives you an estimate of the vector β which you can use to make predictions. Note that the above specification leads to a large number of regression models, at least one for each item. Thus, the above approach of specifying a linear model for each item separately also has a severe dimensionality problem in the number of items: we will need to fit possibly millions of regressions. You might in reality want to group items first, or impose some relationships between the different models. This is often solved by using techniques like PCA (principal component analysis) to reduce the dimensionality of the dataset.

The above model is in practice hardly usable since, especially if we are incorporating all previous ratings, k (the dimensions of the regression problem) becomes very large and hence we run a high risk of overfitting or we find that we are unable to estimate the model since $n > k$ where n is the number of observations. This problem is usually solved using regularization (either ridge / lasso, or Bayesian methods – also discussed in more detail in Lecture 6).

For a very simple introduction to collaborative filtering with Python see: `http://aimotion.blogspot.nl/2009/11/collaborative-filtering-implementation.html`. Also, for a

[4] References for further reading will be provided in the final lecture notes at the end of the course.

simple intro which partly inspired this lecture see http://guidetodatamining.com/guide/ch2/DataMining-ch2.pdf. Note that the literature on Recommender systems and collaborative filtering is huge. Here are some examles: (Ricci et al., 2011; Ochi et al., 2010; Gretzel and Fesenmaier, 2006; Lam et al., 2008; Sarwar et al., 2001)

3.7 Assignment

For this assignment we will be working with the book crossing dataset provided at http://www2.informatik.uni-freiburg.de/~cziegler/BX/. Download the .csv dump, and open the books-ratings file. Here is what I want you to do:

- Find the ten users closest to user "170155" using euclidean distance
- Find the ten items with the highest adjusted co-sine similarity to item "0446520802"
- Suppose user "219459" would arrive at your website, and would look at book "0446520802", which 6 books would you recommend additionally and why?
- Delete, randomly, 20% of the ratings and try to predict them. Keep track of your predication accuracy. Do this in whatever way you think will get you the lowest error. Get some feel for the uncertainty of your procedure.
- Feel free to play around with possible model based approaches. However, note that in Lecture 6 we will be covering (linear) models in more detail.

Chapter 4

Lecture 4: Network statistics

On the web we will often find data that can be thought of as a network linking nodes together with edges. Nodes could for example be webpages, and the edges could be hyperlinks. Or, nodes could be people, and edges could be their friendship on a social network service like Facebook. In this lecture we will deal with network basics: the formal description of networks, and a number of network statistics that are useful to think about and describe network structures.

The goals of this lecture are the following:

- You will learn basic graph theory and notation
- You will learn basic network statistics
- You will learn about *trails* and *paths*
- You will learn how to use python to compute network statistics

4.1 A basic network: formal description of *un*directed networks

Figure 1 shows a basic network. This network has 4 nodes (the red circles), and a set of edges (or lines). In the figure, the edges are *directed*: they have arrow heads and denote a relationship that might be one-way. Thus, we can have *directed* or *undirected* networks. However, we will first start with the mathematical formalism, and some statistics regarding *undirected* networks or *graphs*, $\mathscr{G}$.

A graph $\mathscr{G}$ consists of two sets:

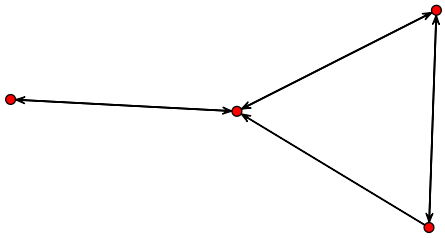


Figure 4.1: A basic network containing 4 nodes.

- A set of nodes, $\mathscr{N} = \{n_1, n_2, \ldots, n_g\}$
- A set of lines, $\mathscr{L} = \{l_1, l_2, \ldots, l_L\}$

and thus there are g nodes and L lines that jointly describe the full graph. Each line is an unordered pair of nodes, $l_k = (n_i, n_j)$ and since its bidirectional $l_k = (n_i, n_j) = (n_j, n_i)$. Usually we exclude (n_i, n_i), called a loop, from the set of lines. A graph without loops and only single, bidirectional relations is called a simple graph. In full notation we denote the simple graph $\mathscr{G}(\mathscr{N}, \mathscr{L})$. Graphs containing a single node are called *trivial*, and maps without any lines are called *empty*. A simple graph can obviously be represented graphically using nodes and undirected edges. Nodes n_i and n_j are called *adjacent* when $l_k = (n_i, n_j) \in \mathscr{L}$.

4.1.1 Subgraphs, Dyads, Triads

A graph $\mathscr{G}_S$ is a subgraph of $\mathscr{G}$ if $\mathscr{L}_S \subseteq \mathscr{L}$ and $\mathscr{N}_S \subseteq \mathscr{N}$. We can generate subgraphs both by selecting a number of nodes first, and then selection all lines associated to the nodes,

or selecting a number of lines and then selecting all relating nodes. Note that often when we do experiments on social networks (more on this later), we use node selection in our sampling and thus have access to only a subgraph of the full network.

A dyad is a (node-generated) subgraph consisting of a pair of nodes. A triad is a subgraph with three nodes. Note that sociologist have been really serious about the analysis of "possible" triads in social networks (we will talk briefly in the lecture about strong and weak ties).

4.1.2 Degree and Density of graphs

Once we have specified a network (or graph), it is of interest to compute summary statistics of the network. We will define a few of these in this and subsequent sections. First, we will focus on undirected networks.

The (nodal) degree of a node is defined as follows:

$$d(n_i) = \texttt{\# adjacent lines for node } n_i$$

The mean nodal degree of a graph $\mathscr{G}$ is

$$\bar{d} = \frac{\sum_{i=1}^{g} d(n_i)}{g} = \frac{2L}{g}$$

and its variance is

$$S_D^2 = \frac{\sum_{i=1}^{g} (d(n_i) - \bar{d})^2}{g}$$

The density of a graph, denoting the *proportion of possible lines* present in the graph, is:

$$\Delta = \frac{L}{g(g-1)/2} = \frac{2L}{g(g-1)}$$

since the maximum number of lines is $\binom{g}{2} = g(g-1)/2$ and quite clearly runs from 0 to 1.

Note that obviously, with a bit of algebra and the notion that the sum of degrees is equal to $2L$, we can show that $\Delta = \bar{d}/(g-1)$. Obviously similar measures can be computed for subgraphs.

4.1.3 Walks, Trails, and Paths

A walk W on a network (of graph) is a sequence of connecting nodes and lines, starting and ending with nodes. For example:

$$W = n_1 l_2 n_4 l_3 n_2 l_3 n_4$$

or more briefly $w = n_1 n_4 n_2 n_4$. Special types of walks are:

- Trail: A walk in which all the lines are distinct (nodes might be included more then once).
- Path: A walk in which all the nodes and all the lines are distinct. Path's are often referred to by their length: the number of lines in the path.
- Closed walk: A walk that begins and ends at the same node.
- Cycle: a closed walk of at leat three nodes in which all lines and nodes (except the begin-end point) are distinct. A graph that contains no cycle at all is called acyclic.
- Tour: A walk in which each line in the graph is used at least once.

And slightly more obscure we have Eulerian trails (closed trail including every line in the graph) and Hamiltonian cycles (including every node exactly once).

If there is a path between nodes n_i and n_j, then the pair is considered *reachable*. If there is a path – no matter how long – between every pair of nodes, then the graph is *connected*. Thus, in a connected graph all pairs of nodes are *reachable*.

Note that a number of the statistics presented above are super simple, but can be hard to compute since they require iterating through the graph multiple times. Developing computationally efficient methods to compute summary statistics of Graphs is an active area of research.

4.1.4 Distances and Geodesics

It is often useful to compute some distance measure between two nodes n_i and n_j. The shortest path (least number of lines) between two nodes is referred to as the *geodesic*, and the geodesic distance $d(i, j)$ is defined as the shortest number of lines between n_i and n_j. The distance is undefined if there is no path. Distances quantify how far apart nodes are, and they are used in centrality measures (see below). The *diameter* of a graph is the maximum distance max $d(i, j)$.

Note that, relating to our previous lecture, the distance between two nodes can also be used to make recommendations for new connections / friends in social networks. Here we

would be looking for nodes that are not adjacent to n (hence not already connected) but still close (e.g., with a small geodesic).

4.2 Directed graphs

We have, in the above, discussed undirected graphs: we were assuming the lines to connect the nodes both ways. However, many relations are directional: Webpage links might go from one page to another but not back, and you might follow people on Twitter who do not follow you. This gives the notion of directed graph $\mathscr{G}_d(\mathscr{N}, \mathscr{L})$, where $\mathscr{N}$ is a set of nodes as before, but $\mathscr{L}$ is a set of *arcs*, where each arc is an ordered pair of nodes $l_k =< n_i, n_j > \neq < n_j, n_i >$. Since each arc is an ordered pair of nodes there are $g(g-1)$ possible arcs in a graph. Nodes n_i and n_j are called *adjacent* when $< n_i, n_j > \in \mathscr{L}$, however this time adjacency of n_i and n_j does not imply adjacency of n_j and n_i. Graphically we usually represent arcs as arrows which indicate the direction, and $< n_i, n_j >$ indicates an arrow from node i to node j. Many of the concepts discussed above are easily extended to directed graphs, although often we need a few more definitions to make sure we identify the directions appropriately.

4.2.1 Indegree, Outdegree

We talked about the degree of a node for directed graphs. For undirected graphs we have the nodal indegree:

$$d_I(n_i) = \texttt{\# adjacent lines adjacent } \textit{to} \ n_i$$

and the outdegree:

$$d_O(n_i) = \texttt{\# adjacent lines adjacent } \textit{from} \ n_i.$$

As before we can compute $\bar{d}_I$ and $\bar{d}_O$, the mean in- and outdegree. Since clearly $\sum_{i=1}^{g} d_I(n_i) = \sum_{i=1}^{g} d_O(n_i) = L$ we note that $\bar{d}_I = \bar{d}_O = L/g$.

4.2.2 Directed walks

The idea of a walk, trail, path, etc. also generalise to directed graphs. Here we note the following:

- Directed walk: walk over a directed graph in the direction of the arcs.
- Directed trail: Directed walk in which no arc is visited more then once.

- Directed path: Directed walk in which no arc and no node is visited more then once.
- Semiwalk: A walk on a directed path in which the direction is ignored (and similarly semitrail and semipath).

Two nodes are set to be weakly connected if they are connected by a semipath, unilaterally connected if they are joined by a path (whichever direction), strongly connected if they are joined by a path back and forth, and recursively connected if they are strongly connected and both paths use the exact same nodes but in reverse order. Take some time to parse the previous sentence.

4.3 Matrix notation for graphs

Above we have covered some properties of graphs or networks. These are really just a basis, many more properties of graphs are known and studied. Please see (Wasserman, 1994) for more information (and actually below we will discuss a few more measures of graphs). But, before we go on we will discuss a different representation of graphs using matrices, and we will dig into the use of this matrix notation when trying to compute (e.g.,) distances.

Let's start with what is often called the or sociomatrix:

$$\boldsymbol{X} = \begin{pmatrix} - & 0 & 0 & 1 & 1 \\ 0 & - & 1 & 0 & 0 \\ 1 & 0 & - & 1 & 0 \\ 0 & 0 & 1 & - & 1 \\ 0 & 1 & 1 & 1 & - \end{pmatrix}$$

and please note that I will quickly drop the boldface on $\boldsymbol{X}$ and replace it by X if it needs no disambiguation. The sociomatrix denotes all the arcs between the the nodes. Hence the sociomatrix is a $g \times g$ matrix, in which each entry (either 0 and 1 for the graphs we have been considering thus far) in the (i, j)th cell (row i, column j) denotes an arc from n_i and n_j. A sociomatrix denotes al the notes and lines of graph $\mathscr{G}$.

Note that another way to denote the information of a graph is what is called an incidence matrix $\boldsymbol{I}$, which is a $g \times L$ matrix with nodes on the rows and lines on the columns (thus each column has two non-zero entries). I do not really think that an incidence matrix is a convenient way to describe graphs, although for some specific statistics the incidence matrix provides an efficient way of storing the data.

4.3.1 Matrix operations

Once we describe a graph or network using a sociomatrix X, we can do all kinds of cool stuff using standard matrix operations (which, clearly, you should already be familiar with). But lets note a few things:

- Transpose: The matrix $\boldsymbol{X}^T$ (or $\boldsymbol{X}'$ or $\boldsymbol{X}^t$ or ... depending on who you are talking to) is the transpose of matrix $\boldsymbol{X}$ where each $x_{ij}^T = x_{ji}$. That basically means "flipping" the matrix around its diagonal. Clearly, for an *undirected* graph $\boldsymbol{X} = \boldsymbol{X}^T$.
- Addition / subtraction: The addition of two matrices (with the same dimensions) is defined as $\boldsymbol{Z} = \boldsymbol{X} + \boldsymbol{Y}$ where $z_{ij} = x_{ij} + y_{ij}$. Similarly for matrix subtraction.
- Multiplication: The operation $\boldsymbol{Z} = \boldsymbol{XY}$ is defined as $z_{ij} = \sum_{l=1}^{k} x_{il} y_{lj}$. Note that $\boldsymbol{X}$ is of dimension $g \times h$ and $\boldsymbol{Y}$ of dimension $h \times k$ (the inner ones match up), and the result $\boldsymbol{Z}$ is of dimension $g \times k$. You should know this.
- Powers: $\boldsymbol{X}^2 = \boldsymbol{XX}$. Done, and similarly for higher powers.
- Boolean multiplication: This is matrix multiplication (see above), but now $\boldsymbol{Z} = \boldsymbol{X} \otimes \boldsymbol{Y}$ and defined as

$$z_{ij} = \begin{cases} 1 & \text{if} \sum_{l=1}^{k} x_{il} y_{lj} > 0 \\ 0 & \text{if} \sum_{l=1}^{k} x_{il} y_{lj} = 0 \end{cases}$$

Network properties using matrix operations

Using the sociomatrix it is relatively easy to compute walks and *reachability*. Note that an entry $x_{ij} = 1$ in $\boldsymbol{X}$ indicates that there is a walk of length one between node n_i and n_j in a graph. Any non-zero entry in $\boldsymbol{X}^p$ denotes the number of walks of length p between n_i and n_j. Hence, taking the powers of the sociomatrix tells you how long walks are between two nodes. We can also define $\boldsymbol{X}^\Sigma = \boldsymbol{X}^1 + \boldsymbol{X}^2 + \cdots + \boldsymbol{X}^{g-1}$ where any non-zero entry in $\boldsymbol{X}^\Sigma$ indicates the n_j is *reachable* from n_i in $g - 1$ steps (which is obviously the maximum number of steps).

Matrix operations on the sociomatrix also help computing distances $d(i, j)$. Basically the first non-zero entry of $\boldsymbol{X}^p$ when increasing powers $p = 1, \ldots, p = g - 1$ gives the distance between two nodes. Formally $d(i, j) = \min_p x_{ij}^p > 0$.

Finally, we can easily use compute the degree of (a node of) a directed graph:

$$d(n_i) = \sum_{j=1}^{g} x_{ij} = \sum_{i=1}^{g} x_{ij} = x_{i+} = x_{j+}$$

and similarly for directed graphs: $d_O(n_i) = x_{i+}$ and $d_I(n_i) = x_{+i}$.

Finally, we can compute the density of a graph:

$$\Delta = \frac{\sum_{i=1}^{g} \sum_{j=1}^{g} x_{ij}}{g(g-1)/2}$$

4.4 Centrality and prestige

A recurring theme in the social network literature is the computation of actor *centrality*. Here, we try to identify actors that are prominent and extensively involved with other actors. Actors (or nodes) are considered prominent if the ties of the actor make her particularly visible to the others. In a directional graph, actor *prestige* (or status) extends the definition of centrality to incoming ties, focussing on the actor as a recipient of ties.

Lets denote $C_A(n_i)$ as a general centrality measure for actor n_i (below you will find specific examples). And, further define $C_A(n^*)$ as the largest value of a particular centrality measure in the network. Freeman (see, e.g., Wasserman, 1994) defines as a group level centrality measure (thus a summary statistic for a graph), the general centralization index:

$$C_A = \frac{\sum_{i=1}^{g} (C_A(n^*) - C_A(n_i))}{\max \sum_{i=1}^{g} (C_A(n^*) - C_A(n_i))}$$

Clearly $0 \leq C_A \leq 1$.

4.4.1 Specific types of centrality

An obvious centrality index is the degree of an actor $C_D(n_i) = d(n_i)$ that we covered earlier. However, since $d(n_i)$ depends on g, this is often standardized: $C_D'(n_i) = d(n_i)/(g-1)$. $C_D(n_i)$ can be uses as a centrality measure to compute a general centralization index. We can use the same idea but now using $d_I(n_i)$ to compute a measure of *prestige*.

Closeness centrality provides another measure of centrality and is defined as $C_C(n_i) = [\sum_{j=1}^{g} d(n_i, n_j)]^{-1}$ where $d(\bullet, \bullet)$ is a distance function between two nodes.

Another measure of prestige is called rank prestige and is defined by $\vec{p} = \boldsymbol{X}^T \vec{p}$ where the entries of $\vec{p}$ contain the rank prestige for each node. You might recognise this as an Eigenvector equation. While this looks simple the idea is pretty refined, and the calculation is not always trivial (for example the above specification assumes $\boldsymbol{X}^T$ to have an Eigenvalue of one, which might not always be true). We will talk about rank prestige more in the next lecture.

4.5 Further remarks on networks and graphs

In this lecture we really only scratched the surface of what is known about graphs, networks, and how to analyze them. However, knowing some of the most basic jargon and operations is useful, since these are often encountered when describing things that are happening on the web. I just want to mention two things that I left out in the above, but that I think you should be aware of:

- Homophily: "Similarity breeds connection. This principlethe homophily principlestructures network ties of every type, including marriage, friendship, work, advice, support, information transfer, exchange, comembership, and other types of relationship. The result is that people's personal networks are homogeneous with regard to many sociodemographic, behavioral, and intrapersonal characteristics. Homophily limits people's social worlds in a way that has powerful implications for the information they receive, the attitudes they form, and the interactions they experience." For more see (Aral et al., 2009). . There is a lot of research on homophily, its strength, and its consequences on the web.
- Network experiments: Now that social networks like Facebook and Twitter and the like clearly give us an overview of the links between people, there is a recent uproar of academic work interested in the estimation of effects of treatments when these are delivered to people that are connected in one way or another (Aral et al., 2011; Bakshy et al., 2012) . It really is non-trivial how one should conduct classical (psychology) experiments in networks. You will experiment with this a bit in this weeks assignment, but make sure to read up on the literature on the topic if you ever find yourself doing an experiment in a network (which is some ways you always are when you are doing an experiment).

4.6 Assignment

For this assignment you will actually be generating your own dataset (since that is good exercise too).

- Write a function to generate a sociomatrix of size $g \times g$ with a density Δ that is set by the user.
- Write a function to generate a sociomatrix for a directed graph of size $g \times g$ which you populate with random ties (each possible arc is a draw from a Bernoulli(p)).
- Plot your graph for $g = 20$.
- Plot your graph for $g = 10^3$.

- Create a function which generates a dataset containing a value $x_i \sim \mathcal{N}(10, 4)$ for each node i for $i = 1, \ldots, i = g$.
- Generate the sociomatrix and values (using your above functions) for $g = 10^3$
- Randomly sample 200 nodes and simulate an experiment in which $n = 100$ nodes (randomly selected from the sample of 200) are "treated" which results in $x_i := x_i + 10 + \epsilon$ where $\epsilon \sim \mathcal{N}(0, 2)$. Estimate the effect of the treatment.
- Now repeat the experiment, but this time assume that there is a network effect: after the effect of the treatment itself, the nodes affect each other using the following formula $x_i = \alpha x_i + (1 - \alpha)(\bar{x}_j)$ where $\bar{x}_j$ denotes the mean value of the nodes adjacent to i.
- Experiment with settings for α, the density Δ, and different sample sizes. How does the "network" effect influence the estimates of the treatment effect?
- Develop different specifications of the network effect.

Chapter 5

Lecture 5: Markov Chains and Pagerank

In this lecture we will continue our discussion of the Web as a network, and will explore one of the most well-known uses of this property: the use of the network structure in Google's Pagerank (in their original version that is — nowadays their algorithm is a bit more complex). However, before we work out Pagerank we have to make a small digression into Markov Chains (you will see why once we get there).

The goals of this lecture are the following:

- You will learn some basics regarding Markov Chains.
- You will understand stationary distributions of Markov Chains
- You will understand the link between the network socio-matrix and transition matrixes.
- You will be able to implement a basic version of Google Pagerank in Python.

5.1 Markov Chain Theory

Let $X_0, X_1, X_2, \ldots$ be a sequence of random variables, for example the (discrete and finite) possible states of a system at discrete timepoints n. The sequence has the Markov property if:

$$\begin{aligned} P(X_{n+1} = j | X_n = i, X_{n-1} = i_{n-1}, \ldots, X_0 = i_0) \\ = P(X_{n+1} = j | X_n = i) \end{aligned}$$

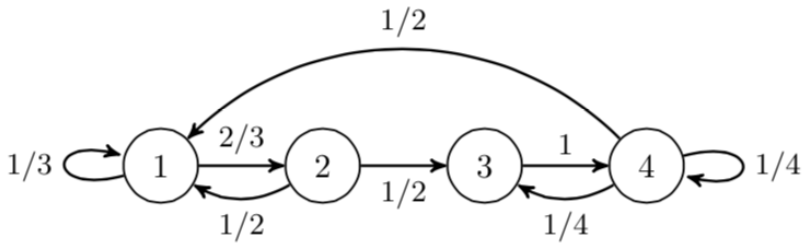


Figure 5.1: Example of a homogeneous Markov Chain with the associated transition probabilities

Basically, if a system has the Markov property then *the past and future are conditionally independent given the present*. Such a stochastic system is called a Markov chain and is used in many places. You should familiarize yourself a bit with Markov theory since it has many useful applications.

If $P(X_{n+1} = j|X_n = i)$ does not depend on time (hence if the probability of reaching a specific state in the future given the current state in the present do not change over time) then we have a homogeneous Markov chain and we can denote the transition probability as:

$$q_{ij} := P(X_{n+1} = j|X_n = i)$$

Obviously we can stack all the q_{ij}'s into a Matrix Q which is called the transition matrix.

Consider the Markov chain displayed in Figure 5.1. The transition matrix is:

$$Q = \begin{pmatrix} \frac{1}{3} & \frac{2}{3} & 0 & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{1}{2} & 0 & \frac{1}{4} & \frac{1}{4} \end{pmatrix}$$

Note that the rows sum to 1.

Suppose that at time n, X_n has distribution s (a row vector denoting the PMF), then

$$\begin{aligned} P(X_{n+1} = j) &= \sum_i P(X_{n+1} = j|X_n = i)P(X_n = i) \\ &= \sum_i q_{ij} s_i \\ &= sQ \end{aligned}$$

so sQ is the distribution of X_{n+1}. More generally, sQ^j is the distribution of X_{n+j} which also implies:

$$P(X_{n+m} = j | X_n = i) = (Q^m)_{ij}$$

5.1.1 Stationary distributions

Let s be some probability vector for a Markov chain with transition matrix Q. We say that s is stationary if:

$$sQ = s$$

Note again here the similarity of an Eigenvector equation (the transpose this time) as briefly referred to in the previous lecture when discussing rank prestige. The stationary distribution can be thought of as the long-run probability that, if you consider the Markov chain as a directed graph with weights on the edges, that you end up at a specific node.

5.1.2 Irreducible Markov chains and stationary distributions

The stationary distribution s is clearly of interest as a summary of the stochastic process (or as a measure of rank prestige. Note that then the sociomatrix needs to be normalised such that the rows sum to 1). However, one could wonder if s actually exists, if it is unique, and if the chain "converges" to s. Obviously, its also kind of interesting to see how we could compute s.

Much is known about Markov Chains that we will not cover in this course, but one theorem is important for our current purposes:

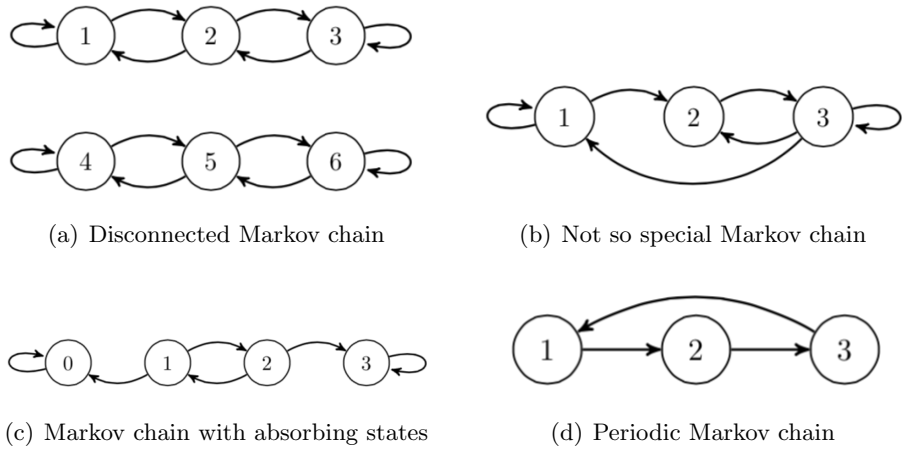
Theorem 1 *For any irreducible Markov chain:*

1. *A stationary distribution s exists*
2. *s is unique*
3. *$s_i = \frac{1}{r_i}$, where r_i is the average time to return to state i starting from state i.*
4. *if Q^m is strictly positive for some m, then given any stating state t*

$$\lim_{n \to +\infty} tQ^n = s \tag{5.1}$$

Which obviously begs the question: when is a Markov chain *irreducible*? Well, that is the case if it is possible (with probability larger then 0) to transit from any one state to any of the other states in a finite number of transitions. Note that this means that all states

are *recurrent*: there is a probability of 1 of recurring to that state after a finite number of transitions. This means that in the figure below 5.2(b) and 5.2(d) are reducible, the other two are not.



(a) Disconnected Markov chain

(b) Not so special Markov chain

(c) Markov chain with absorbing states

(d) Periodic Markov chain

Figure 5.2: Some Markov chain examples

5.2 Google Page Rank

I am sorry if that was too long a ramp up (both networks and Markov chains) to discuss one of "the" algorithms used on the web: Pagerank. Pagerank (Page et al., 1999) was designed as a solution to answer the question: which pages on the internet are more important then others? (or, given a search term, if I get all the pages with that term in them, how do I order the pages?).

The question of identifying "important" pages should make you think of identifying "important" nodes in a network. However initially this is not the way the task was carried out: companies tried human raters to rank the importance of pages, and other companies counted the number of time a search word occurred to estimate importance. Those methods did not really work: the were hard to scale or easy to cheat.

Alta Vista was one of the first companies to recognize the importance of the netwerk structure of the web: webpages can be considered nodes, and links can be considered edges. So, we could use degree (or some standardized version of in-degree) as a measure of importance. However, that is also easy to cheat (just create a lot of fake webpages that link to a page). So, people started looking for a *definition of importance which was based on the network structure of the web, but incorporated not only the degree, but also*

the importance of the nodes linking to a node. That sounds circular, but its exactly what Pagerank does...

Suppose we look at a very small portion of the web (only 4 pages – but the idea remains the same if we scale it) with the following sociomatrix describing the links:

$$X = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

we could then normalise this matrix to get to a transition matrix and treat this snippet of the web as a Markov chain:

$$Q^* = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{5.2}$$

We can then define the importance of a page by looking the stationary distribution s of the Markov chain with transition matrix Q where higher entries in s denote a higher "long-run" probability of ending up at that page and thus a more important page!

That was the key insight: Pagerank is based on the stationary distribution of the web.

5.2.1 Practical issues

If you pay close attention you see that the transition matrix in Equation 5.2 is not valid. The row sums are not equal to 1. This is because page 4 has no outgoing links, so there is no probability of leaving the page in the current specification.

The solution used in pagerank is the following:

$$Q = \begin{pmatrix} 0 & \frac{1}{2} & \frac{1}{2} & 0 \\ \frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 1 \\ \frac{1}{m} & \frac{1}{m} & \frac{1}{m} & \frac{1}{m} \end{pmatrix} \tag{5.3}$$

where m is the number of pages: 4 in this example.

So that solves one problem, however now we are faced with the next problem: if we want to compute $sQ = s$, then we need to know that a stationary distribution even exists. One way of solving it is making sure that the chain described by transition matrix Q is irreducible. This was solved in pagerank using the following:

$$G = \alpha Q + (1 - \alpha)\frac{J}{m}$$

where G is the guaranteed irreducible transition matrix, J is an $m \times m$ matrix with all 1's, $0 \leq \alpha \leq 1$ is a tuning parameter. This forces a small positive transition probability from each state to each state. In the original pagerank article the authors choose $\alpha = .85$.

Now, to get s, or really the pagerank p, we compute $pG = p$. This however is a system of linear equations that might be quite large since m might be 10^9 or even more. With (e.g.,) Gaussian elimination this might be very cumbersome.

5.2.2 Computational issues

The pagerank p is computed by using Equation 5.1: we take some starting position s_0, and then just iterate for "a long time" (how long is something one could have deep philosophical debates about).

so, we compute:

$$p = ((s_0 G)G)\ldots$$

Note that this still is cumbersome, but there are a few things that make it computationally feasible:

- We can break down the computation in terms $s_0 G = \alpha s_0 Q + (1 - \alpha)\frac{s_0 J}{m}$.
- The first term Q is composed largely of 0's, and computationally fast methods for these types of *sparse* matrix multiplications exist.[1]
- Also $s_0 J$ means doing a dot product with a matrix J with all 1's: basically just adding all the components of s_0. That means the result is a vector of all 1's.

5.2.3 Example

Here is some example (sorry) [R] code to make all of the above run for a link-matrix of size $M = 10$:

```
set.seed(14)
# Build and normalize sociomatrix:
M <- 10
X <- matrix(rbinom(M*M,1,.3), nrow=M, byrow=TRUE)
```

[1] Note that this is not a computational methods course, so if you need more on this, try another course.

```
Q.star <- t(apply(X,1,FUN=function(x){return(x/sum(x))}))
Q <- t(apply(Q.star,1,FUN=function(x){
        if(any(is.nan(x))){
                return(rep(1/length(x),length(x)))
        }else{return(x)}}))

# Google pagerank matrix
alpha <- .85
J <- matrix(1,nrow=M,ncol=M)
G <- alpha * Q + (1-alpha) * J/nrow(Q)

# random start
draws <- runif(M)
p <- draws / sum(draws)

# iterate and store:
N <- 50
store.p <- matrix(NA, ncol=N, nrow=M)
for(i in 1:N){
        p <- p{\%}*{\%}Q
        store.p[,i] <- p
}
# plot
plot(store.p[1,], type="l", ylim=c(0,.5))
for(i in 2:M){
        lines(store.p[i,], col=i)
}
```

And a little plot of how fast this actually converges.

Note that this lecture was partly based on a lecture by Joe Blaskovich given at Berkely during Statistics 101. These lectures, which I can highly recommend, can be found on iTunes U: http://itunes.berkeley.edu.

5.3 Assignment

This assignment looks short, but isn't simple at all:

- Write, using e.g., http://scrapy.org a scraper to scrape all links within a specific domain. For example http://www.ru.nl/artificialintelligence/.
  - Thus, you first aim is to create an array of nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_g\}$ and a list of lines $\mathcal{L} = \{l_1, l_2, \ldots, l_L\}$ that describe the /artificialintelligence/ pages
  - You should be aware that you can make http requests to get the html document, that you can parse this document, and that you can navigate the DOM tree.

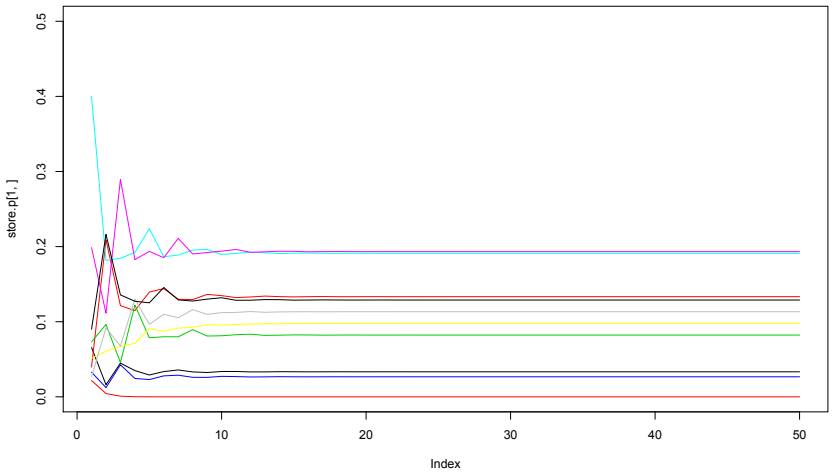


Figure 5.3: Convergence of a random starting distribution p over iterations N for the pagerank algorithm.

  - You are obviously looking for `<a href=‘‘...’’>...<a>` tags.
  - **Make sure you are not querying the same page in an infinite loop!**
  - **Time the number of request you make to the page so you don't get banned!**

- Create the link matrix X, and then the transition matrix Q.
- Compute the pagerank for each of the pages. Make sure to monitor "convergence" in some way.

Chapter 6

Lecture 6: Linear and hierarchical models in Python.

The goals of this lecture are the following:

- You will learn how to fit linear models in Python (and off course you will learn some theory behind them)
- You will learn how to fit generalized linear models in Python
- We will discuss shrinkage models (no-pooling, pooling, partial pooling)
- You will learn about Stein estimation
- You will learn how to fit hierarchical linear models in Python

Note that this lecture, and these lecture notes, focus primarily on the (mathematical) ideas behind these types of models. They are included in the course since they have a large number of applications for web data. Fitting the models and finding your ways through Python is left up to you during the assignments.

If you want to know more about these topics I can strongly recommend the following books: (Gelman and Hill, 2006; Hastie et al., 2013).

6.1 Linear regression

Ok, this section is, I hope, all well known to all of you, but it never hurts to revisit.

A (very) simple linear regression model is given by

$$y_i = \beta_0 + \beta_1 x_i + \epsilon_i$$

where

- y_i is the response of "person" i.
- x_i is the predictor
- β_0 is the unknown intercept of the line
- β_1 is the unknown slope of the line
- $\epsilon_i \sim N(0, \sigma^2)$ is the noise with unknown variance σ_ϵ^2

This implies a few things, namely, y_i is a random quantity due to ϵ_i only, $\mathbb{E}(y) = \beta_0 + \beta_1 x$, $\mathbb{V}(y) = \sigma^2$, and thus, $y_i \sim N(\beta_0 + \beta_1 x_i, \sigma^2)$.

One way to find the "best" fitting line through a set of points – because remind you, that is kind of what we are after here – is called Least Squares. It minimizes:

$$\min_{b_0, b_1} \sum_{i=1}^{n} (y_i - b_0 + b_1 x_i)^2$$

Since this is a minimization problem, taking the derivatives with respect to b_0 and b_1 and setting them equal to zero will result in two equations which are called the **normal equations**:

$$\begin{aligned} nb_0 + (\sum x_i) b_1 &= 0 \\ (\sum x_i) b_0 + (\sum x_i^2) b_1 &= \sum x_i y_i \end{aligned}$$

Or, solving for the parameters of interest:

$$\begin{aligned} b_1 &= \hat{\beta}_1 = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2} \\ b_0 &= \hat{\beta}_0 = \bar{y} - b_1 \bar{x}. \end{aligned}$$

Notice that there is nothing probabilistic about least squares estimation. It's merely an optimization problem where the sum of squared vertical distances from actual points to a line is minimized. There is no underlying distribution assumption. In fact, nothing is treated as random. We just have a cloud of points and we pass a line through them.

However, we said $y_i \sim N(\beta_0 + \beta_1 x_i, \sigma^2)$, and thus assuming that the responses are distributed normally with mean $\beta_0 + \beta_1 x_i$ and variance σ^2 we can also compute a likelihood

over the unknown model parameters β_0, β_1 and σ^2. Maximizing this likelihood will yield the Maximum Likelihood Estimates (MLE).

The density functions for y_j are

$$f(y_i) = \frac{1}{\sqrt{2\pi}\sigma} \exp\Big[\frac{(y_i - \beta_0 + \beta_i x_i)^2}{2\sigma^2}\Big].$$

The loglikelihood is

$$L(\beta_0, \beta_1) = const - \frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \beta_0 + \beta_1 x_i)^2.$$

We then maximize the loglikelihood, which in this case is equivalent to minimizing the sum of the residuals:

$$\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - \beta_0 + \beta_1 x_i)^2.$$

Hence MLE is LSE in linear regression. This is a nice link between purely distance based methods and probabilistic methods. It turns out that under the assumptions we made earlier, the maximum likelihood estimators for β_0 and β_1 are identical to the least squares estimators.

The maximum likelihood estimator for the error variance σ^2 is easily obtained as

$$\hat{\sigma^2} = \frac{\sum_{i=1}^{n}(y_i - b_0 + b_1 x_i)^2}{n}.$$

Note that this is a *biased* estimator for σ^2.[1] To correct for the bias we can subtract the number of parameters estimated prior to the estimation of σ^2 from n. Thus, the unbiased estimator is obtained as

$$s^2 = \frac{\sum_{i=1}^{n}(y_i - b_0 + b_1 x_i)^2}{n-2}.$$

for the 2 parameter case.

Finally, for inferential purposes, it can be shown that $b_1 = \hat{\beta}_1$ is normally distributed with mean $\mathbb{E}(b_1) = \beta_1$ and variance $\mathbb{V}(b_1) = \frac{\sigma^2}{S_{xx}}$ where $S_{xx} = \sum(x_i - \bar{x})^2$. Thus the quantity $z = \frac{b_1 - \beta_1}{\sigma/\sqrt{S_{xx}}}$ would be standard normally distributed. Since we don't know σ^2, if we replace it by its estimator s^2:

$$t = \frac{b_1 - \beta_1}{s/\sqrt{S_{xx}}}$$

[1] Here "bias" means that the expected value of the estimator for σ^2 is not equal to the true value.

has a t distribution with $n-2$ df.

So, that was the simple case. All of this obviously translates to more predictors: $\vec{x}_i$. We can thus now denote:

$$Y = \beta_0 + \beta_1 x_1 + \beta_3 x_3 + ... + \beta_k x_k + \epsilon$$

where

- y is the response
- $x_1, x_2, x_3, ..., x_k$ are the predictors
- $\beta_0, \beta_1, \beta_2, ..., \beta_k$ are unknown regression coefficients
- $\epsilon \sim N(0, \sigma^2)$ is the noise with unknown variance σ^2

When we have n observations from such a model, i.e. $\mathbf{y} = (y_1, y_2, ..., y_n)'$, with we define $\mathbf{X}$ as the design matrix

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1p} \\ 1 & x_{21} & x_{22} & \cdots & x_{2p} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{np} \end{bmatrix}.$$

The least squares (or MLE) solution $\hat{\boldsymbol{\beta}} = (\beta_0, \beta_1, \beta_2, ..., \beta_k)'$ is given by

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}.$$

6.1.1 Note on Ridge regression, Lasso, etc.

Estimation of $\vec{\beta}$ as described above for linear regression works and is super simple (in closed form). It can actually be done in summation form in data streams (we will talk about this more in later lectures).

However, when $k > n$, where n is the total number of individuals, this will fail (why?). There are several solutions to the problem. One is to not minimize $\min_{b_0,b_1} \sum_{i=1}^{n}(y_i - b_0 + b_1 x_i)^2$ (for the simple regression case), but rather something like:

$$\min_{b_0,b_1} \left((\sum_{i=1}^{n}(y_i - b_0 + b_1 x_i)^2) + f(b_0, b_1) \right)$$

where $f(b_0, b_1)$ denotes some function of the b's, which (usually) imposes a penalty for high values of b. *Ridge regression*, *the Lasso*, etc. are all variations of this basic idea. Note that also the *multi-level models* that we cover below can be thought of as a penalty on a batch of parameters (more on this later).

A Bayesian approach

Another solution to $k > n$ is to take a Bayesian approach by placing a prior on $\boldsymbol{\beta}$ which somewhat restricts the solution but will help at least make it tractable. The prior we can[2] use on $\boldsymbol{\beta}$ is

$$\pi(\beta) \propto \exp(-\tau \boldsymbol{\beta}'\boldsymbol{\beta}).$$

We now write down the posterior with respect to $\boldsymbol{\beta}$

$$\text{Post}(\boldsymbol{\beta}) \propto \exp\left(-\frac{(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})'(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})}{2\sigma^2}\right) \times \exp(-\tau \boldsymbol{\beta}'\boldsymbol{\beta}).$$

We now, to obtain our MAP estimate, compute the log-likelihood and maximize

$$\max_{\boldsymbol{\beta}} \left[-\frac{(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})'(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})}{2\sigma^2} - \tau \boldsymbol{\beta}'\boldsymbol{\beta}\right],$$

or

$$\min_{\boldsymbol{\beta}} \left[(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})'(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) + 2\sigma^2\tau \boldsymbol{\beta}'\boldsymbol{\beta}\right].$$

Please note the similarity between the resulting estimator and the Ridge / Lasso specification.

Finally, we can take the derivatives to solve this stuff:

$$\begin{aligned}
\mathbf{0} &= \frac{\partial}{\partial \boldsymbol{\beta}} \left[(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})'(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) + 2\sigma^2\tau \boldsymbol{\beta}'\boldsymbol{\beta}\right] \\
&= -2\mathbf{X}'(\mathbf{y}-\mathbf{X}\boldsymbol{\beta}) + 2\tau\sigma^2\boldsymbol{\beta} \\
&= \mathbf{X}'(\mathbf{X}\boldsymbol{\beta}-\mathbf{y}) + \lambda\boldsymbol{\beta}.
\end{aligned}$$

and with some not too fancy matrix manipulations we get:

$$\begin{aligned}
\mathbf{X}'\mathbf{y} &= \mathbf{X}'\mathbf{X}\boldsymbol{\beta} + \lambda\boldsymbol{\beta} \\
\mathbf{X}'\mathbf{y} &= (\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta} \\
(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}'\mathbf{y} &= (\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})\boldsymbol{\beta} \\
(\mathbf{X}'\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}'\mathbf{y} &= \hat{\boldsymbol{\beta}}.
\end{aligned}$$

Note that $\mathbf{X}'\mathbf{X} + \lambda\mathbf{I}$ is invertible when $\lambda > 0$ and hence solves the issue.

[2] Since this provides us with a relatively simple solution

6.2 Link functions and Generalized Linear Models

Above, we where assuming continuous (and Normal) y. We might not always encounter these. However, note that what we were doing was $E(y_i|x_i) = \beta_0 + \beta_1 x_i$, we were finding the expected value of y given x. What do we do if $y \in \{0, 1\}$? Well, then we could state $E(y_i) = Pr(y_i = 1) = \pi_i$ and the linear regression thus would become:

$$\pi_i = \beta_0 + \beta_1 X_i \tag{6.1}$$

with the full model for y being

$$y_i = \pi_i + \epsilon_i = \beta_0 + \beta_1 X_i + \epsilon_i \tag{6.2}$$

and this is called the *linear probability model*. But that is a stupid idea because:

1. The residuals are supposed to be $\sim \mathcal{N}$. However, with a binary response variable the residual $y - \beta_0 + \beta_1 X_i$ can only take a limited number of values.
2. The variance of e, σ_e^2 should be constant (homoskedasticity). However, $var(y_i) = \pi_i(1-\pi_i) = (\beta_0+\beta_1 X_i)(1-\beta_0+\beta_1 X_i)$ which depends on x_i and is thus heteroskedastic.
3. The model would easily predict values for y_i that are outside of its range $(0, 1)$ and thus would produce insensible results.

So, we are stuck when we do not have continuous data. Well, not really. We could use a so called Link[3] function, which "links" the linear predictor $X\beta$ to $E(y)$. A few examples:

- *The logit link:*

  $$Pr(y_i = 1|X_i) = logit^{-1}(X_i\beta) = \frac{1}{1 + e^{-X_i\beta}}. \tag{6.3}$$

  The logit link is used to model dichotomous data. I guess you are familiar with this model. Note that it cannot be fit in closed form, and thus requires some sort of iterative method. We will cover methods to fit logistic regression later on, for now please just rely on the existing implementations in Python.

- *The probit link:*

  $$Pr(y_i = 1|X_i) = \Phi(X_i\beta), \tag{6.4}$$

  here Φ is the CDF (cumulative density function) of the normal distribution. This model is also used for dichotomous data and is also called *probit regression*. Often the results of probit and logit regression are very similar (since they only differ slightly in the tails). Probit links are used more in economics and are often easier for Bayesian analysis.

[3] That's the jargon used in statistics, in computer science they call the inverse of the link function the activation function.

- *The poisson link:*

$$E(y_i|X_i) = exp(X_i\beta). \tag{6.5}$$

  This distribution has support over the interval $[0, \infty)$ and is often used to model count data. E.g. how many visitors does a website attract within a given time period? Or how many cars run over different types of roads within a set timeframe. The data is assumed to be distributed *Poisson*.

- *The cumulative logit link:* (this one is a bit tricky to get your head around)

$$\begin{aligned} Pr(y_i > 1|X_i) &= logit^{-1}(X_i\beta) && (6.6)\\ Pr(y_i > 2|X_i) &= logit^{-1}(X_i\beta - c_2) && (6.7)\\ Pr(y_i > 3|X_i) &= logit^{-1}(X_i\beta - c_3) && (6.8)\\ &\dots && (6.9)\\ Pr(y_i > K-1|X_i) &= logit^{-1}(X_i\beta - c_{K-1}). && (6.10) \end{aligned}$$

  This is used for ordered categorical data. Basically the linear predictor predicts a number between $-\infty$ and $+\infty$. Next, a number of "cut offs" are determined (recall the latent variable specification we talked about earlier for logistic regression). The value of y_i is predicted by comparing the different cut-offs $c_1, \dots, c_{K-1}$.

Do note that other link functions (such as the Gamma) exist and might be useful for different types of observed data. Also note that for the "standard" regression model the link function is $E(y_i|X_i) = \mu = X_i\beta$.

So, you are now well on par with *linear models* and *generalized linear models*.

6.3 Hierarchical (or mixed) models — LMMs and GLMMs

In reality, data is often not so simple. Especially web data often comes with a structure that is not easily dealt with using (generalized) linear models: Web data often comes in multiple levels or hierarchies, with some dependency structure between observations. For example, we might have the click behavior of individuals on a newsfeed article, $y \in \{0, 1\}$, which might depend on the number of friends who "liked" the article, $x \in \{0, \dots\}$. However, we can observe multiple interactions of the same person with newsfeed articles. Hence we get a data structure like: in which I am (not very realistically, but it serves an illustrative purpose) ignoring which newsfeed article people were actually looking at.

Now the question is, how do I deal with the "userid" in my analysis? One thing I can do (called pooling) would be to just ignore it. I fit $Pr(y_i = 1|X_i) = logit^{-1}(X_i\beta) = \frac{1}{1+e^{-X_i\beta}}$, and use that to generate (e.g.,) predictions as to whether or not a newsfeed with

y	x	userid
0	12	"aaa"
1	240	"aaa"
0	8	"bbb"
1	98	"aaa"
0	0	"ccc"
0	583	"xxx"
...	...	...
0	6	"aaa"

a certain number of likes will be clicked on (and hence should be selected to get more advertising money). But note that I am ignoring differences between people, and it could be that some user who showed up a lot but has awkward clicking behavior totally ruins my predictions.

I could also do something like this:

$$Pr(y_i = 1|X_i) = logit^{-1}(X_i\beta_{j[i]}) \tag{6.11}$$

Where I am using $\beta_{j[i]}$ as a notation that each individual j has his or her own $\vec{\beta}$. Basically this would mean doing separate regressions for each user. However, here I have the problem that at the level of a user, I might not have enough data to actually obtain proper estimates.

Thus, we would like some "intermediate" form. And one way to do this is to fit a multilevel (or hierarchical, or random effects) model. The basic idea here is to assume some distribution over the β_j's: that allows us to both use the information from the other people, but also model the people separately. These models are denoted in several (equivalent) ways:

1. First, we can write our models by emphasising that we allow the coefficients in a linear regression to vary by groups. For a "random intercept" model this leads to the following specification (in Matrix notation):

   $$y_i = \alpha_{j[i]} + X_i\beta + \epsilon_i. \tag{6.12}$$

   Here, X would include a row of 1's for the intercept, and the value of the intercept varies by group of observations j. The second level model is simply $\alpha_j \sim \mathcal{N}(\mu_\alpha, \sigma_\alpha^2)$. We could also write this second level model as $\alpha = \mu_\alpha + \eta_j$, with $\eta_j \sim \mathcal{N}(0, \sigma_\alpha^2)$.

2. We could also explicitly link the local regressions within each group. So with in a group j:

   $$y_i \sim \mathcal{N}(\alpha_j + \beta x_i, \sigma_y^2), \tag{6.13}$$

for $i = 1, \ldots, n_j$. Next, at the second level of the model we include group level predictors:

$$\alpha_i \sim \mathcal{N}(\gamma_0 + \gamma_1 \mu_j, \sigma_\alpha^2). \tag{6.14}$$

3. We can also model the whole thing as a single regression:

$$y_i \sim \mathcal{N}(X_i \beta, \sigma_y^2), \tag{6.15}$$

where we explicitly encode in X vectors corresponding to the *intercepts*, *the group level predictors*, and *indicators of each group*. Next, we specify $\beta_j \sim \mathcal{N}(0, \sigma_\alpha^2)$ for each of the j groups. *Why do we set the mean of the random term to* 0 *in this case?*.

4. Finally, we can write the model as a model with correlated errors. Here

$$y_i \sim \mathcal{N}(X_i \beta, \epsilon_i^{all}), \tag{6.16}$$

and $\epsilon^{all} \sim \mathcal{N}(0, \Sigma)$. Here X is again a matrix with predictors, but the erros ϵ_i^{all} have a covariance matrix $Sigma$. The error matrix is the same as the sum of the two errors at the two levels of the model in earlier specifications. Here the matrix Σ is structured as such that one of the terms is the same for each unit within a group. This last specification is not immediately obvious, and we will not discuss it too deeply: it is more informative to specify the multilevel structure explicitly than to hide it in a (complicated) error matrix. However, its good to know that this is possible (and that it is the most shorthand notation for the model).

Note that ridge regression (or Bayes regression) actually does something similar: it "ties together" the parameters β. The only difference here is that we have "batches" of parameters that are tied together (the β_j's) which are called random effects, and we have parameters that do not depend on each other: the so-called fixed effects. Obviously, random effects models can make use of link functions for different types of dependent outcomes y.

Stein, in 1965, proved that taking the grouped structure into account during estimation improves predictions in terms of average squared prediction error. This is due to the fact that the individual average – with multiple observations for individuals – as a predictor of someones true "ability" (or whichever trait that is measured) produces more prediction error variance than an estimator which takes the grouped structure of the data into account. This process of taking the group structure into account is commonly referred to as *shrinkage*. Shrinkage is a weighing between the individual average and the overall average. Because we make use of binary data, we refer to these proportions as $\bar{p}_j$ for the individual average and $\bar{p}$ for the average over all individuals. The extent to which the estimate of individual level effect is shrunk towards the overall average, $\bar{p}$, is determined by the shrink factor,b:

$$\hat{p}_j = b\bar{p} + (1 - b)\bar{p}_j \quad j = 1, \ldots, N, \tag{6.17}$$

where $\hat{p}_j$ is the estimated individual level effect for person j and N is equal to the total number of individuals. When b is equal to 1 (fully shrunk), the individual level effect is equal to the overall average, $\hat{p}_j = \bar{p}$, while the individual level effect is equal to the individual average, $\hat{p}_j = \bar{p}_j$, in the case that $b = 0$ (not shrunk). Different model specifications (or priors) on the distribution of the β_j's translate – when trying to predict values – to different shrink factors.

Figure 6.1 presents the general effect of shrinkage in a simple case when there are no features x: the individual averages are shrunk towards the overall average. Individual averages that are close to the overall average are hardly shrunk while the more extreme individual averages are shrunk more towards the overall average. While this introduces bias, it reduces the error variances.

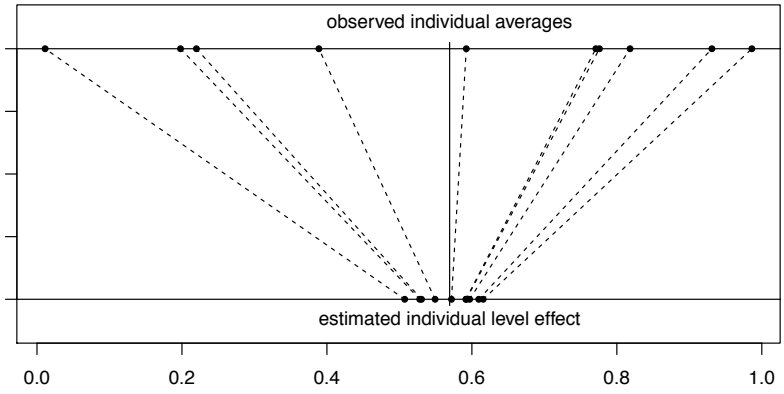


Figure 6.1: An illustration of shrinkage: At the top of the graph are the observed individual averages $\bar{p}_j$ and at the bottom of the graph are the estimated individual level effects $\hat{p}_j$ which are influenced by the other individuals. Each dashed line connects the observed individual average with the estimated individual level effect. The solid line in the middle of the graph indicates the overall average, $\bar{p}$.

6.3.1 A note on estimation

Hierarchical models are difficult to fit, since the likelihood contains "latent" (missing) variables (at the higher levels of the hierarchy we actually do not observer the variables directly). We will not go into detail into how hierarchal models are fit (e.g. how we compute the β's and the variance terms), but here are some remarks about it:

Multilevel models are more demanding to estimate than standard linear models (or generalized linear models). In the case of the Gaussian-Gaussian multilevel model such as the one-way random effects model, we can integrate out all levels of the hierarchy and are left with a likelihood for y which only depends on the fixed effects and the induced covariance structure. Maximum likelihood estimates for the coefficients on the fixed effects as well as the parameters of the variance components can be computed. However, with the maximum likelihood estimation of the standard linear model, the estimates of the parameters of the variance component are biased. This has led to the use of REstricted Maximum Likelihood (REML) methods for estimating variance components from a standard likelihood point of view.

At the heart of the REML method is a partition of the likelihood function into two pieces, one which is free of the fixed effects and one which depends on the fixed effects. The variance components are then estimated by maximizing the piece of the likelihood which only depends on the variance components and not the fixed effects. In the Gaussian-Gaussian setting, this step is achieved through an appropriate linear transformation of the data. The estimate of the variance component is then used when finding estimates of the fixed effects. This second step is equivalent to maximizing the entire likelihood conditioned upon the assumption that the variance component is equal to the one estimated using the restricted likelihood.

Even using the REML approach, computation can be burdensome and time-consuming and so the EM algorithm of Dempster, Laird, and Rubin is often employed when computing estimates. While REML uses a restricted likelihood, the EM algorithm expands the data into a complete data vector by appending some parameters (usually location parameters) to the end of the data vector. This expanded data vector now depends only on the remaining parameters (those of the variance component) and one iterates the Expectation and Maximization steps. In the M step, an MLE (REMLE) for the variance components is found conditioned on the current value of the fixed effects. In the E step, the expected values of the sufficient statistics for the fixed effects is found conditioned on the current value of the variance components.

Beyond these methods for computing ML and REML estimates, the Bayes paradigm offers powerful Markov chain Monte Carlo (MCMC) simulation tools for computing posterior statistics of interest. It has been shown that an empirical Bayes approach provides estimators that are equivalent to those from the REML method. In the Gaussian-Gaussian multilevel model, the hierarchical specifications and conjugate prior (a mathematically convenient form) for the parameters lead to a Gibbs sampler for the parameters of the model. In Gibbs sampling, the full conditional distributions of the parameters are sampled iteratively, producing a Markov chain that eventually converges such that

these conditional draws behave as if they are draws from the marginal posterior distributions for the parameters. When model prior specifications do not lend themselves well to Gibbs sampling, the Metropolis-Hastings algorithm (an iterative modification of rejection sampling) is utilized to produce samples from the appropriate posterior distribution. One possible complication in the utilization of the MCMC techniques is the dimensionality of the random effects. This has become a well-studied problem and there are a wide range of tools for addressing this problem.

In this course we will use "of the shelf" methods for fitting these types of models. However, do you should definitly understand how linear models (and generalized linear models) are estimated, and you should be able to look up – if need be – how the EM algorithm or the MCMC methods referred to above actually work.

You will find that for "web data" – very informally referring to very large datasets – EM and MCMC methods for multi level models are quite slow. Thus, it is an active area of research to work out how we can compute these things fast, how we can distribute the computation, etc.

6.4 Assignment

For this assignment we will examine a dataset provided by (Gelman and Hill, 2006) in Chapter 14. You can find a description of the datafile at:

http://www.stat.columbia.edu/ gelman/arm/examples/election88

The dataset is called "polls.subset.dat".

Please finish the following assignments:

- Let's first fit a very simple logistic model to predict voting based on people's age, ignoring the grouping factor state. What do you see?
  - You can use scikit-learn for Python to fit these types of models.
  - To get started, please run through the tutorial here: `http://nbviewer.ipython.org/github/justmarkham/gadsdc1/blob/master/logistic_assignment/kevin_logistic_sklearn.ipynb`
- We now want to turn to fitting random slopes for this model (e.g., fitting a hierarchical model). But, we run into a problem: Python has libraries available for fitting LMMs (Linear Mixed Models), but not GLMMs. However, we will go ahead anyway:
  - Install [R].

  - Get a hang of [R] for fitting GLMMs by following http://www.ats.ucla.edu/stat/r/dae/melogit.htm
  - Intall http://rpy.sourceforge.net
  - Try some simple example of using [R] code from python. For example try `x <- rnorm(10,0,1)` and return x to obtain random 10 random draws from a Normal distribution using [R] instead of Python.

- Now that we can use [R], we can use `lme4` to fit GLMM's! So, return to the "polls.subset.dat" dataset.
- Now let's examine variability between states. Fit a model with random intercepts per state, and nothing more (no fixed effects). Inspect the model and the estimated *variance of the random effect*. Is it large?
  - The lme4 comment would look like this: `lmer(bush  1 + (1 | state), data=poll, family=binomial(link="logit"))`.
- I guess that is where we will stop, since this is already a lot. However, you should know sickout-learn as a package to fit statistical models (and do a lot of nice Machine Learning stuff, and you should be aware of the opportunity to interface with other languages from python (for example [R]), which discloses all kinds of cool options.

Chapter 7

Lecture 7: SQL and No-SQL databases

After covering some basic methods and tools (networks, generalized linear models), and some of the web's historical AI / Machine learning algorithms (Naive Bayes, Recommender Systems, Pagerank), we are now turning to some more practical issues. Why? Because of you claim you know something about AI / data on the web, you need to not only learn a programming language that you can use for AI / data heavy applications on the web (which you have been doing all along, its called python), but you better also know practically how to deal with data.

Note that most of this lecture (and lecture 8) covers web tutorials on practical issues. I the lectures I will discuss the basic idea's behind the concepts covered. However, in the tutorials you should gain hands on experience: experience that you will need to complete the practicals. Some of you might have encountered these topics already, so my apologies if its boring. On the other hand, these things are really vital to make stuff work on the web.

The goals of this lecture are the following:

- You will learn about relational databases, the basic structure and basic queries.
- You will learn how to interface with MySQL using python.
- You will learn about non relational database structures, and their pro's and con's.
- You will learn how to use Python to interface with a Mongo (NoSQL) database.

7.1 Relational Database: An introduction

Beforehand: If you would like to learn more about relational databases, a good textbook is Database Management Systems, Third Edition, by Ramakrishnan and Gehrke.

A relational database is collection of tables (also called relations). A table is a collection of rows (also called tuples or records).

Each row in a table contains a set of columns (also called fields or attributes). Each column has a type:

- String: VARCHAR(20)
- Integer: INTEGER
- Floating-point: FLOAT, DOUBLE
- Date/time DATE, TIME, DATETIME
- Several others...

An important concept in a relational database is the *primary key*: this key provides a unique identifier for each row (need not be present, but almost always is in practice). Note that in relational databases rows are of a fixed size.

A *schema* contains the structure of the database, including for each table:

- The table name
- The names and types of its columns
- Various optional additional information (constraints, etc.)

SQL is a language for creating and manipulating relational databases. It was nitially created at IBM as part of System-R and is now implemented with modifications in numerous products: Oracle, Sybase, DB-2, SQL Server, MySQL, SQLite, etc. not all of these are completely compatible.

Lets look at some examples of the use of SQL to create databases: Let's create a table for student administration:

```
CREATE TABLE students (
    id INT AUTO_INCREMENT,
    name VARCHAR(30),
    birth DATE,
    gpa FLOAT,
    grad INT,
    PRIMARY KEY(id));
```

and, let's add rows to the students table:

```
INSERT INTO students(name, birth, gpa, grad)
     VALUES ('Anderson', '1987-10-22', 3.9, 2009);
INSERT INTO students(name, birth, gpa, grad)
     VALUES ('Jones', '1990-4-16', 2.4, 2012);
INSERT INTO students(name, birth, gpa, grad)
    VALUES ('Hernandez', '1989-8-12', 3.1, 2011);
INSERT INTO students(name, birth, gpa, grad)
    VALUES ('Chen', '1990-2-4', 3.2, 2011);
```

Delete row(s):

```
DELETE FROM students WHERE name='Anderson';
```

Delete table:

```
DROP TABLE students;
```

7.1.1 Queries

Once you have a bunch of data in a relational database, which is often composed of many tables, the real core of what you can do is write queries. Writing queries is an acquired skill, and really you will learn it by practicing it and doing things, however, here we cover some basic examples:

Show entire contents of a table:

```
SELECT * FROM students;
```

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-04-16	2.4	2012
3	Hernandez	1989-08-12	3.1	2011
4	Chen	1990-02-04	3.2	2011

Show just a few columns from a table:

```
SELECT name, gpa FROM students;
```

name	gpa
Anderson	3.9
Jones	2.4
Hernandez	3.1
Chen	3.2

Filtering: only display a subset of the rows:

```
SELECT name, gpa
    FROM students
    WHERE gpa > 3.0;
+-------------+-------+
| name        | gpa   |
+-------------+-------+
| Anderson    |   3.9 |
| Hernandez   |   3.1 |
| Chen        |   3.2 |
+-------------+-------+
```

Sorting:

```
SELECT gpa, name, grad
    FROM students
    WHERE gpa > 3.0
    ORDER BY gpa DESC;
+-------+-------------+-------+
| gpa   | name        | grad  |
+-------+-------------+-------+
|   3.9 | Anderson    | 2009  |
|   3.2 | Chen        | 2011  |
|   3.1 | Hernandez   | 2011  |
+-------+-------------+-------+
```

Updates:

```
UPDATE students
    SET gpa = 2.6, grad = 2013
    WHERE id = 2;
```

Joins

Joins are a way to manage relationships between tables. A join is a query that merges the contents of 2 or more tables, and displays information from the results. Joins can produce the equivalent of a linked list in a programming language, and many other effects.

Join example: many-to-one relationships

Students have advisors; add new table describing faculty.

```
+----+-----------+------------+
| id | name      | title      |
+----+-----------+------------+
|  1 | Fujimura  | assocprof  |
|  2 | Bolosky   | prof       |
+----+-----------+------------+
```

Add new column `advisor_id` to the students table. This is a *foreign key.*

id	name	birth	gpa	grad	advisor_id
1	Anderson	1987-10-22	3.9	2009	2
2	Jones	1990-04-16	2.4	2012	1
3	Hernandez	1989-08-12	3.1	2011	1
4	Chen	1990-02-04	3.2	2011	1

Example query:

```
SELECT s.name, s.gpa
    FROM students s, advisors p
    WHERE s.advisor_id = p.id AND p.name = 'Fujimura';
```

name	gpa
Jones	2.4
Hernandez	3.1
Chen	3.2

A join creates the cross-product of 2 or more tables. This is (potentially) very computationally expensive! However, in practice, joins are optimized carefully by the database system.

Join example: many-to-many relationship

Courses: students take many courses, courses have many students Add new table describing courses:

id	number	name	quarter
1	CS142	Web stuff	Winter 2009
2	ART101	Finger painting	Fall 2008
3	ART101	Finger painting	Winter 2009
4	PE204	Mud wrestling	Winter 2009

Create a join table `courses_students` describing which students took which courses.

course_id	student_id
1	1
3	1
4	1
1	2

```
|           2 |           2 |
|           1 |           3 |
|           2 |           4 |
|           4 |           4 |
+-------------+-------------+
```

Find all students who took a particular course:

```
SELECT s.name, c.quarter
    FROM students s, courses c, courses_students cs
    WHERE c.id = cs.course_id AND s.id = cs.student_id
    AND c.number = 'ART101';
+-----------+---------------+
| name      | quarter       |
+-----------+---------------+
| Jones     | Fall 2008     |
| Chen      | Fall 2008     |
| Anderson  | Winter 2009   |
+-----------+---------------+
```

7.1.2 Notes

A few final remarks of things you should be aware of but we will not cover in detail. First, tables have *Indexes*: these are used to speed up queries. Second, there is a concept called *Transactions*, these are used to group operations together to provide predictable behavior even when there are concurrent operations on the database. Finally, there is a lot more to be learned on how to scale and speed up a SQL database for specific problems, but we won't get into all of the details...

7.2 MySQL and Python

After covering some basic SQL concepts, we will dig into using python to interface with MySQL. Here I just list the basic steps and a number of examples.

If you do not already have MySQL installed, we must install it.

```
$ sudo apt-get install mysql-server
```

This command installs the MySQL server and various other packages. While installing the package, we are prompted to enter a password for the MySQL root account.

```
$ apt-cache search MySQLdb
```

A few useful packages:

- python-mysqldb - A Python interface to MySQL
- python-mysqldb-dbg - A Python interface to MySQL (debug extension)
- bibus - bibliographic database
- eikazo - graphical frontend for SANE designed for mass-scanning

Here we install the Python interface to the MySQL database. Both _mysql and MySQL modules:

```
$ sudo apt-get install python-mysqldb
```

Next, we are going to create a new database user and a new database. We use the mysql client.

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 30
Server version: 5.0.67-0ubuntu6 (Ubuntu)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
+--------------------+
2 rows in set (0.00 sec)
```

We connect to the database using the root account. We show all available databases with the SHOW DATABASES statement.

```
mysql> CREATE DATABASE testdb;
Query OK, 1 row affected (0.02 sec)
```

We create a new testdb database. We will use this database throughout the tutorial.

```
mysql> CREATE USER 'testuser'@'localhost' IDENTIFIED BY 'test623';
Query OK, 0 rows affected (0.00 sec)

mysql> USE testdb;
Database changed

mysql> GRANT ALL ON testdb.* TO 'testuser'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> quit;
Bye
```

7.2.1 MySQLdb module

MySQLdb is a thin Python wrapper around _mysql. It is compatible with the Python DB API, which makes the code more portable. Using this model is the preferred way of working with the MySQL.

First example

In the first example, we will get the version of the MySQL database.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb
import sys

try:
    con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

    cur = con.cursor()
    cur.execute("SELECT VERSION()")

    ver = cur.fetchone()

    print "Database version : %s " % ver

except mdb.Error, e:

    print "Error %d: %s" % (e.args[0],e.args[1])
    sys.exit(1)

finally:

    if con:
        con.close()
```

In this script, we connect to the testdb database and execute the SELECT VERSION() statement. This will return the current version of the MySQL database. We print it to the console.

```
$ ./version.py
Database version : 5.5.9
```

Creating and populating a table

We create a table and populate it with some data.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

with con:

    cur = con.cursor()
    cur.execute("DROP TABLE IF EXISTS Writers")
    cur.execute("CREATE TABLE Writers(Id INT PRIMARY KEY AUTO_INCREMENT, \
                 Name VARCHAR(25))")
    cur.execute("INSERT INTO Writers(Name) VALUES('Jack London')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Honore de Balzac')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Lion Feuchtwanger')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Emile Zola')")
    cur.execute("INSERT INTO Writers(Name) VALUES('Truman Capote')")
```

We create a "Writers" table and add five authors to it. After executing the script, we can use the mysql client tool to select all data from the Writers table.

```
mysql> SELECT * FROM Writers;
+----+--------------------+
| Id | Name               |
+----+--------------------+
|  1 | Jack London        |
|  2 | Honore de Balzac   |
|  3 | Lion Feuchtwanger  |
|  4 | Emile Zola         |
|  5 | Truman Capote      |
+----+--------------------+
5 rows in set (0.00 sec)
```

Retrieving data

Now that we have inserted some data into the database, we want to get it back.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

with con:

    cur = con.cursor()
```

```
    cur.execute("SELECT * FROM Writers")

    rows = cur.fetchall()

    for row in rows:
        print row
```

In this example, we retrieve all data from the Writers table. Note the location of the SQL query. Obviously here you can use all the fancy joins etc. that we discussed previously.

```
cur.execute("SELECT * FROM Writers")
```

This SQL statement selects all data from the Writers table.

```
rows = cur.fetchall()
```

The fetchall() method gets all records. It returns a result set. Technically, it is a tuple of tuples. Each of the inner tuples represent a row in the table.

Returning all data at a time may not be feasible. We can fetch rows one by one.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb');

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Writers")

    for i in range(cur.rowcount):

        row = cur.fetchone()
        print row[0], row[1]
```

We again print the data from the Writers table to the console. This time, we fetch the rows one by one.

The dictionary cursor

There are multiple cursor types in the MySQLdb module. The default cursor returns the data in a tuple of tuples. When we use a dictionary cursor, the data is sent in a form of Python dictionaries. This way we can refer to the data by their column names.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor(mdb.cursors.DictCursor)
    cur.execute("SELECT * FROM Writers LIMIT 4")

    rows = cur.fetchall()

    for row in rows:
        print row["Id"], row["Name"]
```

In this example, we get the first four rows of the Writers table using the dictionary cursor.

Column headers

Next we will show, how to print column headers with the data from the database table.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()
    cur.execute("SELECT * FROM Writers LIMIT 5")

    rows = cur.fetchall()

    desc = cur.description

    print "%s %3s" % (desc[0][0], desc[1][0])

    for row in rows:
        print "%2s %3s" % row
```

Again, we print the contents of the Writers table to the console. Now, we include the names of the columns too. The column names are considered to be the 'meta data'. It is

obtained from the cursor object.

Prepared statements

Now we will concern ourselves with prepared statements. When we write prepared statements, we use placeholders instead of directly writing the values into the statements. Prepared statements increase security and performance. The Python DB API specification suggests 5 different ways how to build prepared statements. The MySQLdb module supports one of them, the ANSI printf format codes.

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb as mdb

con = mdb.connect('localhost', 'testuser', 'test623', 'testdb')

with con:

    cur = con.cursor()

    cur.execute("UPDATE Writers SET Name = %s WHERE Id = %s",
        ("Guy de Maupasant", "4"))

    print "Number of rows updated:",  cur.rowcount
```

We change the name of the author we are looking for dynamically.

This was only a very basic example of the possibilities of Python in combination with MySQL. You will have a chance to play with these when you are doing the assignments.

7.3 Non relational databases (NoSQL): Introduction

NoSQL is not a tool, but an ecosystem composed of several complimentary and competing tools. The tools branded with the NoSQL monicker provide an alternative to SQL-based relational database systems for storing data. To understand NoSQL, we have to understand the space of available tools, and see how the design of each one explores the space of data storage possibilities.

If you are considering using a NoSQL storage system, you should first understand the wide space of options that NoSQL systems span. NoSQL systems do away with many of the traditional comforts of relational database systems, and operations which were typically

encapsulated behind the system boundary of a database are now left to application designers. This requires you to take on the hat of a systems architect, which requires a more in-depth understanding of how such systems are built.

7.3.1 What's in a Name?

In defining the space of NoSQL, let's first take a stab at defining the name. Taken literally, a NoSQL system presents a query interface to the user that is not SQL. The NoSQL community generally takes a more inclusive view, suggesting that NoSQL systems provide alternatives to traditional relational databases, and allow developers to design projects which use *Not Only* a SQL interface. In some cases, you might replace a relational database with a NoSQL alternative, and in others you will employ a mix-and-match approach to different problems you encounter in application development.

Before diving into the world of NoSQL, let's explore the cases where SQL and the relational model suit your needs, and others where a NoSQL system might be a better fit.

7.3.2 SQL and the Relational Model

SQL is a declarative language for querying data. A declarative language is one in which a programmer specifies *what* they want the system to do, rather than procedurally defining *how* the system should do it. A few examples include: find the record for employee 39, project out only the employee name and phone number from their entire record, filter employee records to those that work in accounting, count the employees in each department, or join the data from the employees table with the managers table.

To a first approximation, SQL allows you to ask these questions without thinking about how the data is laid out on disk, which indices to use to access the data, or what algorithms to use to process the data. A significant architectural component of most relational databases is a *query optimizer*, which decides which of the many logically equivalent query plans to execute to most quickly answer a query. These optimizers are often better than the average database user, but sometimes they do not have enough information or have too simple a model of the system in order to generate the most efficient execution.

Relational databases, which are the most common databases used in practice, follow the *relational data model*. In this model, different real-world entities are stored in different tables. For example, all employees might be stored in an Employees table, and all departments might be stored in a Departments table. Each row of a table has various properties stored in columns. For example, employees might have an employee id, salary, birth date, and first/last names. Each of these properties will be stored in a column of the Employees table.

The relational model goes hand-in-hand with SQL. Simple SQL queries, such as filters, retrieve all records whose field matches some test (e.g., employeeid = 3, or salary ¿ $20000). More complex constructs cause the database to do some extra work, such as joining data from multiple tables (e.g., what is the name of the department in which employee 3 works?). Other complex constructs such as aggregates (e.g., what is the average salary of my employees?) can lead to full-table scans.

The relational data model defines highly structured entities with strict relationships between them. Querying this model with SQL allows complex data traversals without too much custom development. The complexity of such modeling and querying has its limits, though:

- Complexity leads to unpredictability. SQL's expressiveness makes it challenging to reason about the cost of each query, and thus the cost of a workload. While simpler query languages might complicate application logic, they make it easier to provision data storage systems, which only respond to simple requests.
- There are many ways to model a problem. The relational data model is strict: the schema assigned to each table specifies the data in each row. If we are storing less structured data, or rows with more variance in the columns they store, the relational model may be needlessly restrictive. Similarly, application developers might not find the relational model perfect for modeling every kind of data. For example, a lot of application logic is written in object-oriented languages and includes high-level concepts such as lists, queues, and sets, and some programmers would like their persistence layer to model this.
- If the data grows past the capacity of one server, then the tables in the database will have to be partitioned across computers. To avoid JOINs having to cross the network in order to get data in different tables, we will have to denormalize it. Denormalization stores all of the data from different tables that one might want to look up at once in a single place. This makes our database look like a key-lookup storage system, leaving us wondering what other data models might better suit the data.

It's generally not wise to discard many years of design considerations arbitrarily. When you consider storing your data in a database, consider SQL and the relational model, which are backed by decades of research and development, offer rich modeling capabilities, and provide easy-to-understand guarantees about complex operations. NoSQL is a good option when you have a specific problem, such as large amounts of data, a massive workload, or a difficult data modeling decision for which SQL and relational databases might not have been optimized.

7.3.3 NoSQL Inspirations

The NoSQL movement finds much of its inspiration in papers from the research community. While many papers are at the core of design decisions in NoSQL systems, two stand out in particular.

Google's BigTable presents an interesting data model, which facilitates sorted storage of multi-column historical data. Data is distributed to multiple servers using a hierarchical range-based partitioning scheme, and data is updated with strict consistency (a concept that we will eventually define in).

Amazon's Dynamo uses a different key-oriented distributed datastore. Dynamo's data model is simpler, mapping keys to application-specific blobs of data. The partitioning model is more resilient to failure, but accomplishes that goal through a looser data consistency approach called eventual consistency.

We will dig into each of these concepts in more detail, but it is important to understand that many of them can be mixed and matched. Some NoSQL systems such as HBase[1] sticks closely to the BigTable design. Another NoSQL system named Voldemort[2] replicates many of Dynamo's features. Still other NoSQL projects such as Cassandra[3] have taken some features from BigTable (its data model) and others from Dynamo (its partitioning and consistency schemes).

7.3.4 Characteristics and Considerations

NoSQL systems part ways with the hefty SQL standard and offer simpler but piecemeal solutions for architecting storage solutions. These systems were built with the belief that in simplifying how a database operates over data, an architect can better predict the performance of a query. In many NoSQL systems, complex query logic is left to the application, resulting in a data store with more predictable query performance because of the lack of variability in queries

NoSQL systems part with more than just declarative queries over the relational data. Transactional semantics, consistency, and durability are guarantees that organizations such as banks demand of databases. *Transactions* provide an all-or-nothing guarantee when combining several potentially complex operations into one, such as deducting money from one account and adding the money to another. *Consistency* ensures that when a value is updated, subsequent queries will see the updated value. *Durability* guarantees that once a

[1] http://hbase.apache.org/
[2] http://project-voldemort.com/
[3] http://cassandra.apache.org/

value is updated, it will be written to stable storage (such as a hard drive) and recoverable if the database crashes.

NoSQL systems relax some of these guarantees, a decision which, for many non-banking applications, can provide acceptable and predictable behavior in exchange for improved performance. These relaxations, combined with data model and query language changes, often make it easier to safely partition a database across multiple machines when the data grows beyond a single machine's capability.

NoSQL systems are still very much in their infancy. The architectural decisions that go into the systems described in this chapter are a testament to the requirements of various users. The biggest challenge in summarizing the architectural features of several open source projects is that each one is a moving target.

As you think about NoSQL systems, here is a roadmap of considerations:

- Data and query model: Is your data represented as rows, objects, data structures, or documents? Can you ask the database to calculate aggregates over multiple records?
- Durability: When you change a value, does it immediately go to stable storage? Does it get stored on multiple machines in case one crashes?
- Scalability: Does your data fit on a single server? Do the amount of reads and writes require multiple disks to handle the workload?
- Partitioning: For scalability, availability, or durability reasons, does the data need to live on multiple servers? How do you know which record is on which server?
- Consistency: If you've partitioned and replicated your records across multiple servers, how do the servers coordinate when a record changes?
- Transactional semantics: When you run a series of operations, some databases allow you to wrap them in a transaction, which provides some subset of ACID (Atomicity, Consistency, Isolation, and Durability) guarantees on the transaction and all others currently running. Does your business logic require these guarantees, which often come with performance tradeoffs?
- Single-server performance: If you want to safely store data on disk, what on-disk data structures are best-geared toward read-heavy or write-heavy workloads? Is writing to disk your bottleneck?
- Analytical workloads: We're going to pay a lot of attention to lookup-heavy workloads of the kind you need to run a responsive user-focused web application. In many cases, you will want to build dataset-sized reports, aggregating statistics across multiple users for example. Does your use-case and toolchain require such functionality?

7.3.5 NoSQL Data and Query Models

The *data model* of a database specifies how data is logically organized. Its *query model* dictates how the data can be retrieved and updated. Common data models are the relational model, key-oriented storage model, or various graph models. Query languages you might have heard of include SQL, key lookups, and MapReduce. NoSQL systems combine different data and query models, resulting in different architectural considerations.

Key-based NoSQL Data Models

NoSQL systems often part with the relational model and the full expressivity of SQL by restricting lookups on a dataset to a single field. For example, even if an employee has many properties, you might only be able to retrieve an employee by her ID. As a result, most queries in NoSQL systems are key lookup-based. The programmer selects a key to identify each data item, and can, for the most part, only retrieve items by performing a lookup for their key in the database.

In key lookup-based systems, complex join operations or multiple-key retrieval of the same data might require creative uses of key names. A programmer wishing to look up an employee by his employee ID and to look up all employees in a department might create two key types. For example, the key `employee:30` would point to an employee record for employee ID 30, and `employee_departments:20` might contain a list of all employees in department 20. A join operation gets pushed into application logic: to retrieve employees in department 20, an application first retrieves a list of employee IDs from key `employee_departments:20`, and then loops over key lookups for each `employee:ID` in the employee list.

The key lookup model is beneficial because it means that the database has a consistent query pattern—the entire workload consists of key lookups whose performance is relatively uniform and predictable. Profiling to find the slow parts of an application is simpler, since all complex operations reside in the application code. On the flip side, the data model logic and business logic are now more closely intertwined, which muddles abstraction.

Let's quickly touch on the data associated with each key. Various NoSQL systems offer different solutions in this space.

Key-Value Stores

The simplest form of NoSQL store is a *key-value* store. Each key is mapped to a value containing arbitrary data. The NoSQL store has no knowledge of the contents of its payload, and simply delivers the data to the application. In our Employee database example, one

might map the key `employee:30` to a blob containing JSON or a binary format such as Protocol Buffers[4], Thrift[5], or Avro[6] in order to encapsulate the information about employee 30.

If a developer uses structured formats to store complex data for a key, she must operate against the data in application space: a key-value data store generally offers no mechanisms for querying for keys based on some property of their values. Key-value stores shine in the simplicity of their query model, usually consisting of `set`, `get`, and `delete` primitives, but discard the ability to add simple in-database filtering capabilities due to the opacity of their values. Voldemort, which is based on Amazon's Dynamo, provides a distributed key-value store. BDB[7] offers a persistence library that has a key-value interface.

Key-Data Structure Stores

Key-data structure stores, made popular by Redis[8], assign each value a type. In Redis, the available types a value can take on are integer, string, list, set, and sorted set. In addition to `set`/`get`/`delete`, type-specific commands, such as increment/decrement for integers, or push/pop for lists, add functionality to the query model without drastically affecting performance characteristics of requests. By providing simple type-specific functionality while avoiding multi-key operations such as aggregation or joins, Redis balances functionality and performance.

Key-Document Stores

Key-document stores, such as CouchDB[9], MongoDB[10], and Riak[11], map a key to some document that contains structured information. These systems store documents in a JSON or JSON-like format. They store lists and dictionaries, which can be embedded recursively inside one-another.

MongoDB[12] separates the keyspace into collections, so that keys for Employees and Department, for example, do not collide. CouchDB and Riak leave type-tracking to the developer.

[4] http://code.google.com/p/protobuf/
[5] http://thrift.apache.org/
[6] http://avro.apache.org/
[7] http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html
[8] http://redis.io/
[9] http://couchdb.apache.org/
[10] http://www.mongodb.org/
[11] http://www.basho.com/products_riak_overview.php
[12] We will be working with MongoDB.

The freedom and complexity of document stores is a double-edged sword: application developers have a lot of freedom in modeling their documents, but application-based query logic can become exceedingly complex.

BigTable Column Family Stores

HBase and Cassandra base their data model on the one used by Google's BigTable. In this model, a key identifies a row, which contains data stored in one or more Column Families (CFs). Within a CF, each row can contain multiple columns. The values within each column are timestamped, so that several versions of a row-column mapping can live within a CF.

Conceptually, one can think of Column Families as storing complex keys of the form (row ID, CF, column, timestamp), mapping to values which are sorted by their keys. This design results in data modeling decisions which push a lot of functionality into the keyspace. It is particularly good at modeling historical data with timestamps. The model naturally supports sparse column placement since row IDs that do not have certain columns do not need an explicit NULL value for those columns. On the flip side, columns which have few or no NULL values must still store the column identifier with each row, which leads to greater space consumption.

Each project data model differs from the original BigTable model in various ways, but Cassandra's changes are most notable. Cassandra introduces the notion of a supercolumn within each CF to allow for another level of mapping, modeling, and indexing. It also does away with a notion of locality groups, which can physically store multiple column families together for performance reasons.

Graph Storage

One class of NoSQL stores are graph stores. Not all data is created equal, and the relational and key-oriented data models of storing and querying data are not the best for all data. Graphs are a fundamental data structure in computer science, and systems such as HyperGraphDB[13] and Neo4J[14] are two popular NoSQL storage systems for storing graph-structured data. Graph stores differ from the other stores we have discussed thus far in almost every way: data models, data traversal and querying patterns, physical layout of data on disk, distribution to multiple machines, and the transactional semantics of queries. We can not do these stark differences justice given space limitations, but you should be aware that certain classes of data may be better stored and queried as a graph.

[13] http://www.hypergraphdb.org/index
[14] http://neo4j.org/

Complex Queries

There are notable exceptions to key-only lookups in NoSQL systems. MongoDB allows you to index your data based on any number of properties and has a relatively high-level language for specifying which data you want to retrieve. BigTable-based systems support scanners to iterate over a column family and select particular items by a filter on a column. CouchDB allows you to create different views of the data, and to run MapReduce tasks across your table to facilitate more complex lookups and updates. Most of the systems have bindings to Hadoop or another MapReduce framework to perform dataset-scale analytical queries.

Schema-free Storage

A cross-cutting property of many NoSQL systems is the lack of schema enforcement in the database. Even in document stores and column family-oriented stores, properties across similar entities are not required to be the same. This has the benefit of supporting less structured data requirements and requiring less performance expense when modifying schemas on-the-fly. The decision leaves more responsibility to the application developer, who now has to program more defensively. For example, is the lack of a `lastname` property on an employee record an error to be rectified, or a schema update which is currently propagating through the system? Data and schema versioning is common in application-level code after a few iterations of a project which relies on *sloppy-schema* NoSQL systems.

7.3.6 Python and Mongo

You see that there are many database systems out there, and choices will depend on your problem at hand, your experience, etc. However, here is an example using MongoDB with Python.

Before we start, make sure that you have the PyMongo distribution installed. In the Python shell, the following should run without raising an exception:

```
>>> import pymongo
```

This tutorial also assumes that a MongoDB instance is running on the default host and port. Assuming you have downloaded and installed MongoDB, you can start it like so:

```
$ mongod
```

Making a Connection with MongoClient

The first step when working with PyMongo is to create a MongoClient to the running mongod instance. Doing so is easy:

```
>>> from pymongo import MongoClient
>>> client = MongoClient()
```

The above code will connect on the default host and port. We can also specify the host and port explicitly, as follows:

```
>>> client = MongoClient('localhost', 27017)
```

Or use the MongoDB URI format:

```
>>> client = MongoClient('mongodb://localhost:27017/')
```

Getting a Database

A single instance of MongoDB can support multiple independent databases. When working with PyMongo you access databases using attribute style access on MongoClient instances:

```
>>> db = client.test_database
```

If your database name is such that using attribute style access wont work (like test-database), you can use dictionary style access instead:

```
>>> db = client['test-database']
```

Getting a Collection

A collection is a group of documents stored in MongoDB, and can be thought of as roughly the equivalent of a table in a relational database. Getting a collection in PyMongo works the same as getting a database:

```
>>> collection = db.test_collection
```

or (using dictionary style access):

```
>>> collection = db['test-collection']
```

An important note about collections (and databases) in MongoDB is that they are created lazily - none of the above commands have actually performed any operations on the MongoDB server. Collections and databases are created when the first document is inserted into them.

Documents

Data in MongoDB is represented (and stored) using JSON-style documents. In PyMongo we use dictionaries to represent documents. As an example, the following dictionary might be used to represent a blog post:

```
>>> import datetime
>>> post = {"author": "Mike",
...         "text": "My first blog post!",
...         "tags": ["mongodb", "python", "pymongo"],
...         "date": datetime.datetime.utcnow()}
```

Note that documents can contain native Python types (like datetime.datetime instances) which will be automatically converted to and from the appropriate BSON types.

Inserting a Document

To insert a document into a collection we can use the insert() method:

```
>>> posts = db.posts
>>> post_id = posts.insert(post)
>>> post_id
ObjectId('...')
```

When a document is inserted a special key, "_id", is automatically added if the document doesnt already contain an "_id" key. The value of "_id" must be unique across the collection. insert() returns the value of "_id" for the inserted document. For more information, see the documentation on _id.

After inserting the first document, the posts collection has actually been created on the server. We can verify this by listing all of the collections in our database:

```
>>> db.collection_names()
[u'system.indexes', u'posts']
```

Note The system.indexes collection is a special internal collection that was created automatically.

Getting a Single Document With find_one()

The most basic type of query that can be performed in MongoDB is find_one(). This method returns a single document matching a query (or None if there are no matches). It is useful when you know there is only one matching document, or are only interested in the first match. Here we use find_one() to get the first document from the posts collection:

```
>>> posts.find_one()
{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'_id':
    ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'
    pymongo']}
```

The result is a dictionary matching the one that we inserted previously.

Note The returned document contains an "_id", which was automatically added on insert. find_one() also supports querying on specific elements that the resulting document must match. To limit our results to a document with author Mike we do:

```
>>> posts.find_one({"author": "Mike"})
{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'_id':
    ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'
    pymongo']}
```

If we try with a different author, like Eliot, well get no result:

```
>>> posts.find_one({"author": "Eliot"})
>>>
```

Querying By ObjectId

We can also find a post by its _id, which in our example is an ObjectId:

```
>>> post_id
ObjectId(...)
>>> posts.find_one({"_id": post_id})
{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'_id':
    ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'
    pymongo']}
```

Note that an ObjectId is not the same as its string representation:

```
>>> post_id_as_str = str(post_id)
>>> posts.find_one({"_id": post_id_as_str}) # No result
>>>
```

A common task in web applications is to get an ObjectId from the request URL and find the matching document. Its necessary in this case to convert the ObjectId from a string before passing it to find_one:

```
from bson.objectid import ObjectId

# The web framework gets post_id from the URL and passes it as a string
def get(post_id):
    # Convert from string to ObjectId:
    document = client.db.collection.find_one({'_id': ObjectId(post_id)})
```

A Note On Unicode Strings

You probably noticed that the regular Python strings we stored earlier look different when retrieved from the server (e.g. uMike instead of Mike). A short explanation is in order.

MongoDB stores data in BSON format. BSON strings are UTF-8 encoded so PyMongo must ensure that any strings it stores contain only valid UTF-8 data. Regular strings (¡type str¿) are validated and stored unaltered. Unicode strings (¡type unicode¿) are encoded UTF-8 first. The reason our example string is represented in the Python shell as uMike instead of Mike is that PyMongo decodes each BSON string to a Python unicode string, not a regular str.

Bulk Inserts

In order to make querying a little more interesting, lets insert a few more documents. In addition to inserting a single document, we can also perform bulk insert operations, by passing an iterable as the first argument to insert(). This will insert each document in the iterable, sending only a single command to the server:

```
>>> new_posts = [{"author": "Mike",
...               "text": "Another post!",
...               "tags": ["bulk", "insert"],
...               "date": datetime.datetime(2009, 11, 12, 11, 14)},
...              {"author": "Eliot",
...               "title": "MongoDB is fun",
...               "text": "and pretty easy too!",
...               "date": datetime.datetime(2009, 11, 10, 10, 45)}]
>>> posts.insert(new_posts)
[ObjectId('...'), ObjectId('...')]
```

There are a couple of interesting things to note about this example:

- The call to insert() now returns two ObjectId instances, one for each inserted document.
- new_posts[1] has a different shape than the other posts - there is no "tags" field and weve added a new field, "title".
- This is what we mean when we say that MongoDB is schema-free.

Querying for More Than One Document

To get more than a single document as the result of a query we use the find() method. find() returns a Cursor instance, which allows us to iterate over all matching documents. For example, we can iterate over every document in the posts collection:

```
>>> for post in posts.find():
...     post
...
{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'_id':
    ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'
    pymongo']}
{u'date': datetime.datetime(2009, 11, 12, 11, 14), u'text': u'Another post!',
     u'_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'bulk', u'insert
    ']}
{u'date': datetime.datetime(2009, 11, 10, 10, 45), u'text': u'and pretty easy
     too!', u'_id': ObjectId('...'), u'author': u'Eliot', u'title': u'MongoDB
     is fun'}
```

Just like we did with find_one(), we can pass a document to find() to limit the returned results. Here, we get only those documents whose author is Mike:

```
>>> for post in posts.find({"author": "Mike"}):
...     post
...
{u'date': datetime.datetime(...), u'text': u'My first blog post!', u'_id':
    ObjectId('...'), u'author': u'Mike', u'tags': [u'mongodb', u'python', u'
    pymongo']}
{u'date': datetime.datetime(2009, 11, 12, 11, 14), u'text': u'Another post!',
     u'_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'bulk', u'insert
    ']}
```

Counting

If we just want to know how many documents match a query we can perform a count() operation instead of a full query. We can get a count of all of the documents in a collection:

```
>>> posts.count()
3
```

or just of those documents that match a specific query:

```
>>> posts.find({"author": "Mike"}).count()
2
```

Further notes

MongoDB supports many different types of advanced queries. As an example, lets perform a query where we limit results to posts older than a certain date, but also sort the results by author:

```
>>> d = datetime.datetime(2009, 11, 12, 12)
>>> for post in posts.find({"date": {"$lt": d}}).sort("author"):
...     print post
...
{u'date': datetime.datetime(2009, 11, 10, 10, 45), u'text': u'and pretty easy
     too!', u'_id': ObjectId('...'), u'author': u'Eliot', u'title': u'MongoDB
     is fun'}
{u'date': datetime.datetime(2009, 11, 12, 11, 14), u'text': u'Another post!',
     u'_id': ObjectId('...'), u'author': u'Mike', u'tags': [u'bulk', u'insert
    ']}
```

Here we use the special "$lt" operator to do a range query, and also call sort() to sort the results by author.

To make the above query fast we can add a compound index on "date" and "author". To start, lets use the explain() method to get some information about how the query is being performed without the index:

```
>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["cursor"]
u'BasicCursor'
>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["nscanned"]
3
```

We can see that the query is using the BasicCursor and scanning over all 3 documents in the collection. Now lets add a compound index and look at the same information:

```
>>> from pymongo import ASCENDING, DESCENDING
>>> posts.create_index([("date", DESCENDING), ("author", ASCENDING)])
u'date_-1_author_1'
>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["cursor"]
u'BtreeCursor date_-1_author_1'
>>> posts.find({"date": {"$lt": d}}).sort("author").explain()["nscanned"]
2
```

Now the query is using a BtreeCursor (the index) and only scanning over the 2 matching documents.

See also The MongoDB documentation on indexes.

Finally, Mongo support Map/Reduce: a very general way of specifying “queries” that are easily distributed. We will dig into Map/Reduce more in the next lecture.

There are a number of online tutorials I can suggest (some of which inspired the current material):

- http://www.tutorialspoint.com/mysql/mysql-introduction.htm
- http://zetcode.com/db/mysqlpython/
- http://www.hongkiat.com/blog/webdev-with-mongodb-part1/
- http://docs.mongodb.org/ecosystem/drivers/python/
- http://api.mongodb.org/python/2.0/examples/map_reduce.html
- http://www.w3schools.com/json/

7.4 Assignment

- SQL assignments:
  - Install MySQL on your machine
  - Use the MySQL command line interface to create 2 tables. Let one table obtain the names of 10 students, their age, and their study major. Let the other table contain 50 rows which describe for each student what their grades were on 5 different exams.
  - Write a python script to retrieve the mean and standard deviation of all student jointly, and of each student individually.
  - NOTE: You are free to pick names and grades etc. yourself. This is just an exercise.
- MongoDB assignments:
  - Simulate a dataset, using the regression methods we covered in the previous lectures, which contains data of 10.000 houses. The data should contain the number of square feet, the number of rooms, an (incremental) ID for the house, and the price (the latter of which you simulate by making a model that predict prices based on the square feet and the number of rooms.
  - Store the data in MongoDB: how do you do it and why?
  - Add a short description to 10 of the houses.
  - Select all houses that have a description.

  - Write the python code to select all houses with a price higher then x, and a number of rooms y. Inspect how the query is carried out.

- Done? Make sure to explore MongoDB in more detail. How would you store the data we used in Lecture 2 (spam classification) in MongoDB?

Chapter 8

Lecture 8: Map/Reduce & REST API's

This lecture wraps up our 2-session section on practicalities for the use of AI on the web. We have covered SQL and No-SQL databases in the previous lecture, and you were briefly introduced to queries. This lecture we cover two slightly more advanced (and distinct!) topics: (1) Map / Reduce, and (2) REST API's.

The topics are distinct, but both are vital to the current day Web infrastructure and for the use of heavy AI techniques on the web. The first, Map / Reduce (or shortly MR) provides a method of dealing with extremely large (and often distributed) datasets. The second, (REST) API's, provide a method of sending around data and events between servers: this is vital for obvious reasons.

This lecture should teach you:

- The basic ideas behing Map / Reduce (MR)
- The use of MR when using Mongo DB
- You should be able to implement simple MR scripts
- You will know what (REST) API's are.
- You will be able to send around JSON objects (and thus data) using REST API's.

8.1 Map Reduce Basics

Naïve approaches designed for traditional amounts of data do not typically scale to big data. (e.g. invertibility of matrices, curse of dimensionality, etc.)

Usually we can use some combination of the following methods to scale to big data:

1. Assuming that the data has inherently lower-dimensional structure
    - Sparsity
    - Conditional independence (so that we can handle things separately)
2. Fast algorithms
    - Parallelization
    - Typically linear time algorithms or better
3. Methodology that avoids the need to fit the "full" data
    - Consensus Monte Carlo
    - Bag of Little Bootstraps

However, even once we have a method that is able to scale to big data, we have to find out a way to access the huge or complex data sets with practical ways. Storing stuff in Python memory will hardly work as our data grows larger and larger and our web applications serve intelligent functionality to millions of users all over the Globe. So, we need solutions to deal with (extremely) large and often distributed datasets.

A method that has become extremely popular over the last few years to deal with extremely large data is called Map / Reduce. This is what we will cover in this lecture. Note that we will only cover the basics of MR: more info is easily found online, but we don't have the time to really dig into it. Also note that MR solves particular large scale data problems very well (it allows for easy parallelization of analysis tasks), but not others (it is not very good for "real-time" analysis). So despite the fact that we are covering MR in this lecture, not that there is more in live than MR.

There are two-steps to programming a MapReduce program:

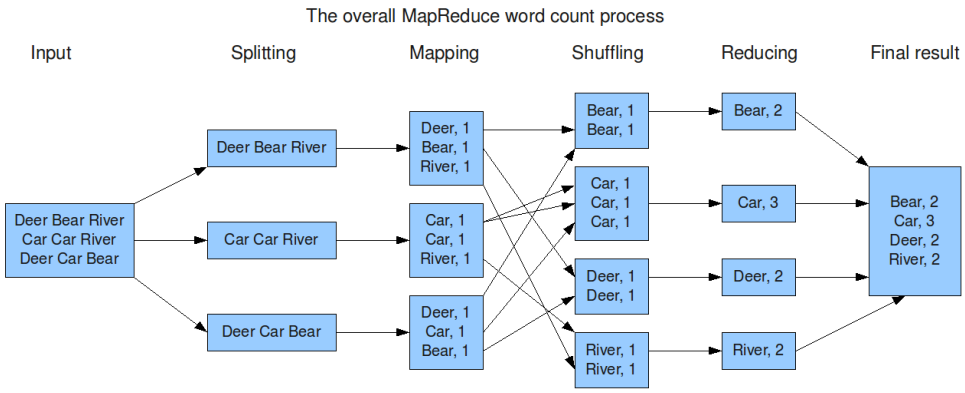
1. Map: For every data element, a function is applied to that element, and it returns a (key, value) pair. So, the mapper "transverses" all DB records (or documents), and "emits" part of these data using a key value pair.

2. Reduce: For every element with the same key, a function is applied to combine the values. So, the reducer "receives" all emits with the same key, and a **list of all values submitted for that key.**

There could be potentially millions or billions of such pairs! You could take the sum of the values, or any useful statistic.

Note that any MR application will do a bunch of stuff "under the hood": the emitted key's will be sorted and shuffled, the whole task will be distributed over nodes, etc. etc. But, the nice thing is that you don't have to worry about these things. However, it is good to have some idea of what happens. Figure 8.1 provides a schematic overview of the processes that happen when using MR for counting unique words. We will cover this example in more detail below.

Figure 8.1: Overview of the MR process for word count.



8.1.1 Example 1: The classic word count.

Word count is the "hello world" of the user of MR. Suppose we want to count the number of times distinct words appear in a large set of text documents.

Let's suppose we are working with only 1 document for now, containing the following text:

```
Angry Bob was angry that little Bob was angry at big Bob.
```

For the *map* step we need to decide on what (key, value) pairs to emit. For each word, emit: (word, 1). This will result in the following key-value pairs that are emitted:

```
(Angry, 1)
(Bob, 1)
(was, 1)
(angry, 1)
(that, 1)
(little, 1)
(Bob, 1)
(was, 1)
(angry, 1)
(at, 1)
(big, 1)
(Bob, 1)
```

The code would look something like this:

```
map(String key, String value):
    //    key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

Next, for the reduce step, (conceptually) all key-value pairs with the same key (i.e. the same words) are combined. The reduce function receives data like this:

```
(Angry, 1)
----------
(Bob, 1)
(Bob, 1)
(Bob, 1)
----------
(was, 1)
(was, 1)
----------
(angry, 1)
(angry, 1)
----------
...
```

We need a reduce function to apply to the set of values within each unique key. Here, that is just a sum. The function looks like this:

```
reduce(String key, Iterator values):
    //    key: a word
    // values: a list of counts
    int word_count = 0;
    for each v in values:
        word_count += ParseInt(v);
    Emit(key, AsString(word_count));
```

If you've really been paying attention, you might be asking yourself: Why didn't we simply use sum = values.length?. This would seem like an efficient approach when you are essentially summing an array of 1s. The fact is that reduce isn't always called with a full and perfect set of intermediate date. As such, reduce must always be idempotent. Eventually, it will output something like:

```
(Angry, 1)
(Bob, 3)
(was, 2)
(angry, 2)
(that, 1)
(little, 1)
(at, 1)
(big, 1)
```

Which means we have now counted all the distinct words in this document using MR. I hope that gives you somewhat a gist of how MR works.

We will now turn to an example using MongoDB and Python.

8.1.2 Map Reduce Using MognoDB and Python.

This example shows how to use the `map_reduce` method to perform map/reduce style aggregations on data. This is again a word count, but now using MongoDB and Python.

To start, well insert some example data which we can perform map/reduce queries on:

```
>>> from pymongo import Connection
>>> db = Connection().map_reduce_example
>>> db.things.insert({"x": 1, "tags": ["dog", "cat"]})
ObjectId('...')
>>> db.things.insert({"x": 2, "tags": ["cat"]})
ObjectId('...')
>>> db.things.insert({"x": 3, "tags": ["mouse", "cat", "dog"]})
ObjectId('...')
>>> db.things.insert({"x": 4, "tags": []})
ObjectId('...')
```

Now well define our map and reduce functions. In this case were performing the same operation as in the MongoDB Map/Reduce documentation - counting the number of occurrences for each tag in the tags array, across the entire collection.

Our map function just emits a single (key, 1) pair for each tag in the array:

```
>>> from bson.code import Code
>>> map = Code("function () {"
...            "  this.tags.forEach(function(z) {"
...            "    emit(z, 1);"
```

```
...             "  });"
...             "}")
```

The reduce function sums over all of the emitted values for a given key:

```
>>> reduce = Code("function (key, values) {"
...                "  var total = 0;"
...                "  for (var i = 0; i < values.length; i++) {"
...                "    total += values[i];"
...                "  }"
...                "  return total;"
...                "}")
```

Note again that we cant just return values.length as the reduce function might be called iteratively on the results of other reduce steps.

Finally, we call `map_reduce` and iterate over the result collection:

```
>>> result = db.things.map_reduce(map, reduce, "myresults")
>>> for doc in result.find():
...   print doc
...
{u'_id': u'cat', u'value': 3.0}
{u'_id': u'dog', u'value': 2.0}
{u'_id': u'mouse', u'value': 1.0}
```

Note that there are many more things your can do, we are really just scratching the surface here.

8.2 REST API's

This section of the course is a bit odd, since it does not really deal with AI, however it does deal with how data is handled on the web, and how you can send data from one server or machine to the other. Obviously this is something you might often use in real life applications: you might have a mobile phone that is measuring data from a user, and that data needs to be send to a bunch of webservers to be analyzed using MR, before it forwarded to the users watch to display the results. Despite the fact that there are many many options to create such functionality, we will briefly dig into REST API's here as a method to make these things work. Note by the way that I am assuming familiarity with JSON (a data structure, see www.json.org). The content we are discussing here is partly taken from http://code.tutsplus.com/tutorials/a-beginners-guide-to-http-and-rest--net-16340

In recent years REST (REpresentational State Transfer) has emerged as the standard architectural design for web services and web APIs.

8.2.1 What is REST?

The characteristics of a REST system are defined by six design rules:

- Client-Server: There should be a separation between the server that offers a service, and the client that consumes it.
- Stateless: Each request from a client must contain all the information required by the server to carry out the request. In other words, the server cannot store information provided by the client in one request and use it in another request.
- Cacheable: The server must indicate to the client if requests can be cached or not.
- Layered System: Communication between a client and a server should be standardized in such a way that allows intermediaries to respond to requests instead of the end server, without the client having to do anything different.
- Uniform Interface: The method of communication between a client and a server must be uniform.
- Code on demand: Servers can provide executable code or scripts for clients to execute in their context. This constraint is the only one that is optional.

8.2.2 What is a RESTful web service?

The REST architecture was originally designed to fit the HTTP protocol that the world wide web uses.

Central to the concept of RESTful web services is the notion of resources. Resources are represented by URIs. The clients send requests to these URIs using the methods defined by the HTTP protocol, and possibly as a result of that the state of the affected resource changes.

The HTTP request methods are typically designed to affect a given resource in standard ways:

The REST design does not require a specific format for the data provided with the requests. In general data is provided in the request body as a *JSON* blob, or sometimes as arguments in the query string portion of the URL.

8.2.3 Designing a simple web service

The task of designing a web service or API that adheres to the REST guidelines then becomes an exercise in identifying the resources that will be exposed and how they will be

HTTP Method	Action	Examples
GET	Obtain information about a resource	http://example.com/api/orders (retrieve order list
GET	Obtain information about a resource	http://example.com/api/orders/123 (retrieve orde
POST	Create a new resource	http://example.com/api/orders (create a new or data provided with the request)
PUT	Update a resource	http://example.com/api/orders/123 (update ord from data provided with the request)
DELETE	Delete a resource	http://example.com/api/orders/123 (delete order

affected by the different request methods.

Let's say we want to write a To Do List application and we want to design a web service for it. The first thing to do is to decide what is the root URL to access this service. For example, we could expose this service as:

```
http://[hostname]/todo/api/v1.0/
```

Here I have decided to include the name of the application and the version of the API in the URL. Including the application name in the URL is useful to provide a namespace that separates this service from others that can be running on the same system. Including the version in the URL can help with making updates in the future, since new and potentially incompatible functions can be added under a new version, without affecting applications that rely on the older functions.

The next step is to select the resources that will be exposed by this service. This is an extremely simple application, we only have tasks, so our only resource will be the tasks in our to do list.

Our tasks resource will use HTTP methods as follows:

HTTP Method	URI	Action
GET	http://[hostname]/todo/api/v1.0/tasks	Retrieve list of tasks
GET	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Retrieve a task
POST	http://[hostname]/todo/api/v1.0/tasks	Create a new task
PUT	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Update an existing task
DELETE	http://[hostname]/todo/api/v1.0/tasks/[task_id]	Delete a task

We can define a task as having the following fields:

- id: unique identifier for tasks. Numeric type.
- title: short task description. String type.

- description: long task description. Text type.
- done: task completion state. Boolean type.

And with this we are basically done with the design part of our web service. All that is left is to implement it! (Note that in this lecture we will only discuss implementations of the GET requests.)

8.2.4 A brief introduction to the Flask microframework

If you read my Flask Mega-Tutorial series you know that Flask is a simple, yet very powerful Python web framework. Before we delve into the specifics of web services let's review how a regular Flask web application is structured.

You should obviously start by installing Flask. .... Great!

Now that we have Flask installed let's create a simple web application, which we will put in a file called app.py:

```
#!flask/bin/python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

if __name__ == '__main__':
    app.run(debug=True)
```

To run this application we have to execute app.py:

```
$ chmod a+x app.py
$ ./app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

And now you can launch your web browser and type

```
http://localhost:5000
```

to see this tiny application in action. Simple, right? Now we will convert this app into our RESTful service!

8.2.5 Implementing RESTful services in Python and Flask

Building web services with Flask is surprisingly simple. There are a couple of Flask extensions that help with building RESTful services with Flask, but the task is so simple that in my opinion there is no need to use an extension.

The clients of our web service will be asking the service to add, remove and modify tasks, so clearly we need to have a way to store tasks. The obvious way to do that is to build a small database, but because databases are not the topic of this lecture we are going to take a much simpler approach. In place of a database we will store our task list in a memory structure. This will only work when the web server that runs our application is single process and single threaded. This is okay for Flask's own development web server. It is not okay to use this technique on a production web server, for that a proper database setup must be used.

Using the base Flask application we are now ready to implement the first entry point of our web service:

```
#!flask/bin/python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the web',
        'done': False
    }
]

@app.route('/todo/api/v1.0/tasks', methods=['GET'])
def get_tasks():
    return jsonify({'tasks': tasks})

if __name__ == '__main__':
    app.run(debug=True)
```

As you can see, not much has changed. We created a memory database of tasks, which is nothing more than a plain and simple array of dictionaries. Each entry in the array has

the fields that we defined above for our tasks.

Instead of the index entry point we now have a `get_tasks` function that is associated with the `/todo/api/v1.0/tasks` URI, and only for the GET HTTP method.

The response of this function is not text, we are now replying with JSON data, which Flask's jsonify function generates for us from our data structure.

Using a web browser to test a web service isn't the best idea since web browsers cannot easily generate all types of HTTP requests. Instead, we will use curl. If you don't have curl installed, go ahead and install it now.

Start the web service in the same way we started the sample application, by running `app.py`. Then open a new console window and run the following command:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 294
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 04:53:53 GMT

{
  "tasks": [
    {
      "description": "Milk, Cheese, Pizza, Fruit, Tylenol",
      "done": false,
      "id": 1,
      "title": "Buy groceries"
    },
    {
      "description": "Need to find a good Python tutorial on the web",
      "done": false,
      "id": 2,
      "title": "Learn Python"
    }
  ]
}
```

We just have invoked a function in our RESTful service!

Now let's write the second version of the GET method for our tasks resource. If you look at the table above this will be the one that is used to return the data of a single task:

```
from flask import abort

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods=['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
```

```
        abort(404)
    return jsonify({'task': task[0]})
```

This second function is a little bit more interesting. Here we get the id of the task in the URL, and Flask translates it into the `task_id` argument that we receive in the function.

With this argument we search our tasks array. If the id that we were given does not exist in our database then we return the familiar error code 404, which according to the HTTP specification means "Resource Not Found", which is exactly our case.

If we find the task then we just package it as JSON with jsonify and send it as a response, just like we did before for the entire collection.

Here is how this function looks when invoked from curl:

```
$ curl -i http://localhost:5000/todo/api/v1.0/tasks/2
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 151
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:50 GMT

{
  "task": {
    "description": "Need to find a good Python tutorial on the web",
    "done": false,
    "id": 2,
    "title": "Learn Python"
  }
}

$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: text/html
Content-Length: 238
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:21:52 GMT

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2 Final//EN">
<title>404 Not Found</title>
<h1>Not Found</h1>
<p>The requested URL was not found on the server.</p><p>If you      entered
    the URL manually please check your spelling and try again.</p>
```

When we ask for resource id #2 we get it, but when we ask for #3 we get back the 404 error. The odd thing about the error is that it came back with an HTML message instead of JSON, because that is how Flask generates the 404 response by default. Since this is a

web service client applications will expect that we always respond with JSON, so we need to improve our 404 error handler:

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify({'error': 'Not found'}), 404)
And we get a much more API friendly error response:

$ curl -i http://localhost:5000/todo/api/v1.0/tasks/3
HTTP/1.0 404 NOT FOUND
Content-Type: application/json
Content-Length: 26
Server: Werkzeug/0.8.3 Python/2.7.3
Date: Mon, 20 May 2013 05:36:54 GMT

{
  "error": "Not found"
}
```

Done.

For this lecture I can again recommend a number of online tutorials:

- http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask
- http://rest.elkstein.org
- http://atbrox.com/2010/02/08/parallel-machine-learning-for-hadoopmapreduce-a-python-
- https://github.com/elsevierlabs/logistic-regression-sgd-mapreduce

For a more formal intro to Map Reduce see (Chu et al., 2007).

8.3 Assignment

Please finish the following assignments:

- Implement linear regression using MR by following http://nerdslearning.wordpress.com/2012/12/04/building-linear-regression-with-mapreduce-on-hadoop/. However, make sure to implement things in Python / MongoDB.
- In the lecture we have "implemented" the two GET calls to the REST API described in Table 8.2.3. Implement the remaining calls yourself.

Chapter 9

Lecture 9: Introduction to streaming or online data analysis

We have covered quite a bit by now, but we have not really explicitly dealt with the fact that on the Web data often comes in continuously instead of in batches as most people are used to. Thus, often we do not "have" a dataset that we can use for analysis, but rather we observe things happening all the time. In this lecture we will make a start with what is called *streaming* or *online* analysis: methods to deal with true data-streams. Note in advance that I think that the jargon is a bit inconsistent here and there, and I will use the words streaming, online, or in summation form in quite a sloppy fashion. In Computer Science explicit definitions exist, as is true in statistics (but these differ...). This lecture however intends to give you a gist of the thing.

This lecture should teach you:

- What online / streaming analysis is, and why its useful.
- Which tools you can use for streaming AI solutions
- How to fit / estimate some standard learning algorithms in data streams
- Reason about the computational and space complexities of streaming algorithms (although not very formally)

9.1 Online / Streaming Analysis, a brief intro

Before we start we need some idea of what online (or streaming) analysis is. So, here we are:

"Online estimation algorithms estimate the parameters of a model when new data is available during the operation of the model. In contrast, if you first collect all the input/output data and then estimate the model parameters, you perform offline estimation. Parameter values estimated using online estimation can vary with time, but parameters estimated using offline estimation do not."

This definition might not be super intuitive, and might not always be the way in which the term(s) are actually used. So let me give a slightly more informal description with a simple counting example to get us up to speed with what we are looking at.

Suppose we have datapoints $x_1, \ldots, x_t$ arriving one by one in a data stream of length T ($T > 0, t \leq T$). Here with data stream we intend to say that we observe the datapoints one by one, that T is possibly very very large, and that it – for all practical purposes – is never finished. Thus, the data keeps coming in, and there is no one point at which we stop to analyze it. We just need to deal with the fact that there is more and more coming.

How do we do this? Well, one of the options to deal with data streams is to simply add *more computational power*. Algorithms could be programmed such that the analyses run simultaneously on multiple cores, known as parallelizing. Therefore, parallelizing algorithms can greatly reduce computation time. Map/Reduce, which we discussed in the previous lecture, provides a method to efficiently parallelize computations. However, the problem of how to deal with additional data entering actually remains.

Another option is to only take the most recent part of the data into account. This is known colloquially as a *sliding window* approach. Sliding window methods take only the most recent n data points into account for fitting statistical models. Using a sliding window reduces the burden of the computer memory since it is fixed to the computation required to analyze the n data points regardless of the actual length of the stream. When new data enter, the window shifts, excluding the oldest data points. In this manner only a limited amount of memory is required to conduct the analyses. An advantage of the sliding window approach is that the researcher can, in advance, determine the amount of computational resources since she herself determines the size of the window. This method however has disadvantages as well; 1) all information of the data previous to the window is forgotten, which is especially problematic when modeling rare events, and 2) a sliding window approach needs domain experience to determine how large the window should be for the current application.

Finally, we can resort to *online learning* (which is often used in combination with the

first option). This means that instead of storing all (or a part of) data points, the data is summarized into a limited set in parameters which take all relevant information of previous data points into account. Contrary to a sliding windows approach, online learning algorithms estimate parameters based on all data available in the stream. As you might have noticed by the title of this lecture, we focus on this latter one.

9.1.1 A pet example of online learning

To get a bit of feel for this idea of online (or streaming) learning, consider one of the most basic operations on data we could think of: summing. Thus, we are interested in the sum $\mathcal{S}(t)$ of all datapoints $x_1, \ldots, x_t$, and nothing else.

How do we compute this? Basically there are two versions:

$$\mathcal{S}(t) = \sum_{i=1}^{t} x_i$$

or

$$\mathcal{S}(t) = \mathcal{S}(t-1) + x_t$$

The first version has the advantage that you can find a sum at each $t' < t$ (obviously you cannot find a sum of objects you have not yet observed). However, it has the (large) drawback that each time you want to know $\mathcal{S}(t)$ you will be iterating through your whole dataset. The second approach will only tell you the sum at t, but comes with the (sometimes huge) advantage that $\mathcal{S}(t)$ is available directly, in memory (note that you would store $x_1, \ldots, x_t$ in computer memory for version one, and only $\mathcal{S}(t-1)$ for version 2).

Version one of our sum will grow increasingly complex as t increases. It will take up more and more memory, and the time to compute the sum will become longer and longer. However, version two of our summing algorithm has a fixed usage of memory (one scalar), and a fixed update time.

There are pro's and con's of each approach. And, to be honest the example is a bit too simple: already the Version one algorithm requires only a single pass through the data, and that is often not too hard (and we can use MR for this!). Also, version one is already in *summation* form, and thus it is easy to see how we mover from version 1 to 2. However, once we need multiple iterations through a dataset to obtain our estimates, or have estimation methods which are not in summation form, finding streaming (or online, or row-by-row) methods is not always easy.

9.1.2 Tools for online or streaming analysis

Before we dig into the streaming computation of some basic statistics and some well-known machine learning models, we briefly look at the existing software packages for streaming analysis:

- Twitter's *Storm*: At some point Twitter open-sources their platform for quick processing. It can be found here: https://storm.apache.org, and you can use all kinds of programming languages to talk to it. Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing.
- RStorm http://cran.r-project.org/package=RStorm a package to model Storm streams in [R], useful for debugging and development.
- Initially originating from Yahoo, there is also *S4*: "a general-purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to easily develop applications for processing continuous unbounded streams of data." See http://incubator.apache.org/s4/.
- There is also *Apache Spark*: which "is a fast and general engine for large-scale data processing." See https://spark.apache.org.
- And many many more...

We will not be covering a specific software package, we will primarily focus on the underlying principles of streaming analysis and on some of the algorithms.

9.2 Online or streaming algorithms

In this section we will discuss a number of online algorithms. In a general sense we are looking for "summation form" descriptions of standard estimators and models. So, given that our task is to estimate θ (possibly a vector), we are restricting ourselves to algorithms that can be noted down as follows:

$$\theta_{t+1} = f(\theta_t, x_t) \tag{9.1}$$

or more concise: $\theta := f(\theta, x_t)$.

9.2.1 Sample mean

Suppose we want to estimate the sample mean of $x_1, \ldots, x_t$. Its standard formulation would be $\bar{x} = \sum_{i=1}^{t} x_i/t$. It is trivial to write $\bar{X}$ in the form of Equation 9.1:

$$S := S + x_t$$
$$n := n + 1$$

where $\theta = \{S, n\}$, and $\bar{x} = S/n$.

With this, it might seem that implementations of the streaming estimation of a mean are trivial. However, the term S_t is ever growing in t and might "explode". This might lead to numerical problems if $t \to \infty$. Furthermore, in the above specification $\bar{x}$ is not immediately available.

Alternatively one could specify:

$$\bar{x} := \bar{x} + (x_t - \bar{x})/(n+1)$$
$$n := n + 1$$

where $\theta = \{\bar{x}, n\}$. This latter method is more stable numerically

9.2.2 Sample Variance

Now, let's estimate the sample variance $\sigma^2 = \frac{1}{t-1}\sum_{i=1}^{t}(x_i - \bar{x})^2$, where $\bar{x}$ is the sample mean. Non-streaming computation of σ^2 is often done using the so called two-pass algorithm: first, the sample mean is computed and second the sample variance is computed. However, using the sum of squares we can compute a fully streaming sample variance:

$$n := n_t + 1$$
$$S := S + x_t$$
$$SS := SS + x_t^2$$

where $\theta = \{SS, S, n\}$ and the sample variance at any point t is given by $\sigma_t^2 = 1/(n(n-1))(nSS - S^2)$.

Similar to the sample mean case, the above method of computing the sample variance quickly suffers from numerical problems as $t \to \infty$. Numerical instability is especially problematic when when $\sigma^2 << \mu$. Alternatively we can compute the streaming variance

using:

$$
\begin{aligned}
n &:= n + 1 \\
\bar{x} &:= x_t + (x_t - \bar{x})/(n+1) \\
S &:= S + (x_t - \bar{x})(x_t - \bar{x})
\end{aligned}
$$

where $\theta = \{S, \bar{x}, n\}$ and which conveniently gives both $\bar{x}$ and $\sigma^2 = S/(n-1)$ in a single stream and is more stable numerically.

We will work out the streaming computation of the sample covariance during the lecture.

9.2.3 Linear regression

After estimating means and variances, a next logical step is to estimate a simple linear regression model (LMs). Here we regard a single event z_t a vector containing components $(y_t, x_{1t}, \ldots, x_{kt})$ where y_t is the state of the dependent variable at time t which is to be predicted by k independent variables (or features) $x_{1t}, \ldots, x_{kt}$. In the non-streaming analysis of the linear regression model the observations $y_{1,\ldots,t}$ are predicted as a linear function of the independent variables which, discarding t since we are analysing the full data set, can be written as $y \sim \mathcal{N}\left(\beta_0 + \beta_1 x_1 + \cdots + \beta_k x_k, \sigma^2_{err}\right)$. The parameters of primary interest are the regression weights $\beta_{1,\ldots,k}$. Traditionally these are obtained using the Normal Equations that we discussed previously:

$$
\hat{\beta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{9.2}
$$

where $\mathbf{X}$ is a $t \times k$ matrix with all observations on all explanatory variables k, and $\mathbf{y}$ is a vector of length t with all the observations on the response variable. This solution minimises the squared-error loss function. We can write this in the form of Equation **??** by stating

$$
\begin{aligned}
A &:= A + \mathbf{x}_t\mathbf{x}_t^T \\
b &:= b + \mathbf{x}_t y_t
\end{aligned}
$$

where $\theta = \{A, b\}$, and $\hat{\beta} = A^{-1}b$.

9.2.4 And a few more...

Now that we have covered some of the real basics, here are a number of descriptions of other types of models and their implementations (note that these are based on (Chu et al., 2007)):

- *Locally Weighted linear regression:* LWLR is solved by finding the solution of the normal equations $A\theta = b$, where

$$A = \sum_{i=1}^{t} w_i(x_i x_i^T)$$
$$b = \sum_{i=1}^{t} w_i(x_i y_i)$$

  Note that this is just linear regression as discussed above if $w_i = 1$.

- *Naive Bayes:* In NB (discussed in lecture 1), we have to estimate $P(x_j = k|y = 1)$, $P(x_j = k|y = 0)$, and $P(y)$ from the training data. In order to do so, we need to sum over $x_j = k$ for each y label in the training data to calculate $P(x|y)$. We can do this using a number of sums of indicator functions:

$$\theta_{j1} := \theta_{j1} + \mathbb{1}x_j = k|y = 1$$
$$\theta_{j0} := \theta_{j0} + \mathbb{1}x_j = k|y = 0$$
$$\theta_1 := \theta_1 + \mathbb{1}y = 1$$
$$\theta_0 := \theta_0 + \mathbb{1}y = 0$$

  (Where I leave it up to you to work out the exact details).

- *Principal component Analysis (PCA):* PCA computes the principle eigenvectors of the covariance matrix $\Sigma = \frac{1}{t}\left(\sum_{i=1}^{t} x_i x_i^T\right) - \mu\mu^T$. Clearly, the first term is already in summation form. The mean vector μ can also be computed in summation form: $\mu = \frac{1}{t}\sum_{i=1}^{t} x_i$.

You will be able to find many more "streaming" or "online" or "in summation form" algorithms for standard AI techniques (often approximations). However, I think you get the idea. Formally, we are looking for a number of data sufficient statistics θ, and some update rule of these to estimate our models.

9.3 Complexity considerations

It is important to understand why streaming / online analysis is more feasible computationally (despite being often less straightforward to compute or approximate). There are

really two conceptual things that change:

- *Memory:* In non-streaming methods we often save all observations $\vec{x}_1, \ldots, \vec{x}_t$. Thus, as t grows, the memory usage grows. In steaming methods we only store θ, with dimension k, and we assume $k << t$.
- *Computation:* The real crux in computation is that classical methods revisit $\vec{x}_1, \ldots, \vec{x}'_t$ if estimates are needed at t'. If this is often, then the datapoints will be visited many-many times to compute the quantities of interest. With online learning, the datapoints are only visited once.

You can find more info on online learning and streaming estimation here (Opper and Winther, 1998; Alpcan and Bauckhage, 2009; Bottou, 1998; Gaber et al., 2005), and in many other places...

9.4 Assignment

Since there are no more "tutorials" scheduled at this point, the following assignments are basically voluntary. However, I would suggest you give them a try.

- Implement a streaming computation of a variance in Python.
- Implement a streaming linear regression in Python.
- Compare, when you would like to make a prediction at each point in time t, the difference in computational and memory between streaming and non-streaming linear regression.

Chapter 10

Lecture 10: Stochastic Gradient Descent (SGD)

In the previous lecture we discussed streaming / online estimation and analysis methods. In this lecture we will discuss stochastic gradient descent as a general online estimation method, and look at two instances (logistic regression and regularized logistic regression).

This lecture should teach you:

- What stochastic gradient descent (SGD) is, and why its useful for online estimation.
- How to use SGD to fit logistic regression at a large scale.
- How to use SGD to fit generalized logistic regression.

10.1 (Stochastic) Gradient Descent

Both statistical estimation and machine learning often consider the problem of minimizing an objective function that has the form of a sum:

$$Q(w) = \sum_{i=1}^{n} Q_i(w)$$

where the parameter w is to be estimated and where typically each summand function $Q_i()$ is associated with the i-th observation in the data set (used for training).

In classical statistics, sum-minimization problems arise in least squares and in maximum-likelihood estimation (for independent observations). The general class of estimators that arise as minimizers of sums are called M-estimators.

When used to minimize the above function, a standard (or "batch") gradient descent method would perform the following iterations :

$$w := w - \alpha \nabla Q(w) = w - \alpha \sum_{i=1}^{n} \nabla Q_i(w)$$

where α is a step size (or learning rate), and $\nabla Q(w)$ is the gradient of the objective function. Basically we update the parameters w by making a step "in the direction of the gradient" of the objective function. The term $\alpha \sum_{i=1}^{n} \nabla Q_i(w)$ is a summation over all data points, and is (sometimes) expensive to carry out. Especially in data streams, this might be infeasible.

10.1.1 Online Gradient Descent

In stochastic (or "on-line") gradient descent, the true gradient of Q(w) is approximated by a gradient at a single example:

$$w := w - \alpha \nabla Q_i(w)$$

As the algorithm sweeps through the training set, or data stream, it performs the above update for each training example. On a static dataset one can make several passes over the training set until the algorithm converges. In a stream we assume the data to be random draws of the underlying process and thus we can motivate convergence (for stationary processes) in a similar fashion. Typical implementations may use an adaptive learning rate so that the algorithm converges.

In pseudocode, stochastic gradient descent can be presented as follows:

```
Choose an initial vector of parameters w and learning rate $\alpha$.
Randomly shuffle examples in the training set.
Repeat until an approximate minimum is obtained:
For i=1, 2, ..., n, do:
        w := w - alpha nabla Q_i(w).
```

A compromise between the two forms called "mini-batches" computes the gradient against more than one training examples at each step. This can perform significantly better than true stochastic gradient descent because the code can make use of vectorization libraries rather than computing each step separately. It may also result in smoother convergence, as the gradient computed at each step uses more training examples.

The convergence of stochastic gradient descent has been analyzed using the theories of convex minimization and of stochastic approximation. When the learning rates α decrease with an appropriate rate, and subject to relatively mild assumptions, stochastic gradient descent converges almost surely to a global minimum when the objective function is convex or pseudoconvex, and otherwise converges almost surely to a local minimum.[1]

10.1.2 An example of SGD: (simple) Linear regression

Let's suppose we want to fit a straight line $y = w_1 + w_2 x$ to a training set of two-dimensional points $(x_1, y_1), \ldots, (x_n, y_n)$ using least squares.

The objective function to be minimized is:

$$Q(w) = \sum_{i=1}^{n} Q_i(w) = \sum_{i=1}^{n} (w_1 + w_2 x_i - y_i)^2 .$$

The last line in the above pseudocode for this specific problem will become:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix} := \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \alpha \begin{bmatrix} 2(w_1 + w_2 x_i - y_i) \\ 2x_i(w_1 + w_2 x_i - y_i) \end{bmatrix} .$$

10.2 SGD for Logistic Regression

One fairly simple way (and extremely scalable way) to implement logistic regression is using stochastic gradient descent (for which actually no closed-form solution exists, unlike the linear regression example above).

We estimate the probability p that an example $\vec{x} = \langle x_1, \ldots, x_d \rangle$ in the log-odds form (using a logit link):

$$\log \frac{p}{1-p} = \alpha + \sum_{j=1} d\beta_j x_j$$

We can merge the intercept α, into the β's by including a feature x_0 with only 1's. Then:

$$p = \frac{\exp \beta^T \vec{x}}{1 + \exp \beta^T \vec{x}}$$

[1] Learning rates are often assumed to satisfy the following conditions: $\sum_{i=1}^{T} \alpha_i = \infty$, and $\sum_{i=1}^{T} \alpha_i^2 < \infty$.

It's convenient to consider examples of the form $(\vec{x}, y)$ where $y = 0$ or $y = 1$. The log of the conditional likelihood is

$$\mathcal{L}(\vec{x}, y) = \log p$$

if $y = 1$ and

$$\mathcal{L}(\vec{x}, y) = \log(1 - p)$$

if $y = 0$, where p is computed as above. With a little calculus you can show that for a positive example,

$$\frac{\partial}{\partial \beta_j}\mathcal{L}(\vec{x}, y) = \frac{1}{p}\frac{\partial}{\partial \beta_j}p$$

and for a negative example,

$$\frac{\partial}{\partial \beta_j}\mathcal{L}(\vec{x}, y) = \frac{1}{1-p}(-\frac{\partial}{\partial \beta_j}p)$$

and that

$$\frac{\partial}{\partial \beta_j}p = p(1 - p)x_j$$

and putting this together we get that if $y = 1$

$$\frac{\partial}{\partial \beta_j}\mathcal{L}(\vec{x}, y) = (1 - p)x_j$$

and if $y = 0$ then

$$\frac{\partial}{\partial \beta_j}\mathcal{L}(\vec{x}, y) = -px_j$$

so in either case

$$\frac{\partial}{\partial \beta_j}\mathcal{L}(\vec{x}, y) = (y - p)x_j \tag{10.1}$$

So an update to the β's that would improve most would be along the gradient—i.e., for some small step size λ, let

$$\beta_j = \beta_j + \lambda(y - p)x_i$$

Notice that if $x_i = 0$ then β_j is unchanged.

So this leads to this streaming algorithm, which is very fast (assuming you have enough memory to hash all the parameter values).

```
Initialize a hashtable B
For i=1, ...,t
   For each non-zero feature:
      If feature j is not in B, set B[j]=0.
      Set B[j] = B[j] + lambda (y-p) x_i
Output the parameters B.
```

10.3 Efficient regularized logistic regression using SGD

Logistic regression tends to overfit when there are many rare features. One fix is to penalize large values of β, by optimizing, instead of $\mathcal{L}$, some function such as $\mathcal{L} - \mu \sum_{j=1}^{d} \beta_j^2$ Here μ controls how much weight to give to the penalty term. The update for β_j becomes

$$\beta_j = \beta_j + \lambda((y - p)x_i - 2\mu\beta_j)$$

or equivalently

$$\beta_j = \beta_j + \lambda(y - p)x_i - \lambda 2\mu\beta_j$$

Experimentally this greatly improves overfitting - but unfortunately, this makes the computation more expensive, because now *every* β_j needs to be updated, not only the ones that are non-zero. However, it can already be done in a data stream.

One trick to making this even more efficient is to break the update into two parts. One is the usual update of adding $\lambda(y - p)x_i$. Let's call this the " $\mathcal{L}$" part of the update. The second is the "regularization part" of the update, which is to replace β by

$$\beta_j = \beta_j - \lambda 2\mu\beta_j = \beta_j \cdot (1 - 2\lambda\mu)$$

So we could perform our update of β_j as follows:

- Set $\beta_j = \beta_j \cdot (1 - 2\lambda\mu)$
- If $x_j \neq 0$, set $\beta_j = \beta_j + \lambda(y - p)x_i$

Following this up, we note that we can perform m successive "regularization" updates by letting $B_j = B_j \cdot (1 - 2\lambda\mu)^m$. The basic idea of the new algorithm is to not perform regularization updates for zero-valued x_j's, but instead to simply keep track of how many such updates would need to be performed to update β_j, and perform them only when we would normally perform "$\mathcal{L}$" updates (or when we output the parameters at the end of the day). This latter version is called "Lazy sparse stochastic gradient descent for regularized logistic regression".

For more info see also `http://cilvr.cs.nyu.edu/diglib/lsml/bottou-sgd-tricks-2012.pdf`, or (amongst others Gardner, 1984; Zinkevich et al., 2010; Opper and Winther, 1998; Poggio et al., 2011).

10.4 Assignment

Since there are no more "tutorials" scheduled at this point, the following assignments are basically voluntary. However, I would suggest you give them a try.

- Implement linear regression using stochastic gradient descent in Python. Compare it to the online solution we discussed in the previous lecture.
- Implement logistic regression in Python
- Implement regularized logistic regression in Python.
- Plot the convergence of the parameters β for logistic regression and regularized logistic regression as a function of time t for a dataset you simulate yourself.

Chapter 11

Lecture 11: Bandit problems

In this lecture we will discuss reinforcement learning, and more prominently bandit problems, since these arise often on the Web. We will briefly introduce the topic, and then examine "policies" to solve the problem. At the end you should be:

- Able to explain the difference between supervised and unsupervised learning
- Understand the formulation of Bandit problems and their omnipresence
- Understand the exploration-exploitation trade-off
- Discuss what it means for a policy to "solve" the bandit problem
- Be able to implement different solutions to bandit problems
- Understand Thompson Sampling

11.1 Reinforcement Learning

Reinforcement learning is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, statistics, and genetic algorithms. In the operations research and control literature, the field where reinforcement learning methods are studied is called approximate dynamic programming. The problem has been studied in the theory of optimal control, though most studies there are concerned with existence of optimal solutions and their characterization, and not with the learning or approximation

aspects. In economics and game theory, reinforcement learning may be used to explain how equilibrium may arise under bounded rationality.

In machine learning, the environment is typically formulated as a Markov decision process (MDP) as many reinforcement learning algorithms for this context utilize dynamic programming techniques. The main difference between the classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible.

Reinforcement learning differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). The exploration vs. exploitation trade-off in reinforcement learning has been most thoroughly studied through the multi-armed bandit problem and in finite MDPs.

The basic reinforcement learning model consists of:

- a set of environment states $\mathcal{S}$;
- a set of actions $\mathcal{A}$;
- rules of transitioning between states;
- rules that determine the scalar immediate reward r of a transition; and
- rules that describe what the agent observes.

The rules are often stochastic. The observation typically involves the scalar immediate reward associated with the last transition. In many works, the agent is also assumed to observe the current environmental state, in which case we talk about full observability, whereas in the opposing case we talk about partial observability. Sometimes the set of actions available to the agent is restricted (e.g., you cannot spend more money than what you possess).

A reinforcement learning agent interacts with its environment in discrete time steps. At each time t, the agent receives an observation o_t, which typically includes the reward r_t. It then chooses an action a_t from the set of actions available, which is subsequently sent to the environment. The environment moves to a new state s_{t+1} and the reward r_{t+1} associated with the transition (s_t, a_t, s_{t+1}) is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The agent can choose any action as a function of the history and it can even randomize its action selection.

When the agent's performance is compared to that of an agent which acts optimally from the beginning, the difference in performance gives rise to the notion of *regret*.

We will not dig into reinforcement learning as a topic of its own, but you should obviously know it exists, and has applications on the Web. We refer to (Sutton and Barto 1998) for a good introduction. In this course we will now move on and discuss multi-armed bandit problems. We will formalize "policies" and "regret" in this context. Note that the bandit literature is large, and that (again), we are only scratching the surface here....

11.2 Bandit problems

The multi-armed bandit problem derives its name from one of the classical ways of setting up the problem: A one-armed bandit is a slot machine with one arm to pull. In the multi-armed bandit we assume a player faces multiple arms (or slot machines), which might have differing pay-offs. However, the pay-offs are unknown to the player and have to be "learned" as she goes along. The question then becomes what strategy (or often "policy") should a player follow to maximize her profits from playing the machines.

Formally, bandit problems can be described as follows: at each time $t = 1, \ldots, T$, we have a set of possible actions $\mathcal{A}$. After choosing $a_t \in \mathcal{A}$ we observe reward r_t. The aim is to find a policy $\mathcal{P}$ to select actions a such that the cumulative reward $\mathcal{R}_c = \sum_{t=1}^{T} r_t$ is as large as possible. If we denote that actions $a = 1, \ldots, a = K$ for the $k = 1, \ldots, k = K$ different arms then we can denote μ_k, the expected reward of arm k. The arm (or action) k has an unknown reward distribution that the player needs to learn.

11.2.1 Exploration vs. Exploitation

Obviously, if μ_k is known with perfect certainty, then the multi-armed bandit problem is not at all a problem: you just play the action k with the highest expectation. However, in practice you often don't know μ_k with full certainty: we have to estimate μ_k using the rewards obtain thus far. Because of this uncertainty the player now has conflicting interests: the player might play the action that she believes has the highest pay-off, but is uncertain about. However, she might also choose to try the arms which she is more unsure about, but which potentially have a higher reward. This is called the exploration-exploitation trade-off: on one hand a player wants to explore the different actions to see what they are worth, but on the other hand the player wants to utilize her knowledge by playing the action she believes has the highest pay-off.

11.2.2 Strategies or Policies

In the bandit literature researchers examine "Policies". These are distinct algorithms to choose actions at $t = t'$ based on the actions and rewards at $t < t'$. A policy might be simple

(and stupid), and it might involve elaborate models of actions (and possible dependencies between actions), rewards (with possible effects over time), etc.

In its essence a policy contains 2 parts: (1) the policy specifies a model to summarize or model the data $(a_1, \ldots, a_t, r_1, \ldots, r_t)$ and subsequently (2) the policy consists of a decision procedure: given the model, what choice does the player make? We will look at a few policies shortly.

11.2.3 Performance of Policies: Regret

Before we examine different policies it is good to think about the performance of policies: when is a policy "good"? This is often specified in terms of the total expected regret:

$$R_T = \mu^* T - \sum_{k=1}^{K} \mu_j \mathbb{E}[T_j(T)]$$

where μ^* is the expected reward of the action with the highest expected reward and $\mathbb{E}[T_k(T)]$ denotes the expected number of times that a policy $\mathcal{P}$ will select action k. It is also often denoted (and computed) as:

$$R_T = \mu^* T - \sum_{t=1}^{T} \mathcal{P}(t)$$

where $\mathcal{P}(t)$ denotes the reward observed at t when using policy $\mathcal{P}$. In empirical simulations we often repeat a policy multiple times and compute the expectation over simulation runs of $\sum_{t=1}^{T} \mathcal{P}(t)$.

Note that R_T is a function of T. Also note that when selecting the "best" action, there is no increase in R_T: if the best action is selected all the time R_T should converge to 0. If a wrong action is selected all the time (thus $\mu' < \mu^*$), then the regret increases linearly with time.

An algorithm is said to "solve" the multi armed bandit problem if it can match the lower bound $R_T = O(\log T)$. In words: an optimal policy selects the "best" action exponentially more often than any of the other actions.

11.2.4 Discount

For the analysis of bandit problems we analyze the regret R_T as a function of T. However, how do we deal with this as $T \to \infty$: we introduce discounts. We thus consider

$r_1\gamma_1, \ldots, r_t\gamma_t, \ldots r_{t+\ldots}\gamma_{t+\ldots}$ as our rewards where γ denotes how we discount historical or future rewards.

A very common discount structure is the n-horizon uniform discount where $\gamma_1 = 1, \ldots, \gamma_t = 1, \gamma_{t+1} = 0, \ldots, \gamma_{t+\ldots} = 0$. We will primarily consider this structure (basically giving 0 value to future rewards). Also common is the geometric discount $\gamma = 1, \beta, \beta^2, \beta^3, \ldots$.

11.2.5 Why is this problem important?

In case you have not been able to come up with any other application then gambling for the multi-armed bandit problem, here are a few other cases where the problem arises and needs attention:

- Advertisement selection online: each add will have a different (but at the start unknown) pay-off: how do we decide which ads to show?
- News article selection (see above)
- Medication testing: in its simplest case we have two competing pills, both with uncertain pay-offs. How do we go about to cure as many people as possible?
- ...

With some imagination you must be able to come up with a long list of possible applications of bandit problems (and their extensions – we will cover the "contextual" bandit in the next lecture).

11.3 Simple Policies

Let's first look at some simple (but not optimal) policies that are in use for the multi-armed bandit problem:

- Greedy: Always play the action with the currently highest estimate μ (highest expected reward).
- ϵ-greedy: Play action with the highest expected reward with probability $1-\epsilon$, choose a random action with probability ϵ (giving a $\frac{\epsilon}{(K-1)}$ probability for each).
- ϵ-first: Play random action with probability $\frac{1}{K}$ for $t < N$. For $t > N$ choose action with highest expected reward based on the initial trial. This is what we do in clinical trials.
- Play the Winner: Choose a random action, if it wins play it again, if it doesn't then switch (randomly) to one of the others.

11.4 UCB methods

A well studied and broad class of policies for bandit problems are called Upper Confidence Bound (or UCB) methods. Here is one version:

"At each timepoint T, select the arm k with the largest value of $B_{k,n_k,T}$ defined as:

$$B_{k,n_k,T} = \frac{1}{n_k}\sum_{i=1}^{n_k} r_{k,i} + \sqrt{\frac{2\log T}{n_k}}$$

Note that this will play arms with a high mean reward (the first term), but also those with high uncertainty (the second term). In this way exploration and exploitation are balanced. For a number of problems UCB methods are optimal (meaning they obtain the $R_T = O(\log T)$ bound). However, UCB methods are hard to generalize to complex bandit problems (with e.g. related actions or with a context), and they are sometimes hard to compute. Hence, despite their theoretical appeal, they are not super frequently used to solve bandit problems online.

11.5 Thompson Sampling

Thompson sampling is a recently popular policy, that was only proven optimal in 2013. The basic idea of Thompson sampling is simple and intuitive: one randomly selects an action a at time t according to its estimated probability of being optimal (e.g., leading to the highest reward). Thompson sampling is formalized easily within a Bayesian framework (cf. Scott, 2010). The set of past observations $\mathcal{D}$ consists of the actions $a_{(1,\ldots,t)}$ and the rewards $r_{(1,\ldots,t)}$. The rewards are modeled using a parametric likelihood function: $\Pr(r|a,\theta)$ where θ is a set of parameters. Using Bayes rule it is, in some problems, easy to compute or sample from $\Pr(\theta|\mathcal{D})$. Given that we can compute $\Pr(\theta|\mathcal{D})$ we can select an action according to its probability of being optimal:

$$\int \mathbb{1}\left[\mathbb{E}(r|a,\theta) = \max_{a'} \mathbb{E}(r|a',\theta)\right] \Pr(\theta|\mathcal{D}) d\theta \tag{11.1}$$

where $\mathbb{1}$ is the indicator function. In practice it is not necessary to compute the above integral: it suffices to draw a random sample θ^* from the posterior at each round and select the action with the highest estimated reward given the current draw.

When it is easy to sample from $\Pr(\theta|\mathcal{D})$, Thompson sampling is easy to implement.

A commonly used example of a bandit problem is the K-arm Bernoulli bandit problem, where $r_t \in \{0,1\}$, and the action a is to select an arm $k = 1,\ldots,K$ at time t. The reward

of the k-th arm follows a Bernoulli distribution with true mean θ_k. The implementation of Thompson sampling using Beta priors for each arm is straightforward: For each arm k one sets up a Beta-Bernoulli model and at each round one obtains a single draw θ_k^* from each of the Beta posteriors, plays the arm $\hat{k} = \arg\max_k \theta_k^*$, and subsequently uses the observed reward r_t to update the Beta posterior of arm $\hat{k}$.

There is a large literature on MAB problems. For an accesible intro see (Berry and Fristedt, 1985). For more recent or detailed work: (see, e.g., Garivier and Cappé, 2011; Press, 2009; Bubeck et al., 2011; Scott, 2010; Gittins, 1979; Whittle, 1980).

11.6 Assignments

- Implement, in Python, a comparison of the greedy, ϵ-first (with $N = 1000$, UCB, and Thompson sampling policies for a K arm bandit problem with continuous rewards. Compute the regret R_T for each method for $t = 1, \ldots, T = 10^6$ and average over 100 simulation runs for each. Describe the outcomes.
- Are your above policies implemented "online"?
- Which quantities do you need for each of the versions?

Chapter 12

Lecture 12: Contextual Bandit problems and applications

We have discussed simple bandit problems in the previous lecture, and we discussed some of the real-world problems that can be described using a bandit formulation. In this lecture we will dig a bit further into bandit problems and we will consider "contextual bandit problems". You wil learn:

- What a contextual bandit problem is.
- How to implement Thompson sampling for a contextual bandit problem (both with continuous as well as dichotomous rewards)
- To gain intuition into the hierarchical structures that often arise in Contextual Bandit problems.
- To implement Bootstrap Thompson sampling

12.1 The Contextual Bandit Problem

One of the fundamental underpinnings of the internet is advertising based content. This has become much more effective due to targeted advertising where ads are specifically matched to interests. Everyone is familiar with this, because everyone uses search engines and all search engines try to make money this way. The problem of matching ads to interests is a natural machine learning problem since there is a lot of data available about "who clicks on what". A fundamental problem with this data is that it is not supervisedin particular a click-or-not on one ad doesnt generally tell you if a different ad would have been clicked on. This implies we have a fundamental *exploration-exploitation* problem.

A standard mathematical setting for this situation is the multi-armed bandit, as we discussed last lecture. This setting (and its variants) however fail to capture a critical phenomenon: each of these displayed ads are done in the context of a search or other webpage. To model this, we might think of a different setting where on each round:

- The world announces some context information x (think of this as a high dimensional vector if that helps).
- A policy chooses arm a from 1 of k actions (i.e. 1 of k ads).
- The world reveals the reward r_a of the chosen action (i.e. whether the ad is clicked on).

Thus, now we suddenly have data $x_1, \ldots, x_t, a_1, \ldots, a_t, r_1, \ldots, r_t$ as opposed to $a_1, \ldots, a_t, r_1, \ldots, r_t$ as we had in the previous lecture. This is often quite realistic: we often know something about the environment or context before making a decision which impact the expected reward of the actions. We now thus need to learn a model that does not only describe the relationship between actions and rewards, but this time we need a model that also models (possible) interactions with the context x.

One naive way to do this is to just increase the number of actions as a factor of the context: if the context x is discrete valued, we could denote new actions ax and just use the methods we have developed before. However, with a very large set of possible context this is probably not feasible. Luckily, Thompson sampling generalizes quite well: all we need to do is setup a Bayesian model relating the actions, context, and rewards, and sample from its posterior to decide on the action to select. Below are some examples.

12.2 Thompson sampling for a simple contextual bandit problem

Thompson sampling "requires" a Bayesian model. Here are two versions for both continuous and discrete $\{0, 1\}$ rewards.

12.2.1 Continuous rewards

Consider a standard linear regression problem, in which for $i = 1, ..., n$ we specify the conditional distribution of y_i given a $k \times 1$ predictor vector $\mathbf{x}_i$:

$$y_i = \mathbf{x}_i^{\mathrm{T}} \boldsymbol{\beta} + \epsilon_i,$$

where $\boldsymbol{\beta}$ is a $k \times 1$ vector, and the ϵ_i are independent and identical normally distributed random variables: $\epsilon_i \sim N(0, \sigma^2)$. We can now treat the y_i's as the rewards r_t, and within

$\mathbf{x}_i$ we encode both the effect of the actions a as well as the context x on the rewards that we expect. We continue with the notation y_i, x_i.

As we have seen, the likelihood of this model is:

$$\rho(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta}, \sigma^2) \propto (\sigma^2)^{-n/2} \exp\left(-\frac{1}{2\sigma^2}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})\right).$$

which is maximized by:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

where $\mathbf{X}$ is the $n \times k$ design matrix, each row of which is a predictor vector $\mathbf{x}_i^{\mathrm{T}}$.

This is a frequentist approach, and it assumes that there are enough measurements to say something meaningful about $\boldsymbol{\beta}$. In the Bayesian approach, the data are supplemented with additional information in the form of a prior probability distribution. The prior belief about the parameters is combined with the data's likelihood function according to Bayes theorem to yield the posterior belief about the parameters $\boldsymbol{\beta}$ and σ. The prior can take different functional forms depending on the domain and the information that is available a priori. Here we consider the a conjugate prior for the Bayesian linear regression model.

A prior $\rho(\boldsymbol{\beta}, \sigma^2)$ is conjugate to this likelihood function if it has the same functional form with respect to $\boldsymbol{\beta}$ and σ. Since the log-likelihood is quadratic in $\boldsymbol{\beta}$, the log-likelihood is re-written such that the likelihood becomes normal in $(\boldsymbol{\beta} - \hat{\boldsymbol{\beta}})$. Write

$$\begin{aligned}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\boldsymbol{\beta}) = (\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}}) \\ + (\boldsymbol{\beta} - \hat{\boldsymbol{\beta}})^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X})(\boldsymbol{\beta} - \hat{\boldsymbol{\beta}}).\end{aligned}$$

The likelihood is now re-written as

$$\begin{aligned}\rho(\mathbf{y}|\mathbf{X}, \boldsymbol{\beta}, \sigma^2) \propto (\sigma^2)^{-v/2} \exp\left(-\frac{vs^2}{2\sigma^2}\right) (\sigma^2)^{-(n-v)/2} \\ \times \exp\left(-\frac{1}{2\sigma^2}(\boldsymbol{\beta} - \hat{\boldsymbol{\beta}})^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X})(\boldsymbol{\beta} - \hat{\boldsymbol{\beta}})\right),\end{aligned}$$

where $vs^2 = (\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})^{\mathrm{T}}(\mathbf{y} - \mathbf{X}\hat{\boldsymbol{\beta}})$, and $v = n - k$, where k is the number of regression coefficients. This suggests a form for the prior:

$$\rho(\boldsymbol{\beta}, \sigma^2) = \rho(\sigma^2)\rho(\boldsymbol{\beta}|\sigma^2)$$

where $\rho(\sigma^2)$ is an inverse-gamma distribution: $\rho(\sigma^2) \propto (\sigma^2)^{-(v_0/2+1)} \exp\left(-\frac{v_0 s_0^2}{2\sigma^2}\right)$. This is the density of an Inv-Gamma(a_0, b_0) distribution with $a_0 = v_0/2$ and $b_0 = \frac{1}{2}v_0 s_0^2$ with v_0

and s_0^2 as the prior values of v and s^2, respectively. The conditional prior density $\rho(\boldsymbol{\beta}|\sigma^2)$ is a normal distribution,

$$\rho(\boldsymbol{\beta}|\sigma^2) \propto (\sigma^2)^{-k/2} \exp\left(-\frac{1}{2\sigma^2}(\boldsymbol{\beta}-\boldsymbol{\mu}_0)^{\mathrm{T}}\boldsymbol{\Lambda}_0(\boldsymbol{\beta}-\boldsymbol{\mu}_0)\right)).$$

In the notation of the normal distribution, the conditional prior distribution is $\mathcal{N}\left(\boldsymbol{\mu}_0, \sigma^2\boldsymbol{\Lambda}_0^{-1}\right)$.

With the prior now specified, the posterior distribution can be expressed as

$$\begin{aligned}\rho(\boldsymbol{\beta},\sigma^2|\mathbf{y},\mathbf{X}) &\propto \rho(\mathbf{y}|\mathbf{X},\boldsymbol{\beta},\sigma^2)\rho(\boldsymbol{\beta}|\sigma^2)\rho(\sigma^2)\\ &\propto (\sigma^2)^{-n/2}\exp\left(-\frac{1}{2\sigma^2}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})^{\mathrm{T}}(\mathbf{y}-\mathbf{X}\boldsymbol{\beta})\right)\\ &\times (\sigma^2)^{-k/2}\exp\left(-\frac{1}{2\sigma^2}(\boldsymbol{\beta}-\boldsymbol{\mu}_0)^{\mathrm{T}}\boldsymbol{\Lambda}_0(\boldsymbol{\beta}-\boldsymbol{\mu}_0)\right)\times(\sigma^2)^{-(a_0+1)}\exp\left(-\frac{b_0}{\sigma^2}\right).\end{aligned}$$

With some re-arrangement, the posterior can be re-written so that the posterior mean $\boldsymbol{\mu}_n$ of the parameter vector $\boldsymbol{\beta}$ can be expressed in terms of the least squares estimator $\hat{\boldsymbol{\beta}}$ and the prior mean $\boldsymbol{\mu}_0$, with the strength of the prior indicated by the prior precision matrix $\boldsymbol{\Lambda}_0$

$$\boldsymbol{\mu}_n = (\mathbf{X}^{\mathrm{T}}\mathbf{X}+\boldsymbol{\Lambda}_0)^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{X}\hat{\boldsymbol{\beta}}+\boldsymbol{\Lambda}_0\boldsymbol{\mu}_0)$$

The posterior can be expressed as a normal distribution times an inverse-gamma distribution:

$$\begin{aligned}\rho(\boldsymbol{\beta},\sigma^2|\mathbf{y},\mathbf{X}) &\propto (\sigma^2)^{-k/2}\exp\left(-\frac{1}{2\sigma^2}(\boldsymbol{\beta}-\boldsymbol{\mu}_n)^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X}+\boldsymbol{\Lambda}_0)(\boldsymbol{\beta}-\boldsymbol{\mu}_n)\right)\\ &\times(\sigma^2)^{-(n+a_0)/2-1}\exp\left(-\frac{b_0+\mathbf{y}^{\mathrm{T}}\mathbf{y}-\boldsymbol{\mu}_n^{\mathrm{T}}(\mathbf{X}^{\mathrm{T}}\mathbf{X}+\boldsymbol{\Lambda}_0)\boldsymbol{\mu}_n+\boldsymbol{\mu}_0^{\mathrm{T}}\boldsymbol{\Lambda}_0\boldsymbol{\mu}_0}{2\sigma^2}\right).\end{aligned}$$

Therefore the posterior distribution can be parametrized as follows:

$$\rho(\boldsymbol{\beta},\sigma^2|\mathbf{y},\mathbf{X}) \propto \rho(\boldsymbol{\beta}|\sigma^2,\mathbf{y},\mathbf{X})\rho(\sigma^2|\mathbf{y},\mathbf{X}),$$

where the two factors correspond to the densities of $\mathcal{N}\left(\boldsymbol{\mu}_n, \sigma^2\boldsymbol{\Lambda}_n^{-1}\right)$, and Inv-Gamma (a_n, b_n) distributions, with the parameters of these given by

$$\begin{aligned}\boldsymbol{\Lambda}_n &= (\mathbf{X}^{\mathrm{T}}\mathbf{X}+\boldsymbol{\Lambda}_0),\\ \boldsymbol{\mu}_n &= (\boldsymbol{\Lambda}_n)^{-1}(\mathbf{X}^{\mathrm{T}}\mathbf{X}\hat{\boldsymbol{\beta}}+\boldsymbol{\Lambda}_0\boldsymbol{\mu}_0)\\ a_n &= a_0+\frac{n}{2}\\ b_n &= b_0+\frac{1}{2}(\mathbf{y}^{\mathrm{T}}\mathbf{y}+\boldsymbol{\mu}_0^{\mathrm{T}}\boldsymbol{\Lambda}_0\boldsymbol{\mu}_0-\boldsymbol{\mu}_n^{\mathrm{T}}\boldsymbol{\Lambda}_n\boldsymbol{\mu}_n).\end{aligned}$$

After setting up this Bayesian Linear model all we need to do at each round is to take a draw from the posterior β and select the action a which maximizes r given x.

12.2.2 Discrete rewards

For discrete rewards $r \in \{0, 1\}$ the above model does not work and we need a Bayesian model with some link function (e.g. logistic regression, probit regression, etc.). Here we choose a probit model.

Probit models

Suppose response variable Y is binary, that is it can have only two possible outcomes which we will denote as 1 and 0. For example Y may represent presence/absence of a certain condition, success/failure of some device, answer yes/no on a survey, etc. We also have a vector of regressors X, which are assumed to influence the outcome Y. Specifically, we assume that the model takes the form

$$\Pr(Y = 1 \mid X) = \Phi(X'\beta),$$

where Φ is the Cumulative Distribution Function (CDF) of the standard normal distribution. The parameters β are typically estimated by maximum likelihood.

It is possible to motivate the probit model as a latent variable model. Suppose there exists an auxiliary random variable

$$Y^* = X'\beta + \varepsilon,$$

where ϵ $N(0, 1)$. Then Y can be viewed as an indicator for whether this latent variable is positive:

$$Y = \begin{cases} 1 & \text{if } Y^* > 0 \text{ i.e. } -\varepsilon < X'\beta, \\ 0 & \text{otherwise.} \end{cases}$$

The use of the standard normal distribution causes no loss of generality compared with using an arbitrary mean and standard deviation because adding a fixed amount to the mean can be compensated by subtracting the same amount from the intercept, and multiplying the standard deviation by a fixed amount can be compensated by multiplying the weights by the same amount.

To see that the two models are equivalent, note that

$$\begin{aligned}
\Pr(Y = 1 \mid X) &= \Pr(Y^* > 0) = \Pr(X'\beta + \varepsilon > 0) && (12.1)\\
&= \Pr(\varepsilon > -X'\beta) && (12.2)\\
&= \Pr(\varepsilon < X'\beta) \quad \text{(by symmetry of the normal dist)} && (12.3)\\
&= \Phi(X'\beta) && (12.4)
\end{aligned}$$

Gibbs Sampling

Gibbs sampling of a probit model is possible because regression models typically use normal prior distributions over the weights, and this distribution is conjugate with the normal distribution of the errors (and hence of the latent variablesY*). The model can be described as

$$\begin{aligned}
\boldsymbol{\beta} &\sim \mathcal{N}(\mathbf{b}_0, \mathbf{B}_0) && (12.5)\\
y_i^* \mid \mathbf{x}_i, \boldsymbol{\beta} &\sim \mathcal{N}(\mathbf{x}_i'\boldsymbol{\beta}, 1) && (12.6)\\
y_i &= \begin{cases} 1 & \text{if } y_i^* > 0 \\ 0 & \text{otherwise} \end{cases} && (12.7)
\end{aligned}$$

From this, we can determine the full conditional densities needed:

$$\begin{aligned}
\mathbf{B} &= (\mathbf{B}_0^{-1} + \mathbf{X}'\mathbf{X})^{-1} && (12.8)\\
\boldsymbol{\beta} \mid \mathbf{y}^* &\sim \mathcal{N}(\mathbf{B}(\mathbf{B}_0^{-1}\mathbf{b}_0 + \mathbf{X}'\mathbf{y}^*), \mathbf{B}) && (12.9)\\
y_i^* \mid y_i = 0, \mathbf{x}_i, \boldsymbol{\beta} &\sim \mathcal{N}(\mathbf{x}_i'\boldsymbol{\beta}, 1)[y_i^* < 0] && (12.10)\\
y_i^* \mid y_i = 1, \mathbf{x}_i, \boldsymbol{\beta} &\sim \mathcal{N}(\mathbf{x}_i'\boldsymbol{\beta}, 1)[y_i^* \geq 0] && (12.11)
\end{aligned}$$

The only trickiness is in the last two equations. The notation $[y_i^* < 0]$ is the Iverson bracket, sometimes written $\mathcal{I}(y_i^* < 0)$ or similar. It indicates that the distribution must be truncated within the given range, and rescaled appropriately. In this particular case, a truncated normal distribution arises. Sampling from this distribution depends on how much is truncated. If a large fraction of the original mass remains, sampling can be easily done with rejection sampling simply sample a number from the non-truncated distribution, and reject it if it falls outside the restriction imposed by the truncation. If sampling from only a small fraction of the original mass, however (e.g. if sampling from one of the tails

of the normal distribution for example if $\mathbf{x}_i'\boldsymbol{\beta}$ is around 3 or more, and a negative sample is desired), then this will be inefficient and it becomes necessary to fall back on other sampling algorithms.

One can use draws from the Gibbs sampler to implement Thompson sampling. Note that this does not scale very well computationally (it is not "online"). However, many standard implementation of Bayesian Probit regression can be found and used if the problem is not too large.

12.3 Hierarchical Structures

Already above, for the probit case, computation of the posterior β was somewhat annoying and we resorted to Gibbs Sampling. This obviously does not scale very well. However, the above two examples were really simple linear models and generalized linear model versions for the contextual bandit problem. In reality these are often too simple: in many real problems the data is nested or hierarchical, and we thus need to resort to linear mixed models or generalized mixed models. For these it might be computationally even harder to sample from the posterior, and thus to implement Thompson sampling. However, we should not just ignore the grouping structure. Developing scalable and efficient methods for Thompson sampling in the cases of hierarchical contexts is an active research field.

12.4 Bootstrap Thompson Sampling

When it is easy to sample from $\Pr(\theta|\mathcal{D})$, Thompson sampling is easy to implement. However, to be practically feasible for many problems, and thus scalable to large T or to complex likelihood functions, Thompson sampling requires computationally efficient sampling from $\Pr(\theta|\mathcal{D})$. In practice $\Pr(\theta|\mathcal{D})$ might not always be easily available: already in situations in which a logit or probit model is used to model the expected reward of the actions, $\Pr(\theta|\mathcal{D})$ is not available in closed form and is then often computed using MCMC methods, which can be computationally costly. Furthermore, for many likelihood functions it is hard to update the posterior online (i.e., row-by-row) thus requiring inspection of the full dataset $\mathcal{D}$ at each iteration. Both of these properties make that for a number of applied problems the scalability of Thompson sampling might be limited. Also, Thompson sampling is a parametric method, so its performance depends on the accuracy of the model that is used to compute $\Pr(r|a,\theta)$. Thus, Thompson sampling may not be very robust to common forms of model misspecification.

A modification of Thompson sampling that we call *bootstrap Thompson sampling* (BTS) aims to solve some of these problems. BTS replaces the posterior $\Pr(\theta|\mathcal{D})$ by a bootstrap

distribution of the point estimate $\hat{\theta}$. Some bootstrap methods are especially computationally appealing. In particular, bootstrap methods that involve randomly reweighting data (Rubin, 1981), rather than resampling data, can be conducted online (Lee and Clyde, 2004; Owen and Eckles, 2012; Oza, 2001). For BTS we use a bootstrap method in which, for each bootstrap replicate $j \in \{1, \ldots, J\}$, each observation gets a weight $w_{tj} \sim 2 \times \text{Bernoulli}(1/2)$. Following Owen and Eckles (2012, §3.3), we refer to this bootstrap as the *double-or-nothing bootstrap* (DoNB) or *online half-sampling*.[1]

Statisticians have noted relationships between bootstrap distributions and Bayesian posteriors. With a particular weight distribution and nonparametric model of the distribution of observations, the bootstrap distribution and the posterior coincide (Rubin, 1981). In other cases, the bootstrap distribution $\tilde{\theta}$ can be used to approximate a posterior (e.g., Efron, 2011; Newton and Raftery, 1994), e.g., as a proposal distribution in importance sampling. Moreover, sometimes the difference between the bootstrap distribution and the Bayesian posterior is that the bootstrap distribution is more robust to model misspecification, such that if they differ substantially the bootstrap distribution may even be preferred (Liu and Rubin, 1994; Szpiro et al., 2010).

See for more on BTS: (Kaptein and Eckles, 2014). For alternative solutions to continuous bandit problems see (Kaptein and Iannuzzi, 2014; Agarwal et al., 2011). For optimality proves on bandits see (Agrawal et al., 1988; Agrawal, 2012, 2014).

12.5 Applications

Please try the following assignment:

- Implement in Python a simulation of a contextual bandit solution to the "web advertisement" problem: suppose the context presents itself as $x \in \{1, \ldots, m\}$ webpages that a user could visit. Subsequently, you have the choice of $1, \ldots, K$ actions (or ads) to serve. Finally, you observe the reward (and let's ignore the user grouping for now). Implement this both in the continuous and in the discrete (using a probit model) case.

[1] Since the absolute scale of the weights does not matter for most estimators, it is equivalent to have the weights be 0 or 1, rather than 0 or 2. Other weight distributions could be used for various reasons. For example, using exponential weights is the so-called Bayesian bootstrap (Rubin, 1981). In that case, each observation has positive weight in each replicate, which can avoid numerical problems in some settings, but requires updating all replicates for each observation. Owen and Eckles (2012, §3.3) compare weight distributions for the bootstrapping the sample mean.

Bibliography

Agarwal, A., Foster, D. P., Hsu, D., Kakade, S. M., and Rakhlin, A. (2011). Stochastic convex optimization with bandit feedback. pages 1–26.

Agrawal, R., Hedge, M., and Teneketzis, D. (1988). Asymptotically efficient adaptive allocation rules for the multiarmed bandit problem with switching cost.

Agrawal, S. (2012). Further Optimal Regret Bounds for Thompson Sampling. *CoRR*, none:1–14.

Agrawal, S. (2014). Thompson Sampling for Contextual Bandits with Linear Payoffs.

Alpcan, T. and Bauckhage, C. (2009). A distributed machine learning framework. *Proceedings of the 48h IEEE Conference on Decision and Control CDC held jointly with 2009 28th Chinese Control Conference*, pages 2546–2551.

Aral, S., Muchnik, L., and Sundararajan, A. (2009). Distinguishing influence-based contagion from homophily-driven diffusion in dynamic networks. *Proceedings of the National Academy of Sciences of the United States of America*, 106(51):21544–21549.

Aral, S., Muchnik, L., and Sundararajan, A. (2011). Engineering Social Contagions : Optimal Network Seeding and Incentive Strategies. *Business*, 1770982.

Bakshy, E., Eckles, D., Yan, R., and Rosenn, I. (2012). Social Influence in Social Advertising : Evidence from Field Experiments. In *Electronic Commerce 2012*, volume 1.

Berry, D. A. and Fristedt, B. (1985). *Bandit Problems: Sequential Allocation of Experiments*. Springer.

Bishop, C. M. et al. (2006). *Pattern recognition and machine learning*, volume 4. springer New York.

Bottou, L. (1998). Online Algorithms and Stochastic Approximations. In Saad, D., editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK.

Bubeck, S., Munos, R., and Stoltz, G. (2011). Pure exploration in finitely-armed and continuous-armed bandits. *Theoretical Computer Science*, 412(19):1832–1852.

Chu, C.-t., Kim, S. K., Lin, Y.-a., and Ng, A. Y. (2007). Map-Reduce for Machine Learning on Multicore. *Advances in neural information processing systems*, 19(23):281.

Efron, B. (2011). Bayesian computation and the parametric bootstrap. *Working Paper*, pages 1–24.

Gaber, M. M., Zaslavsky, A., and Krishnaswamy, S. (2005). Mining data streams. *ACM SIGMOD Record*, 34(2):18.

Gardner, W. (1984). Learning characteristics of stochastic-gradient-descent algorithms: A general study, analysis, and critique. *Signal Processing*, 6(2):113–133.

Garivier, A. and Cappé, O. (2011). The KL-UCB Algorithm for Bounded Stochastic Bandits and Beyond. *Bernoulli*, 19(1):13.

Gelman, A. and Hill, J. (2006). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press.

Gittins, J. C. (1979). Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society Series B Methodological*, 41(2):148–177.

Gretzel, U. and Fesenmaier, D. (2006). Persuasion in Recommender Systems. *International Journal of Electronic Commerce*, 11(2):81–100.

Hastie, T., Tibshirani, R., and Friedman, J. (2013). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, volume 11. Springer Science & Business Media.

Kaptein, M. and Eckles, D. (2014). Thompson Sampling with the Online Bootstrap. *arXiv preprint arXiv:1410.4009*.

Kaptein, M. C. and Iannuzzi, D. (2014). Lock in Feedback in Sequential Experiments.

Lam, X. N., Vu, T., Le, T. D., and Duong, A. D. (2008). Addressing cold-start problem in recommendation systems. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication - ICUIMC '08*, page 208, New York, New York, USA. ACM Press.

Lee, H. K. H. and Clyde, M. A. (2004). Lossless online Bayesian bagging. *Journal of Machine Learning Research*, 5:143–151.

Liu, J. S. and Rubin, D. B. (1994). Comment on "Approximate Bayesian inference with the weighted likelihood bootstrap". *Journal of the Royal Statistical Society. Series B (Methodological)*, page 40.

Marlin, B. (2004). *Collaborative filtering: A machine learning perspective*. PhD thesis, University of Toronto.

Newton, M. A. and Raftery, A. E. (1994). Approximate Bayesian inference with the weighted likelihood bootstrap. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 3–48.

Ochi, P., Rao, S., Takayama, L., and Nass, C. (2010). Predictors of user perceptions of web recommender systems: How the basis for generating experience and search product recommendations affects user responses. *International Journal of Human-Computer Studies*, 68(8):472–482.

Opper, M. and Winther, O. (1998). A Bayesian approach to on-line learning. In Saad, D., editor, *Learning*, chapter 16, pages 363–378. Cambridge University Press.

Owen, A. B. and Eckles, D. (2012). Bootstrapping data arrays of arbitrary order. *The Annals of Applied Statistics*, 6(3):895–927.

Oza, N. (2001). Online bagging and boosting. In *Systems, man and cybernetics, 2005 IEEE international conference on*, volume 3, pages 2340–2345. IEEE.

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). The pagerank citation ranking: Bringing order to the web.

Poggio, T., Voinea, S., and Rosasco, L. (2011). Online Learning, Stability, and Stochastic Gradient Descent. *Artificial Intelligence*, 8(11):11.

Press, W. H. (2009). Bandit solutions provide unified ethical models for randomized clinical trials and comparative effectiveness research. *Proceedings of the National Academy of Sciences of the United States of America*, 106(52):22387–92.

Ricci, F., Rokach, L., Shapira, B., and Kantor, P. B. (2011). Recommender Systems Handbook. *Media*, 54(11):217–253.

Rubin, D. B. (1981). The Bayesian bootstrap. *The Annals of Statistics*, 9:130–134.

Sarwar, B., Karypis, G., Konstan, J., and Riedl, J. (2001). Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM.

Scott, S. L. (2010). A modern Bayesian look at the multi-armed bandit. *Applied Stochastic Models in Business and Industry*, 26(6):639–658.

Szpiro, A. A., Rice, K. M., and Lumley, T. (2010). Model-robust regression and a Bayesian "sandwich" estimator. *The Annals of Applied Statistics*, 4(4):2099–2113.

Wasserman, S. (1994). *Social network analysis: Methods and applications*, volume 8. Cambridge university press.

Whittle, P. (1980). Multi-armed bandits and the Gittins index. *Journal of the Royal Statistical Society Series B Methodological*, 42(2):143–149.

Williams, D. and Williams, D. (2001). *Weighing the odds: a course in probability and statistics*, volume 548. Springer.

Zinkevich, M. A., Smola, A., and Weimer, M. (2010). Parallelized Stochastic Gradient Descent. *Advances in Neural Information Processing Systems 23*, 23(6):1–9.