Math~Blog
Mathematics is wonderful!

Faster, Better, Cheaper Search Engines

By John F. McGowan, Ph.D. for Math-Blog.com

Introduction

Searching for documents and other items on the Web or computers is often tedious and time consuming. Time is money. Highly paid professionals spend hours, days, and even longer searching for information on the Web or computers. Most search today is done using key word and phrase matching, often combined with various ranking schemes for the search results. Occasionally more advanced methods such as logical queries, e.g. search for "rocket scientist" and NOT "space", and regular expressions are used. All of these methods have significant limitations and often require lengthy human review and further manual searching of the search results. The dream search engine would search by topic, by the detailed content of the items searched, ideally finding the desired information immediately. Actual understanding of text remains a unfulfilled promise of artificial intelligence. Statistical language processing can achieve a degree of searching by topic. This article introduces the basic concepts and mathematics of statistical language processing and its applications to search. It gives a brief introduction and overview of more advanced techniques in statistical language processing as applied to search. It also includes sample Ruby code illustrating some simple statistical language processing methods.

Professionals spend substantial amounts of time and money searching for documents and information. For example, programmers use the Web to locate solutions for obscure bugs, often reports by other programmers who have already encountered the bug, in widely used programs such as Excel. With search engines programmers can sometimes find these solutions in a few minutes, but a search for a bug report often take hours or even days to find a solution using a Web search. Experienced programmers also spend hours, even days, relearning how to do things they already know how to do in a new programming language, finding out how to use a rarely used feature in a familiar programming language, and identifying undocumented functions. This is often done using the search features of online help systems and code editors, often little more than simple word and phrase matching (or occasionally regular expression matching).

Lawyers search for legal cases, laws, regulations, Law Review articles, and so forth. Medical doctors search for papers and other information on medical conditions and treatments. Research scientists search for research papers, doctoral dissertations, patents, and tables of previously measured data. Engineers search for books and papers, technical data, technical drawings of working machines, mathematical formulas for computing useful results, and so forth. Business analysts search for financial and marketing data. With professional salaries of tens to even hundreds of dollars per hour, lengthy searches cost hundreds to thousands of dollars per search. Some busy professionals may conduct hundreds of important searches per year. More powerful search engines can save time and money and bring success.

Hourly Rate	Duration of Search	Cost of Search	Cost of 100 Searches
$20	30 minutes	$10	$1,000
$20	2 hours	$40	$4,000
$20	20 hours	$400	$40,000
$50	30 minutes	$25	$2,500
$50	2 hours	$100	$10,000
$50	20 hours	$1000	$100,000
$200	30 minutes	$100	$10,000
$200	2 hours	$400	$40,000
$200	20 hours	$4000	$400,000
$500	30 minutes	$250	$25,000
$500	2 hours	$500	$50,000
$500	20 hours	$10000	$1,000,000
$1000	30 minutes	$500	$50,000
$1000	2 hours	$2000	$200,000
$1000	20 hours	$20000	$2,000,000

Search engines remain based primarily on matching words and phrases often weighted by the popularity of documents, advertising dollars, and other adjustments. In practice, end users may be unable to find a relevant document or spend many minutes, hours, or even days paging through search results and/or trying many different search words and phrases in hopes of turning up a relevant document or item. Popularity is not always a good measure of either the relevance or the quality of a document or item. Rankings based on advertising dollars may not match the needs of the end user of a search engine. More powerful methods are needed.

The cause of this costly and frustrating state of search is that present-day search engines cannot understand either the search queries or the documents or items searched. For example, if an end user enters the search phrase "rocket scientist", they are probably looking for information on actual rocket scientists, but not always. Rocket scientist is sometimes used as generic term for highly skilled technical professionals in scientific, engineering, and similar fields. A search for "rocket scientist" will occasionally turn up articles on Wall Street financial engineers, also known as "quants", who are sometimes referred to as "rocket scientists" in the financial literature. More powerful search engines will need understanding or a way to emulate some or all aspects of human understanding.

The dream search engine should find documents or other items *by topic*, not by word or phrase, and return only documents or items related to the topic of interest. Stephen Wolfram's *Alpha* is being marketed as such a search engine. Time will tell if *Alpha* is or will become the dream search engine. To date, actual understanding of natural language by computers has proven extremely difficult, like most problems in artificial intelligence (AI), and successes have been few and limited. Hence, search engines continue to rely on simple word and phrase matching. While actual understanding is probably decades if not centuries in the future, statistical language processing methods can achieve a degree of searching *by topic* today. Statistical language processing involves measuring and using the frequency of words and phrases as well as the absolute and relative positions of words and phrases in text.

Closely related problems also occur in recommendation engines and data loss prevention (sometimes identified by the cryptic acronym DLP). Recommendation engines such as NetFlix's Cinematch system, which recommends DVD rentals to customers, recommend possible purchases based on buying patterns and other data. However, recommendation engines do not *understand* customer preferences, products, product descriptions, and so forth. Recommendation engines use statistical methods to guess what other products or services customers are likely to purchase, giving the illusion of true understanding.

Data loss prevention consists of systems designed to prevent the accidental or intentional loss or theft of sensitive information from companies and organizations, for example an e-mail with customer credit card numbers sent to a credit card fraud group overseas. In some cases, for example social security numbers or credit card numbers, the sensitive information can be identified by matching simple patterns, such as regular expressions. However, some sensitive information, such as a sensitive marketing plan, may lack easily identifiable text or numerical patterns. A human being reading the document can identify it as sensitive easily but a data loss prevention algorithm would fail. Again, even without understanding, statistical patterns in the text of the document may be able to identify a sensitive document.

Reproducing human intelligence on computers, artificial intelligence, has proven baffling. Actual understanding of spoken or written text remains far beyond the state of computer science. However, statistical language processing can imitate some aspects of human intelligence and yield more "intelligent" results in speech recognition, machine translation, and search. The rest of this article gives an introduction and overview of the basic concepts and mathematics of statistical language processing and its applications to improving the speed and lowering the cost of search.

A Brief Introduction to *Mathematica*

This article presents mathematical formulas in the *Mathematica* programming language used to compute results and display graphics as well as in standard mathematical notation. *Mathematica* is an algorithm prototyping and mathematical research tool similar to MatLab. *Mathematica* and MatLab are scripting language like Python or Ruby with comprehensive well-integrated mathematical, numerical, and statistical functions. The *Mathematica* code is retained for greater clarity and detail. The *Mathematica* code in this article is usually simple, clear and contains comments in plain English explaining what the code does, but may be ignored by those unfamiliar with *Mathematica*. In addition, this article includes sample code in the popular Ruby scripting language. Ruby is free, open-source software available for Windows, Macintosh, and Unix platforms. It can be downloaded from the Ruby web site http://www.ruby-lang.org/.

Technically, everything in *Mathematica* code is an *expression*. The key to both understanding *Mathematica* and the power of *Mathematica* is a type of expression known as a *list*. As might be expected, a list is simply a list of elements. A table or matrix in *Mathematica* is simply a list of lists. *Mathematica* can represent almost anything as a list.

```
data = {0,Pi/8,2 Pi/8,3 Pi/8.0,4 Pi/8.0};  (* a simple list in Mathemat
```

The mathematical functions that are built into *Mathematica* are mostly set up to be "listable" so that they act separately on each element of a list.

```
Sin[data]  (* the built in trigonometric sine function in Mathematica i
```

$$\left\{0,\ \text{Sin}\left[\frac{\pi}{8}\right],\ \frac{1}{\sqrt{2}},\ 0.92388,\ 1.\right\}$$

However, some functions are not "listable". Some functions act on a list as a whole.

```
μ = Mean[data]  (* the average or arithmetic mean function is NOT "list
0.785398
```

This article uses several functions that act on a list as whole including **Length[list]**, **Mean[list]**, **Variance[list]**, **Covariance[list, list]**, **Correlation[list, list]**, and **ListPlot[list]**.

Most mathematical, numerical, statistical, and graphics functions in *Mathematica* take lists as arguments and often return lists. The common but flexible list data type enables tight integration of the many mathematical functions in *Mathematica*. Like a set of LEGO blocks or a TinkerToy, one can quickly create complex mathematical systems as needed.

Mathematica derives from the tradition of list processing languages such as LISP and Scheme as well as functional programming that began in the 1960's and has recently enjoyed a renaissance. *Mathematica* has many more advanced list and functional programming features, but for this article and indeed many common tasks the brief introduction above is all one needs to know.

Mathematica has a forbidding reputation in some circles because it includes extensive symbolic manipulation capabilities. *Mathematica* is considered a type of *computer algebra system*. Some people believe that one must know and use these symbolic capabilities to use *Mathematica* or that *Mathematica* is only for symbolic manipulation. These symbolic capabilities are actually only one subset of *Mathematica*'s capabilities. *Mathematica* has the same extensive numerical, statistical, mathematical, and graphics features as Matlab or other competing products (Maple, AXIOM, Maxima, R, and many others).

Mathematica does not require that one use its symbolic capabilities. One can use *Mathematica* heavily without ever using the symbolic manipulation capabilities. The author worked on prototypes of image and video processing algorithms at NASA for years without needing to use the symbolic features. The symbolic features are not used in this article. Even quite complex mathematical models of the frequency and location of words and phrases in documents that might be developed for search applications would generally not require using the symbolic manipulation features of *Mathematica*. *Mathematica* and Ruby are used for illustration; the needed statistical Language models can be researched and developed in many other programming languages.

Correlations

A correlation coefficient is a simple concept from basic statistics. It provides a simple numerical measure ranging from -1.0 to 1.0 for the correlation between two variables. The main definition of correlation in English is *a mutual relationship or connection*. It can have a more precise meaning: *the degree of relative correspondence, as between two sets of data (a* correlation *of 75 percent)*. In statistics, a correlation coefficient can be any one of several measures of concomitant variation in two or more variables. In this article and the accompanying sample code, the simplest and most common correlation coefficient from introductory statistics is used.

The correlation coefficient is built from three simpler concepts in basic statistics: the average, the variance, and the covariance. The average of a series of data points is the sum of the values of the data points divided by the number of data points. The average of (0,0,0) is 0. The average of (1,2,3) is 2.

The Average (also known as the arithmetic mean) is shown in the following *Mathematica* code snippet:

```
<<Statistics`DescriptiveStatistics`  (* load the descriptive statistic
<< "MultivariateStatistics`"; << "ComputationalGeometry`"
<< "BarCharts`"; << "Histograms`"; << "PieCharts`"

SeedRandom[17553]; (* set the seed for the Mathematica random number g
data = Table[RandomReal[], {i, 10}];
```

```
M = Length[data]  (* Length[list] returns the number of elements in a
10
HoldForm[Average = Sum[data[[i]], {i, M}] / (M) ] (* display formula fo
```

$$\text{Average} = \frac{\sum_{i=1}^{M} \text{data}[\![i]\!]}{M}$$

```
Average = Sum[data[[i]], {i, Length[data]}] / Length[data] // HoldForm
```

$$\frac{\sum_{i=1}^{\text{Length}[\text{data}]} \text{data}[\![i]\!]}{\text{Length}[\text{data}]}$$

```
μ = Mean[data]  (* compute average of data using built in Mean functio
0.54752
```

Once we have the average of a series of data points, we can compute the variance, a measure of the spread or *dispersion* of the data points.

The Variance (often represented by σ^2) is shown in the following *Mathematica* code snippet:

```
HoldForm[ σ^2 = Sum[ (data[[i]] - μ )^2, {i, Length[data]}] / (Length[
```

$$\sigma^2 = \frac{\sum_{i=1}^{\text{Length}[\text{data}]} (\text{data}[\![i]\!] - \mu)^2}{\text{Length}[\text{data}] - 1}$$

The standard deviation, usually represented as σ (the Greek letter *sigma*), is the square root of the variance:

```
σ = Sqrt[σ^2] (* display formula for the standard deviation σ *)
```

$$\sqrt{\frac{\sum_{i=1}^{\text{Length}[\text{data}]} (\text{data}[\![i]\!] - \mu)^2}{\text{Length}[\text{data}] - 1}}$$

The *covariance* of two variables *A* and *B*, usually represented as $\sigma_{A,B}$ (the Greek letter *sigma* with subscripts A and B) is defined as:

```
HoldForm[\!\(TraditionalForm\`σ\_\(A, B\)\) = (1/(Length[a]-1)) Sum[ (a
```

$$\sigma_{A,B} = \frac{\sum_{i=1}^{\text{Length}[a]} (a[\![i]\!] - \text{Mean}[a])\,(b[\![i]\!] - \text{Mean}[b])}{\text{Length}[a] - 1}$$

```
Cov[a_,b_] := Sum[ (a[[i]] - Mean[a]) (b[[i]] - Mean[b]), {i, Length[a]
data = Table[RandomReal[], {i, 1000}];
```

```
Length[data]  (* Length[list] returns the number of elements in list *
1000
Mean[data]  (* Mean[list] computes the average, arithmetic mean, of lis
0.505141
Cov[data,data]  (* my Covariance function, data is correlated with its
0.0815323
Covariance[data,data]  (* built-in Covariance function in Mathematica
0.0815323
AntiData = Table[RandomReal[], {i, Length[data]}];
Cov[data, AntiData]  (* my Covariance function *)
-0.00128841
Covariance[data, AntiData]  (* built-in Covariance function in Mathema
-0.00128841
```

The correlation coefficient, often represented by $\rho_{A,B}$ (Greek letter *rho* with subscripts A and B), is the covariance of the variables A and B $\sigma_{A,B}$ (Greek letter *sigma* with subscripts A and B) divided by the product of the standard deviation of A and the standard deviation of B.

$\rho_{A,B} = \sigma_{A,B} / \sigma_A \sigma_B$

$$\sigma_{a,b} = \frac{\sum_{i=1}^{\text{Length}[a]} (a[[i]]-\text{Mean}[a])\,(b[[i]]-\text{Mean}[b])}{\text{Length}[a]}$$

```
HoldForm[ ρ = Sum[ (a[[i]] - Mean[a]) (b[[i]] - Mean[b]), {i, Length[a
```

$$\rho = \frac{\sum_{i=1}^{\text{Length}[a]} (a[[i]] - \text{Mean}[a])\,(b[[i]] - \text{Mean}[b])}{\text{Length}[a]\,\sqrt{\text{Variance}[a]\,\text{Variance}[b]}}$$

```
Corr[a_,b_] := Sum[ (a[[i]] - Mean[a]) (b[[i]] - Mean[b]), {i, Length[a
Corr[data, data]  (* my Correlation function, defined above *)
1.
Correlation[data,data]  (* built in Mathematica Correlation function *
1.
Corr[data, AntiData] (* correlation calculated using the user defined f
-0.0156761
```

```
Correlation[data, AntiData]  (* correlation calculated using the built
```

```
-0.0156761
```

The Limits of Correlation Coefficients

The standard correlation coefficient only works well for linear relationships. In the example below the random variables *X* and *Y* are closely related by the equation *Y* = *Sin(X)*, a non-linear relationship. However, the *expectation value* of the standard correlation coefficient in the example below is 0.0. The variables appear unrelated if the standard correlation coefficient is used. The expectation value of a discrete random variable *X* is $\sum x P(x)$ where *x* is the value of *X* and *P(x)* is the probability of that value. For example, consider flipping a fair coin. If one gives the value 1.0 to heads and -1.0 to tails, the expectation value of the coin flip is 0.0 = 0.5*1.0 + 0.5*(-1.0).

```
X = Table[π RandomReal[], {i, 1000}]; Y = Sin[X];
```

```
Correlation[X, X] (* the correlation of a data sample with itself is 1
```

```
1.
```

```
Correlation[X, 2 X] (* the correlation of a data sample with 2.0 times
```

```
1.
```

```
Correlation[X, -X]  (* the correlation of a data sample with -1.0 time
```

```
-1.
```

```
Correlation[X, -2 X] (* the correlation of a data sample with -2.0 tim
```

```
-1.
```

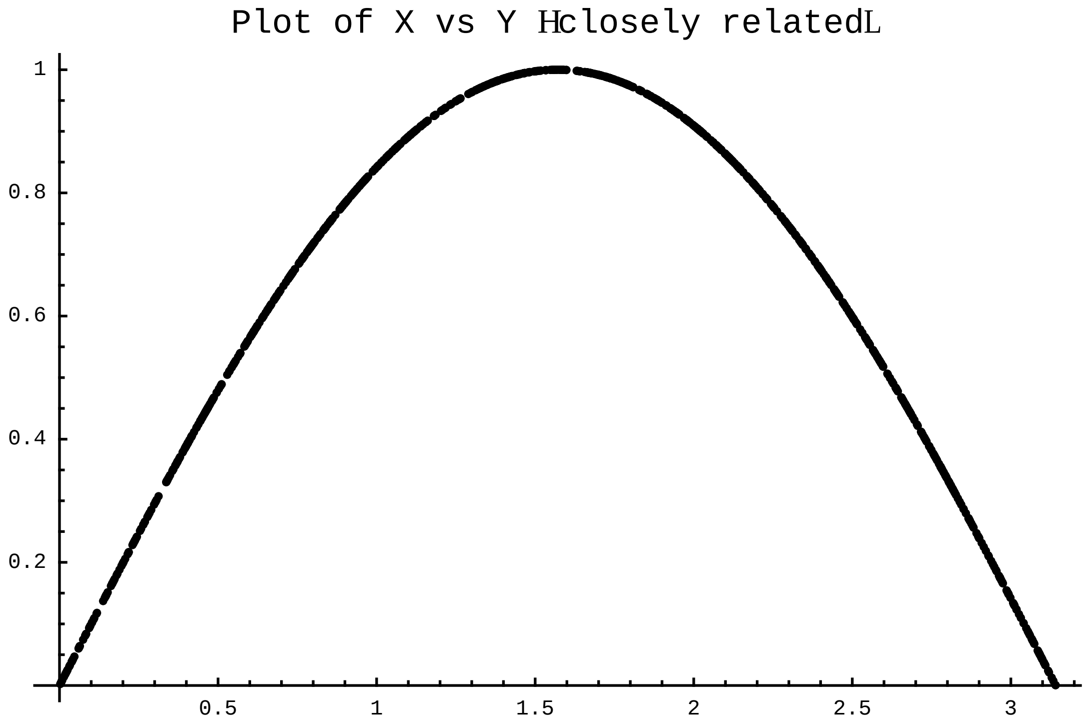
The variables X and Y are essentially uncorrelated as measured by the standard correlation coefficient. The standard correlation coefficient is almost zero. In fact, the expectation value of the correlation coefficient is exactly zero in this example.

```
Correlation[X,Y] (* the correlation between a data sample X and Y = Si
```

```
-0.00465432
```

However, the variables *X* and *Y* are actually closely related because *Y* = *Sin(X)*, a non-linear relationship that is not detected by the correlation coefficient:

```
ListPlot[Transpose[{X, Y}],
 PlotLabel→Style["Plot of X vs Y (closely related)", FontSize→16],
 ImageSize→{72×7, 72×7}]
```



More sophisticated statistical methods and statistical models can detect and measure a non-linear relationships such as *Y=Sin(X)*. These methods often involve either explicitly or implicitly fitting a mathematical model of the relationships to the data. A simple mathematical model of the frequency of words and phrases in articles is shown below.

Correlation does not mean causation. If two variables X and Y are correlated, X may cause Y, Y may cause X, or X and Y may share a common cause. Even perfectly correlated variables can have a tenuous or indirect relationship. Consider for example articles on the web covering a niche issue such as scram-jet engines. All articles in this case might be written by one aerospace journalist, say "Karman Von Theodore", All articles on scram-jets would be perfectly correlated with "Karman Von Theodore" but the correlation would be quite tenuous. If "Karman Von Theodore" retired or ceased writing articles on scram-jet engines, the correlation would suddenly disappear. It requires more than a correlation to determine the nature of the relationship between two or more variables.

N-Grams

An N-Gram is a simple concept used in speech recognition, machine translation, and language processing. A 1-gram or unigram is simply a single word such as "the" or "rocket". A 2-gram or bigram is simply a pair of adjacent words such as "ice cream" or "I scream". A 3-gram or trigram is simply three adjacent words such as "ice cream cone" or "I scream loudly". An N-gram is simply N adjacent words where N is any integer (1,2,3,...).

Standard Hidden Markov Model (HMM) based speech recognition algorithms make heavy use of N-Grams, often trigrams, to recognize speech. Even if speech recognition algorithms could recognize the sounds in speech, the so-called phonemes such as the "AH" sound in "father", as well as human beings, speech contains homonyms and near-homonyms such as "I scream" and "ice cream" that cannot be distinguished by sound alone. Human beings resolve homonyms and near-homonyms by actually understanding the speech and determining the correct words from context. Computer programs are a long way from actual understanding of language. However, speech recognition algorithms can use statistical models of the frequency of N-grams to determine what was said. For example, consider the trigrams "I scream loudly" and "ice cream cone". People are less likely to say "I scream cone" or "ice cream loudly" than "I scream loudly" or "ice cream cone". The words "loudly" and "cone" are not homonyms or near homonyms. They are acoustically distinctive. Thus, a speech recognition algorithm can use the frequency of trigrams such as "I scream loudly" and "ice cream cone" to successfully resolve "I scream" and "ice cream".

Current HMM speech recognition algorithms use statistical language models of the frequency of N-Grams to significantly improve the accuracy of speech recognition. The actual accuracy of current speech recognition algorithms in correctly identifying the basic sounds in speech, the phonemes such as the "AH" sound in "father", is quite poor compared to human beings, probably less than 80% accurate compared to 95-100% accuracy for humans depending on conditions. The automatic speech recognition systems used in telephone help lines, where context information is limited, are usually designed to use a small vocabulary of words and phrases that are very acoustically distinctive such as "YES" and "NO" to get around the limitations of current speech sound recognition (known as the acoustic model in the jargon of speech recognition). Even so, errors still occur. Even the spectra of "YES" and "NO" are more similar than one would think based on human hearing and it is more difficult than most people would think to develop an algorithm to discriminate even between these two sounds.

N-Grams can be correlated with the topic of a document or the subject of a conversation. For example, "ice cream" is more likely to be encountered in a story about a birthday party than a story about a funeral. "I scream" is more likely to be encountered in a horror story than a romantic comedy.

Combining Correlations and N-Grams

Correlation coefficients and N-Grams can be combined to illustrate finding documents by topic using statistical language processing methods. One needs a training set of documents labeled by topic. Assign a topic variable a value of 1 if the document is about the topic, a value of 0 if the document is not about the topic. Assign an N-Gram variable a value of 1 if the N-Gram, e.g "rocket scientist", is in a document, 0 if the N-Gram is not present. Then, one can compute correlation coefficients for the topic and the N-Gram. As a first pass, one is looking for N-Grams that are highly correlated with the topic. A correlation coefficient of 1.0 would indicate an N-Gram that is perfectly correlated with the topic in the training set of documents. When an end user specifies a topic to search for, the search engine will use the N-Grams identified from the training set to locate documents that are likely to be related to the topic of interest.

Example Ruby Code

The Ruby code that follows takes a training set and outputs a sorted list of correlation coefficients between unigrams, bigrams, and trigrams in the training set and the topics. The program takes an input file, *train.txt*, that lists the training set files and the topics. It is a simple example of the concept. An actual search engine would be more complex (obviously) and use more sophisticated statistical methods than correlation coefficients. A brief overview of more advanced statistical methods follows the sample code.

`train.txt (a sample training file with documents labeled by topic -- test or rocket)`

```
test.txt test
test2.txt test
test3.txt rocket
test4.txt rocket
test5.txt rocket
```

`test.txt  (sample document from training set)`

```
This is a test.
This is only a test.
This is still a test.
This was a test.
If this had been real, you would be dead.
```

test2.txt (sample document from training set)

```
my test
this is my test
it is not your test
your test is dismal
my test is better
```

test3.txt (sample document from training set)

```
rockets are very difficult to design and build
The failure rate of rockets is very high
About 2% of manned rockets fail.
About 5% of unmanned rockets fail.
rocket engines are the most powerful engines on Earth.
```

test4.txt (sample document from training set)

```
my rocket is bigger than your rocket
my rocket is faster than your rocket
my rocket is cheaper than your rocket
my rocket looks snazzier than your rocket
```

test5.txt (sample document from training set)

```
rockets are actually quite old
rockets date back over one thousand years
black powder or gunpowder was known long before it became an effective
weapon
rockets and fireworks were made using black powder long before cannons
rockets made from black powder are very unreliable and difficult to aim
During World War II Jack Parsons found an alternative to black powder
that was much better.
```

trigram.rb

```
=begin

Name: Trigram
Author: John F. McGowan, Ph.D.
Contact: jmcgowan11@earthlink.net
Web: http://www.jmcgowan.com
Copyright (C) 2009, John F. McGowan

Description: Trigram counts the ngrams in a training set of text files
and computes
the correlation coefficient between the trigram and a topic (e.g.
"rockets") over the training set.
An n-gram is a sequence of n words in a file.  A unigram is just a single
word (e.g. "rocket").  A
bigram is two words in sequence (e.g. "rocket exploded").  A trigram is
three words in sequence
(e.g. "rocket exploded violently").  A topic such as "rocket" refers to
the meaning, the semantics, of
a web page or document.  An ngram such as "rocket" or "rocket scientist"
can appear in a web page or
document that has nothing to do with the topic.  For exmaple, "the rocket
scientists on Wall Street
wrecked the global economy using the Gaussian copula model." in an
article on mortgage backed
```

securities.

The correlation coefficient ranges from -1.0 to 1.0. 1.0 means a perfect correlation in the training set. The ngram is always present in documents about the topic and never present in documents not about the topic. 0.0 means no correlation. The presence or absence of the ngram tells nothing about the topic. For example, common words such as "the" or "is" are found in almost all documents.
-1.0 means the ngram is perfectly negatively correlated with the topic. If the ngram is present, the document is not about the topic.

Statistical models of the frequency of n-grams are used in speech recognition and machine translation.
They can also be used in web or other document search applications to locate target web pages or documents by topic (e.g. rocketry). One needs to find statistical patterns of words that are highly correlated with the search topic. Web pages or other documents can be hand labeled or one can use keyword tags, meta tags, and so forth to identify the topic(s) of the document for training purposes.

At present the trigram script is a simple demonstration of the concept. A real search engine would use more sophisticated statistical models.

```
Files:  trigram.rb  # main script
        train.txt   # demo list of training files
        triout.txt  # sample output file using the sample training files

        Training File   Topic  (in train.txt)
        test.txt test
        test2.txt test
        test3.txt rocket
        test4.txt rocket
        test5.txt rocket
```

Platform: Developed and tested (briefly) on Windows XP Service Pack 2 (2.6 GHz) Sony Vaio with Ruby 1.86 installed.

```
About the Author

John F. McGowan, Ph.D. is a software developer, research scientist, and
consultant. He works
primarily in the area of complex algorithms that embody advanced
mathematical and
logical concepts, including speech recognition and video compression
technologies. He has many
years of experience developing software in Visual Basic, C++, and many
other programming languages
and environments. He has a Ph.D. in Physics from the University of
Illinois at Urbana-Champaign
and a B.S. in Physics from the California Institute of Technology
(Caltech).
He can be reached at jmcgowan11@earthlink.net.

=end

class Trigram
    def initialize(file_name)
        @file = file_name
        @tokens = []
        @unigrams = {}
        @bigrams = {}
        @trigrams = {}
        @allgrams = {}

        puts "reading #{file_name}"
        @lines = IO.readlines(file_name)
        puts "read #{@lines.length} lines "
        for i in 0..@lines.length-1
            @tokens = @tokens + @lines[i].split
            # puts "\nLine #{i}"
            #puts @tokens
        end

        for i in 0..@tokens.length-1
            puts "checking #{@tokens[i]}"

            if @tokens[i][-1].chr == "." || @tokens[i][-1].chr == "," ||
@tokens[i][-1].chr == "!" || @tokens[i][-1].chr == "?"
                puts "removing trailing ."
                @tokens[i] = @tokens[i].chop
            end
            @tokens[i] = @tokens[i].upcase  # don't worry about case for 
now
        end


        @tokens.each { |tok|

                # puts "checking #{tok} "
                if @unigrams[tok]
                    @unigrams[tok] = @unigrams[tok] + 1
                                else
                    @unigrams[tok] = 1
                end
        }  # collect histogram of 1-grams

        for i in 0..@tokens.length-2
            if @bigrams[@tokens[i] + "." + @tokens[i+1] ]
                @bigrams[@tokens[i] + "." + @tokens[i+1]] = 
@bigrams[@tokens[i] + "." + @tokens[i+1]] + 1
            else
                @bigrams[@tokens[i] + "." + @tokens[i+1]] = 1
            end
        end  # collect histogram of bigrams

        for i in 0..@tokens.length-3
            trig = @tokens[i] + "." + @tokens[i+1] + "." + @tokens[i+2]
            if @trigrams[trig]
                @trigrams[trig] = @trigrams[trig] + 1
            else
                @trigrams[trig] = 1
            end
        end  # collect histogram of trigrams


        @allgrams = @unigrams.merge(@bigrams)
        @allgrams = @allgrams.merge(@trigrams)

    end  # initialize

    def show
        puts "1-grams #{@file}"
        puts @unigrams
        puts "\n2-grams #{@file}"
        puts @bigrams
        puts "\n3-grams #{@file}"
        puts @trigrams

        puts "\nAll N Grams #{@file}"
        puts @allgrams
    end

    def unigrams
        return @unigrams.keys
    end

    def bigrams
        return @bigrams.keys
    end

    def trigrams
        return @trigrams.keys
    end

    def allgrams
        return @allgrams.keys
    end

    def has_ngram?(ngram)
        return @allgrams.has_key?(ngram)
    end

end


class Trainer
    def initialize(file_name)
        @file = file_name  # file contains a list of file <space> topic
        @training_file = []
        @topic = []
        @topic_file = {}
        @topic_average = {}
        @StatData = {}
        @number_files = {}
        @allmatches = {}
        @allfilem = {}
        @allngrams = []
        @correlation = {}  # indexed by n-gram and topic

        @lines = IO.readlines(file_name)

        @lines.each { |line|
            @dummy = line.split
            @training_file.push @dummy[0]  # text file
            @topic.push @dummy[1]          # topic
            @topic_file[@dummy[0]] = @dummy[1]  # quick lookup of file
topic
            @StatData[@dummy[0]] = Trigram.new(@dummy[0])
            @StatData[@dummy[0]].show
            @local_tokens = []

            if @number_files[@dummy[1]]  # number of files on topic
                @number_files[@dummy[1]] = @number_files[@dummy[1]] + 1
            else
                @number_files[@dummy[1]] = 1.0
            end

# count how many files contain a unigram
            @matches = @StatData[@dummy[0]].unigrams  # list of unigrams
for file

            @matches.each { |match|
                key = match + "." + @dummy[1]  # combine keyword and topic
                if @allmatches[key]
                    @allmatches[key] = @allmatches[key] + 1.0
                else
                    @allmatches[key] = 1.0
                end

                if @allfilem[match]
                    @allfilem[match] = @allfilem[match] + 1.0
                else
                    @allfilem[match] = 1.0
                end

            } # end of block

# count how many files contain a bigram
```

```
            @matches = @StatData[@dummy[0]].bigrams  # list of bigrams
for file

            @matches.each { |match|   # match is the bigram
                key = match + "." + @dummy[1]  # combine keyword and topic
                if @allmatches[key]
                    @allmatches[key] = @allmatches[key] + 1.0
                else
                    @allmatches[key] = 1.0
                end

                if @allfilem[match]
                    @allfilem[match] = @allfilem[match] + 1.0
                else
                    @allfilem[match] = 1.0
                end

            }  # end of block

# count how many files contain a trigram
            @matches = @StatData[@dummy[0]].trigrams  # list of trigrams
for file

            @matches.each { |match|   # match is the trigram
                key = match + "." + @dummy[1]  # combine keyword and topic
                if @allmatches[key]
                    @allmatches[key] = @allmatches[key] + 1.0
                else
                    @allmatches[key] = 1.0
                end

                if @allfilem[match]
                    @allfilem[match] = @allfilem[match] + 1.0
                else
                    @allfilem[match] = 1.0
                end

            }  # end of block

            @allngrams = @allngrams + @StatData[@dummy[0]].allgrams  #
build list of all ngrams in all files

        } # loop over files in training set
        @number_files["ALL"] = @lines.length.to_f

        # @allmatches = @allmatches / @number_files

        @allmatches.each_key { |key|
                @local_tokens = key.split(".")
                @allmatches[key] = @allmatches[key] /
@number_files[@local_tokens[-1]]
        }

        @allfilem.each_key { |key|
                @allfilem[key] = @allfilem[key] / @number_files["ALL"]
        }       # proportion of training files with ngram.topic compound
key

        @topic.each { |top|
            @topic_average[top] = @number_files[top]/@number_files["ALL"]
        }   # proportion of training files with topic

        @allngrams.each { |ngram|  # loop over ngrams in training set
            @topic.each { |top|  # loop over topics in training set

                puts "computing correlation coefficient for #{ngram} AND
TOPIC: #{top}"

                compound_key = ngram + top

                @correlation[compound_key] = 0.0  # start at zero
                covariance = 0.0
                variance_ngram = 0.0
                variance_topic = 0.0

                # @correlation[ngram.topic] =
covariance/Math.sqrt(variance_ngram*variance_topic)


                @training_file.each { |file|
                    if @topic_file[file] == top
                        delta_top = (1.0 - @topic_average[top])
                    else
                        delta_top = (0.0 - @topic_average[top])
                    end
                    variance_topic = variance_topic + delta_top**2

                    if @StatData[file].has_ngram?(ngram)
                        delta_ngram = (1.0 - @allfilem[ngram])
                    else
                        delta_ngram = (0.0 - @allfilem[ngram])
                    end
                    variance_ngram = variance_ngram + delta_ngram**2

                    covariance = covariance + delta_top*delta_ngram

                }   # loop over files in training set

                @correlation[compound_key] = covariance/
Math.sqrt(variance_ngram*variance_topic)
            }
        }

    end

    def show
        puts "\nTraining Files"
        puts @training_file
        puts "\nTopics"
        puts @topic

        puts @StatData

        puts "\nUnigram Match Statistics"
        sorted = @allmatches.sort {|a,b| a[1]<=>b[1]}  # sort result by
value (not key)
        sorted.each { |match| puts match }
        puts "\nUnigram Statistics over All Files"
        sorted = @allfilem.sort {|a,b| a[1] <=> b[1]}  # sort by value
        sorted.each { |match| puts match }

        puts "\nTopic Average"
        puts @topic_average


        puts "\nCorrelations"
        sorted = @correlation.sort {|a,b| a[1] <=> b[1]}  # sort by value
        sorted.each { |match| puts match }
    end
end

# simple demo

#t = Trigram.new("test.txt")

#t.show

set = Trainer.new("train.txt")

set.show
```

Advanced Statistical Models of Text

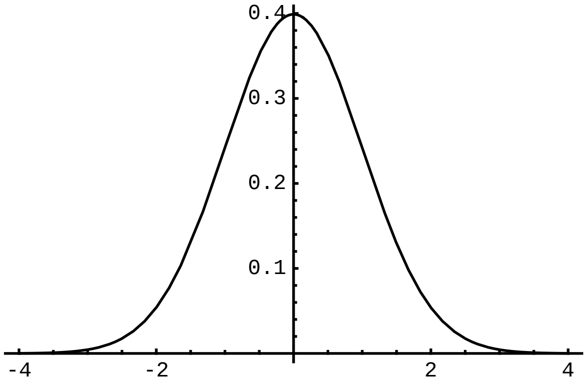
An advanced search engine searches by *topic* not by direct key word or phrase matching. For example, one end user might search for articles on the topic "Wall Street", meaning the financial industry, not the exact phrase "Wall Street". Another end user might search for articles on the topic "NASA," the National Aeronautics and Space Administration. Instead of searching for the phrase "Wall Street" or "NASA", an advanced search engine searches for statistical patterns of words and phrases that occur when the topic "Wall Street" or "NASA" is discussed. For example, both "space" and "rocket scientist" tend to occur in articles about NASA. However, the phrase "rocket scientist" also occurs in articles about so-called financial engineering on Wall Street (meaning in the financial industry). In general, an advanced search engine needs sophisticated statistical models of the frequency and/or location of words and phrases in a document as a function of *topic*.

The Gaussian (also known as the Normal or Bell Curve) distribution or function is a common function in standard statistics. The Gaussian is frequently used as a key building block of statistical and mathematical models. The Gaussian is probably overused in mathematical modeling. However, for illustrative purposes, the Gaussian is used here to illustrate more advanced statistical models of the frequency and location of words and phrases (n-grams) in text.

```
HoldForm[ N[x, μ, σ_] = (1.0/ (Sqrt[2 Pi] σ) Exp[ -(1./2.)(x - μ)^2/σ^
```

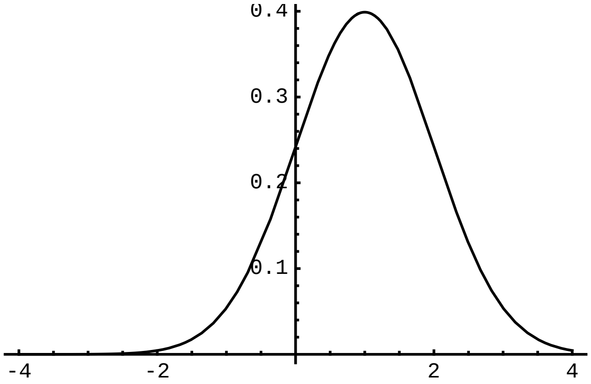
$$N[x, \mu, \sigma\_] = \frac{1.\, e^{-\frac{1.\,(x-\mu)^2}{2.\,\sigma^2}}}{\sqrt{2\pi}\,\sigma}$$

```
Gaussian[x_, μ_:0.0, σ_:1.0] := (1.0/ (Sqrt[2 Pi] σ) Exp[ -(1./2.)(x -
Plot[Gaussian[x], {x, -4,4}]  (* make a plot of the Gaussian or Normal
```



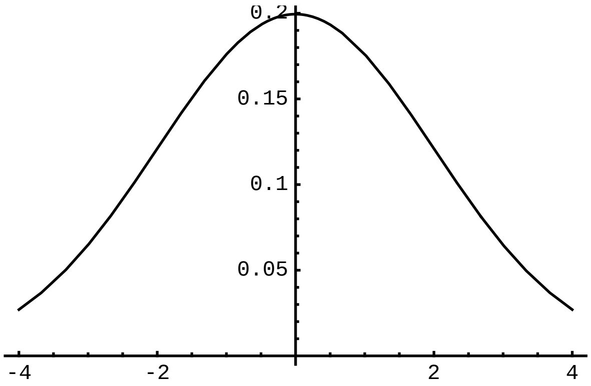
```
- Graphics -
```

```
Plot[Gaussian[x, 1], {x, -4,4}]  (* make a plot of the Gaussian with a
```



```
- Graphics -
```

```
Plot[Gaussian[x, 0, 2], {x, -4,4}]  (* make a plot of the Gaussian witl
```

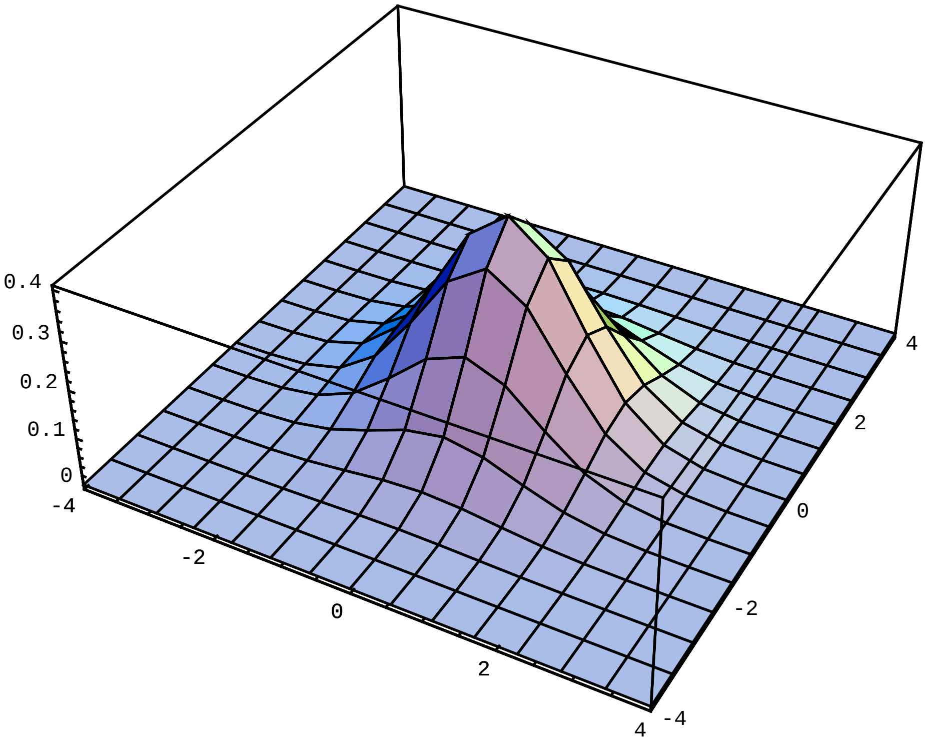


```
- Graphics -
```

The Gaussian can be extended to two (or more) dimensions. For statistical language processing, one can define the two dimensions *x* and *y* as the frequency of two different words or phrases in a document, for example "space" and "rocket scientist" in articles about NASA and Wall Street financial engineering.

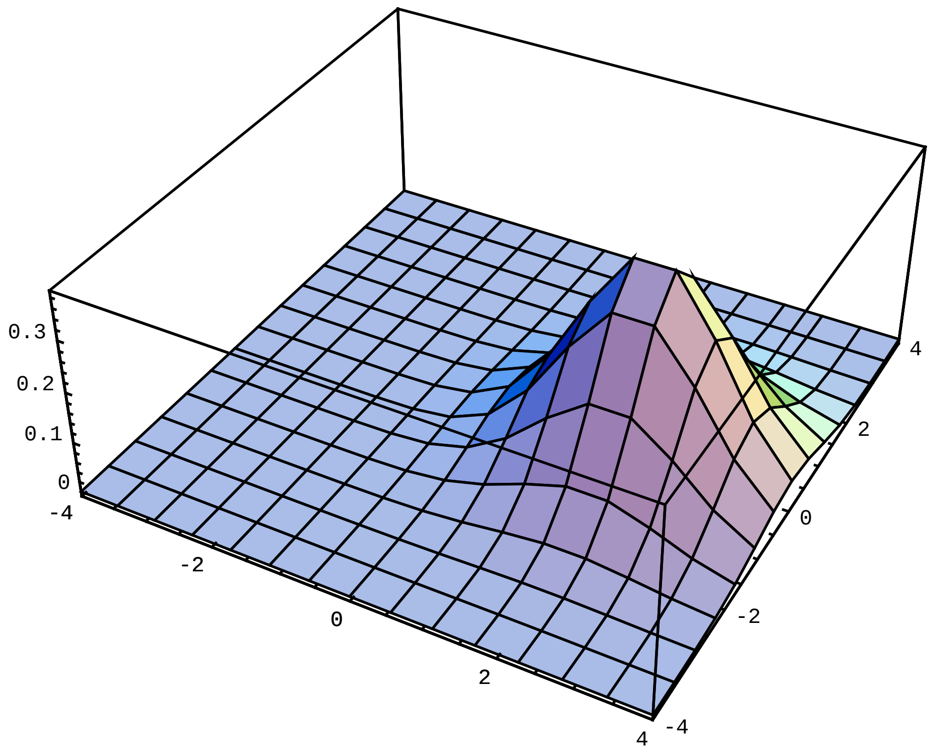
```
Gaussian2D[x_, μ_:{0.0, 0.0}, σ_:{{1.0, 0.0}, {0.0, 1.0}}] := (1.0/ (S

Plot3D[ Gaussian2D[{x,y}], {x, -4, 4}, {y, -4, 4}, PlotRange->All, Ima
```



```
- SurfaceGraphics -
```

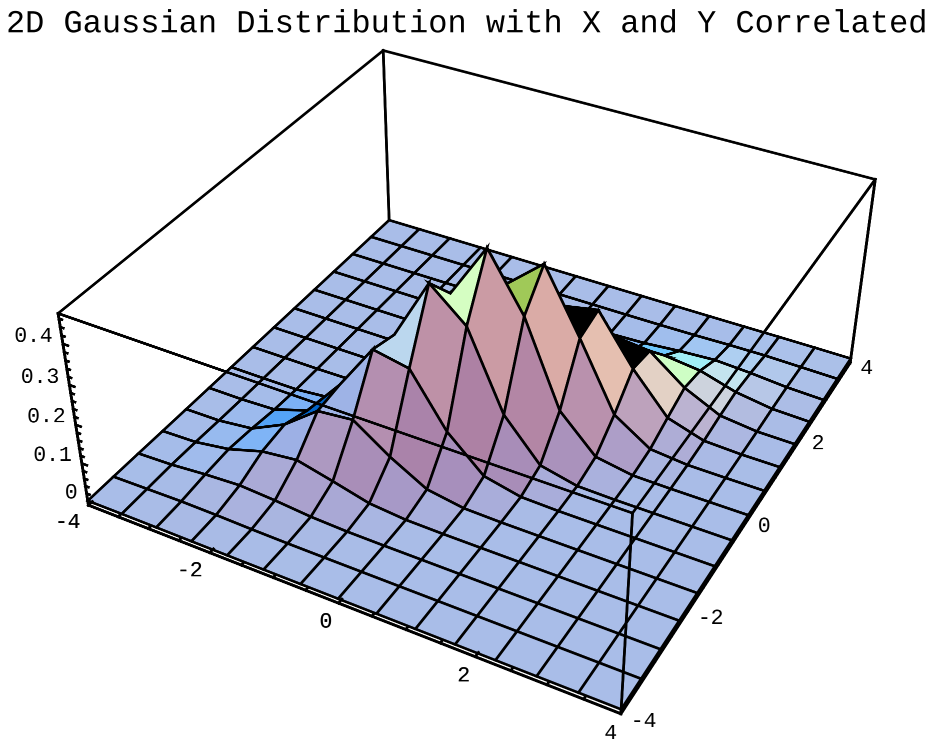
```
Plot3D[ Gaussian2D[{x,y}, {2., 0.0}], {x, -4, 4}, {y, -4, 4}, PlotRang
```



```
- SurfaceGraphics -
```

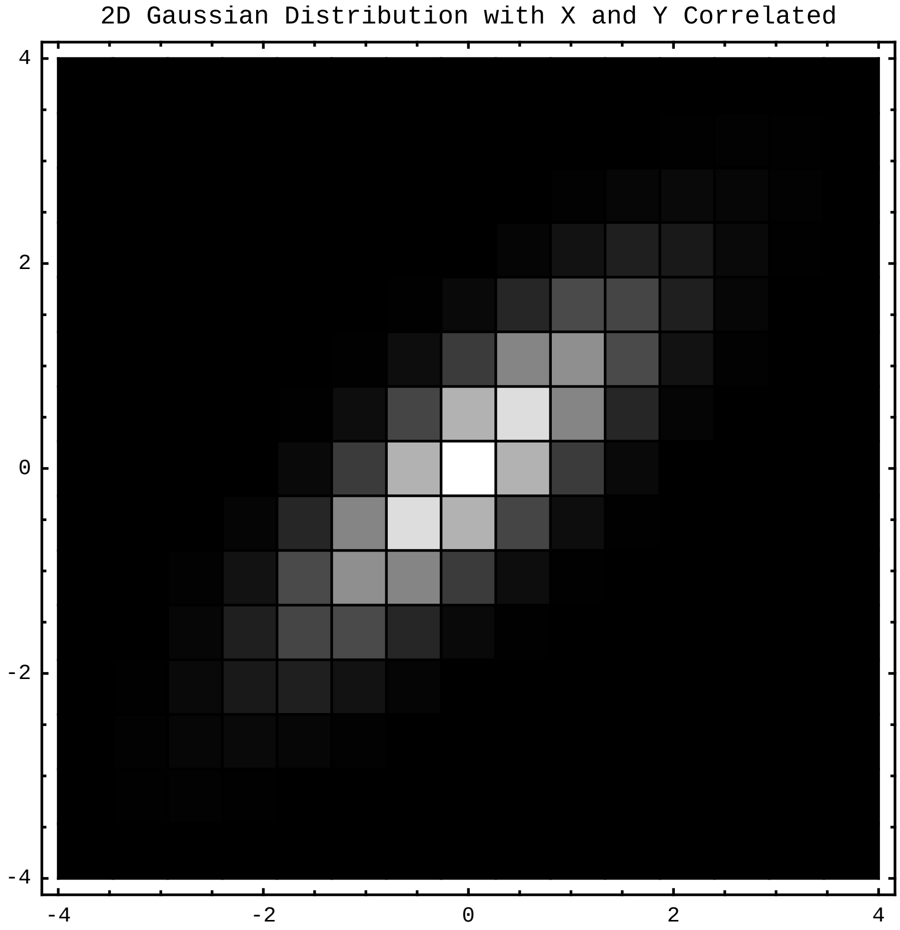
The plots below illustrate the distribution of *X* (use of "space") and *Y* (use of "rocket scientist") in articles about NASA. The frequency of "space" and "rocket scientist" is correlated in articles about NASA.

```
Plot3D[Gaussian2D[{x, y}, {0.`, 0.`}, {{1.`, 0.5`}, {0.5`, 1.`}}],
 {x, -4, 4}, {y, -4, 4}, PlotRange → All,
 PlotLabel → Style["2D Gaussian Distribution with X and Y Correlated",
   FontSize → 15], ImageSize → {72 × 6, 72 × 6}]
```



```
- SurfaceGraphics -
```

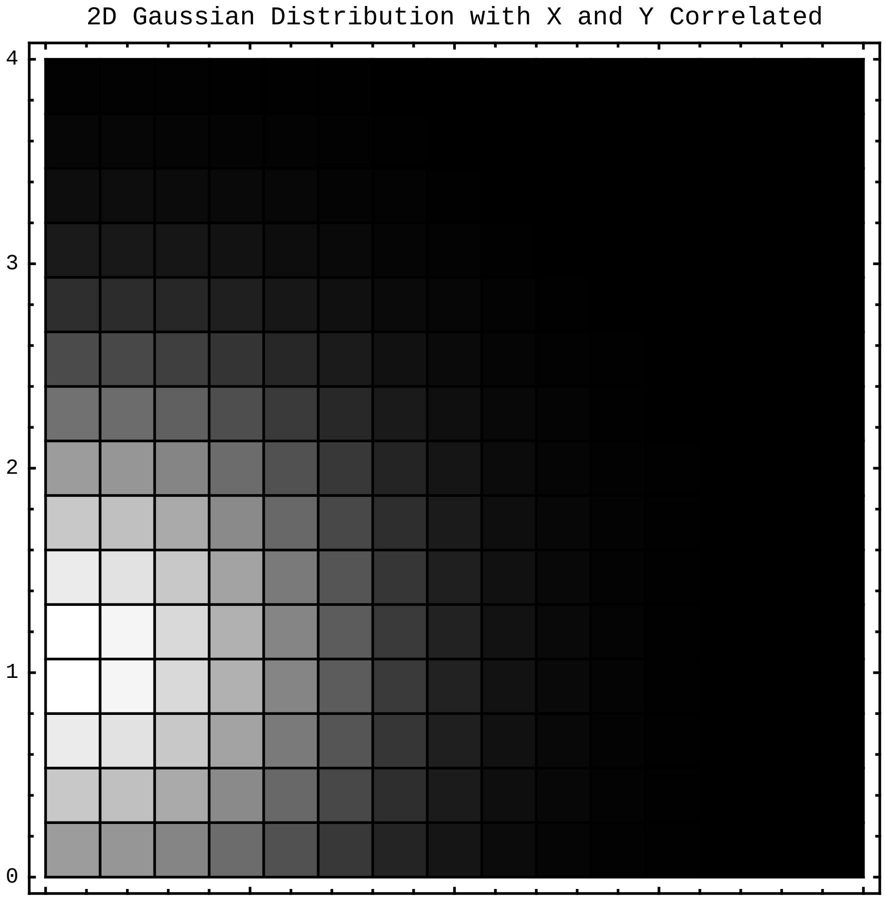
```
DensityPlot[Gaussian2D[{x, y}, {0.`, 0.`}, {{1.`, 0.5`}, {0.5`, 1.`}}],
 {x, -4, 4}, {y, -4, 4}, PlotRange → All,
 PlotLabel → Style["2D Gaussian Distribution with X and Y Correlated",
   FontSize → 12], ImageSize → {72 × 6, 72 × 6}]
```



```
- DensityGraphics -
```

The distribution (below) of X (use of "space") versus Y (use of "rocket scientist") for articles about Wall Street. "Rocket scientist" is used 1% of time. "Space" is uncorrelated with the use of "rocket scientist".

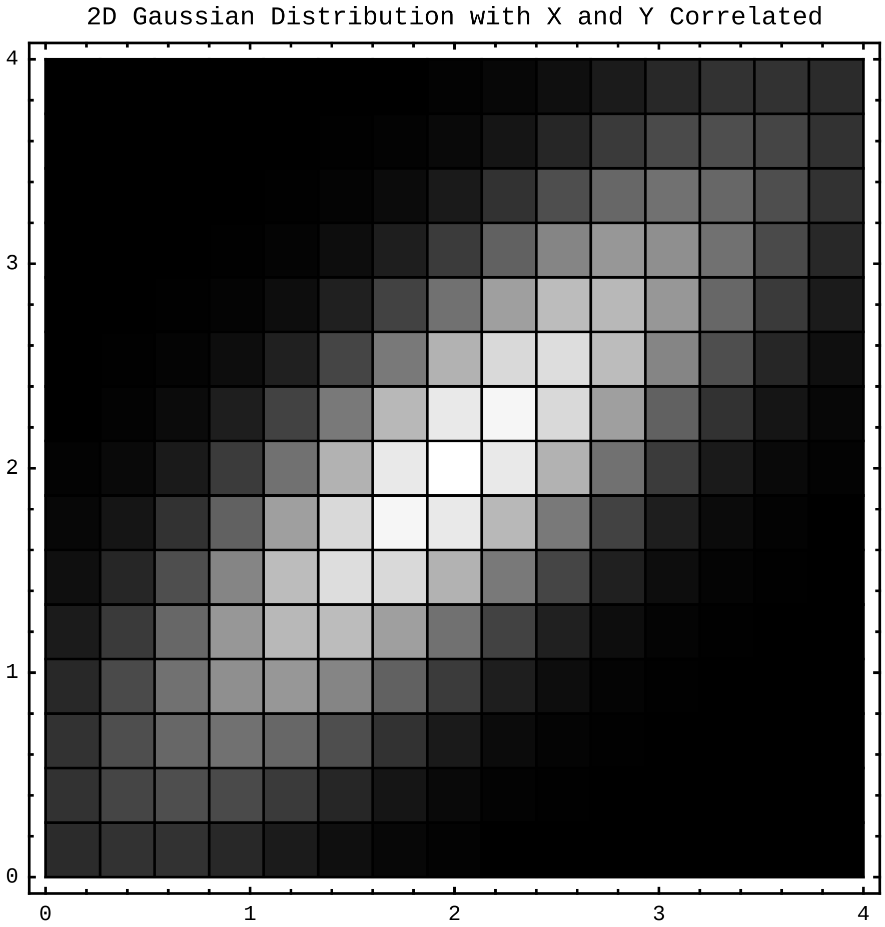
```
WallStreet =
 DensityPlot[Gaussian2D[{x, y}, {0.`, 1.`}, {{1.`, 0.`}, {0.`, 1.`}}],
  {x, 0, 4}, {y, 0, 4}, PlotRange → All, PlotLabel →
   Style["2D Gaussian Distribution with X and Y Correlated",
    FontSize → 12], ImageSize → {72 × 6, 72 × 6}]
```



```
- DensityGraphics -
```

The distribution (below) of X (use of "space") versus Y (use of "rocket scientist") for articles about NASA. "Rocket scientist" is used 2% of time and "space" is used 2% of time on average. "Space" is highly correlated with the use of "rocket scientist".

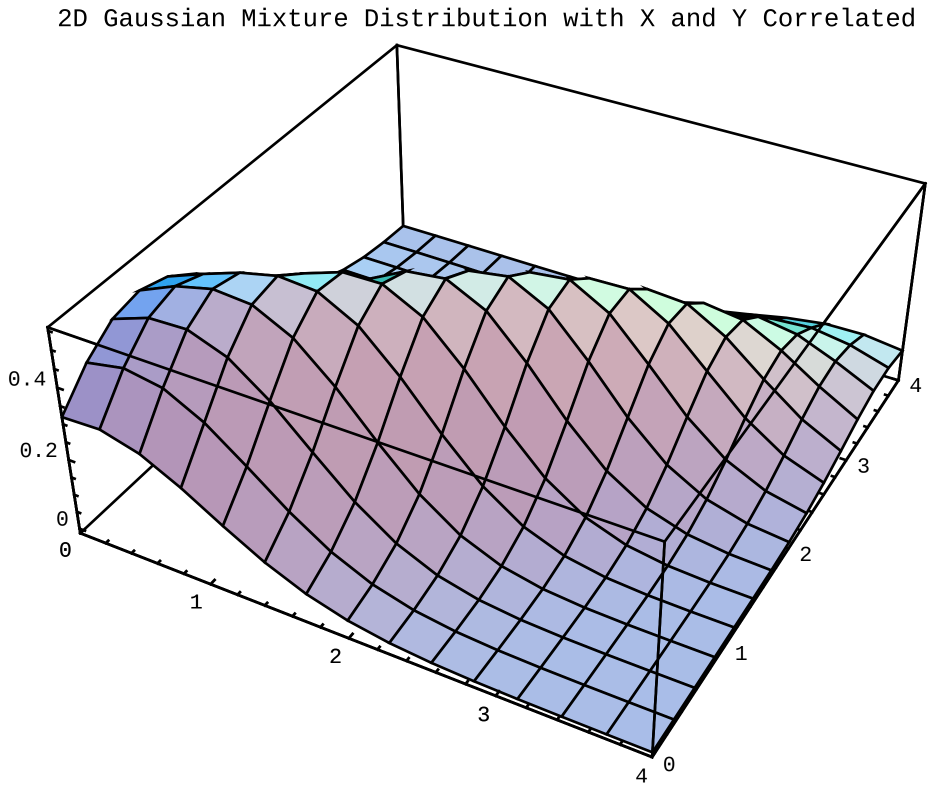
```
NASA =
 DensityPlot[Gaussian2D[{x, y}, {2.`, 2.`}, {{1.`, 0.5`}, {0.5`, 1.`}}],
  {x, 0, 4}, {y, 0, 4}, PlotRange → All, PlotLabel →
   Style["2D Gaussian Distribution with X and Y Correlated",
    FontSize → 12], ImageSize → {72 × 6, 72 × 6}]
```



```
- DensityGraphics -
```

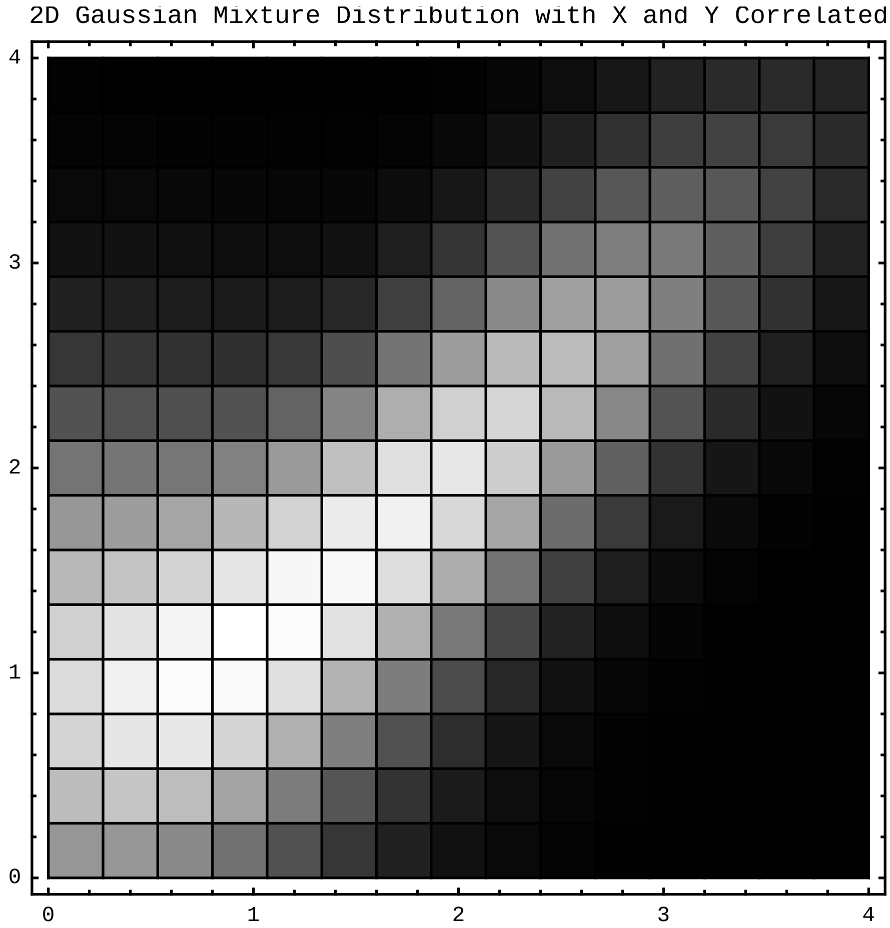
The plots below illustrate the distribution of *X* (use of "space") and *Y* (use of "rocket scientist") in all articles, articles about both NASA and Wall Street.

```
Both3D = Plot3D[Gaussian2D[{x, y}, {0.`, 1.`}, {{1.`, 0.`}, {0.`, 1.`}}] +
    Gaussian2D[{x, y}, {2.`, 2.`}, {{1.`, 0.5`}, {0.5`, 1.`}}],
   {x, 0, 4}, {y, 0, 4}, PlotRange → All, PlotLabel →
    Style["2D Gaussian Mixture Distribution with X and Y Correlated",
     FontSize → 12], ImageSize → {72 × 6, 72 × 6}]
```



```
- SurfaceGraphics -
```

```
WallSteetAndNASA =
 DensityPlot[Gaussian2D[{x, y}, {0.`, 1.`}, {{1.`, 0.`}, {0.`, 1.`}}] +
   Gaussian2D[{x, y}, {2.`, 2.`}, {{1.`, 0.5`}, {0.5`, 1.`}}],
  {x, 0, 4}, {y, 0, 4}, PlotRange → All, PlotLabel →
   Style["2D Gaussian Mixture Distribution with X and Y Correlated",
    FontSize → 12], ImageSize → {72 × 6, 72 × 6}]
```



```
- DensityGraphics -
```

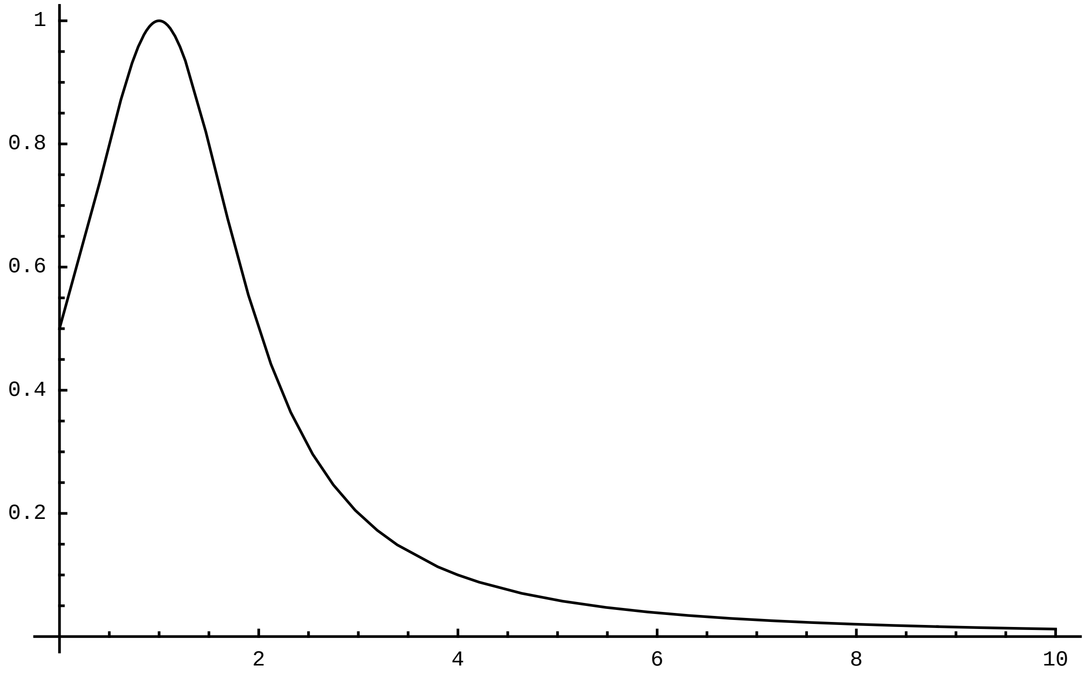
Notice that there is some overlap in the example above. If both "rocket scientist" and "space" make up 2% of the words and phrases (N-Grams) in a document (article), then it is almost certainly about NASA. If "rocket scientist" is used 2% of the time and "space" is never used (0% of the time), then the document (article) is almost certainly about Wall Street. If both "rocket scientist" and "space" are used 1 % of the time, the document could be about either Wall Street or NASA. One can in fact compute a probability that the article is about Wall Street (e.g. 30%) or about NASA (e.g. 70%) based on the statistical model of the frequency of "rocket scientist" and "space" in documents. To resolve the overlaps would require a more sophisticated model of additional N-Grams. For example, one might include the frequency of other words and phrases such as "NASA" or "bond trader" in the document to improve the accuracy.

Note however that statistical models cannot resolve some cases. They can only imitate human understanding. For example, former US Under Secretary of the Treasury and Goldman Sachs executive Neel Kashkari actually worked for NASA prior to moving to Wall Street. His NASA background was frequently mentioned in news articles about him. Consequently, a statistical model would have difficulty identifying an article about Neel Kashkari as an article about Wall Street. Some Wall Street "rocket scientists" actually are rocket scientists. Note also that many articles about NASA discuss the federal budget, the US Treasury, and other government finance issues making separation based on financial key words and phrases problematic (probably the presence of "Goldman Sachs" in an article would solve the problem in Neel Kashkari's case). Only actual understanding of the text could guarantee a perfect search result in some cases.

Real world distributions are rarely Gaussians. They are usually more complex and often exhibit long "non-Gaussian" tails. The Gaussian is used here only for illustrative purposes. Incidentally, a number of prominent Wall Street financial models including notably the famous Black-Scholes option pricing model and the Gaussian copula model used to value mortgage backed securities make heavy use of the Gaussian even though most financial assets exhibit non-Gaussian tails. These financial models tend, therefore, to understate the risk (and overstate the value) of various financial assets. A real statistical model of English would use more complex models than these simple Gaussian examples to imitate human understanding of text.

```
Plot[1.`/(1.` + (x - 1.`)^2), {x, 0.`, 10.`},
 PlotLabel → Style["A Distribution with a Non-Gaussian Tail",
   FontSize → 18], ImageSize → {7 × 72, 7 × 72}]
```

A Distribution with a Non-Gaussian Tail



```
- Graphics -
```

The advantage of using statistical models of language is that proven statistical methods, including some advanced pattern recognition methods, already exist, whereas actual understanding of language is an unsolved basic research problem. A company or organization need not invest in years of high risk basic research to achieve improved search results.

■ Mathematical Models

The example above is a simple example of type of mathematical model known as a Gaussian Mixture Model (GMM) composed of sums of two or more Gaussian distributions. A Gaussian Mixture Model can, with difficulty, approximate any other distribution. Unless the distribution being approximated is actually a Gaussian Mixture Model formed from a finite number of Gaussians, the approximation will often require a very large number of Gaussians and still have significant errors. Present day speech recognition algorithms use extremely complex Gaussian Mixture Models to recognize the basic sounds in speech, known as phonemes, for example, the "AH" sound in "father". The Gaussian Mixture Models in speech recognition algorithms today achieve rather limited accuracy, as any user of automatic speech recognition systems such as telephone customer service lines can attest to. Other, hopefully simpler, mathematical models may be needed to achieve higher performance in speech recognition or in text search.

Constructing mathematical models that work well is something of a "black art". There are standard techniques such as Gaussian Mixture Models but these often require tweaking to use effectively. There are a range of fitting methods used to find mathematical models but these also have many pitfalls and often require "hand tuning" by the mathematical modeler. It usually requires a significant amount of trial and error as well as insight to find a good mathematical model.

Conclusion

Searching for documents and other items on the Web or computers often takes a long time. Highly paid professionals spend hours, days, and even longer searching for information on the Web or computers. With professional salaries of tens to hundreds of dollars per hour, lengthy searches cost hundreds to thousands of dollars per search. Highly paid professionals may conduct hundreds, even thousands, of searches each year. The cost of lengthy searches can add up to tens of thousands, hundreds of thousands, even millions of dollars for just one professional. The cost to organizations with more than one professional can easily be many millions of dollars. The cost to the economy as a whole is probably billions of dollars. More powerful search engines can save time, money, and frustration – and ensure success.

Search engines today are based primarily on matching words and phrases often weighted by the popularity of documents, advertising dollars, and other adjustments. In practice, end users often spend many minutes, hours, or days paging through search results and/or trying many different search words and phrases trying to find a relevant document or item. More powerful search methods are needed.

The cause of the costly state of search is that present-day search engines do not understand either the search queries or the documents or items searched. More powerful search engines need understanding or a way to emulate aspects of human understanding of text. The dream search engine should find documents or other items *by topic*, not by word or phrase, and return only documents or items related to the topic of interest. Actual understanding of natural language by computers has proven extremely difficult, like most problems in artificial intelligence (AI), and successes have been few and limited. Hence, search engines continue to rely on simple word and phrase matching. While actual understanding is probably decades, if not centuries, in the future, statistical language processing methods based on the frequency and location of words and phrases in documents can achieve a degree of searching *by topic* today.

About the Author

John F. McGowan, Ph.D. *is a software developer, research scientist, and consultant. He works primarily in the area of complex algorithms that embody advanced mathematical and logical concepts, including speech recognition and video compression technologies. He has extensive experience developing software in C, C++, Visual Basic, Mathematica, and many other programming languages. He is probably best known for his AVI Overview, an Internet FAQ (Frequently Asked Questions) on the Microsoft AVI (Audio Video Interleave) file format. He has worked as a contractor at NASA Ames Research Center involved in the research and development of image and video processing algorithms and technology. He has published articles on the origin and evolution of life, the exploration of Mars (anticipating the discovery of methane on Mars), and cheap access to space. He has a Ph.D. in physics from the University of Illinois at Urbana-Champaign and a B.S. in physics from the California Institute of Technology (Caltech). He can be reached at* jmcgowan11@ear-think.net