# MODULE1 · INTRODUCTION TO TIME SERIES



## WORKING WITH TIME SERIES



## FORECASTING IN CONTEXT



## TIME SERIES AND FORECASTING LAB

## MODULE2 · INTRODUCTION TO SPATIAL DATA

- INTRODUCTION
- GETTING STARTED WITH SPATIAL DATA
- DEMO: SPATIAL DATA FRAMES IN R
- DEMO: PLOTTING SPATIAL DATA
- DEMO: EXPLORING SPATIAL RELATIONSHIPS
- KERNEL DENSITY ESTIMATION (KDE)
- DEMO: KDE
- K-NEAREST NEIGHBOUR

### WORKING WITH SPATIAL DATA

- INTRODUCTION
- SPATIAL POISSON PROCESSES
- VARIOGRAMS
- DEMO: VARIOGRAMS
- KRIGING (PART 1, 2, 3)
- DEMO: KRIGING

### SPATIAL DATA IN CONTEXT

- INTRODUCTION
- A SPATIAL DATA SCENARIO
- DEMO: USING R IN SQL SERVER
- DEMO: WORKING WITH SPATIAL DATA IN SQL SERVER
- DEMO: VISUALIZING SPATIAL DATA IN POWER BI

### SPATIAL DATA ANALYSIS LAB

## MODULE3 · INTRODUCTION TO TEXT ANALYTICS

- INTRODUCTION
- WORD FREQUENCY
- DEMO: TEXT NORMALIZATION IN R
- DEMO: TEXT NORMALIZATION IN PYTHON
- DEMO: REMOVING STOPWORDS IN R
- DEMO: REMOVING STOPWORDS IN PYTHON
- STEMMING
- DEMO: STEMMING IN R
- DEMO: STEMMING IN PYTHON
- THE MOST POPULAR WORDS

### WORKING WITH TEXT

- INTRODUCTION
- CALCULATING WORD IMPORTANCE
- INTRODUCTION TO NATURAL LANGUAGE PROCESSING
- DEMO: SENTIMENT ANALYSIS WITH R
- DEMO: SENTIMENT ANALYSIS WITH PYTHON

### TEXT ANALYTICS IN CONTEXT

- INTRODUCTION
- A TEXT ANALYSIS SCENARIO
- DEMO: STREAMING SENTIMENT ANALYSIS
- DEMO: REALTIME SENTIMENT REPORTING

### TEXT ANALYTICS LAB

## MODULE4 · INTRODUCTION TO IMAGE ANALYSIS

- INTRODUCTION
- DEMO: IMAGE BASICS
- DEMO: IMAGE PROPERTIES
- DEMO: IMAGE MANIPULATION
- BLURRING AND DENOISING
- DENOISING WITH A MEDIAN FILTER
- DEMO: USING FILTERS

### WORKING WITH IMAGES

- INTRODUCTION
- EDGE DETECTION
- DEMO: EDGE DETECTION
- DEMO: SEGMENTATION
- CORNER DETECTION
- DEMO: CORNER DETECTION

### INTRODUCTION TO MATHEMATICAL MORPHOLOGY

- DEMO: EROSION
- DEMO: DILATION
- OPENING AND CLOSING
- DEMO: OPENING
- DEMO: CLOSING

### IMAGE ANALYSIS IN CONTEXT

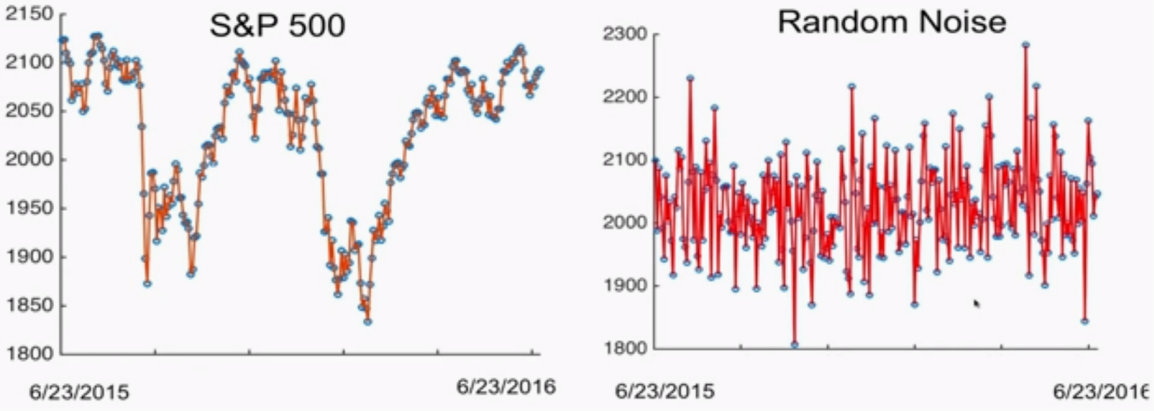
- INTRODUCTION
- AN IMAGE ANALYSIS SCENARIO
- DEMO: COGNITIVE SERVICES APIS
- DEMO: FACE DETECTION WITH PYTHON
- DEMO: FACE COMPARISON WITH C#

### IMAGE ANALYSIS LAB

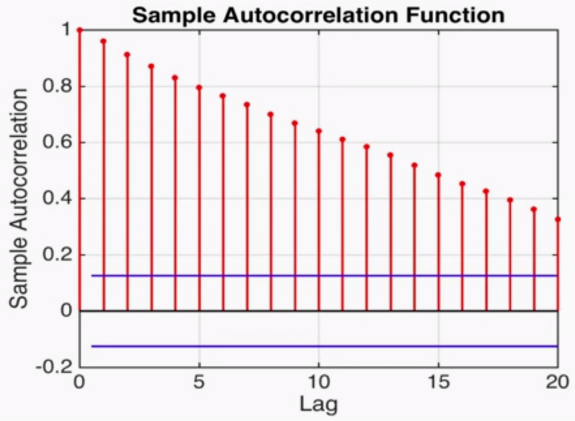
# module1 · time series and forecasting

## introduction to 〰 time series

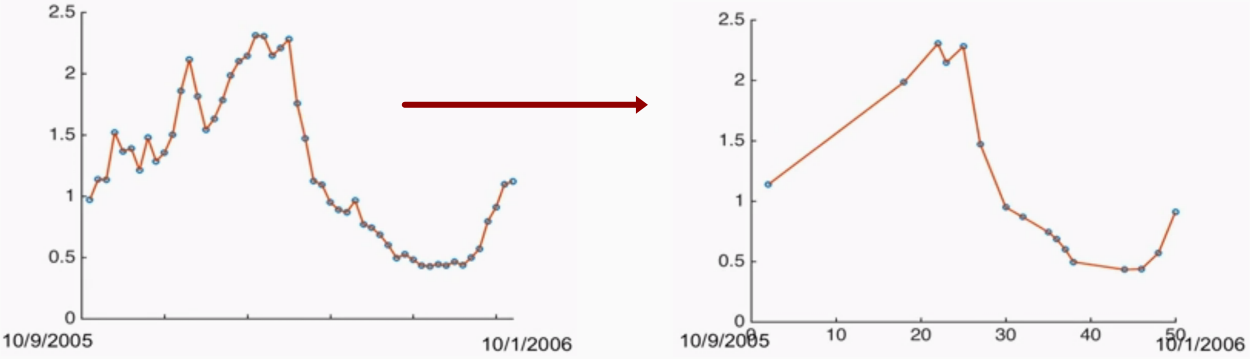
times series data does not have the same characteristics of random independent numbers



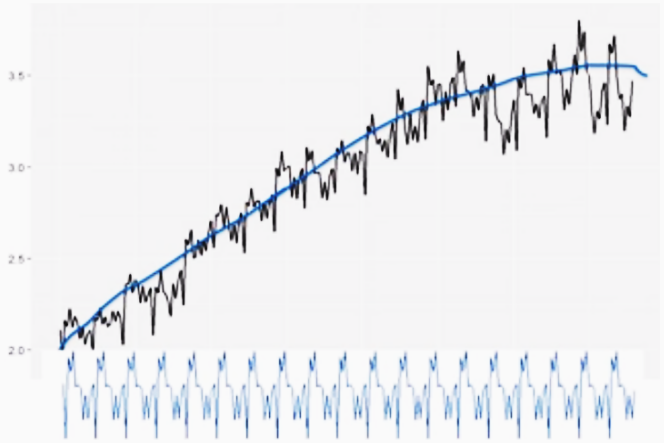

Time series data has periodicity qualities that are capable of being predicted

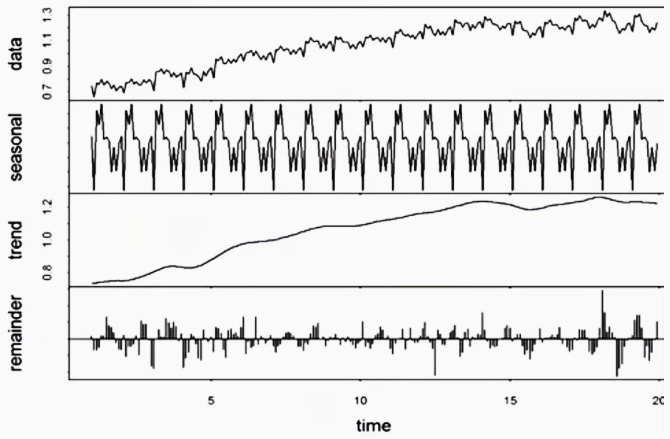



With time series data, irregular reporting times can affect the quality or representation of data




A trendline can be decomposed into various components. For example, the following data consists of a periodic signal and a stationary signal. Periodic referring to a repetitive pattern and stationary referring to data whose mean and standard deviation remain constant over time






Performing trend decomposition automatically is achieved through lowess/loess regression

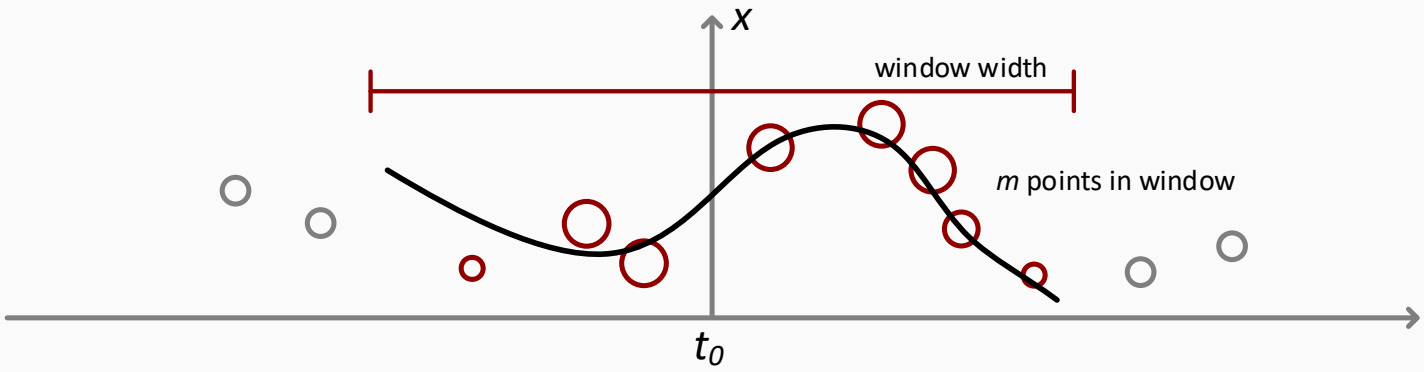
Lowess – locally weighted scatterplot smoothing

Loess – local regression

Each essentially fits local polynomial models and merges accordingly

Focusing on Loess regression:

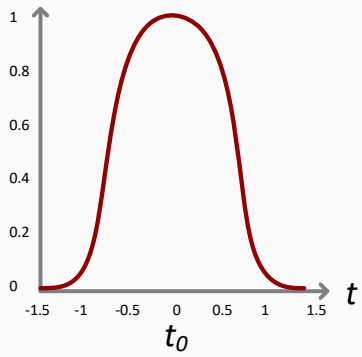
1. the window width defined as $m$ uses $m$ nearest neighbors in the window for local regression
2. a weighting function applies weights to points nearest to the center
3. the data is trended through regression using polynomial terms and the assigned weights
4. repeat the above for each value of $t$




the weighting function normalizes the data by scaling the data to 1 through the tri-weight function:

$$W(u) = \begin{cases} (1-|u|^3)^3 & \text{if } |u| \le 1 \\ 0 & \text{if } |u| > 1 \end{cases}$$

$$(1 - \text{distance}(t, t_0)^3)^3$$




the scaled distance to neighbor $i$ is $u_i = \frac{t_i - t_0}{\text{width}} = \frac{\text{distance from the center}}{\text{scaling factor}}$

weight for neighbor $i$ is: $w_i = W(u_i)$

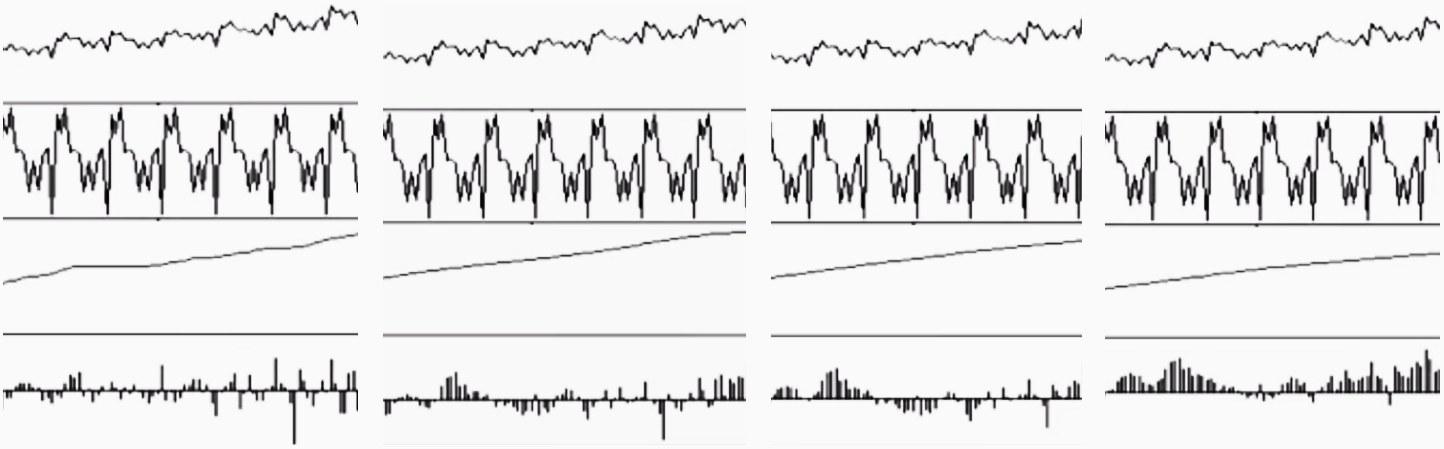
the data is fitted through taylor's theorem from calculus:

$$f(t) \approx \beta_0 + \beta_1(t_i - t_0) + \frac{1}{2}\beta_2(t_i - t_0)^2 + \cdots$$

$$\hat{\beta} = \arg\min_{\beta_0,\beta_1,\beta_2} \sum_{i=1}^{n} w_i(x_i - \hat{x}_i)^2 \qquad \text{where} \qquad \hat{x}_i = \hat{\beta}_0 + \hat{\beta}_1(t_i - t_0) + \frac{1}{2}\hat{\beta}_2(t_i - t_0)^2 \qquad \text{if } t_i = t_0\text{: } \hat{x}_i = \hat{\beta}_0$$

the window with is determined; larger windows will produce flatter trends and larger residuals

5% of data | 25% of data | 50% of data | 95% of data

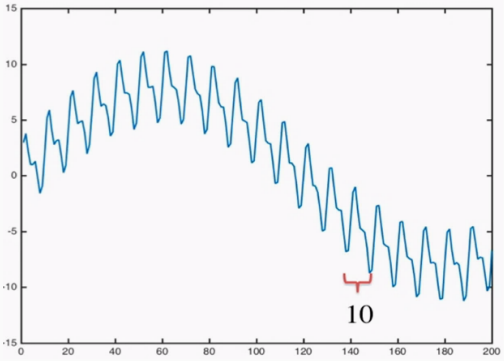


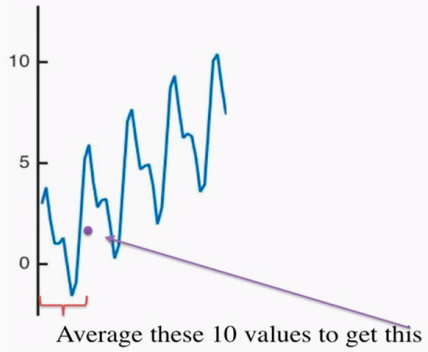
after loess has been applied to understand the trend of the given data, moving average smoothing removes the period signals to obtain a stationary signal

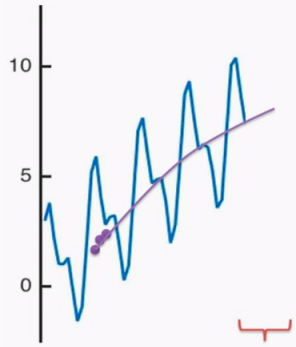
this is another way to remove an obvious periodic signal through the use of a low-pass filter (moving averages)

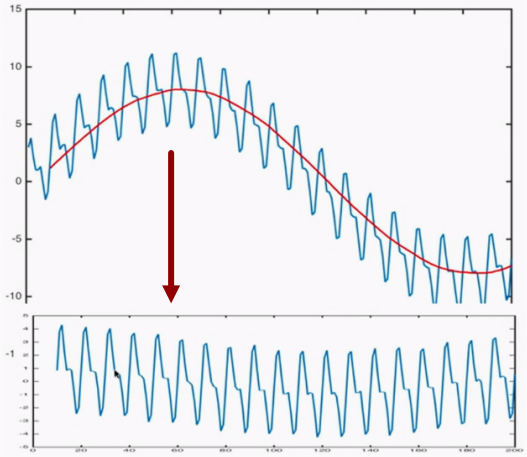
the example (left) illustrates a signal with obvious periodicity. the method is applied simply by averaging the prior 10 values in obtaining each average example (middle figure). the formula is applied forward to compute each successive point and then trended over the plots. the formula for estimate is the average of the previous $c$ points where $c$ is the periodicity of the signal: $\hat{x}_i = \frac{1}{c}(x_i + x_{i-1} + \cdots + x_{i-c+1})$

if $c$ is chosen incorrectly, the formula could fail to compute correctly

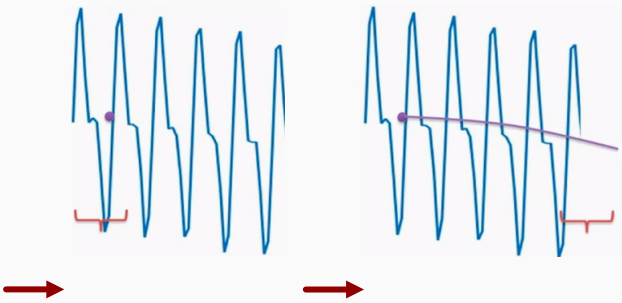


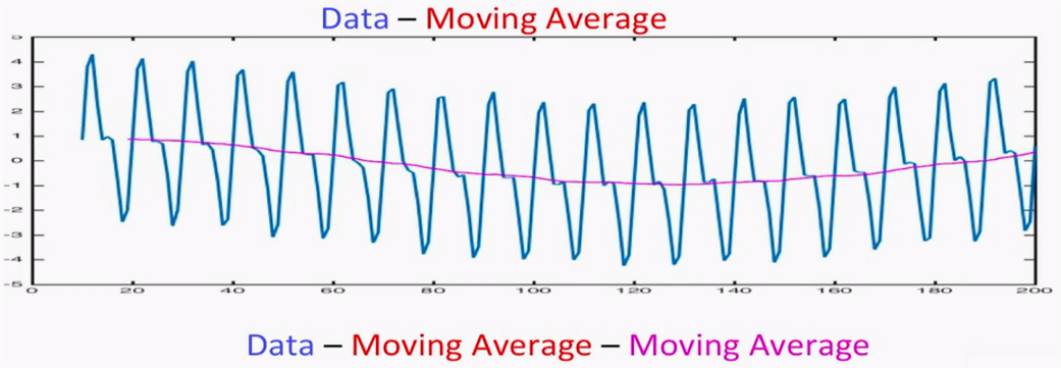



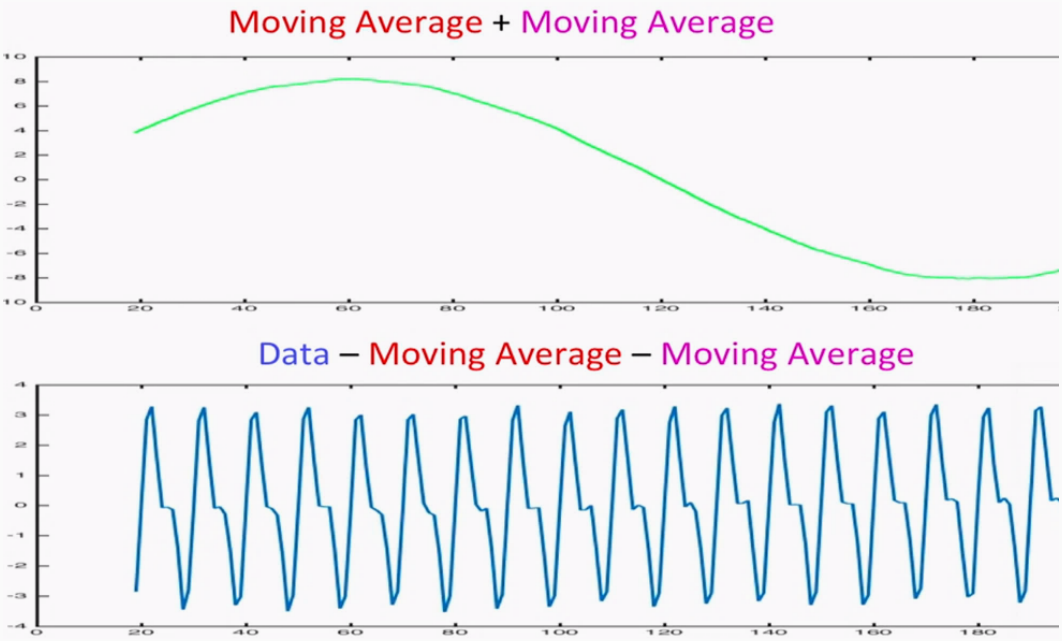


after a first attempt, the moving averaged data still appears to have a trend when the moving average is subtracted from the data (below figure on the left). the approach to refine the model reiterates through the process:

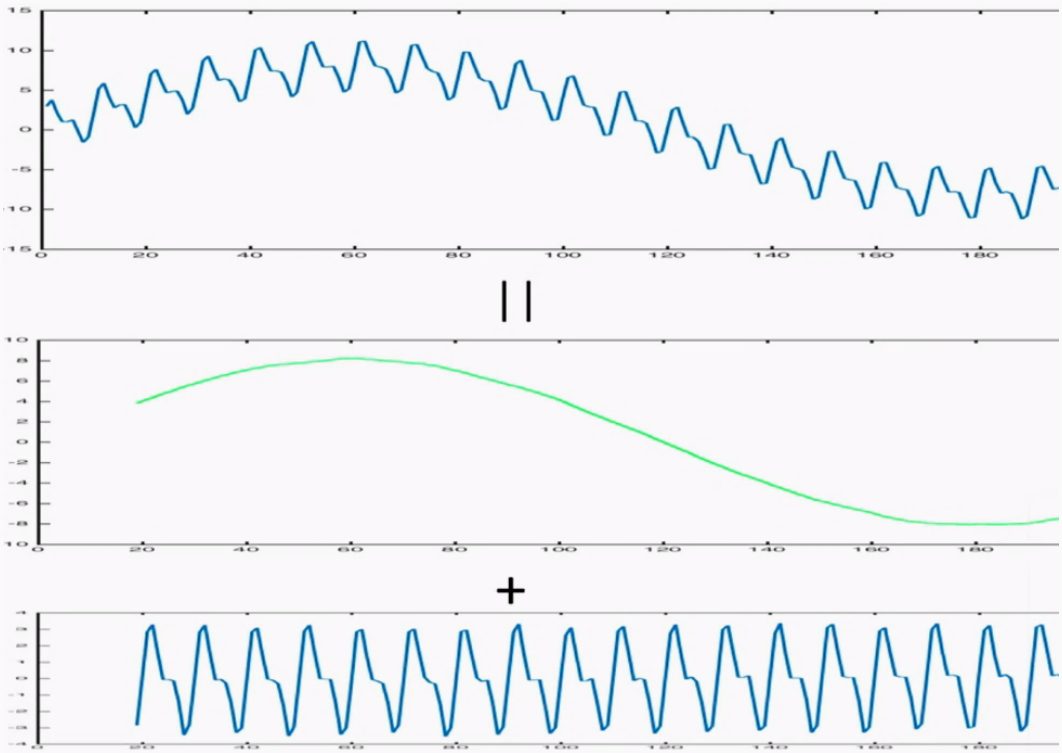





examing the reiterative trend after separating the trend and periodicity contains little to no residual:




the summary decomposition performed through moving averages is as follows:



## theorhetical recap:

loess – finds trends using polynomial modeling.

- need to determine bandwidth parameter
- coarse, looks at data over possibly many cycles

moving average smoothing – finds trends by averaging over one cycle length

- will not work well if the trend is not smooth
- the length of each periodic cycle needs to be known
- fine-grained smoothing and looks only at the previous cycle (single cycle length), nothing else

moving average smoothing does not provide long-term smoothing like loess

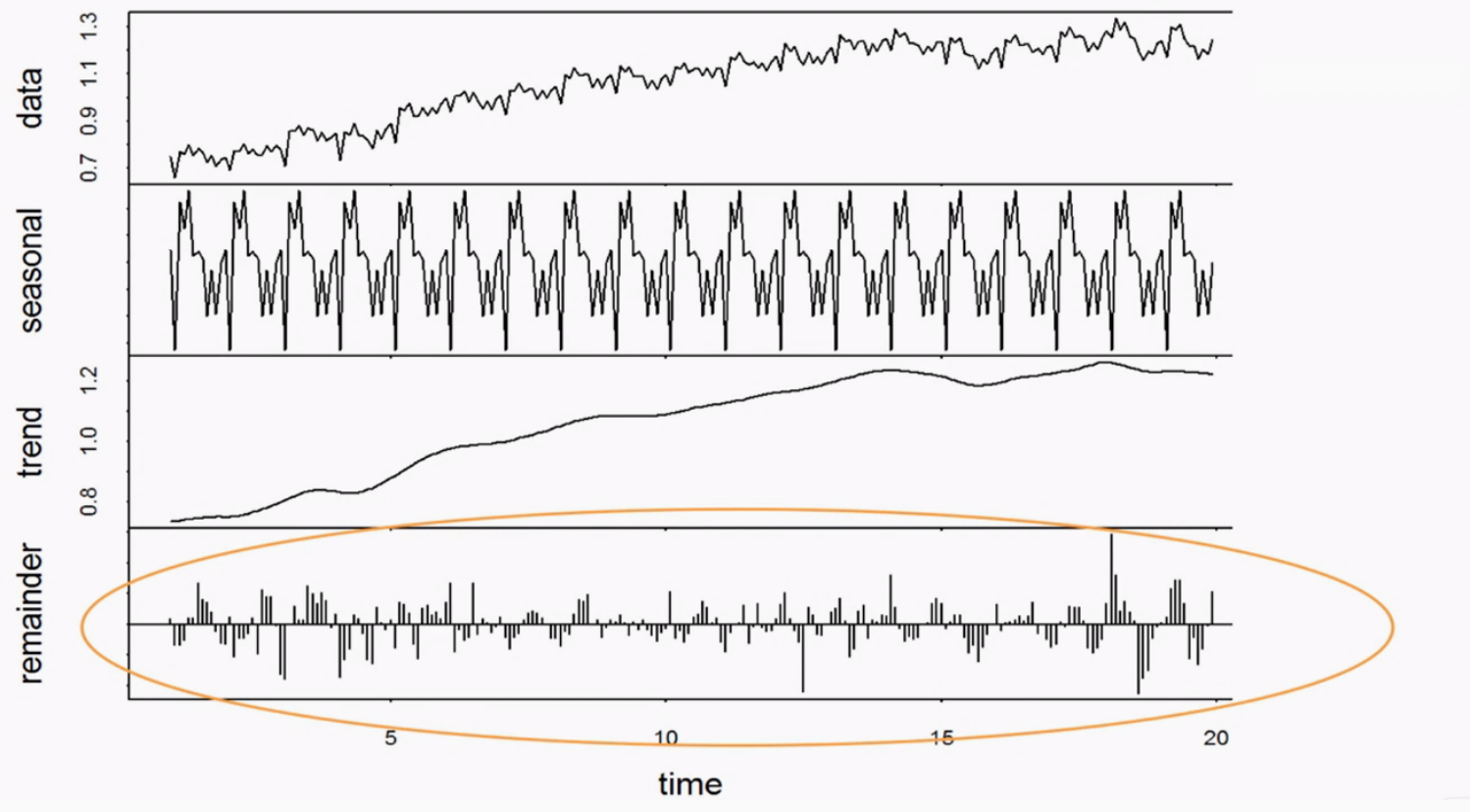
# Enhanced Seasonal Decomposition of Time Series by Loess

## R package '**stlplus**' Date 2016-01-05

**Description**: Decompose a time series into seasonal, trend, and remainder components using an implementation of Seasonal Decomposition of Time Series by Loess (STL) that provides several enhancements over the STL method in the stats package. These enhancements include handling missing values, providing higher order (quadratic) loess smoothing with automated parameter choices, frequency component smoothing beyond the seasonal and trend components, and some basic plot methods for diagnostics.

## STL package (abbreviated)

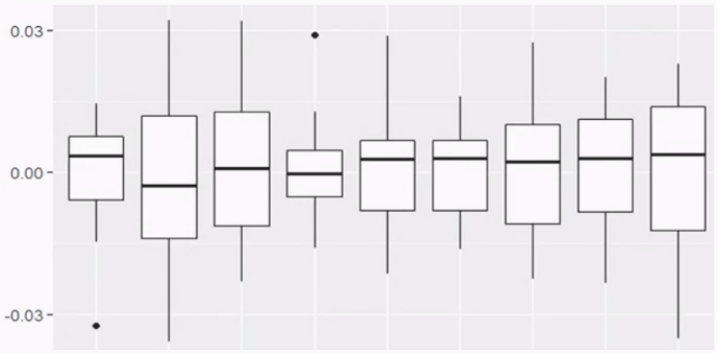
- uses loess to find a general trend $T$
- then subtacts the trend: $X\text{-}T$
- then uses moving average smoothing to find more trend $C$
- then subtracts the moving average trend: $S=X\text{-}T\text{-}C$
- the seasonal componenet, $S$ remains (seasonal because the stationary signals are smoothed and the trend is subtracted)
- then smooths the total trend $X\text{-}S$ with loess to get total trend $V$ (seasonal signal is removed from the original data, leaving the sum of the two remaining prior two trends $X\text{-}S = T+C$)
- the remainder is $R=X\text{-}S\text{-}V$ (remove the seasonal component, the trend $V$, and the remainder $R$ remains from the original signal)

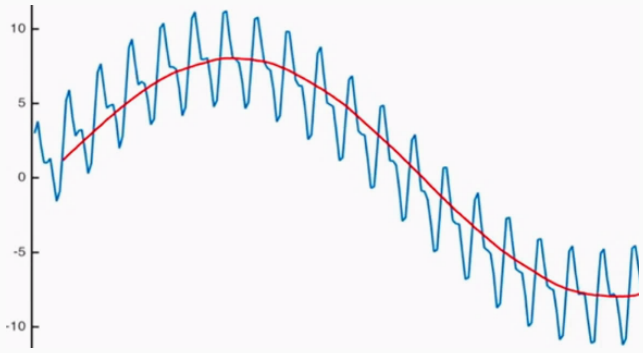


## defining stationary timeseries data

- stationary timeseries have no trend
- second-order stations conditions:
  - constant mean (does not change over time)
  - constant variance (does not change over time)
  - an autocovariance that is not time-dependent

boxplots (left) and trend models (right) are both usefull for considering stationarity:




variations in box plat variances/means and existacne of a trend in a trend model indicates stationarity

## Autocovariance and Autocorrelation (ACF) ("Auto" = "Self")

$$\text{Autocorrelation}_h(X_t) = \frac{\text{Autocovariance}_h(X_t)}{\text{Std}(X_t)\text{Std}(X_{t-h})}$$

$$\text{Autocovariance}_h(X_t) = Cov(X_t, X_{t-h})$$

Autocovariance is convariance of timeseries with lagged version of itself.

for clarity, the covariance of $(X_t, X_{t-h})$ is $X$ a time $t$ and a lagged version of itself $t-h$ with $h$ functioning as the lag. $X_t$ is a random variable true for all aspects of timeseries data. Autocorrelation is simply a normalized vertsion of the autocovariance so that the lowest value is $1$.

If the signal is weakly stationary, then the varaince (and std) are constant over time:

$$\text{Autocorrelation}_h(X_t) = \frac{\text{Autocovariance}_h(X_t)}{\text{Std}(X_t)\mathbf{Std}(\boldsymbol{X_{t-h}})} \rightarrow \text{Autocorrelation}_h(X_t) = \frac{\text{Autocovariance}_h(X_t)}{\text{Std}(X_t)\mathbf{Std}(\boldsymbol{X_t})}$$

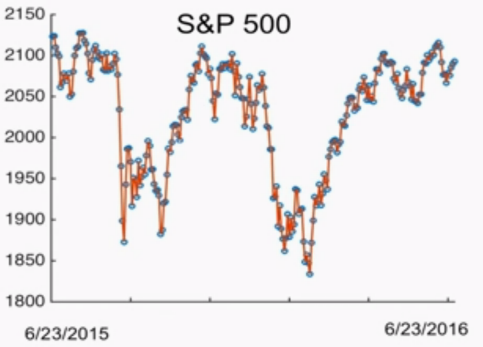
and simplifies as: $$\text{Autocorrelation}_h(X_t) = \frac{\text{Autocovariance}_h(X_t)}{\text{Var}(X_t)}$$

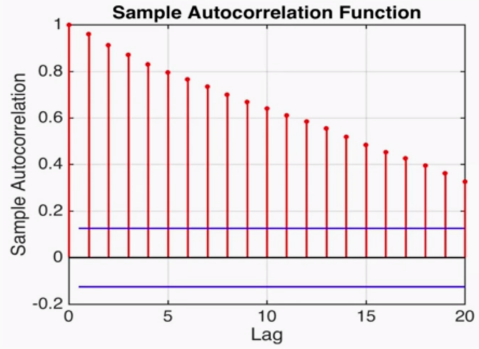
furthermore, if the signal is weakly stationary, autocorrelation is constant over time. The latter property allows a sample to be taken from the data due to its lack of time dependency; therefore:

$$\text{Autocorrelation}_h(X_t) = \frac{\text{Autocovariance}_h(X_t)}{\text{Var}(X_t)}$$

$$\text{SampleAutocorr}(X) = \frac{\text{SampleCov}([x_h, \dots, x_t], [x_1, \dots, x_{t-h}])}{\text{SampleVar}([x_1, \dots, x_t])}$$

the above illustrates the use of Sample Covariance and Sample Variance in place of the Autocovariance and variance as defined formally. To compute the Sample Convariance, it is noted in the expression that the timeseries and a lagged version of itself is necessary to compute Sample Autocorrelation in the resulting depictions illustrated below:



# working with 📈 time series

## Moving Average Models: MA(q)

note: Moving Average Models are **not** the same as the Moving Average Smoothing discussed earlier

Using the Price of Microsoft Stock as a working example:

Day1: Microsoft makes an announcement that affects the stock by $\varepsilon$ on that day

Day2: Fulle impact of the announcement affects the stock by $\theta_1\varepsilon$ on that day

Day3: lingering effects of the announcements affects the stock by $\theta_2\varepsilon$ on that day with no further effects

Assuming that Microsoft makes an announcement everyday, what is the effect on stock at day $t$?

The answer is comprised of the summartion of the current market effect, plus the prior days:

$X_t = \mathcal{E}_t + \theta_1\mathcal{E}_{t-1} + \theta_2\mathcal{E}_{t-2}$ ← exactly a moving average model of order 2

Generally, the value at time $t$ for a moving average model derives from the mean $\mu +$ noise in the form of today's announcement $\mathcal{E}_t$ and yesterday's announcement $\mathcal{E}_{t-1}$, etc:

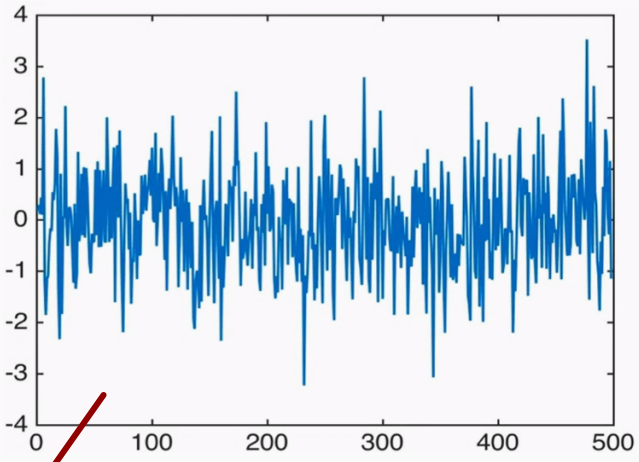
$X_t = \mu + \mathcal{E}_t + \theta_1\mathcal{E}_{t-1} + \cdots + \theta_q\mathcal{E}_{t-q}$ ← weighted average of current/previous noise terms

Moving average models are always stationary (the above is "order $q$)

The models look up to $q$ days ago and does not consider noise outside of the recent ($>q$ days ago)

looking at data from a moving average model:

$X_t = \cancel{\mu} + \mathcal{E}_t$



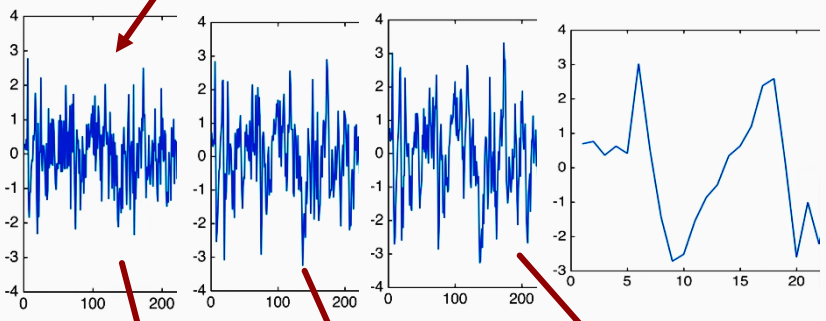
The example is a basic model assuming $\mu = 0$

(assuming $\mu = 0$ in examples going foreward)

If $\mu \neq 0$ a trend results and will subsequently be subtracted out as illustrated in decomposition

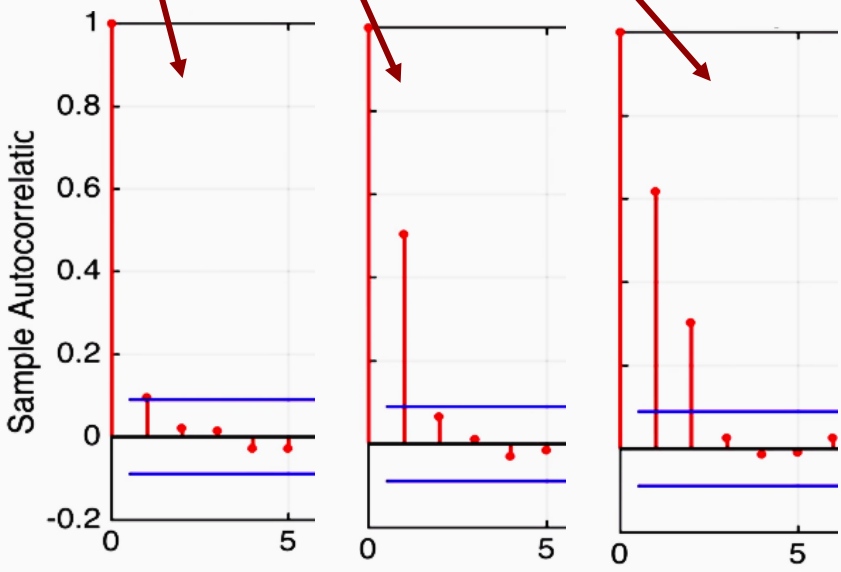
When the model is increased to the order of 1 and succinctly the order of 2, the noice remains indecipherable between the models as seen below:

Each figure to the right illustrates $X_t$ as a function of just $\mathcal{E}_t$ (1st), the order of 1 (2nd), and the order of 2 (3rd); each being generally indistinguishable from the other.



Zooming into the data frame clarifies the noise but continues to lack analytical value. Such is solved through the use of an Autocorrelation plot:

The illustratons depict the Autocorrelation function derived earlier. For the noise series $X_t = \mathcal{E}_t$, the Autocorrelation function is expectedly $0$ (independent noise is not correlated with the past). When additional orders are included in the model, non-zero terms are represented appropriately.




Given a set of timeseries data, plotting a sample autocorrelation is an appropriate way to determine if the data can be modeled as opposed to visually determining the latter.

If data arises from an MA model of order $q$ (MA($q$)), the autocorrelation function will sharply drop past $q$; making an MA($g$) model a valuble way to check if timeseries data can/should be modeled well.

Subsequent to identifying data to be modeled using a Moving Average Model of order $q$, the parameters $\theta_1, \ldots \theta_q$ of the model must be determined. This is achieved by fitting the coeficients to the data using standard least squares regression.

$$X_t = \mu + \mathcal{E}_t + \theta_1 \mathcal{E}_{t-1} + \cdots + \theta_q \mathcal{E}_{t-q}$$

# working with time series

## Autoregressive (AR) Models

$$X_t = \mathcal{E}_t + \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p}$$

The above generally states that today's value is slightly different froma combination of the last few day's values.

Autocorrelated Models are stationary, but the same cannot be said for Autoregressive Models.

The stationary nature of Autoregressive Models (AR) is dependent on the parameters:

Example Simulation:

Assuming 50,000 noise values generated from N(0,1) (normal distribtuion with mean $\mu$ of 0 and variance $\sigma^2$ of 1) to be $\mathcal{E}_1, \ldots, \mathcal{E}_{50,000}$

$X_1 = \mathcal{E}_1$

For $t = 2$ to 50,000:

$X_1 = \mathcal{E}_1 + \mathbf{0.5} X_{t-1}$

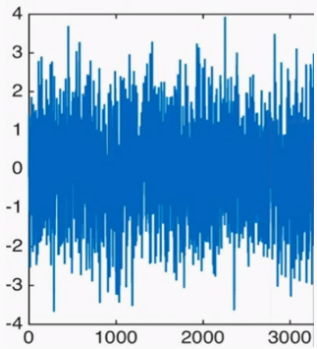
→ appears stationary

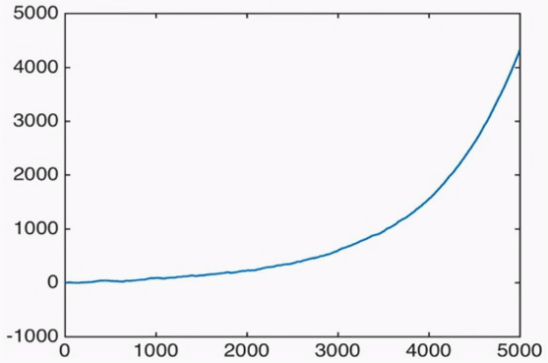
$X_1 = \mathcal{E}_1 + \mathbf{1.001} X_{t-1}$

→ appears exponential (positive)

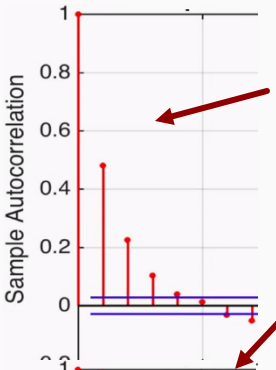
$X_1 = \mathcal{E}_1 + \mathbf{0.999} X_{t-1}$

→ appears exponential (negative)



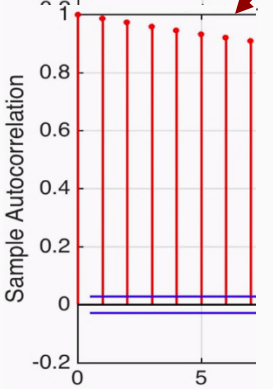




Determining the stationary nature of an Autoregressive Model by examining Autocorrelated Models:




AR(1)

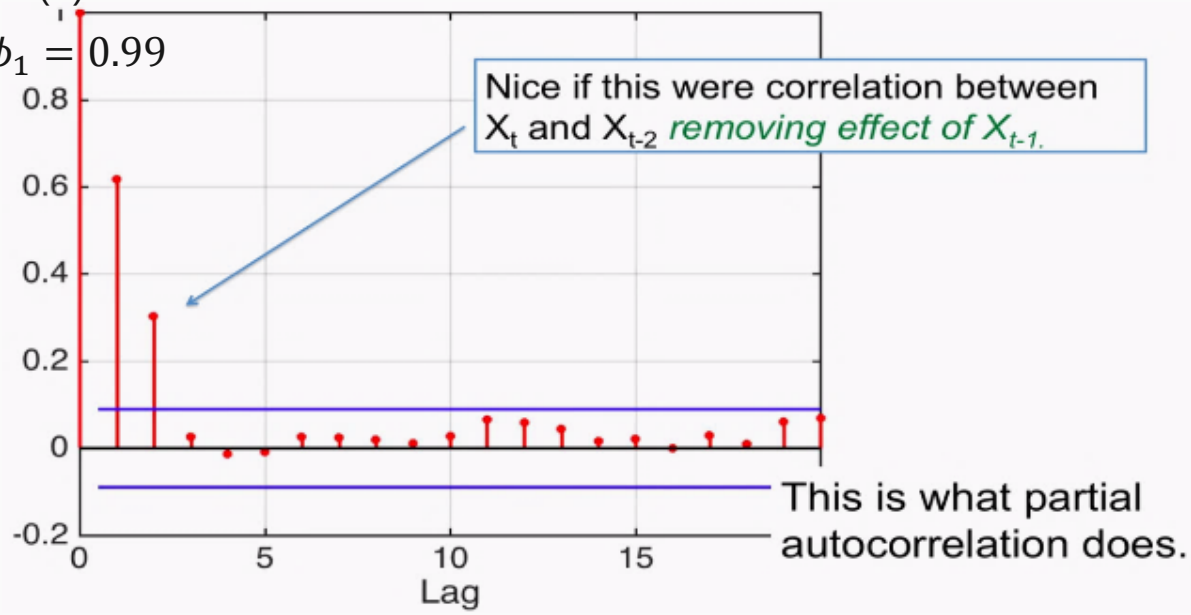
$\phi_1 = 0.5$




AR(1)

$\phi_1 = 0.99$

Both models exhibit exponential decay, with the second model occuring at a slower rate; meaning that the Autocorrelation Model **cannot be used** to determine the order of an Autoregressive Model:



Because Autocorrelation experiences exponential decay, the order of an Autoregressive Model cannot be determined when evaluating the stationary nature of the data. Instead, the Partial Autocorrelation function is examined for stationary inference of the data

## Partial Autocorrelation

The Partial Autocorrelation of $X_1$ and $X_{t-u}$ is the correlation not accounted for by lags $1, \dots, u-1$:

Written in terms of lag *(u)*: $\text{PartialAutoCorr}(u) = \text{Cor}\left(X_t, X_{t-u} \mid X_{t-1}, \dots, X_{t-(u+1)}\right)$

The PAC is the correlation between $X_t$ and $X_{t-u}$ given everything that happened in between the two.

Using the Sleep Model as a working example (the effect of sleep 2 nights ago on tonight's sleep):

In terms of order 2: $\text{PartialAutoCorr}(2) = \text{Cor}(X_t, X_{t-2} \mid X_{t-1})$

→ The correlation between tonight's sleep and two nights ago that is not accounted for last night

To compute the Partial Autocorrelation: Predict correction to $X_t$ from $X_{t-1}$: $\hat{X}_t = \beta_1 X_{t-1}$

$\beta_1$ is the correlation between $X_t$ and $X_{t-1}$.

$\hat{X}_t$ is predicted based upon the sole fact that $X_t$ is correlated with $X_{t-1}$.

All predict correction to $X_{t-2}$ from $X_{t-1}$: $\hat{X}_{t-2} = \beta_1 X_{t-1}$

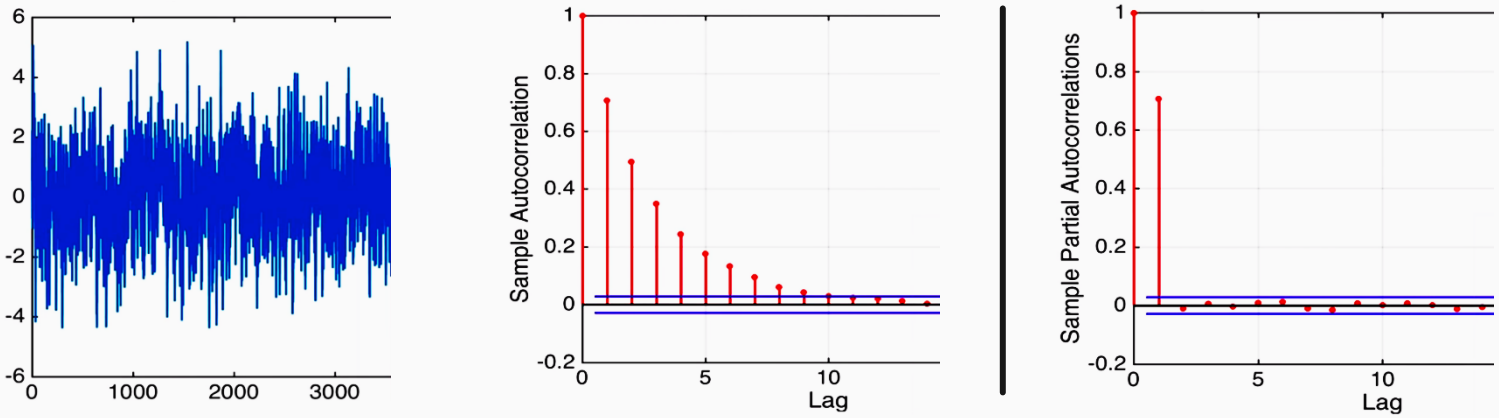
$\beta_1$ is the correlation between $X_{t-2}$ and $X_{t-1}$, which is that same as the correlation between $X_t$ and $X_{t-1}$. This is attributed to the stationary nature of the series (mean and variance constant)

In terms of order 2: $\text{PartialAutoCorr}(2) = \text{Cor}\left(X_t - \hat{X}_t, X_{t-2} - \hat{X}_{t-2}\right)$

the above formulates the question that arises: Is it possible to use what cannot be explained about $X_{t-2}$ from $X_{t-1}$ to predict what cannot be explained about $X_t$ from $X_{t-1}$?

The Partial Autocorrelation computes the correlation of the prediction error at time $t$ and the prediction error at time $t-2$.

For example: assuming the model $X_1 = \mathcal{E}_1 + \mathbf{0.7} X_{t-1}$ and stationary timeseries data:




the Sample Autocorrelation Model is not useful as expected, other than proving that the model is not a true moving average model (or else the Sample Autocorrelation plot would have a cutoff). The exponential decay additionally hints the data is an Autoregressive Model (with order still unknown). In contrast, the Sample Partial Autocorrelation Model depicts an order of 1 (AR(1)) due to a single non-zero term in the Partial Autocorrelation Function.

in general terms $(u)$:

$$\text{PartialAutoCorr}(u) = \text{Cor}\left(X_t - \hat{X}_t, X_{t-u} - \hat{X}_{t-u}\right)$$

The Partial Autocorrelation for lag $u$ is the correlation between the prediction error at time $t$ and the prediction error at time t-u; where allowed to explain $X_t$ and $X_{t-u}$ using **everything** in between:

$$\hat{X}_t = \beta_1 X_{t-1} + \beta_2 X_{t-2} + \cdots + \beta_{u-1} X_{t-(u-1)}$$
$$\hat{X}_{t-u} = \beta_1 X_{t-(u+1)} + \beta_2 X_{t-(u+2)} + \cdots + \beta_{u-1} X_{t-1)}$$

The least squares method will be used to determine the value of the $\beta$'s

Example: use a linear model to predict from the lags:

$$\hat{X}_t = \beta_1 X_{t-1} + \beta_2 X_{t-2} + \cdots + \beta_{u-1} X_{t-(u-1)}$$

use least squares to get the coefficients in a minimized error fashion: $\min_{\beta_0,\beta_1,\ldots,\beta_{u-1}} \left(\hat{X}_t - X_t\right)^2$

then use a linear model to predict $X_{t-u}$ from the lags:

$$\hat{X}_{t-u} = \beta_1 X_{t-(u+1)} + \beta_2 X_{t-(u+2)} + \cdots + \beta_{u-1} X_{t-1)}$$

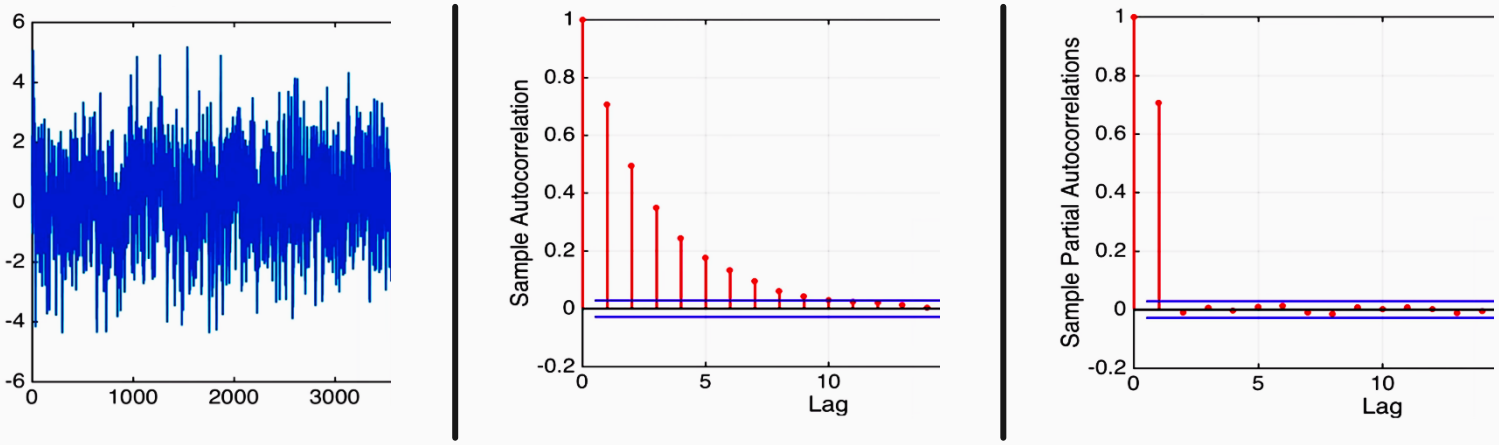
the coeficients are the same in both models because least squares is about correlations (which are not directed forward or backwards in time; no directionality):

$$\text{PartialAutoCorr}(u) = \text{Cor}\left(X_t - \hat{X}_t, X_{t-u} - \hat{X}_{t-u}\right)$$

The Partial Autocorrelation describes the unexplained part of $X_t$ from the unexplained part of $X_{t-u}$; where explinations are derived from all of the lags in between.

## Theorhetical Summary of Autoregressive Models:

- AR models are not always stationary; this depends on the parameters
- The autocorrelation functions for stationart AR(p) models experiences exponential decay and thus, order cannot be determined
- The partial autocorrelation for stationary (AR(p) models experiences a strict cutoff at (p) and thus, the order can effectively be determined



# working with 📈 time series

## Autoregressive Moving Average (ARMA) Models

A combination of Autoregressive Models and Moving Average Models → ARMA(p,q)

$$X_t = \mathcal{E}_t + \phi_1 X_{t-1} + \cdots + \phi_p X_{t-p} \qquad \text{AR(p)}$$

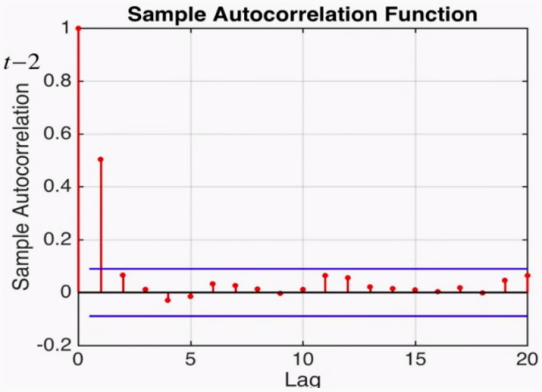
$$X_t = \mu + \mathcal{E}_t + \theta_1 \mathcal{E}_{t-1} + \cdots + \theta_q \mathcal{E}_{t-q} \qquad \text{MA(q)}$$

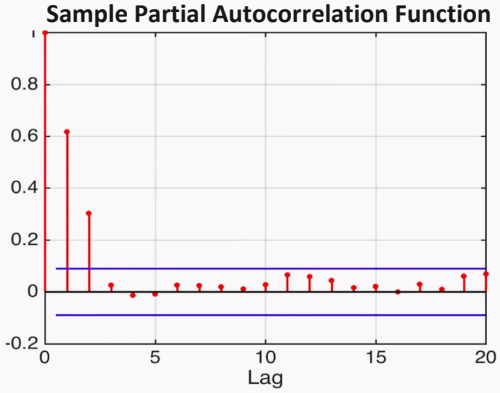
for example: the price of the product today depends on the price yesterday, the day before, etc. but also depends on the announcements of the company's welfare today, the day before, etc.

if $p$ and $q$ are known (order of both parts), regression can be performed to obtain all parameters

*Remember: pure MA models experience a sharp cutoff for the Autocorrelation Function
*Remember: pure AR models experience a sharp cutoff for the Partial Autocorrelation Function

Example: $X_t = \mathcal{E}_t + 0.6\mathcal{E}_{t-1} + 0.4\mathcal{E}_{t-2}$

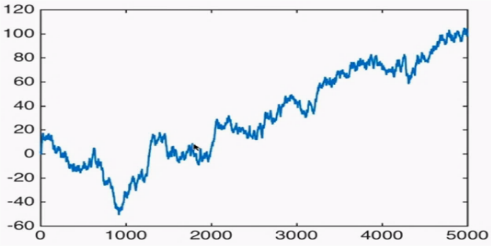



*However: if both the Autocorrelation and Partial Autocorrelation functions experience slow exponential decay, both the AR and MA terms are needed

## Differencing

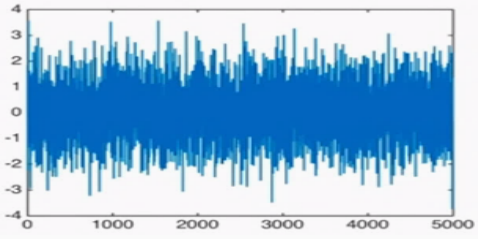
illustrated as 'a random walk' $X_{t-1} + \mathcal{E}_t$, with yesterday's position's represented as $X_{t-1}$ and the random step for today's position (noise) denoted as $\mathcal{E}_t$.



a 'random' walk illustrated to the right is **not stationary**; indicating prior models are not applicable alone. computation of the 'difference series': $X_t - X_{t-1}$

(e.g. the difference between today and yesterday)

$$X_t - X_{t-1} = X_{t-1} + \mathcal{E}_t - X_{t-1} = \mathcal{E}_t$$



application of differencing creates a series that **is stationary.** Therefore, nonstationary data might benefit from computing the difference series prior to modeling (bottom graphic)

an integrative moving average example:

$$\underbrace{X_{t-1} + \varepsilon_t}_{\text{'random walk' terms}} + \underbrace{\theta_1 \varepsilon_{t-1}}_{\text{MA(1) term}}$$

→ also **not stationary**

the above appears as both a Moving Average and a Random Walk; regardless it is still not stationary

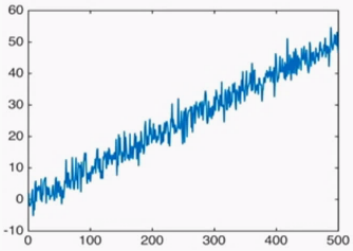
$$X_t - X_{t-1} = X_{t-1} + \varepsilon_t + \theta_1 \varepsilon_{t-1} - X_{t-1} = \varepsilon_t + \theta_1 \varepsilon_{t-1} \rightarrow \text{stationary}$$

the above applies Differencing and results in simply a Moving Average Series to the order of 1 (MA(1)); this series is in fact stationary and can modeled effectively.

## Differencing Continued

assuming a linear trend plus a stationary series:

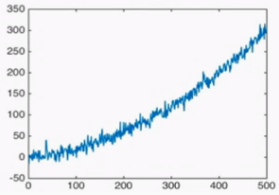
$$X_t = \beta_0 + \beta_1 t + Z_t \longrightarrow$$



$$X_t - X_{t-1} = \cancel{\beta_0} + \beta_1 t + Z_t - (\cancel{\beta_0} + \beta_1 (t-1) + Z_{t-1})$$

$$= \cancel{\beta_1 t} - \beta_1 \cancel{(t-1)} + (Z_t - Z_{t-1})$$

$$= \underbrace{\beta_1}_{\text{constant}} + \underbrace{(Z_t - Z_{t-1})}_{\text{difference series of a stationary series}} \rightarrow \text{stationary}$$



the above is similar to taking a derivative (e.g. discrete derivative): $\nabla X_t = \beta_1 + \nabla Z_t$

assuming a quadratic trend plus a stationary series:

$$X_t = \beta_0 + \beta_1 t + \beta_1 t^2 + Z_t \longrightarrow$$

$$X_t - X_{t-1} = (\cancel{\beta_0} + \beta_1 t + \cancel{\beta_2 t^2}) - (\cancel{\beta_0} + \beta_1 \cancel{(t-1)} + \beta_2 \cancel{(t-1)^2} + \nabla Z_t)$$

$$= \underbrace{(\beta_1 - \beta_2)}_{\text{constant}} + \underbrace{2\beta_2 t}_{\text{linear trend}} + \underbrace{\nabla Z_t}_{\text{difference series of a stationary series}}$$

$$\nabla X_t = \text{constant} + 2\beta_1 t + \nabla Z_t \rightarrow \text{linear trend + stationary}$$

the above is addressed by taking yet another difference:

$$\nabla^2 X_t = 2\beta_1 + \nabla^2 Z_t \rightarrow \text{stationary}$$

Differencing allows techniques for stationary timeseries' to model nonstationary series'.
Higher order trends turn into stationary models through repeated differencing.

## Autoregressive Integrated Moving Average Models (ARIMA Models)

If a Difference Series results in an ARMA model, the original series is an ARIMA model.

$$Y_t = \nabla^d X_t$$

The original series is denoted as $X_t$; the difference $\nabla^d$ is taken to get to $Y_t$ ;

*In order for $X_t$ to be an ARIMA model, $Y_t$ has to be an ARMA Model*

$$Y_t = \underbrace{\mu}_{\text{constant}} + \underbrace{\phi_1 Y_{t-1} + \cdots + \phi_p Y_{t-p}}_{\text{Autoregressive Function}} - \underbrace{\mathcal{E}_t - \theta_1 \mathcal{E}_{t-1} - \cdots - \theta_q \mathcal{E}_{t-q}}_{\text{Moving Average Function}}$$

In general, if the Difference Series is either a purely Moving Average Model or the Difference Series is a purely Autoregressive Model, then the original series is still an ARIMA Model:

$\text{ARIMA}(d, p, q) \rightarrow d =$ order of differences; $p =$ order of AR Model; $q =$ order of MA Model

## Properties of ARIMA Models

- As long as the difference series of any order is an ARMA model, the model is an AIMA model; differences can continuously be taken in testing if an ARMA model exists.
- Differences can be computed easily; take differences until the model becomes stationary
- The Autocorrelation function can be used to determine the order of a purely MA model, or the Partial Autocorrelation to determine the order of a purely AR model.

# working with time series

## Exponentially Weighted Moving Averages (EWMA)

## aka Simple Exponential Smoothing Model (SES)

EWMA is a special case of an ARIMA Model; ARIMA(0,1,1); which is essentially an ARMA Model considering $d = 0$ and 1 AR term and 1 MA term

EWMA makes predictions as follows:

$$\hat{X}_t = \alpha X_{t-1} + (1-\alpha)X_{t-1}$$

The above is a mixture of what is being predicted and the previous value

To expand for intuition:

$$\begin{aligned}\hat{X}_t &= \alpha X_{t-1} + (1-\alpha)\hat{X}_{t-1} \\ &= \alpha X_{t-1} + \hat{X}_{t-1} - \alpha\hat{X}_{t-1} \\ &= \hat{X}_{t-1} + \alpha\left(X_{t-1} - \hat{X}_{t-1}\right)\end{aligned}$$

Using $e_{t-1} = X_{t-1} - \hat{X}_{t-1}$ to define the prediction error $X$ at time $t-1$ (actual value – predicted value):

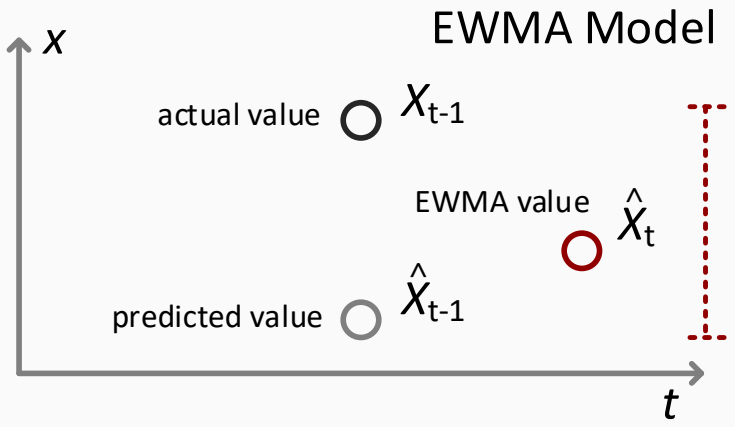
$$\hat{X}_t = \hat{X}_{t-1} - \alpha e_{t-1} \quad \rightarrow \text{the last forecast is adjusted by its error}$$

The prediction of is simply the prediction plus some adjustment times the prediction error

(The prediction of time $t$ is a mixture of what occurred yesterday and a measure of what occurred today)

Assuming the model was fitted or some other method of obtaining alpha:

the prediction $\hat{X}_{t-1}$ turned out to be much lower than its actual value $X_{t-1}$; the EWMA Model in turn takes the combination of the two through the adjusting to compute the final prediction $\hat{X}_t$ (forecast)




Returning to the EWMA Model for Intuition:

$$\begin{aligned}\hat{X}_t &= \alpha X_{t-1} \\ &\quad + (1-\alpha)\hat{X}_{t-1} \\ &= \alpha X_{t-1} + (1-\alpha)X_{t-1} - (1-\alpha)X_{t-1} + (1-\alpha)\hat{X}_{t-1} \\ &= X_{t-1} - (1-\alpha)\left(X_{t-1} - \hat{X}_{t-1}\right)\end{aligned}$$

using $e_{t-1} = X_{t-1} - \hat{X}_{t-1}$...

$$\hat{X}_t = \hat{X}_{t-1} - (1-\alpha)e_{t-1} \rightarrow \text{ARMA Model with 1 AR and 1 Ma term (ARIMA(0,1,1)}$$

# forecasting and time series lab

Time series models are used in a wide range of applications, particularly for forecasting.

Perform analyses on a time series of California dairy data. Specifically exploring the structure of the time series and forecast the monthly production of fresh milk in the state of California.

This exploration is performed in two steps:

- Explore the characteristics of the time series data.
- Decompose the time series of monthly milk production into trend, seasonal components, and remainder components.
- Apply time series models to the remainder component of the time series.
- Forecast the production of monthly milk production for a 12 month period.

The header of the data loaded from scripts of R code to:

- The data is read from a dataset in Azure Machine Learning subscription.
- A new column, of type POSIXct, is created. POSIXct is a flexible R data-time class. The strtime function formats a text string for conversion to the date-time class.
- The Month column is converted to an ordered R factor class and unnecessary columns removed

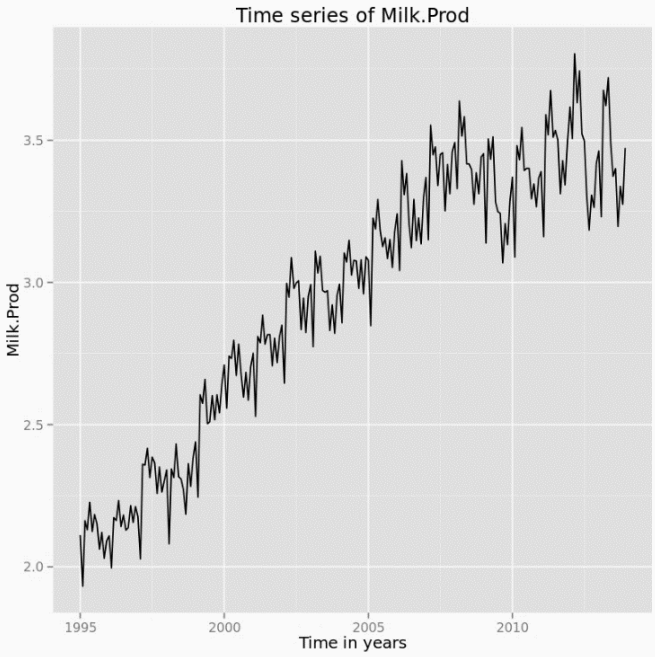
| | Year | Month | Cotagecheese.Prod | Icecream.Prod | Milk.Prod | N.CA.Fat.Price | dateTime |
|---|---|---|---|---|---|---|---|
| 1 | 1995 | Jan | 4.37 | 51.595 | 2.112 | 0.9803 | 1995-01-01 |
| 2 | 1995 | Feb | 3.695 | 56.086 | 1.932 | 0.8924 | 1995-02-01 |
| 3 | 1995 | Mar | 4.538 | 68.453 | 2.162 | 0.8924 | 1995-03-01 |
| 4 | 1995 | Apr | 4.28 | 65.722 | 2.13 | 0.8967 | 1995-04-01 |
| 5 | 1995 | May | 4.47 | 73.73 | 2.227 | 0.8967 | 1995-05-01 |
| 6 | 1995 | Jun | 4.238 | 77.994 | 2.124 | 0.916 | 1995-06-01 |

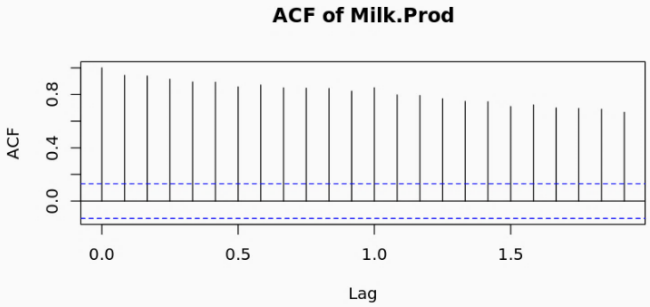
The POSIXct column is used to create the time axis on a Time Series plot of Milk Production (ggplot2)

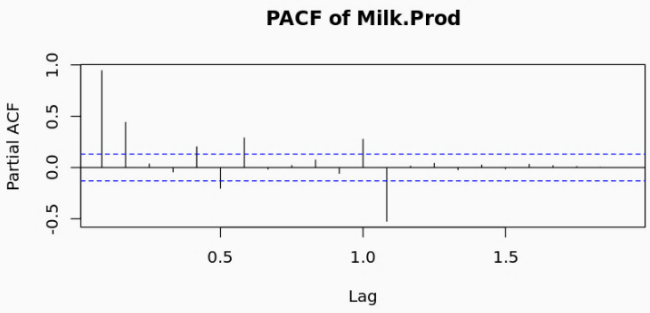
The plot shpws Milk Production increasing over the years with a decline in 2009 (the receission).

Additionally, the Time series exhibits a strong seasonal component with an annual cycle.

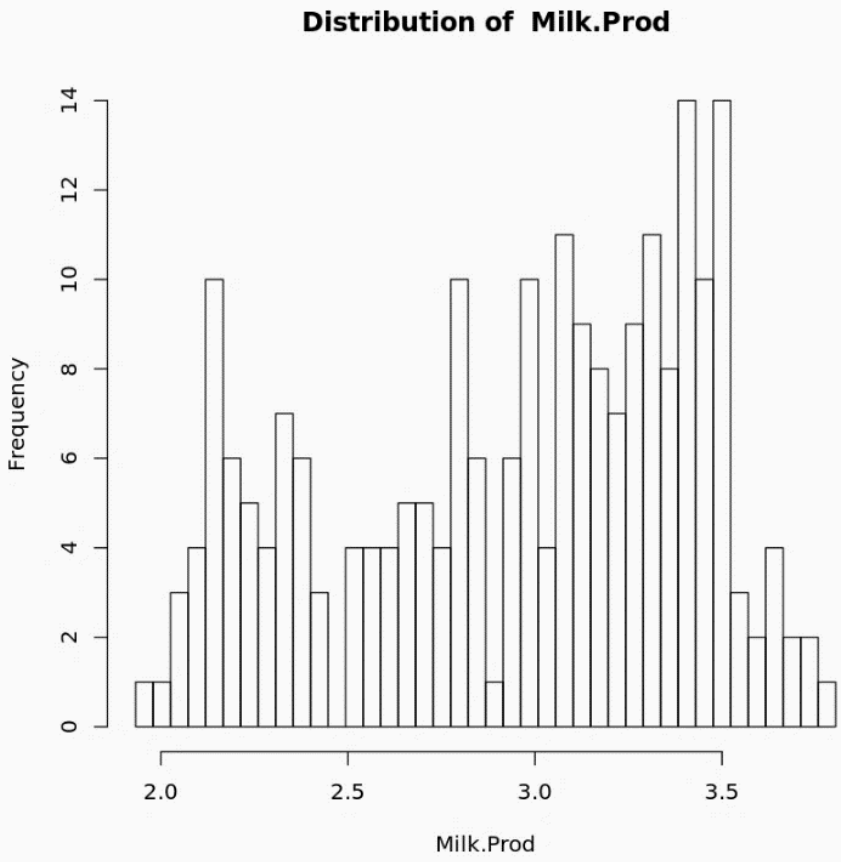
The Autocorrelation and Partial Correlation Functions are computed on the Time Series next:




The values of the ACF decays slowly between lags. This indicates considerable serial correlation between the time series values at the various lags, likely from the trend.




Autocorrelation is a fundamental property of time series.
The **Autocorrelation Function** or **ACF** provides information on the dependency of the time series values of previous values. The results of a ACF analysis is used later on to estimate the order of moving average processes. The **Partial Autocorrelation Function** or **PACF**, measures the correlation of the time series with its own lag values. Later in this lab you will use a **PACF** to estimate the order of an autoregressive process.

Plotting a histogram provides information on the distribution of values of the time series:

The histogram of the full milk production time series shows considerable dispersion. Again, such behavior is likely the result of the trend.

## Simple Moving Average Decomposition of the Time Series

Time series are typically decomposed into three components: trend, seasonal, and the remainder, or residual. Trend can be modeled by several methods; beginning with a Simple Moving Average Model using the Moving Window Method. The Moving Window Method computes the average timeseries over a specified span, or order of operator.

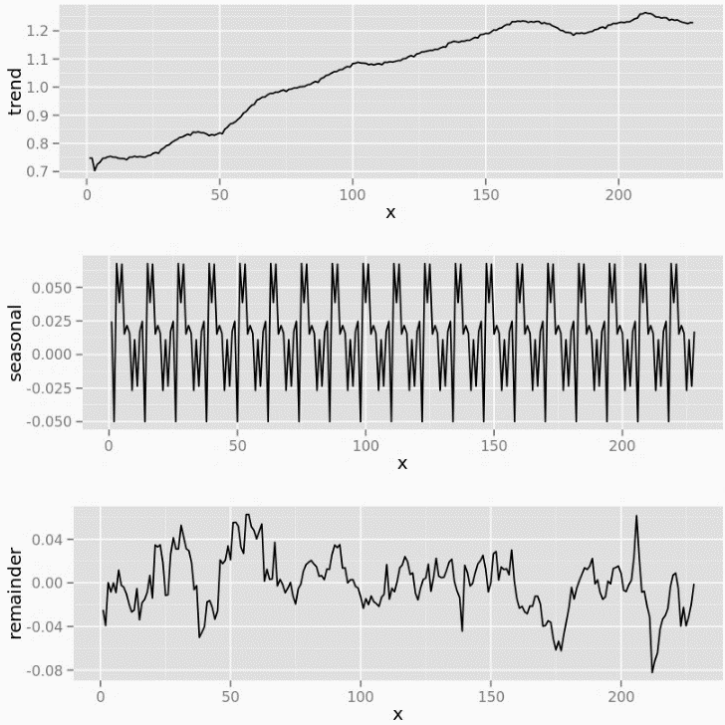
Once the trend has been removed, the seasonal component must be modeled and removed. The Seasonal Component is computed as a function of the month of the year using a linear model.

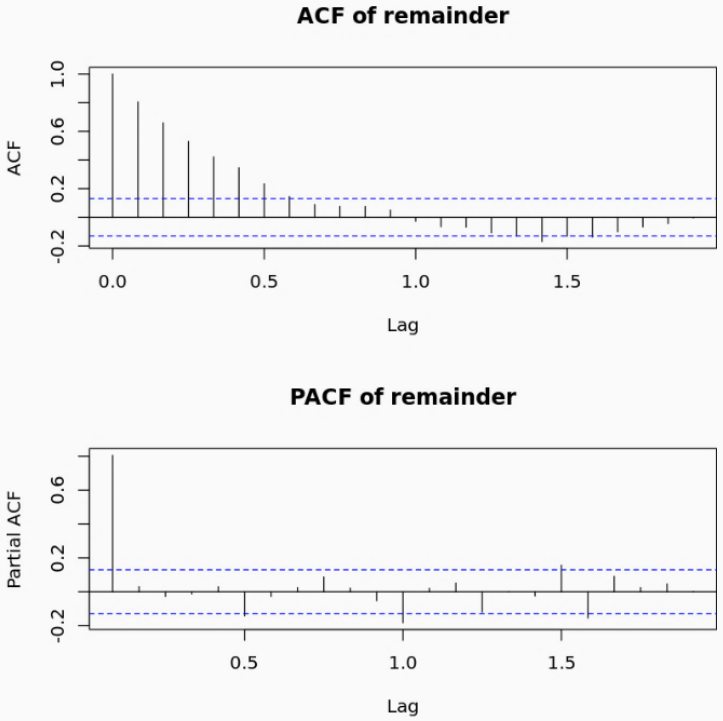
The final step is to take a Multiplicative Decomposition of the timeseries by taking a log of the values.

The resulting data frame has three components for trend, seasonal and remainder.

| | trend | seasonal | remainder |
|---|---|---|---|
| 1 | 0.7476354 | 0.02471208 | -0.02471208 |
| 2 | 0.7476354 | -0.04996016 | -0.03911947 |
| 3 | 0.7030956 | 0.06783954 | 9.862991e-05 |
| 4 | 0.7257416 | 0.03872409 | -0.008343715 |
| 5 | 0.7333367 | 0.06733682 | -1.813267e-05 |
| 6 | 0.7468004 | 0.01530435 | -0.008803681 |

The Decomposed Timeseries data is plotted using **ggplot2** package in R. The trend and seasonal components are clearly separated in the plot on the following page. The remainder plot appears random as expected. However, the remainder will need to be tested if stationary or not, the ACF of the Remainder will determine if the remainder is stationary or not.



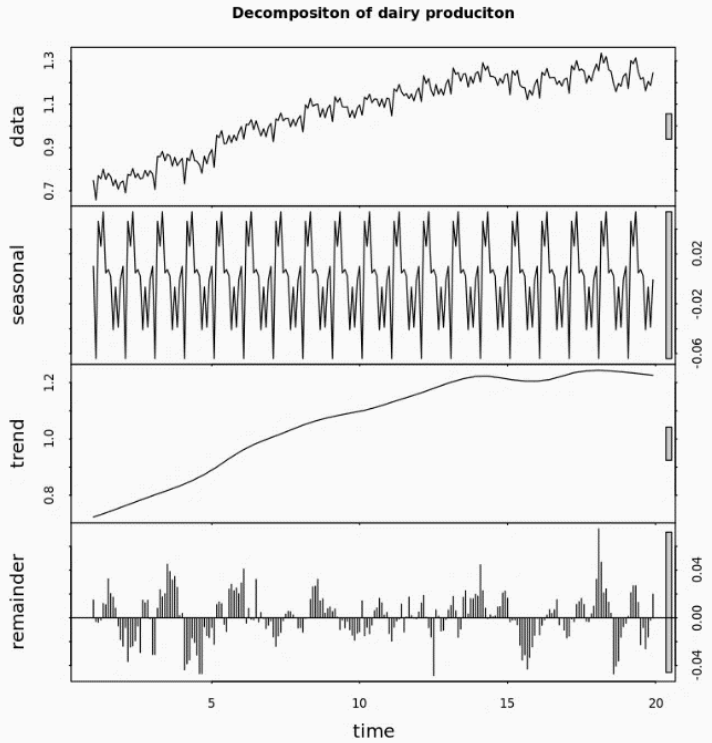


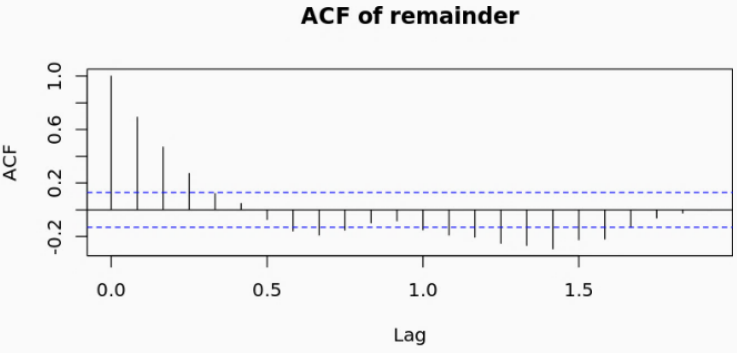


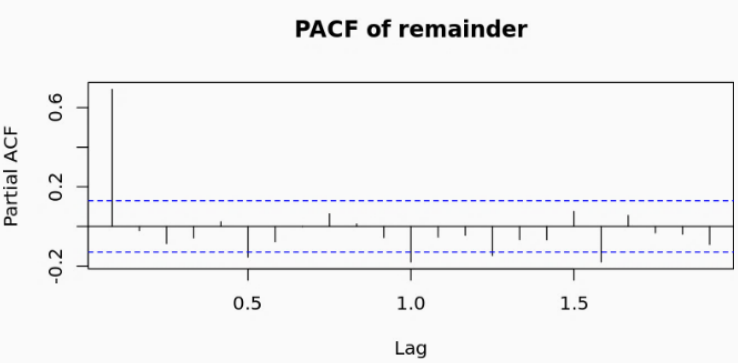
The ACF has 7 significant lag values, indicating the remainder is not, in fact, stationary.

## Exploring the Multiplicative Model with Lowess

Subsequent to applying an MA model to the data, a Lowess Model will be used to determine the trend. Lowess is a sophisticated non-linear regression. The lowess trend model is combined with a moving window seasonal component model into the R **stl** function. The **stl** function decomposes the time series and the columns of the timeseries decomposition are added to the data frame.

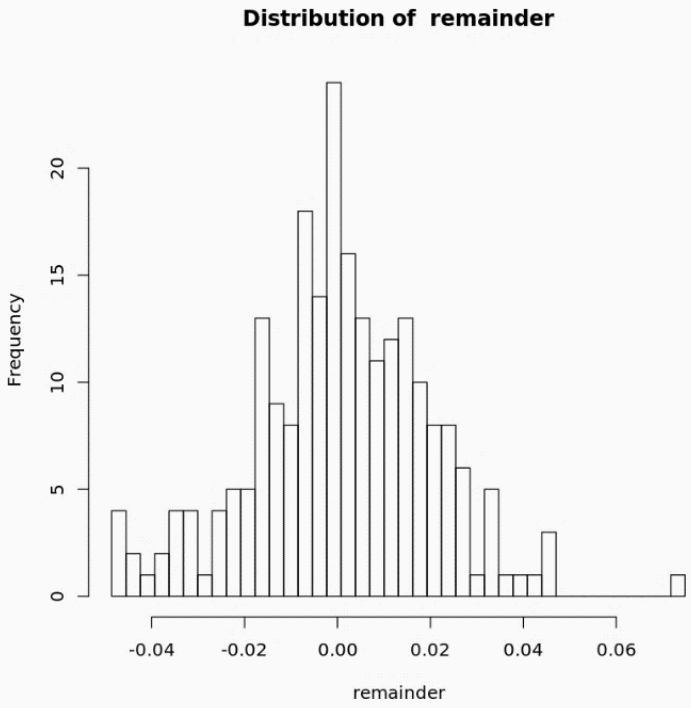


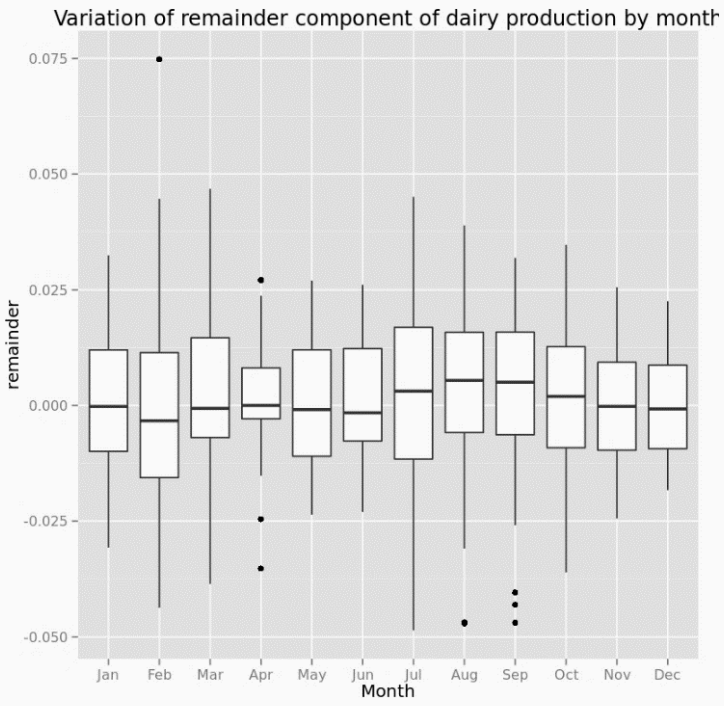




The time series charts show the original time series along with the components of the decomposition. The trend is a bit smoother than was obtained with the simple moving average decomposition. To determine if stationary, the ACF and PCF of the remainder indicate the remainder is still not stationary.

The first 4 lag values of the ACF have significant values, indicating that the remainder series **is not stationary**. Compared to the behavior of the ACF for the simple moving average decomposition, the behavior of the remainder is improved. The Histogram and Box-Plots for the Remainder (non-seasonal residual) Distribution examined next:




The distribution of the remainder values is much closer to a Normal distribution than for the original time series created earlier. This result combined with the ACF plot shows **stl** decomposition effective.

The remainder component shows only limited variation from month to month. The differences are within the interquartile range, indicating that the seasonal model is a reasonably good fit.

## Moving Average Models

Subsequent to Decomposition of the Timeseries, the process moves to constructing and testing an Autoregressive Moving Average (ARMA) Model for the Timeseries Remainder; requiring three steps:

- Create a Moving Average Model (MA)
- Create an Autoregressive Model (AR)
- Creating an Autoregressive Moving Average (ARMA) Model

Autoregressive Integrative Moving Average (ARIMA) model:

The summary statistics for the model are printed and the model object returned. By assigning values to the order of each operator, different time series models can be specified: as order of **MA** model, order of **Integrative** model, and order of **AR** model. Since the de-trended remainder is being modeled, the **include.mean** argument is set to FALSE in the **arima** function.

The ACF of the remainder from the **stl** decomposition of the milk production time series had 4 significant lag values. As an initial model, you will now create an **MA** model of order 4. The summary results are on the following page:

```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ma1     ma2     ma3     ma4
      0.7259  0.5308  0.2976  0.0193
s.e.  0.0659  0.0776  0.0748  0.0589

sigma^2 estimated as 0.0001876:  log likelihood = 654.34,  aic = -1298.67
```

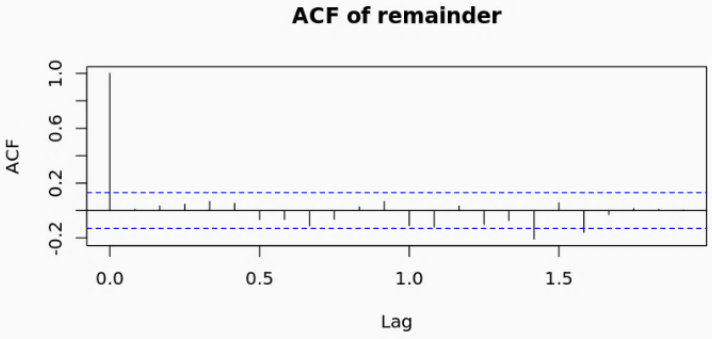
Note the SE of the **ma4** coefficient is > the value of the coefficient itself. This indicates that the value of this coefficient is **poorly determined** and should likely be set to **zero**.

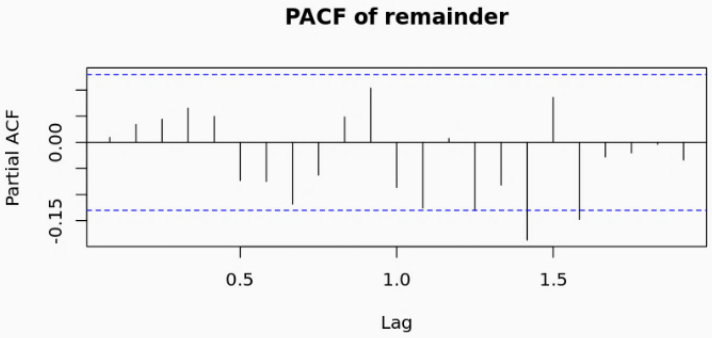
The result indicates that the order of the MA model should be reduced. Generally, the order of an MA model is reduced in unit steps until all the coefficients appear to be significant; an **MA(3)** is run next:

```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ma1     ma2     ma3
      0.7224  0.5211  0.2861
s.e.  0.0645  0.0698  0.0660

sigma^2 estimated as 0.0001877:  log likelihood = 654.28,  aic = -1300.56
```

The small standard error compared to the magnitude of the coefficients indicates that the order of the model is reasonable.

To test how well this model fits the data, and produces a stationary result, plot the ACF of the residuals of the MA(3) model:




Note that only the **0 lag** of the ACF is significant and that there are no significant lags for the PACF; This indicates that the MA(3) model is a good fit.

## Autoregressive Models (AR)

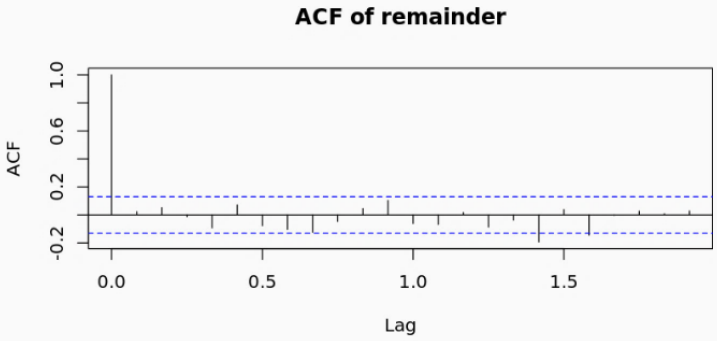
The **MA(3)** model has been shown to be effective. An Autoregressive **(AR)** Model will be tested next. The **PACF** of the reminder indicates that an **AR** model might not be the best choice. None the less, a low order **AR(2)** model might fit the data:

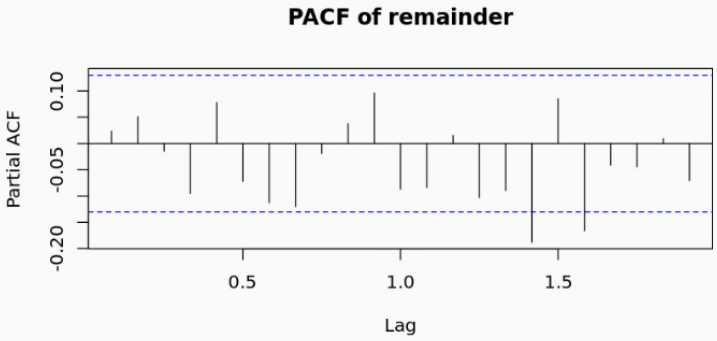
```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ar1      ar2
      0.7148  -0.0288
s.e.  0.0665   0.0665

sigma^2 estimated as 0.0001899:  log likelihood = 653.04,  aic = -1300.08
```

Note that the standard error of the second coefficient is of the same magnitude as the first coefficient. The **AR(2)** model is over parameterized; an **AR(1)** will be run next:

```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ar1
      0.6946
s.e.  0.0475

sigma^2 estimated as 0.00019:  log likelihood = 652.95,  aic = -1301.9
```

The standard error of the **AR(1)** model is an order of magnitude less than the value of the coefficient, which is promising. The next step is to plot the **ACF** and **PACF** of the **AR(1)** model. The graphs are found on the following page:



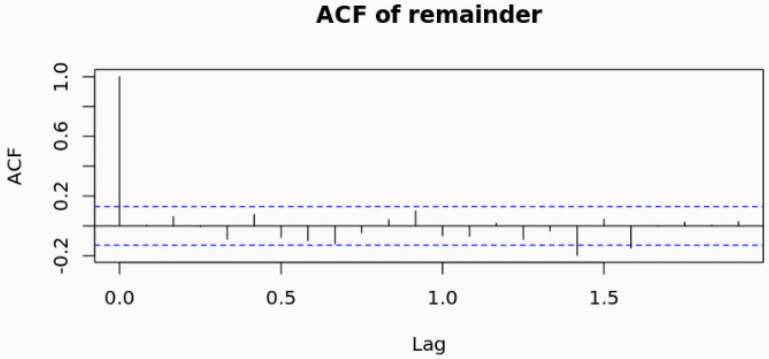



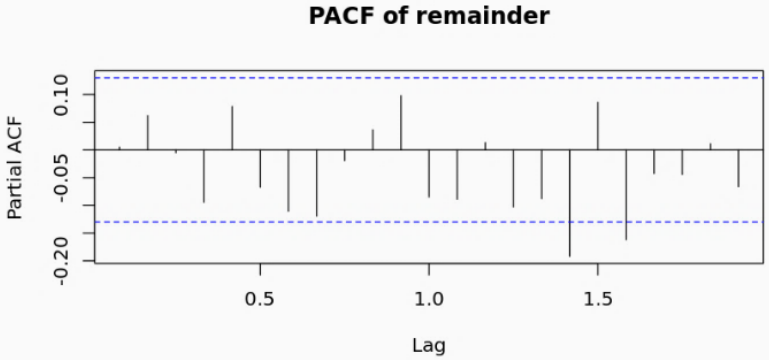

In each case, the standard error is same order of magnitude as the value of the coefficient, indicating this model as a poor fit to the data.

An **ARMA(1)** Model is tested next:

In both cases, the **AR(1)** and the **MA(1)** are good fits to the data. This is proven when as the **ACF** and the **PACF** indicate **no significant features** outside of **0** for the **ACF**.

Differencing Method follows in the next section.




Note that only the **0** lag of the **ACF** is significant and that there are **no significant lags** for the **PACF**. These observations indicate that the **AR(1)** model is a good fit. Compare these results to those of the **MA(3)** model, noting that they are nearly identical. Evidently, either the **MA(3)** or **AR(1)** model is a good choice for this data.

## Autoregressive Moving Average Models (ARMA)

Both **MA(3)** and **AR(1)** models are good fits to the remainder series; an Autoregressive Moving Average **(ARMA)** model will be tested next on the remainder series; starting with an **ARMA(1,3)** model:

```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ar1     ma1     ma2     ma3
      0.1532  0.5750  0.4288  0.2274
s.e.  0.3874  0.3889  0.2588  0.1805

sigma^2 estimated as 0.0001876:  log likelihood = 654.39,  aic = -1298.78
```

```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ar1     ma1
      0.6777  0.0330
s.e.  0.0661  0.0856

sigma^2 estimated as 0.0001899:  log likelihood = 653.02,  aic = -1300.05
```

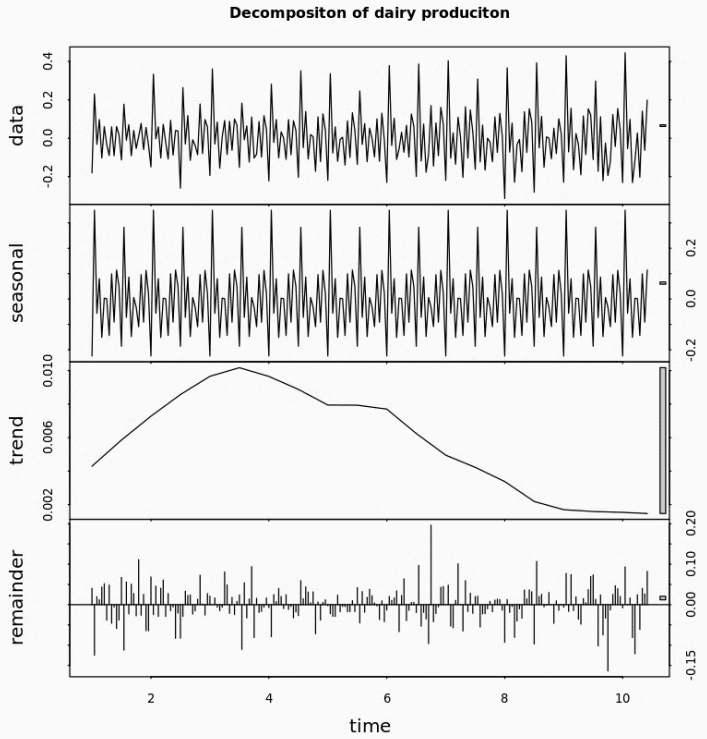
## Exploring the Difference Series

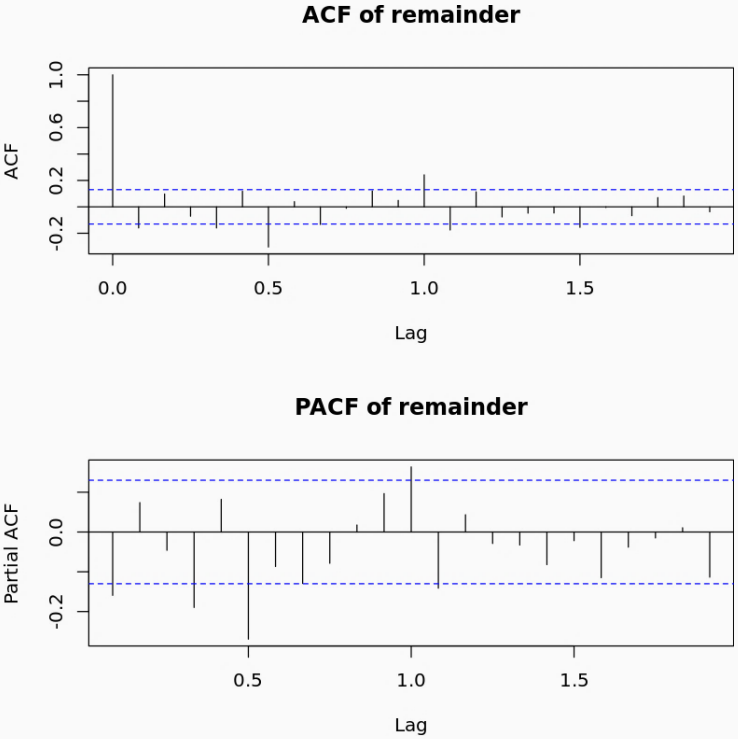
Difference series is a method to remove trend from a time series. The difference can be computed for any number of lag values, depending on the order of the trend. In this case a first order difference series is used to model the trend in the milk production.

Note the difference series is necessarily of length one less than the original series.

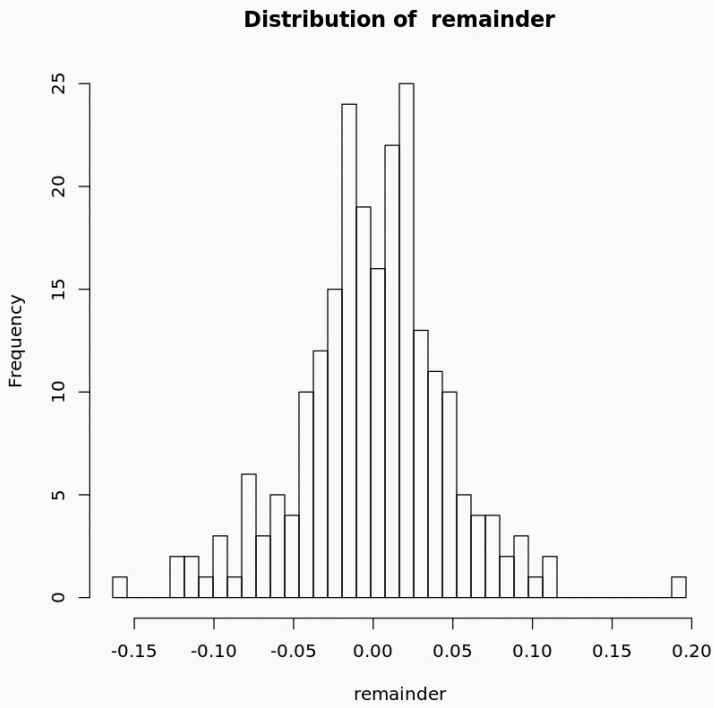
The stl decomposition of the difference series is computed next. Considering working with a difference series, which has positive and negative values, an additive model can be used. No logarithm is taken. The decomposition of the difference series is on the following page:

The difference series is shown in the upper most plot. Note the small magnitude of the remaining trend indicating that the first order difference model removed most of the trend. However, the seasonal series exhibits a pattern with a 24 month cycle which is a bit odd.




The **ACF** and **PACF** represents a couple of significant features at **0**, **0.5**, and **1.0**. The data does not appear to be stationary, but is close. The distribution looks relatively normal with slight variation.





It is clear from the exploration of the **ARMA** model, that the remainder of the decomposition of the dairy production time series is not stationary.

## Autoregressive Integrative Moving Average Model

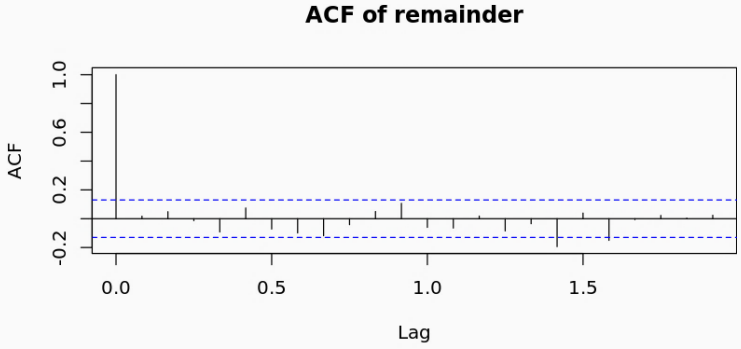
The remainder series is modeled with an autoregressive integrative moving average (ARIMA) model.

An **ARIMA(1, 1, 1)** Model is initially run:

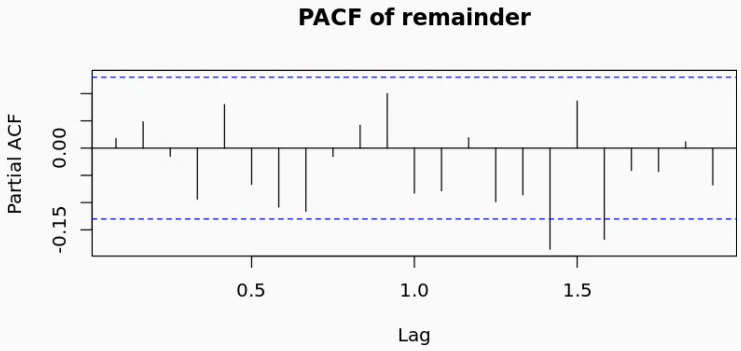
```
Call:
arima(x = ts, order = order, include.mean = FALSE)

Coefficients:
         ar1      ma1
      0.7015  -1.0000
s.e.  0.0482   0.0114

sigma^2 estimated as 0.0001908:  log likelihood = 648.11,  aic = -1290.22
```

The standard error of the **AR1** coefficient is only about half its value. This model seems to be a reasonable fit.

Next, the ACF and PACF of the model are plotted to determine if stationary:




Note that only the **0** lag of the **ACF** is significant and that there are no significant lags for the **PACF**. These observations indicate that the **ARIMA(1,1,1)** model is a good fit. Comparing these results to those of the **MA(3)** and **AR(1)** models, they are nearly identical. The **ARIMA(1,1,1)** model is a good choice for this data as well.




## Modeling and Forecasting

After exploring the properties of the Decomposed Timeseries, the forecasts of Dairy Production are computed next. The **R Forecast** package is used to forecast the next 12 months of Diary Production.

The R **forecast** package contains the **auto.arima** function which automatically steps through the **ARIMA** model parameters to find the best fit to the data. The **ARIMA** model used in the **forecast** package also includes modeling of seasonal differences. The **auto.arima** function has multiple arguments, specifying the parameter range values to search. The first argument is a time series object of class **ts**. The code acts as follows:

- Creates a time series of class **ts**.
- Automatically finds and computes an **ARIMA** model.
- Prints a summary of the **ARIMA** model.

```
Series: temp
ARIMA(0,1,1)(0,1,2)[12]

Coefficients:
          ma1     sma1    sma2
      -0.1506  -0.9076  0.1129
s.e.   0.0743   0.0794  0.0838

sigma^2 estimated as 0.0002547:  log likelihood=577.8
AIC=-1147.6   AICc=-1147.41   BIC=-1134.12

Training set error measures:
                       ME       RMSE        MAE         MPE    MAPE      MASE
Training set -0.0003536657 0.01549906 0.01109068 -0.01955938 1.05342 0.2902694
                    ACF1
Training set 0.005145456
```

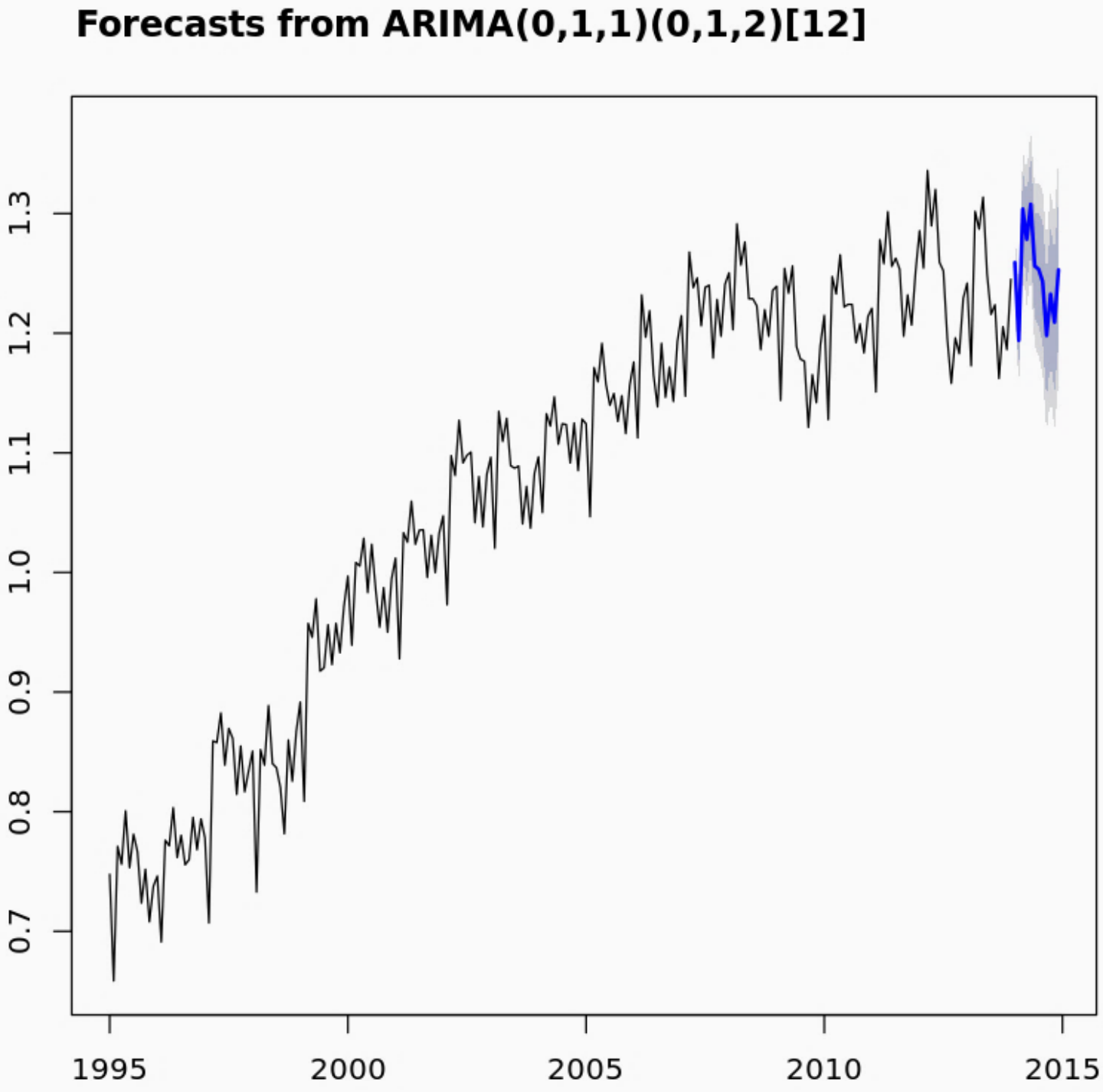
- The model uses an **MA(2)** model for the seasonal difference. The coefficients of this model, **sma1** and **sma2**, along with their standard errors can be seen in the summary.
- The model of the remainder is and **MA(1)** model. The coefficient and its standard error can be seen in the summary above.
- Error metrics, including **RMSE**, are provided in the summary. Notice that the **RMSE** is much smaller than the values of the milk production time series indicating good model performance.

The **forecast** function is used to compute the forecast of the next 12 months using the model created using **auto.arima**:

```
Forecast method: ARIMA(0,1,1)(0,1,2)[12]

Model Information:
Series: temp
ARIMA(0,1,1)(0,1,2)[12]

Coefficients:
          ma1     sma1    sma2
      -0.1506  -0.9076  0.1129
s.e.   0.0743   0.0794  0.0838

sigma^2 estimated as 0.0002547:  log likelihood=577.8
AIC=-1147.6   AICc=-1147.41   BIC=-1134.12

Error measures:
                       ME       RMSE        MAE          MPE    MAPE      MASE
Training set -0.0003536657 0.01549906 0.01109068 -0.01955938 1.05342 0.2902694
                    ACF1
Training set 0.005145456

Forecasts:
         Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
Jan 2014       1.259101 1.238648 1.279555 1.227820 1.290382
Feb 2014       1.193990 1.167154 1.220825 1.152948 1.235031
Mar 2014       1.303803 1.271835 1.335772 1.254912 1.352695
Apr 2014       1.278470 1.242086 1.314854 1.222825 1.334115
May 2014       1.307799 1.267480 1.348118 1.246136 1.369462
Jun 2014       1.256228 1.212325 1.300130 1.189084 1.323371
Jul 2014       1.253456 1.206240 1.300671 1.181246 1.325665
Aug 2014       1.243199 1.192889 1.293509 1.166256 1.320141
Sep 2014       1.198112 1.144887 1.251337 1.116711 1.279512
Oct 2014       1.232666 1.176677 1.288655 1.147039 1.318293
Nov 2014       1.209167 1.150544 1.267789 1.119512 1.298821
Dec 2014       1.252931 1.191789 1.314074 1.159422 1.346440
```

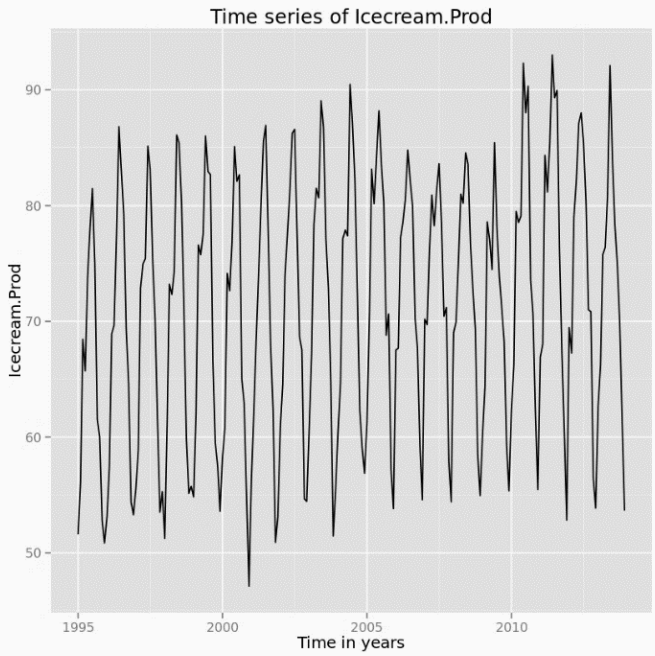
Much of the summary is the same as before. A 12 month forecast is printed below the model summary. There is a point forecast (the expected value) along with 80 and 95 percent confidence intervals. Note, that the confidence intervals generally get wider for forecasts further out in time. It is not surprising that the forecast has more uncertainty as time increases from the present.

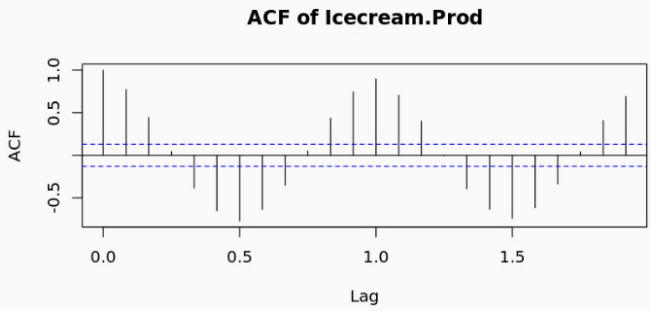
The forecast is plotted on the following page:

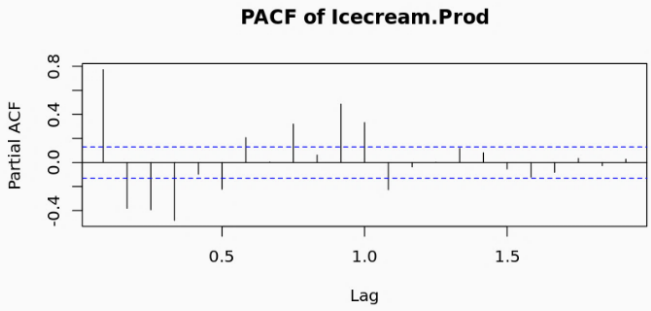



The original time series of milk production is shown in black in the plot above. The forecast is shown in **Blue**. The 80 and 95 percent confidence intervals are shown in lighter shades of **blue-gray**.

## Plotting Ice Cream Production

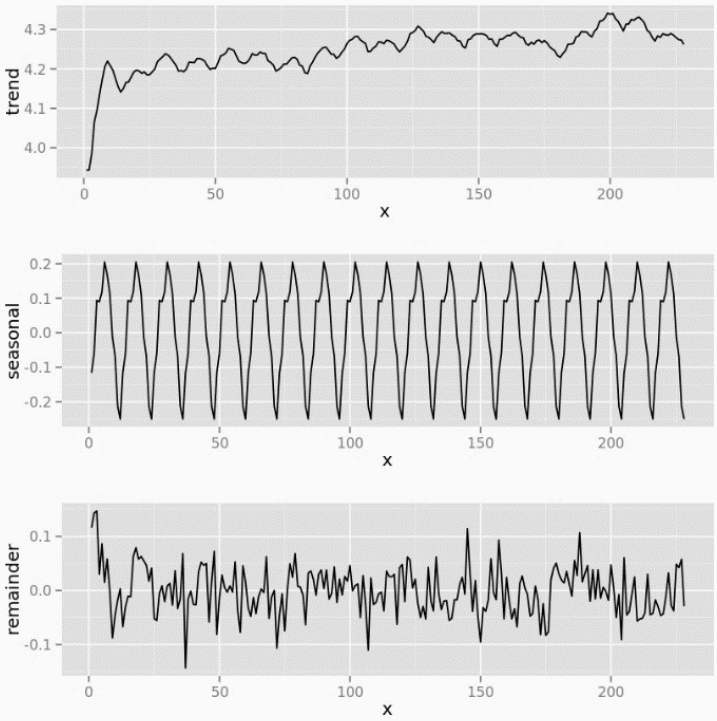


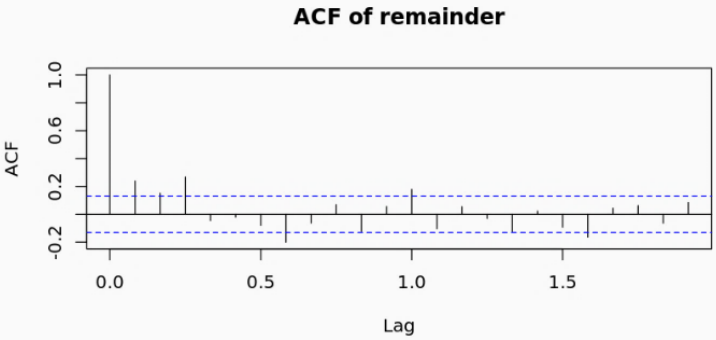



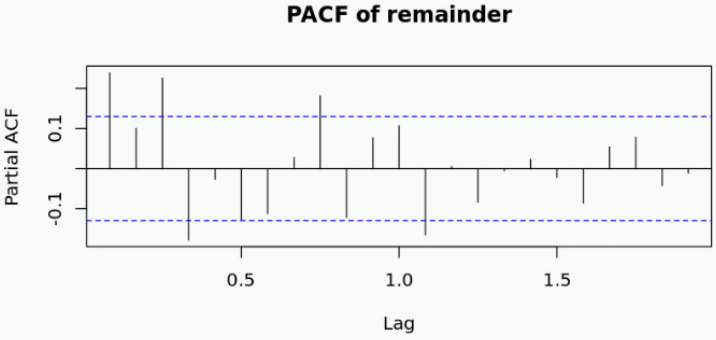




## Simple Moving Average Decomposition of the Time Series

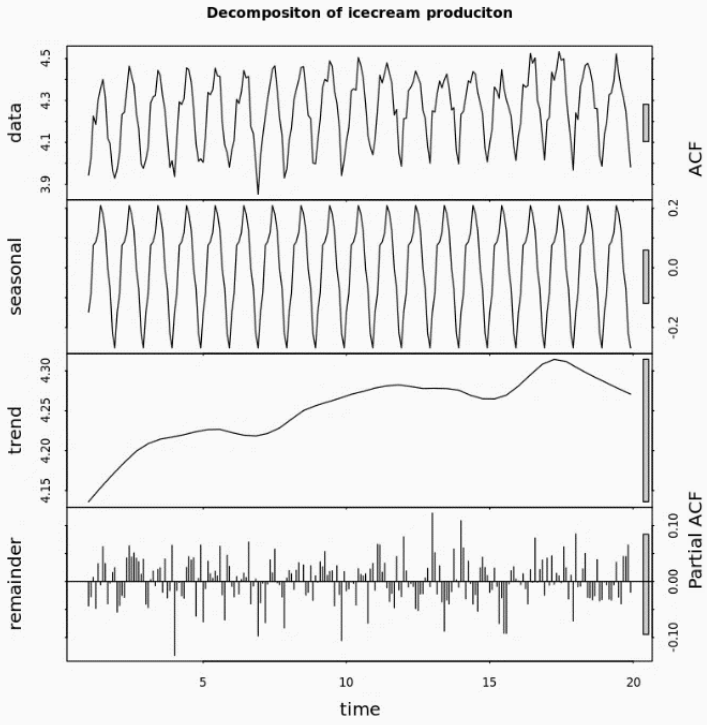
| | trend | seasonal | remainder |
|---|---|---|---|
| 1 | 3.943425 | -0.1161525 | 0.1161525 |
| 2 | 3.943425 | -0.05987141 | 0.1433329 |
| 3 | 3.985155 | 0.09401897 | 0.1469729 |
| 4 | 4.065486 | 0.08971517 | 0.03023243 |
| 5 | 4.095473 | 0.1186856 | 0.08625111 |
| 6 | 4.13646 | 0.205117 | 0.01505456 |

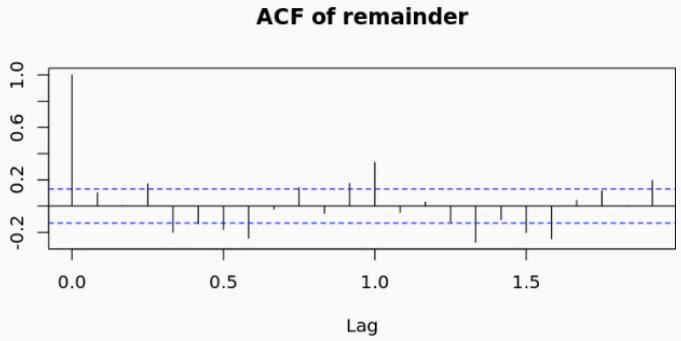


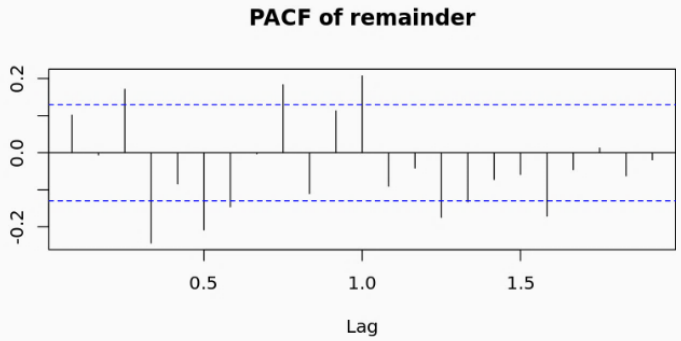



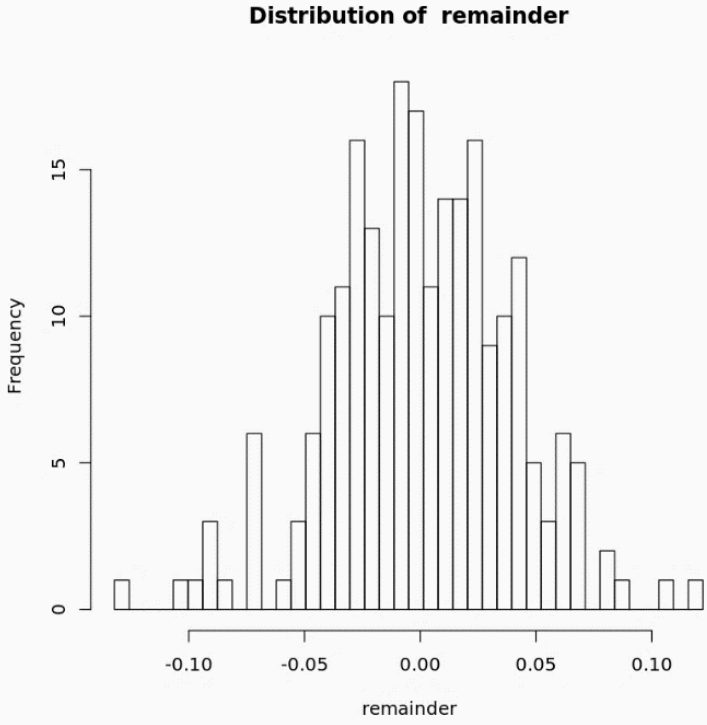



# Exploring the Multiplicative Model with Lowess




## Modeling and Forecasting Ice cream Production

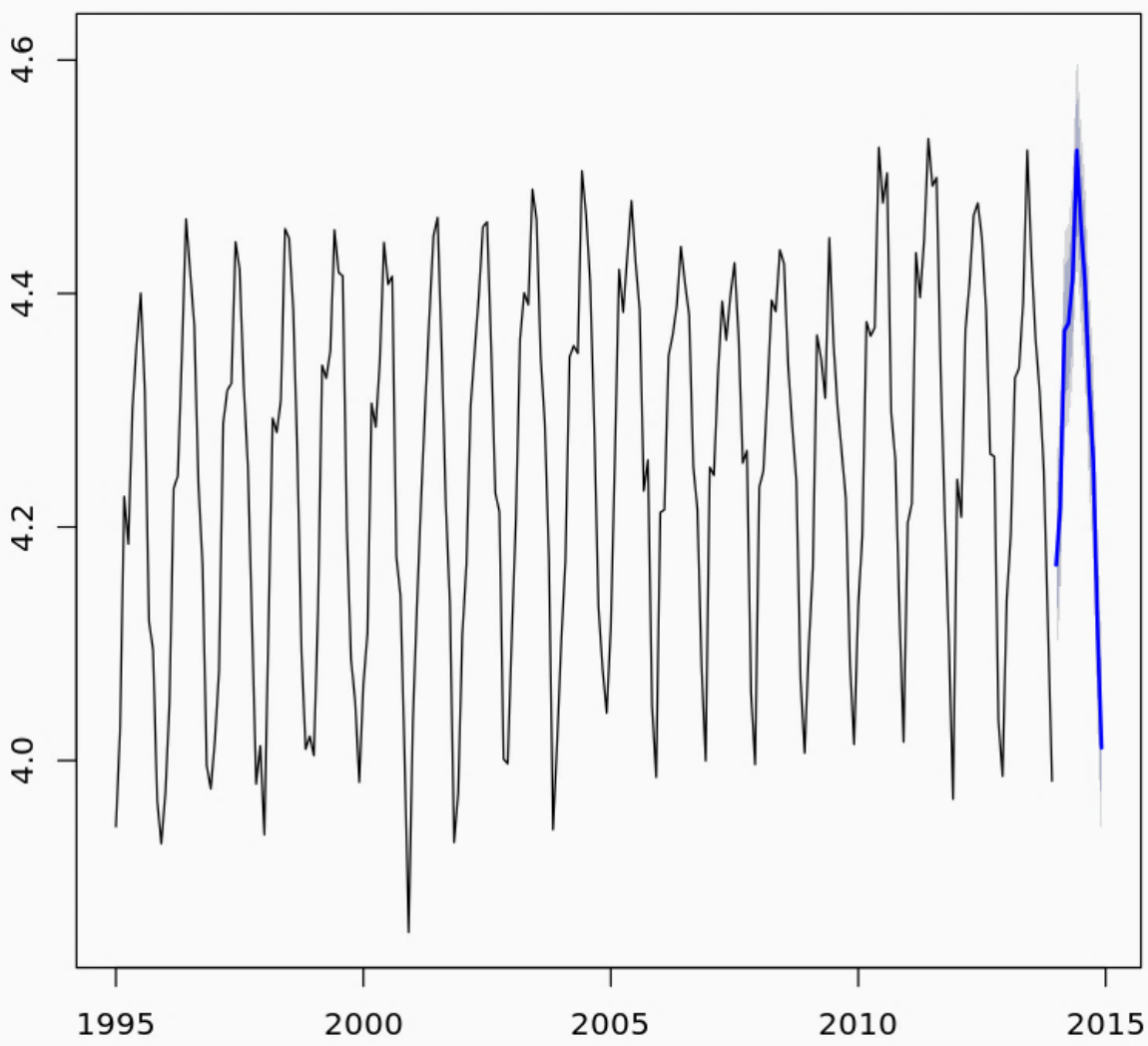
The forecast the production of **icecream** for a 12 month period following these steps.

- Create a time series object with **frequency = 12** and **start = 1995** from the **Icecream.Prod** column of the **dairy** data frame.
- Fit a model with the **auto.arima** function, following the hint given below.
- Print a summary of the model. What is the order of the **MA** and **AR** components of the seasonal and remainder models? Note there will be a drift term, which accounts for linear trend in the time series.
- Compute the forecast of icecream production for the next 12 months.
- Plot the forecast and note the behavior.

The **ts**, **auto.arima**, **summary**, **forecast**, and **plot** functions are used to create the new model, the summaries are printed to make the plots. **Hint** use the time series of icecream production as the first argument to the **auto.arima** function. The other arguments from the milk production model can be copied and pasted, but **max.p = 1** to prevent having an over-parameterized model.

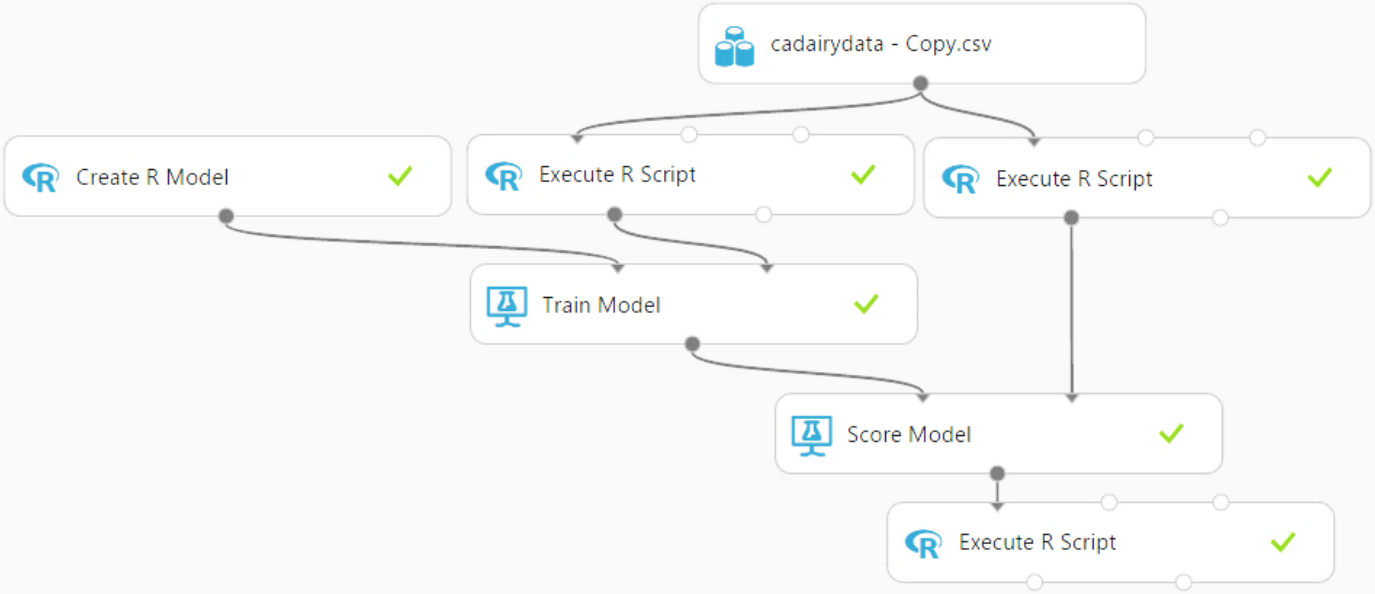
```
Series: tempIC
ARIMA(1,0,1)(0,1,2)[12] with drift

Coefficients:
         ar1      ma1     sma1     sma2  drift
      0.8676  -0.6761  -0.5038  -0.2193  6e-04
s.e.  0.0696   0.0955   0.0694   0.0663  2e-04

sigma^2 estimated as 0.001759:  log likelihood=359.75
AIC=-707.5   AICc=-707.1   BIC=-687.25

Training set error measures:
                      ME       RMSE        MAE         MPE      MAPE      MASE
Training set 0.001246958 0.03967756 0.03089997 0.02618669 0.7287523 0.7882469
                    ACF1
Training set -0.03374896
```

```
Forecast method: ARIMA(1,0,1)(0,1,2)[12] with drift

Model Information:
Series: tempIC
ARIMA(1,0,1)(0,1,2)[12] with drift

Coefficients:
         ar1      ma1     sma1     sma2  drift
      0.8676  -0.6761  -0.5038  -0.2193  6e-04
s.e.  0.0696   0.0955   0.0694   0.0663  2e-04

sigma^2 estimated as 0.001759:  log likelihood=359.75
AIC=-707.5   AICc=-707.1   BIC=-687.25

Error measures:
                      ME       RMSE        MAE        MPE      MAPE      MASE
Training set 0.001246958 0.03967756 0.03089997 0.02618669 0.7287523 0.7882469
                    ACF1
Training set -0.03374896

Forecasts:
         Point Forecast    Lo 80    Hi 80    Lo 95    Hi 95
Jan 2014       4.167588 4.113846 4.221330 4.085397 4.249780
Feb 2014       4.216966 4.162247 4.271684 4.133281 4.300651
Mar 2014       4.368172 4.312730 4.423615 4.283381 4.452964
Apr 2014       4.374599 4.318619 4.430580 4.288984 4.460215
May 2014       4.412652 4.356269 4.469035 4.326422 4.498883
Jun 2014       4.522673 4.465990 4.579357 4.435983 4.609364
Jul 2014       4.459753 4.402844 4.516662 4.372718 4.546788
Aug 2014       4.408627 4.351549 4.465705 4.321333 4.495920
Sep 2014       4.320613 4.263408 4.377818 4.233125 4.408100
Oct 2014       4.256768 4.199468 4.314069 4.169135 4.344402
Nov 2014       4.117332 4.059960 4.174704 4.029589 4.205075
Dec 2014       4.010979 3.953553 4.068405 3.923154 4.098805
```




## Forecasting in Microsoft AzureML

DAT203.3x: Milk Production Forecast



## Statistics

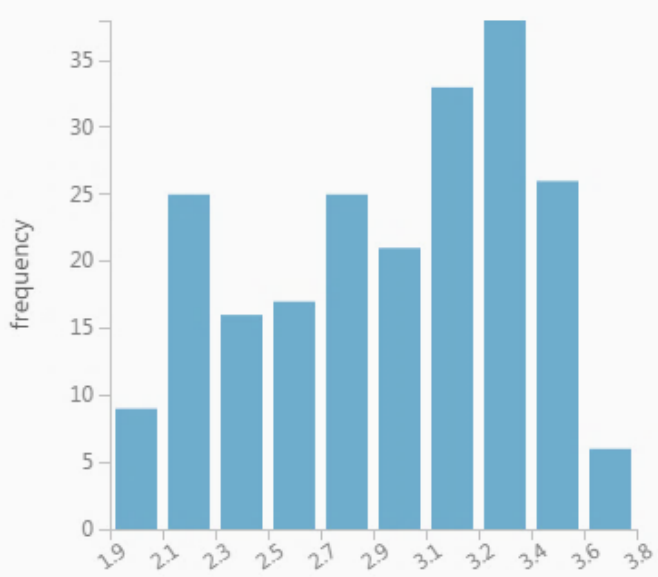
| | |
|---|---|
| Mean | 2.9199 |
| Median | 3.002 |
| Min | 1.932 |
| Max | 3.804 |
| Standard Deviation | 0.4742 |
| Unique Values | 205 |
| Missing Values | 0 |
| Feature Type | Numeric Feature |

## Visualizations

Milk.Prod

Histogram

compare to None




## Statistics

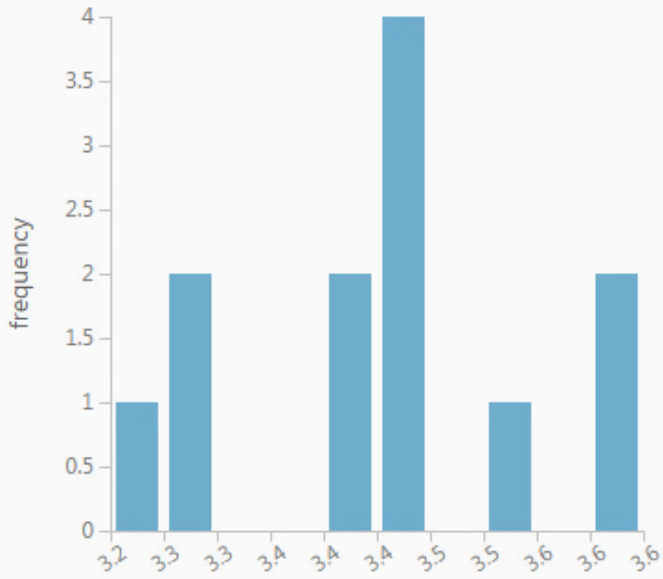
| | |
|---|---|
| Mean | 3.4477 |
| Median | 3.464 |
| Min | 3.2465 |
| Max | 3.637 |
| Standard Deviation | 0.1184 |
| Unique Values | 12 |
| Missing Values | 0 |
| Feature Type | Numeric Feature |

## Visualizations

forecast

Histogram

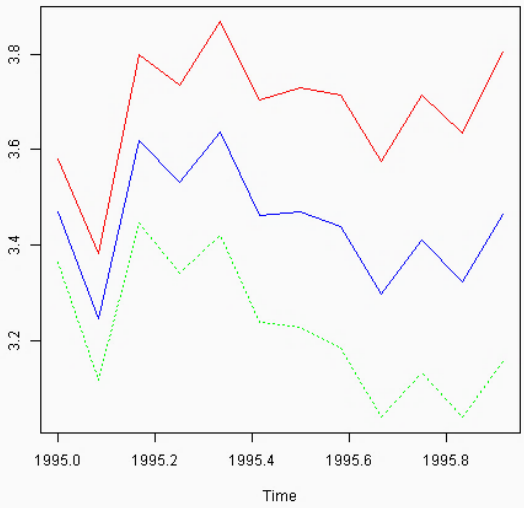
compare to None




## Standard Output

RWorker pushed "port1" to R workspace.
Beginning R Execute Script

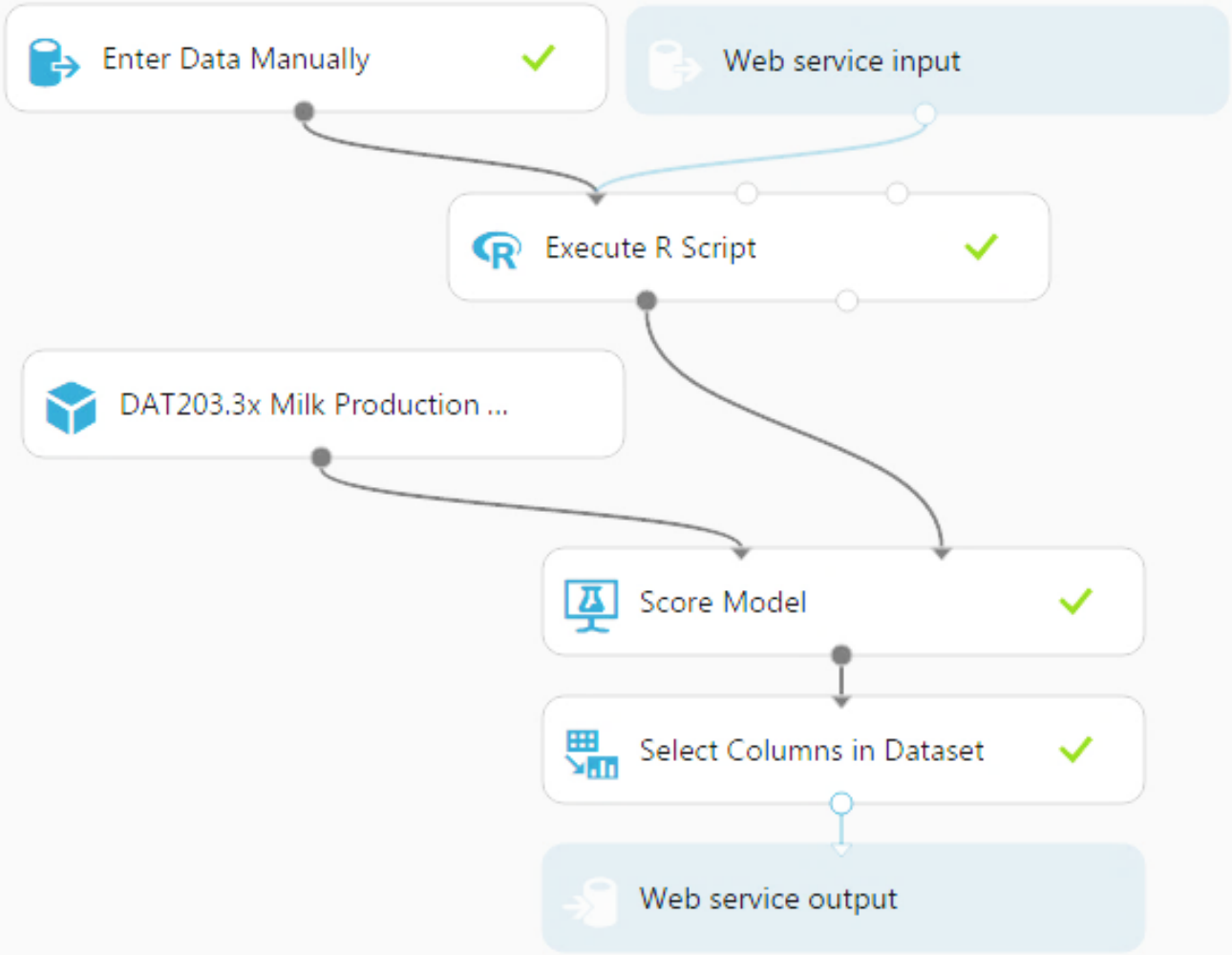
[1] 56000
Loading objects:
port1
[1] "Loading variable port1..."




## Standard Error

R reported no errors.

# DAT203.3x: Milk Production Forecast [Predictive Exp.]




## Summary

This lab worked with and analyzed time series data.

Specifically the following:

- Examined the properties of time series objects.
- Plotted time series data.
- Decomposed time series data into its trend, seasonal, and remainder components.
- Modeled the remainder components as AR, MA, ARMA and ARIMA models.
- Created and evaluated difference series methods.
- Constructed and evaluated a forecasting model.

| forecast | upper95 | lower95 |
|---|---|---|
| 3.469836 | 3.580358 | 3.362726 |
| 3.246524 | 3.382647 | 3.115879 |
| 3.619187 | 3.800553 | 3.446476 |
| 3.532431 | 3.734513 | 3.341284 |
| 3.636989 | 3.868202 | 3.419597 |
| 3.463174 | 3.703532 | 3.238416 |
| 3.469722 | 3.729337 | 3.22818 |
| 3.439218 | 3.71405 | 3.184723 |
| 3.296013 | 3.575274 | 3.038566 |
| 3.410717 | 3.715334 | 3.131075 |
| 3.323451 | 3.634856 | 3.038725 |
| 3.464765 | 3.804016 | 3.155769 |

# module2 · spatial data analysis

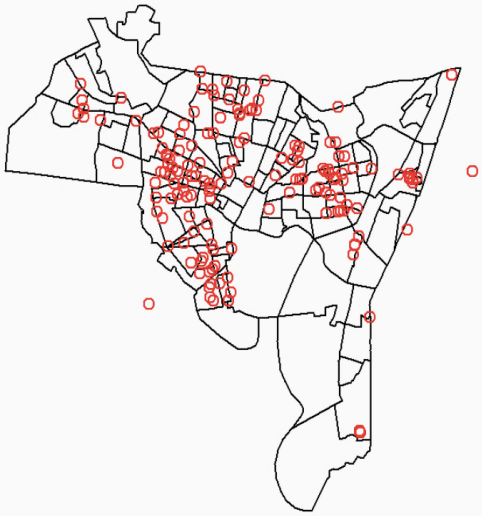
## introduction to spatial data

### introduction

Spatial data encompasses **physical space** including geography and geometry: data obtained from cellular phones, connected devices, biomedical data, oil exploration, real estate prediction, agriculture, internet, natural gas networks, transportation networks, and many other **physical data networks**.

It is important to initially **map** spatial data for understanding of the orientation and **density** for creating predictions of future spatial data characteristics

### initializing the spatial data analysis process



Spatial data has unique characteristics. For example, the illustration (right) maps Forced Entry Burglaries in New Haven CT, Sept 2014. **Point Data** numerically represents the location of each burglary; but often **Raster Data** (a count of point data within each cell).

**Point Data** is not always uniformly distributed (if one area has a higher frequency density than another or if the data is not captured consistently in all circumstance)

A traditional option would be to compute the physical distance (**Euclidean distance**) between the points. However, this is not a best practice with spatial data. An example for reason is if in the example of home burglaries, if the homes were separated by a fence, the **Euclidean distance** would not take into account the barrier between data points.

Alternative measures of defining **distance** between data points could take into account the characteristics of the population. For example, driving distance could account for barriers between points. Additionally, a mixture of distance definitions could be applied based on local circumstance.

| | Crime 1 | Crime 2 | Crime 3 | Crime 4 | Crime 5 | ... | |
|---|---|---|---|---|---|---|---|
| Crime 1 | 0 | 3.329 | 1.309 | ... | ... | | |
| Crime 2 | 2.463 | 0 | 2.235 | ... | ... | | |
| Crime 3 | 1.125 | 2.376 | 0 | | | | |
| Crime 4 | : | : | : | | | | |
| Crime 5 | | | | | | | |
| : | | | | | | | |

A **Distance Matrix** is often required to plot the distance between all points. This is not always applicable for problems that do not have a finite number of entities to compute distances on.

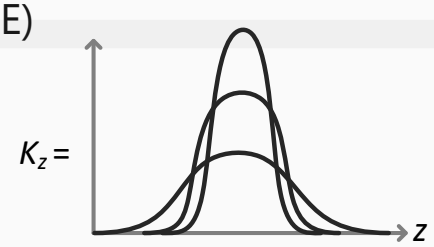
### Spatial Data Warning Considerations

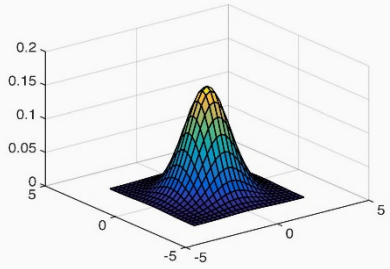
- coordinate systems should not be confused with each other (state plane and gps coordinates)
- Euclidean distances along the Earth should not be manually computed as the Earth is curved
- Various Visualizations will aid in sanity checking the analysis methods

Other types of Spatial Data include **Space-Time Data** that comprise occurrence times of the various Spatial Events. Additionally, **Labeled Spatial Data** could include Binary Classification of an observed classification; **Real-Valued Labels** are also equally probable.

## Kernel Density Estimation (KDE)

$$\hat{f}_h(x) - \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x-x_i}{h}\right)$$




KDE is the most commonly used method of **Density Estimation**.

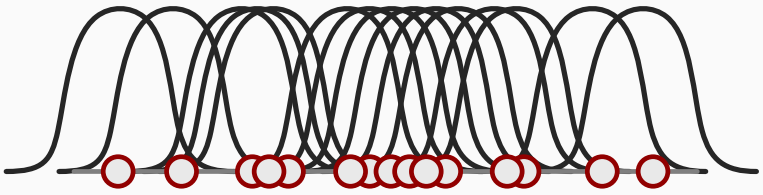
KDE can be applied to both **single** and **multivariate** dimensional density estimation.

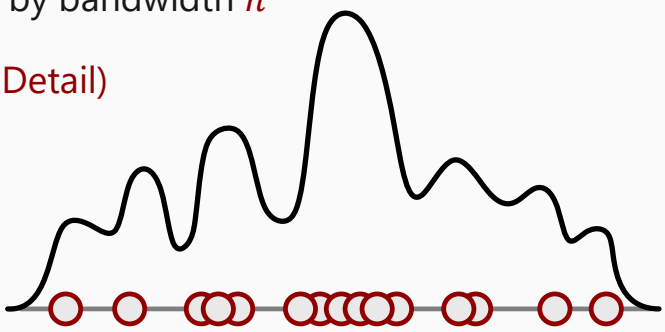
KDE is **Nonparametric** meaning each data point possesses a characteristic distribution

KDE commonly uses the **Gaussian Kernel** that expects a rounded distribution, making it difficult to apply anything other than physical (Euclidean) distance.

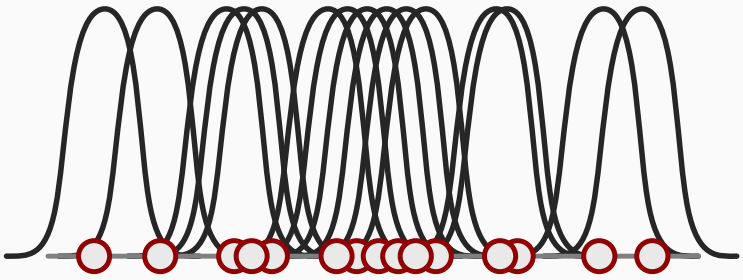
Kernels are centered at the points $x_i$ and scaled by bandwidth $h$

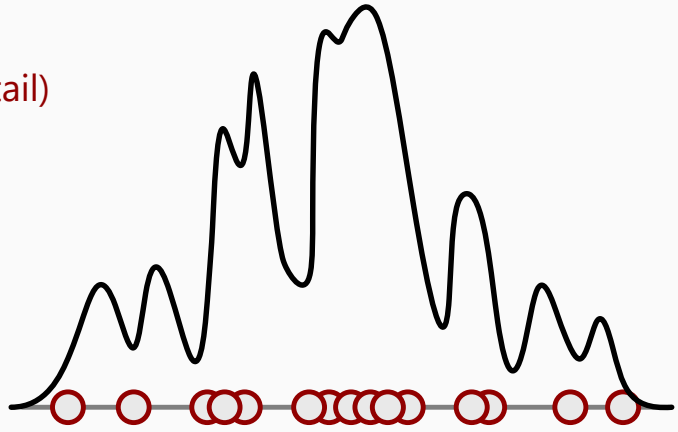
### Adequate KDE Bandwidth (Appropriate Density Detail)



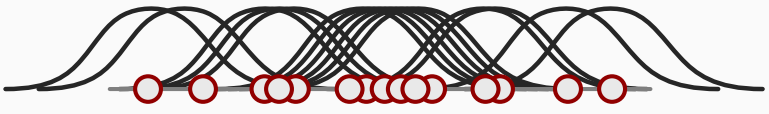


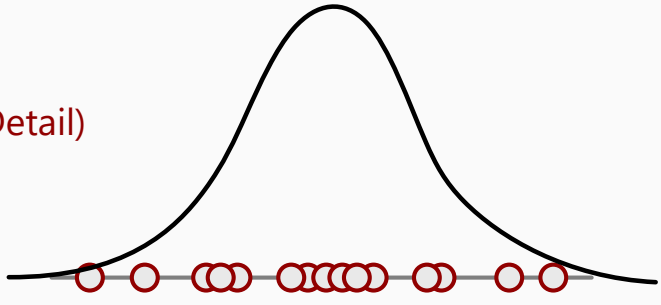
### Narrow KDE Bandwidth (Overfitted Density Detail)





### Excessive KDE Bandwidth (Underfitted Density Detail)





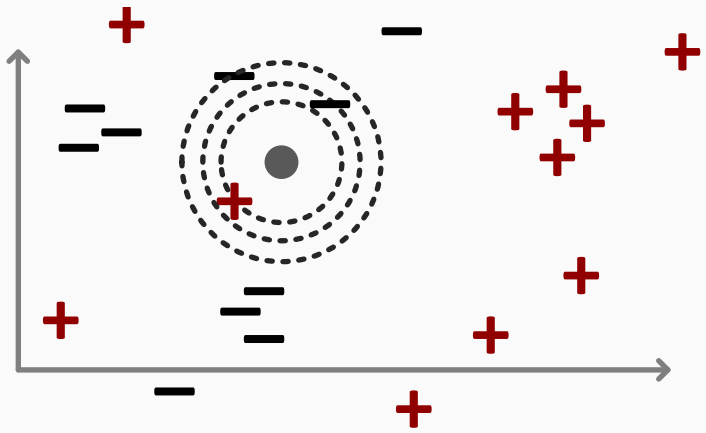
## K Nearest Neighbour ⁘ (KNN)

KNN is one of the most basic machine learning algorithms because it does not require a training set or proprietary model to implement.

KNN can be used for both **Classification** and **Regression** problems by using x's K-Nearest Neighbors to predict what x's next label will likely be (majority vote through proximity ranges)

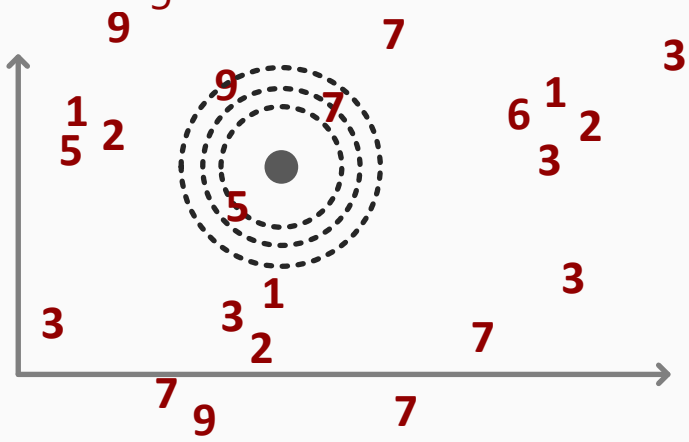
The choice of K-Neighbours determines the classification or label applied to new values based upon proximity to the initializing point:

### KNN Classification



The illustration above:
1-NN is classified as **+**
2-NN is tied between **+** and **–**
3-NN is classified as **–**

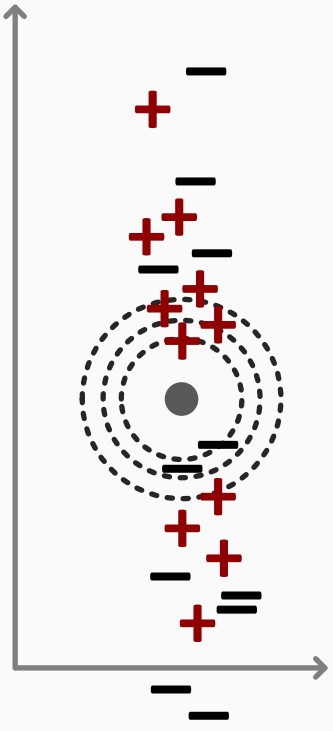
### KNN Regression




The illustration above:
1-NN is classified as 5
2-NN is classified as 6 ([5+7]/2)
3-NN is classified as 7 ([5+7+9]/3)

The parameter ***k*** controls overfitting the dataset. Cross-validation can possibly help. K-Nearest Neighbours does not account for the physical distance between the nearest points; weighting cannot be assigned to Neighbours that are closer than others.

Changing how distance is measured can impact how the algorithm classifies Neighbour proximity. The distance measure has to be meaningful. Separately scaled attributes can be misclassified. For example, if two variables are income and height, the income scale will be large (thousands of dollars) while the height will be less significant (inches). Therefore, the model will classify the vertical axis containing the range of nearest neighbors which is truly not representative of the dataset (right figure).



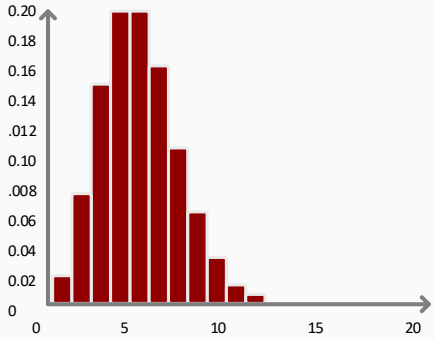
### Pros/Cons to KNN

Simple and powerful method that can be used instead of other classification/regression techniques. Training is not necessary and new examples easily added ("lazy model").

However, KNN is expensive and slow. Computation of a new point requires re-running the entire model to compute all distances again. Additionally, KNN might require heavy feature scaling.

# working with spatial data

## Spatial Poisson Processes




The Poisson Distribution:

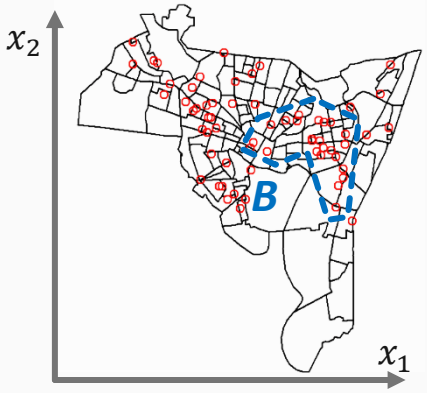
The **Probability Mass Function (PMF)** assigns a random variable ***X*** as **Poisson** with rate parameter lambda ($\lambda$)

$$X \sim \text{Pois}(\lambda)$$

$$P(X = x) = \frac{e^{-\lambda}\lambda^x}{x!}, x = 1, 2, 3, \ldots$$

$$E(X) = \lambda, \quad \text{Var}(X) = \lambda$$

Assuming a model of crime rate occurring in a given month within area $B$ designated spatially:




Choose any region $B$ and any number of crimes $n$.

Determine how often the number of points in $B = n$.

$$P(\text{Number of points } in\ B = n) = \cdots$$

- Should depend on the size of area $B$
- Should depend on the value of $n$
- Should depend on the rate of events $\lambda$

$$P(\text{Number of points } in\ B = n) = \frac{(\lambda\ \text{size}(B))^n}{n!} e^{-\lambda\ \text{size}(B)}$$

The expression above is referred to as a **Homogeneous Point Process** with parameter $\lambda$

**There is a flaw in the logic:** The distribution assumes crimes can happen equally anywhere. If the crime rate should be represented as a non-constant, then lambda $\lambda$ should be represented as non-constant; it should change depending on where the model is spatially. So when the latter is true, the expression above is becomes an **Inhomogeneous Point Process** with parameter $\lambda(x_1, x_2)$ the rate of events should $\lambda$ depend on $x_1, x_2$. Considering the area $B$ can have more concentrated rates of crime $\lambda$ than others, the rate $\lambda$ is integrated over the area $B$:

$$P(\text{Number of points } in\ B = n) = \cdots$$

$$\Lambda(B) = \int_B \lambda(x_1, x_2) dx_1 dx_2$$

$$P(\text{Number of points } in\ B = n) = \frac{(\Lambda(B))^n}{n!} e^{-\Lambda(B)}$$

possibly $\lambda(x_1, x_2) = ax_1 + bx_2 + c$

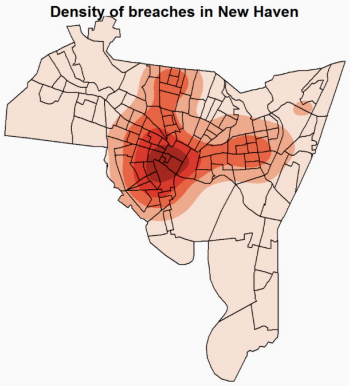
possibly $\lambda(x_1, x_2) = \text{polynomial}(x_1, x_2)$

possibly $\lambda(x_1, x_2) = \sum_i K_h([x_1, x_2], [x_1^i, x_2^i])$

possibly $\lambda(x_1, x_2) = f(\text{proximity to nearest drug house})$

The expression is similar to before with the rate integrated over $B$ with various dependencies (above).

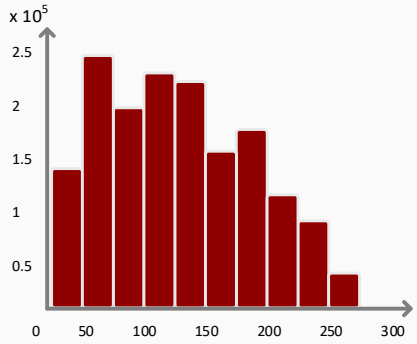
The scientist will do the modeling and the coding packages will fit the model accordingly. The code will determine the parameters using **Maximum Likelihood Estimate** and help determine the **bandwidth**. This process goes a step further than **Density Estimation**; a model is now created and parameters can be fitted accordingly.

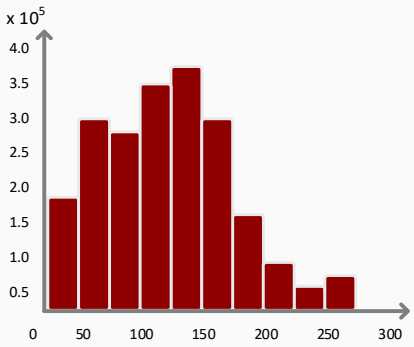


# working with spatial data

## Variograms – used to describe spatial correlations in a way that Histograms do not.

In the case of the two images below, histograms were computed on the grayscale values of each:



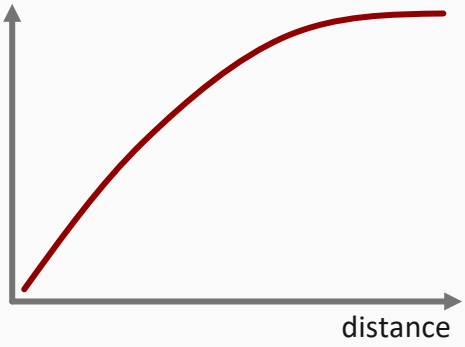




The Histograms ploted against the frequency of each images greyscale points are insignificantly different from each other; Additionally, the **Histograms cannot capture** any similiarities between nearby sprinkles as seen in the images themslevles.

Therefore, if the sprinkles in the pictures (or greyscale points) were randomly rearranged, there would be no actual effect on the histograms produced from the image data.

In contrast, **Varigrams show *how*** nearby points are correlated. The left image illustrates how nearby points are equally correlated as far away points, representing the actual image. The right image curves smoothly demonstrating nearby points as more correlated than further points represented in the image.

Variogram


### Formally Defining Variograms

$(x_1, x_2)$ is a point in space

$y(x_1, x_2)$ is the label at point $(x_1, x_2)$

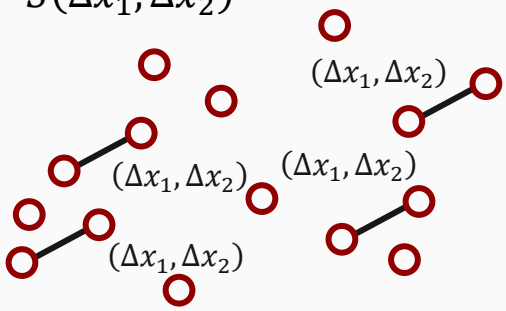
$y(x_1 + \Delta x_1\,, x_2 + \Delta x_2)$ is the label at point $(x_1 + \Delta x_1\,, x_2 + \Delta x_2)$

$\Delta x_1$ and $\Delta x_2$ are separations between points (lags)

Variogram:

$$\gamma(\Delta x_1, \Delta x_2) = \frac{1}{2} E[y(x_1 + \Delta x_1\,, x_2 + \Delta x_2) - y(x_1, x_2)]^2$$

$S(\Delta x_1, \Delta x_2)$




The expression measures the question: On average, how much does the label $y(x_1, x_2)$ change at distance delta $\Delta$ away from the initialized point in space $(x_1, x_2)$?

The expression also includes an expectation $E$ that needs to be estimated to compute:

For each change in distance $\Delta x_1$ and the change in distance $\Delta x_2$, find all pairs that are separated by those two distances $\Delta x_1$ and $\Delta x_2$ to denote $S(\Delta x_1, \Delta x_2)$.

$S(\Delta x_1, \Delta x_2) =$ the set (number count) of $i, j$ pairs sharing the same distances:

$$x_1^i - x_1^j \approx \Delta x_1 \text{ and } x_2^i - x_2^j \approx \Delta x_2$$

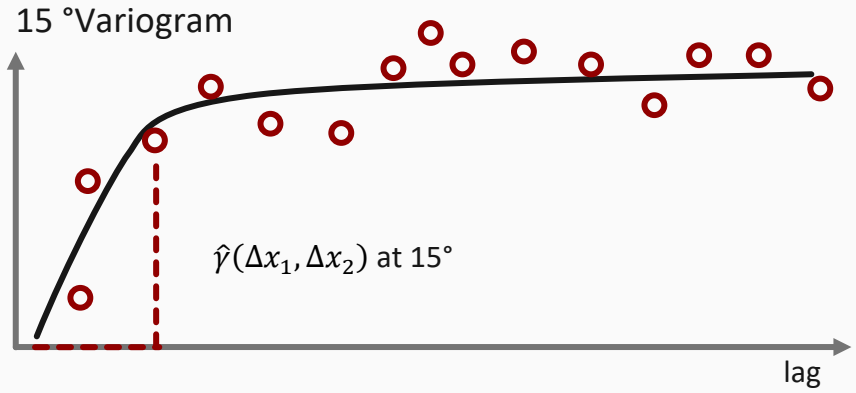
$$|S(\Delta x_1, \Delta x_2)| = \text{the number of pairs in S}$$

Therefore, the Varigram Expresion with Estimated Expectation is computed by taking the points in to approximate the Variogram:

$$\hat{\gamma}(\Delta x_1, \Delta x_2) = \frac{1}{2}\frac{1}{|S(\Delta x_1, \Delta x_2)|}\sum_{i,j\ in\ |S(\Delta x_1, \Delta x_2)|}\left(y^i - y^j\right)^2$$

Essentially, the expression above produces a variogram for each individually measured angle:

Assuming a $(\Delta x_1, \Delta x_2)$ of 15° (left illustration), all point pairs that are 15° apart will be included in the set $|S(\Delta x_1, \Delta x_2)|$. The values will then be squared and averaged over all the pairs in $|S(\Delta x_1, \Delta x_2)|$. providing a point on the 15° Variogram. The process used to compute the 15° lag distance is repeated at different lags and multiple directions to produce a comprohensive Variogram representive of the actual image.

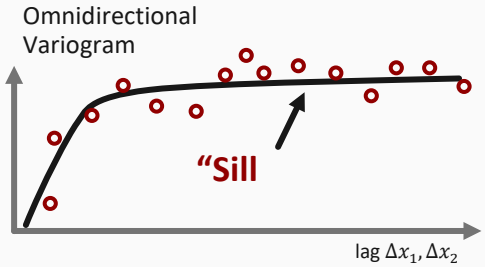



In some circumstances, the Variograms will be similar between multiple directions applied (15°, 45°, 60°, etc.). In this case, it is not necessary to show multiple Variograms and can instead use a singe **Omnidirectional Variogram** representative of all the measured lags.

Omnidirectional Variograms can be computed with ***any*** directions measured by simple **Euclidean Distance**.

An important note: A Variogram is everything needed to compute a Covariance Estimate between $y$ and it's spatially lagged version; simply using the negative of the Variogram and flipping the Sill (the flat portion of the Variogram Curve):

Covariance: $k((x_1 + \Delta x_1, x_2 + \Delta x_2) = \textbf{"Sill"} - \hat{\gamma}(\Delta x_1, \Delta x_2)$

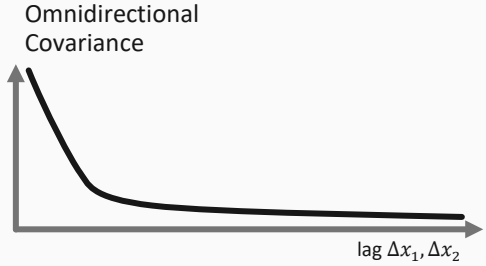



Additional items needed from the Variogram for Kriging or Gaussian Processes:

- A way to etimate pairwise covariances (the Variogram provides this)
- A full matrix of pairwise variances for training points in the dataset.
- Be able to estimate labels at a new point (variance of label at point $x$)

$$\begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ k(x_n,x_1) & \cdots & \cdots & k(x_n,x_n) \end{bmatrix} \quad \begin{bmatrix} k(x,x^1) \\ k(x,x^2) \\ \vdots \\ k(x,x^n) \end{bmatrix} \quad k(x,x)$$

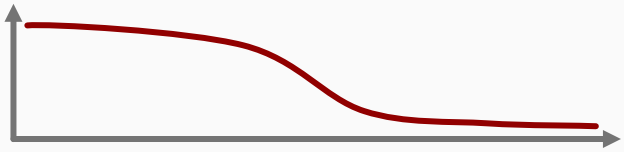
$K^{\text{neighbors}}$ $K^x$



The machine learning community uses a simpler method:

Use the **Gaussian PDF** (Probability Density Function) formula to estimate covariance. The intuition is the covariance decreasing according to the Gaussian PDF with a particular bandwidth, in all directions:

$$k(x, x^1) = C\exp\left[\frac{-(x - x^1)^2}{2h^2}\right]$$



However, regardless of the method chosen, the matrix of numbers $K^{\text{neighbors}}$, vector of numbers $K^x$, and single number $k(x, x)$, are still needed in order to proceed to Kriging.

The final term for Variograms is the concept of a Semi-Variogram; removing the fraction $\mathbf{\frac{1}{2}}$.

Variogram:

$$\hat{\gamma}(\Delta x_1, \Delta x_2) = \mathbf{\frac{1}{2}}\frac{1}{|S(\Delta x_1, \Delta x_2)|}\sum_{i,j\ in\ |S(\Delta x_1, \Delta x_2)|}\left(y^i - y^j\right)^2$$

SemiVariogram:

$$\hat{\gamma}(\Delta x_1, \Delta x_2) = \frac{1}{|S(\Delta x_1, \Delta x_2)|}\sum_{i,j\ in\ |S(\Delta x_1, \Delta x_2)|}\left(y^i - y^j\right)^2$$

A SemiVariogram is just twice (2x) the same Variogram.

# working with spatial data

## Kriging (Part I) · also referred to as Gaussian Processes and Spatial Regression

Beginning with the three items obtained from the Variogram:

- $K^{\text{neighbors}}$ matrix containing the estimated covariance between pairs of training points
- $K^{x}$ vector containing estimated covariance between new points and each training point
- an estimate of the covariance $k(x,x)$

$$\begin{bmatrix} k(x_1,x_1) & k(x_1,x_2) & \cdots & k(x_1,x_n) \\ k(x_2,x_1) & k(x_2,x_2) & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ k(x_n,x_1) & \cdots & \cdots & k(x_n,x_n) \end{bmatrix} \qquad \begin{bmatrix} k(x,x^1) \\ k(x,x^2) \\ \vdots \\ k(x,x^n) \end{bmatrix} \qquad k(x,x)$$

$$\boldsymbol{K^{\text{neighbors}}} \qquad \boldsymbol{K^{x}}$$

$$\begin{bmatrix} 1.9 & 1.2 & \cdots & 0.005 \\ 1.2 & 1.9 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 0.005 & \cdots & \cdots & 1.9 \end{bmatrix} \qquad \begin{bmatrix} 0.5 \\ 0.6 \\ \vdots \\ 0.2 \end{bmatrix} \qquad k(x,x) = 1.9$$

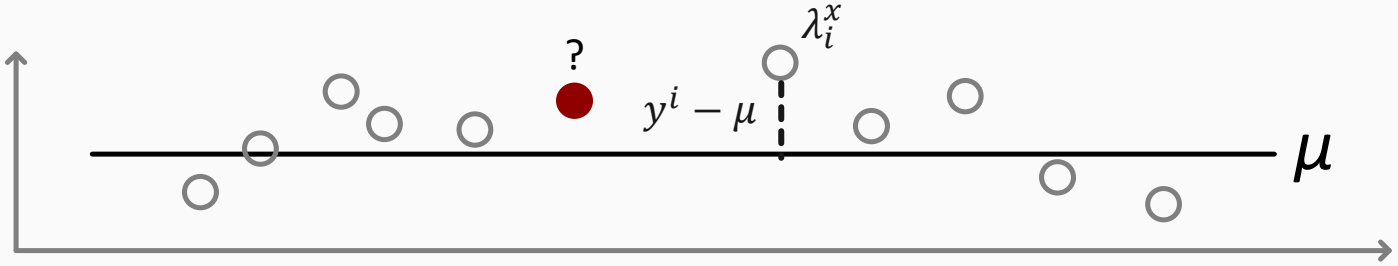
Two ways to derive this method lead to the same result; Maximum Likelihood (MLE) or Optimization

Maximum Likelihood Estimation is often thought as Kriging and Optimization as Gaussian Processes.

$$\hat{\gamma}(x) = \mu + \sum_{i \text{ neighbours}} \lambda_i^x \cdot (y^i - \mu)$$

The estimation is anything is varying in the sample space (label at point $x$). The estimation is the trend $\mu$ plus the summation over the residuals (distances between neighbours) $(y^i - \mu)$ weighted $\lambda_i^x$ for the neighbours that depend on $x$. Highly correlated neighbors do not all have to be taken into account.

The question is: How to get a prediction for $y$ at this point?




Examine the neighbors and determine how far above the trend each is $(y^i - \mu)$, assuming the distance is about the same as all the neighbours. The weighted average is computed of the distances from the trend $\lambda_i^x$.

The remaining questions: How to compute mu $\mu$ and the lambda $\lambda$ for a particular $x$.

# Kriging (Part II) · also referred to as Gaussian Processes and Spatial Regression

$$\hat{y}(x) = \mu + \sum_{i \text{ neighbours}} \lambda_i^x \cdot (y^i - \mu)$$

Mu $\mu$ is simple to compute in terms of Kriging as it estimates the trend by the empirical mean of all the examples. However, the mean $\mu$ can easily be removed by normalizing everything to 0 and assuming the mean is then 0.

Determining lambda $\lambda$ in terms of Kriging is more involved:

Define prediction error as the predicted value less the actual value: $\hat{y}(x) - y(x)$, Kriging functions to minimize the prediction error.

Prediction Error intuition: $\hat{y}(x) - y(x) = (\hat{y}(x) - \mu) - (y(x) - \mu)$ to get distances from the mean. These distances in turn, can be referred to as remainders: $(\hat{R} - R)$ which is the remainder of the empirical (estimated) value and the actual value.

Knowing that Kriging tries to minimize the prediction error $(\hat{R} - R)$ so the variance of the prediction error $\hat{\sigma}^2(x)$ is as follows:

$$\hat{\sigma}^2(x) = \text{Var}(\hat{R}) + \text{Var}(R) - 2\text{Cov}(\hat{R}, R)$$

Plugging in the variables from above:

$$\hat{\sigma}^2(x) = \sum_i \sum_k \lambda_i^x \lambda_k^x (x^i, x^k) + k(x,x) - 2\sum_i \lambda_i^x k(x^i, x)$$

The method Kriging minimizes variance is by setting the derivatives to 0:

$\frac{\partial \hat{\sigma}^2(x)}{\partial \lambda_i^x} = 2\sum_k \lambda_k^x k(x^1, x^k) - 2k(x^i, x) = 0$ → Taking the partial derivative of the variance $\hat{\sigma}^2$ with respect to the first component of lambda $\lambda$.

The expression can be vectorized as follows for setting all derivatives to 0:

$\lambda^x K^{\text{neighbors}} = K^x$ and solve for $\lambda$ with an inverse: $\lambda^x = K^x K^{\text{neighbors}^{-1}}$

$\mathbb{R}^{1 \times \text{neighbours}}$ $\quad$ $\mathbb{R}^{\text{neighbours} \times \text{neighbours}}$ $\quad$ $\mathbb{R}^{1 \times \text{neighbours}}$

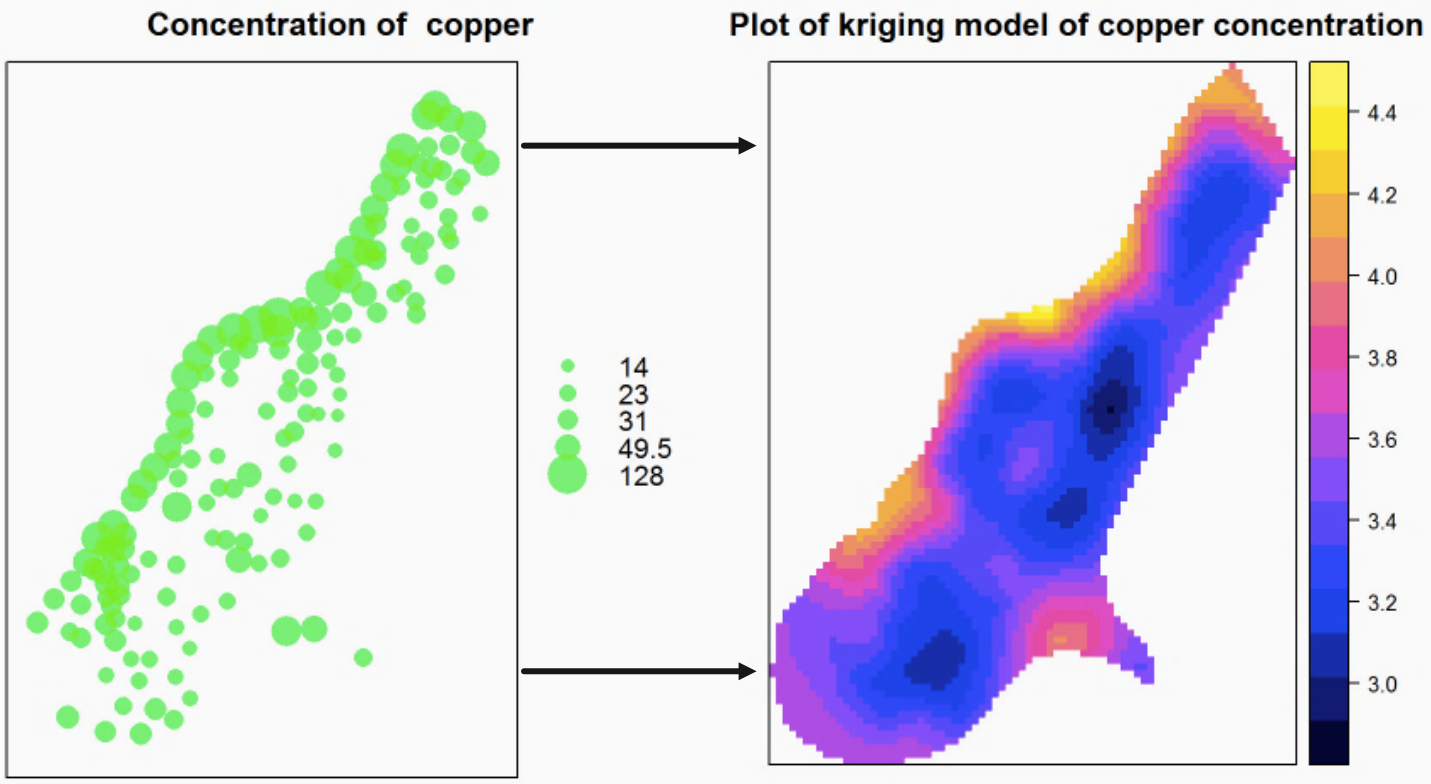
To reiterate, the process of Kriging allows us to compute all necessary components completely:

$$\underset{\boldsymbol{K^{\text{neighbors}}}}{\begin{bmatrix} 1.9 & 1.2 & \cdots & 0.005 \\ 1.2 & 1.9 & \cdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ 0.005 & \cdots & \cdots & 1.9 \end{bmatrix}} \quad \underset{\boldsymbol{K^x}}{\begin{bmatrix} 0.5 \\ 0.6 \\ \vdots \\ 0.2 \end{bmatrix}} \qquad k(x,x) = 1.9 \qquad \mu = \sum_y y^i$$

$$\lambda^x = K^x K^{\text{neighbors}^{-1}}$$

$$\hat{y}(x) = \mu + \sum_{i \text{ neighbours}} \lambda_i^x \cdot (y^i - \mu)$$

$$\hat{\sigma}^2(x) = k(x,x) - \sum_i \lambda_i^x k(x^i, x)$$




To further simplify the equation, the mean $\mu$ can be removed and substitute the vectorized notation:

From:

$$\lambda^x = K^x K^{\text{neighbors}^{-1}}$$

$$\hat{\gamma}(x) = \mu + \sum_{i \text{ neighbours}} \lambda_i^x \cdot (y^i - \mu)$$

$$\hat{\sigma}^2(x) = k(x,x) - \sum_i \lambda_i^x k(x^i, x)$$

To:

$$\lambda^x = K^x K^{\text{neighbors}^{-1}}$$

$$\hat{\gamma}(x) = \sum_{i \text{ neighbours}} \lambda_i^x \cdot y^i = \lambda^x = K^x \cdot K^{\text{neighbors}^{-1}} \cdot y^{\text{neighbors}}$$

$$\hat{\sigma}^2(x) = k(x,x) - K^x \cdot K^{\text{neighbors}^{-1}} \cdot K^{xT}$$

## Kriging (Part III) · Focused on the Gaussian Processes Derivation

Assuming the lables $y^i$ come from a normal distribution $N$ with a mean $\mu = 0$ and a covariance matrix where the top rows represent all of the neighbours with the last row representing the $xy$ pair where a prediction is made.

Maximum Likelihood Interpetration leads to the same math:

$$\begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y \end{bmatrix} \sim N\left(0, \begin{bmatrix} k(x^1,x^1) & k(x^1,x^2) & k(x^1,x^3) & \cdots & k(x,x^1) \\ & k(x^2,x^2) & \cdots & & k(x,x^2) \\ & & \ddots & & \vdots \\ k(x,x^1) & & & k(x,x^n) & k(x,x) \end{bmatrix}\right)$$

Covariance of the neighbors with each other
Covariance of the neighbours with $x$
Variance of $x$ with itself

The above in matrix-vector notation:

$$\begin{bmatrix} y^{\text{neighbors}} \\ \\ y \end{bmatrix} \sim N\left(0, \begin{bmatrix} K^{\text{neighbours}} & K^{xT} \\ K^x & k(x,x) \end{bmatrix}\right)$$

The distribution of $y$ given the value of all the neighboring labels is normal $N$ with the mean and the variance:

likelihood: $p(y|y^{\text{neighbors}})$ is $N(K^x K^{\text{neighbours}^{-1}} y^{\text{neighbors}}, k(x,x) - K^x K^{\text{neighbours}^{-1}} K^{xT})$

$y$ is estimated to be the mean: $\hat{y}(x) = K^x K^{\text{neighbours}^{-1}} y^{\text{neighbors}}$

The Variance of $y$ is estimated: $\text{Var}(\hat{y}) = k(x,x) - K^x K^{\text{neighbours}^{-1}} K^{xT}$

Proof of the Kriging Optimization Process identical to the Gaussian Process Derivation:

From optimization:

Predictions $\hat{y}(x) = \sum_{i \text{ neighbours}} \lambda_i^x \cdot y^i = \lambda^x = K^x \cdot K^{\text{neighbors}^{-1}} \cdot y^{\text{neighbors}}$

Variance of Prediction Error $= \hat{\sigma}^2(x) = k(x,x) - K^x \cdot K^{\text{neighbors}^{-1}} \cdot K^{xT}$

## Notes regarding Kriging · Gaussian Processes · Spatial Regression

- Excellent modeling tool.
- The dimensionality of the input space does not matter; just estimate of covariances $k(x,x)$.
- $k$ can be arbitrarily complicated.
- The technique is easily adapted to use different local means.
- Vanilla Gaussian Processes scale poorly (cubically) in the number of points; it cannot be done for more than a few thousand points without having to use fancier techniques.

# spatial data analysis 🌐 lab

Using **R** to work with spatial data; using **Kriging** to interpolate density values in a spatial data frame.

Using **R** in a Jupyter Notebook:

- Heavy metals pollution data
- Sampled from the Meuse River in Belgium
- The spatial sampling is not performed on a close-spaced regular grid. Therefore, spatial interpolation or Kriging is used to create a map of the expected concentration of metal pollution.
  - Load and examine the properties of these data.
  - Explore the spatial dependency using the Variogram of the metal pollution data.
  - Use Kriging to interpolate and create a map of metal pollution.

The Meuse data is loaded into **R** and the **data.frame** is converted to a **SpatitalPointsDataFrame**.

A spatial grid is required to interpolate the Pollution Data; the **meuse.grid** data frame is loaded.

The **meuse.grid** data frame is converted to a **SpatialPixelsDataFrame** using the function as before:

```
In [7]: summary(meuse.grid)
```

```
Object of class SpatialPixelsDataFrame
Coordinates:
     min    max
x 178440 181560
y 329600 333760
Is projected: TRUE
proj4string :
[+proj=stere +lat_0=52.15616055555555 +lon_0=5.38763888888889
+k=0.999908 +x_0=155000 +y_0=463000 +ellps=bessel +units=m +no_defs
+towgs84=565.2369,50.0087,465.658,
-0.406857330322398,0.350732676542563,-1.8703473836068, 4.0812]
Number of points: 3103
Grid attributes:
  cellcentre.offset cellsize cells.dim
x            178460       40        78
y            329620       40       104
Data attributes:
     part.a           part.b            dist          soil     ffreq
 Min.   :0.0000   Min.   :0.0000   Min.   :0.0000   1:1665   1: 779
 1st Qu.:0.0000   1st Qu.:0.0000   1st Qu.:0.1193   2:1084   2:1335
 Median :0.0000   Median :1.0000   Median :0.2715   3: 354   3: 989
 Mean   :0.3986   Mean   :0.6014   Mean   :0.2971
 3rd Qu.:1.0000   3rd Qu.:1.0000   3rd Qu.:0.4402
 Max.   :1.0000   Max.   :1.0000   Max.   :0.9926
```

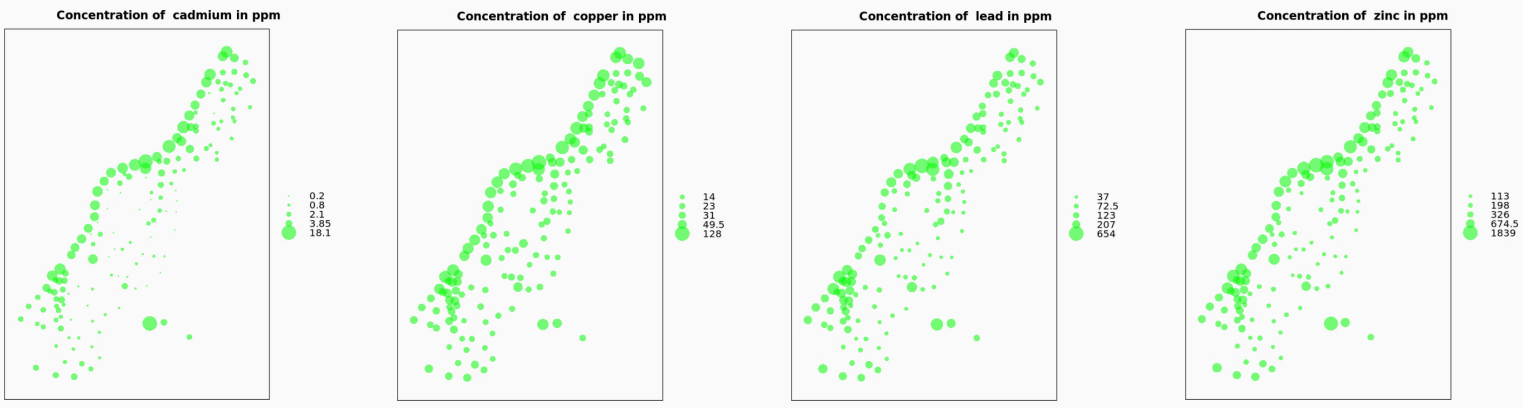
```
In [4]: summary(meuse)
```

```
Object of class SpatialPointsDataFrame
Coordinates:
     min    max
x 178605 181390
y 329714 333611
Is projected: TRUE
proj4string :
[+proj=stere +lat_0=52.15616055555555 +lon_0=5.38763888888889
+k=0.999908 +x_0=155000 +y_0=463000 +ellps=bessel +units=m +no_defs
+towgs84=565.2369,50.0087,465.658,
-0.406857330322398,0.350732676542563,-1.8703473836068, 4.0812]
Number of points: 155
Data attributes:
    cadmium          copper           lead            zinc
 Min.   : 0.200   Min.   : 14.00   Min.   : 37.0   Min.   : 113.0
 1st Qu.: 0.800   1st Qu.: 23.00   1st Qu.: 72.5   1st Qu.: 198.0
 Median : 2.100   Median : 31.00   Median :123.0   Median : 326.0
 Mean   : 3.246   Mean   : 40.32   Mean   :153.4   Mean   : 469.7
 3rd Qu.: 3.850   3rd Qu.: 49.50   3rd Qu.:207.0   3rd Qu.: 674.5
 Max.   :18.100   Max.   :128.00   Max.   :654.0   Max.   :1839.0

      elev             dist              om          ffreq   soil    lime
 Min.   : 5.180   Min.   :0.00000   Min.   : 1.000   1:84   1:97   0:111
 1st Qu.: 7.546   1st Qu.:0.07569   1st Qu.: 5.300   2:48   2:46   1: 44
 Median : 8.180   Median :0.21184   Median : 6.900   3:23   3:12
 Mean   : 8.165   Mean   :0.24002   Mean   : 7.478
 3rd Qu.: 8.955   3rd Qu.:0.36407   3rd Qu.: 9.000
 Max.   :10.520   Max.   :0.88039   Max.   :17.000
                                    NA's   :2
    landuse       dist.m
 W      :50   Min.   :  10.0
 Ah     :39   1st Qu.:  80.0
 Am     :22   Median : 270.0
 Fw     :10   Mean   : 290.3
 Ab     : 8   3rd Qu.: 450.0
 (Other):25   Max.   :1000.0
 NA's   : 1
```

The plots demonstrate the following:

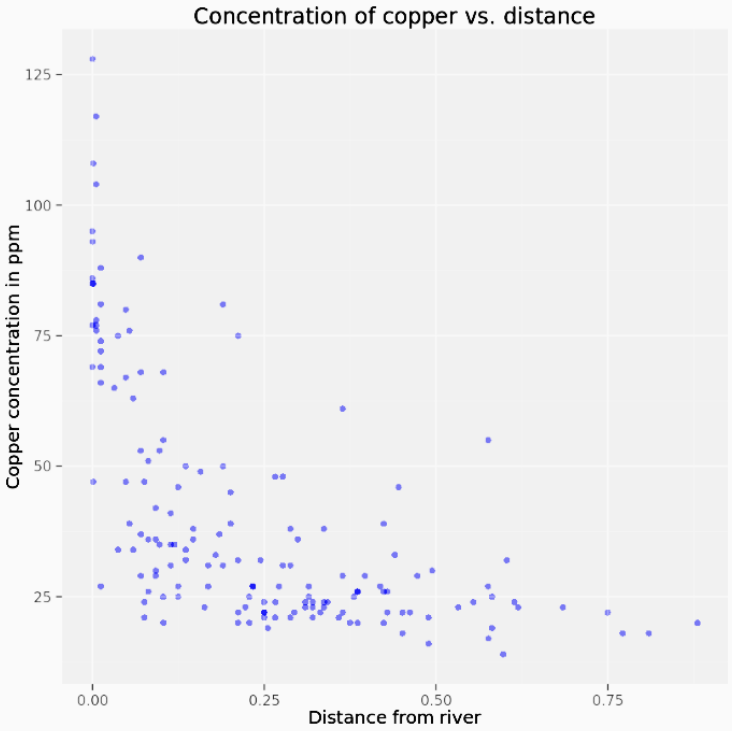
- The distribution of sample points is not uniform in space. The densest sampling occurs along the Meuse River, from the upper right to the lower left of the diagram.
- The distribution of metal concentration is not uniform in space. The highest concentration occurs along the path of the Meuse River. The metals are likely carried by the

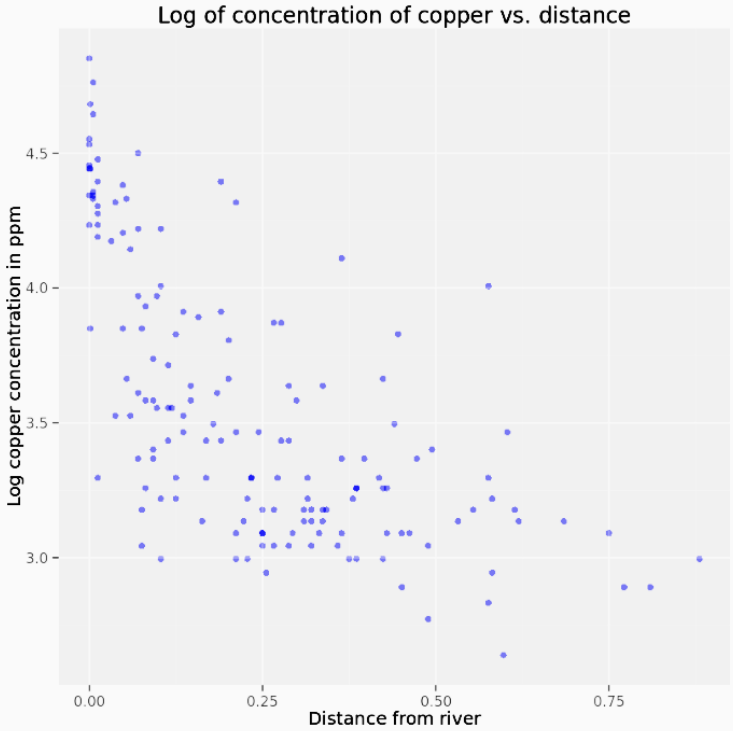
Spatial Maps of the sampled metal coordinates where the bubble size represents a numeric value:



The spatial dependency structure of the metal concentration using **Variogram** methods.

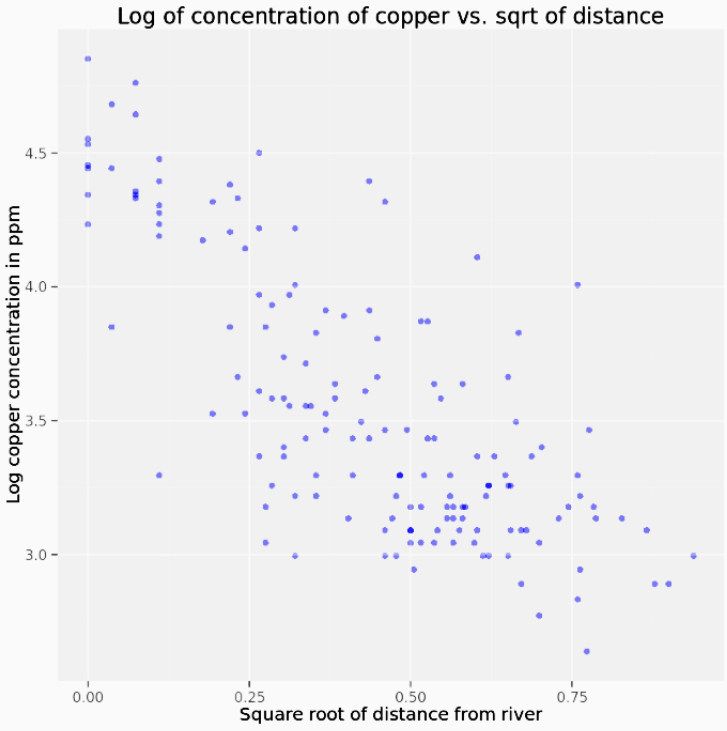
The **dist** column measuring distance from the river is used to compute spatial dependency:

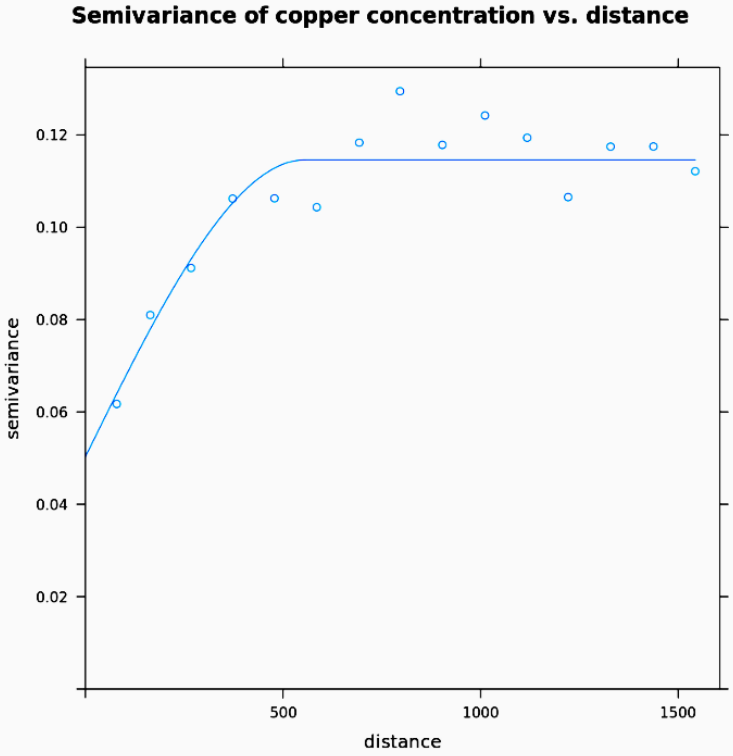



The lack of linearity between copper and distance is more linear after applying logarithmic scale; although still not linear. The scale is further transformed through the log of copper concentration plotted against the squared root of distance from the river; the data is now close to Linear

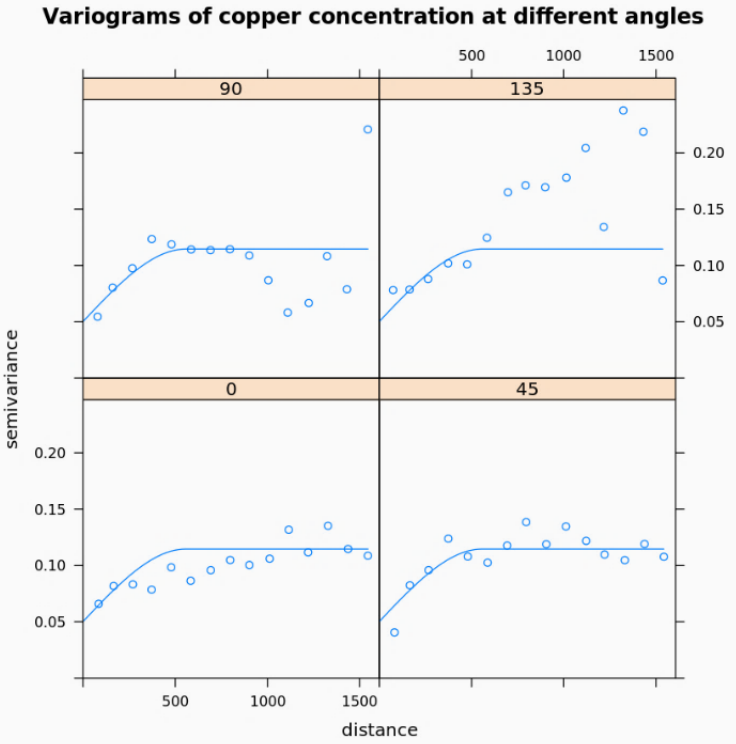
A Variogram can now be properly fitted to the data in the linear diagram

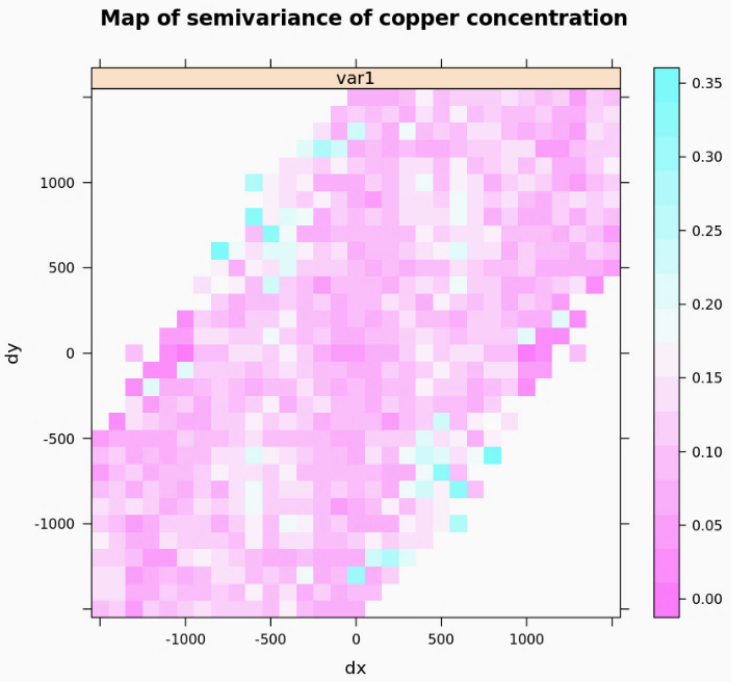



The **semivariance** versus **distance** relationships shows a reasonably good fit.
The **Semivariogram** was computed using the **orthogonal distances** from the river.
However, there can be dependency structures in a spatial diagram in any direction.

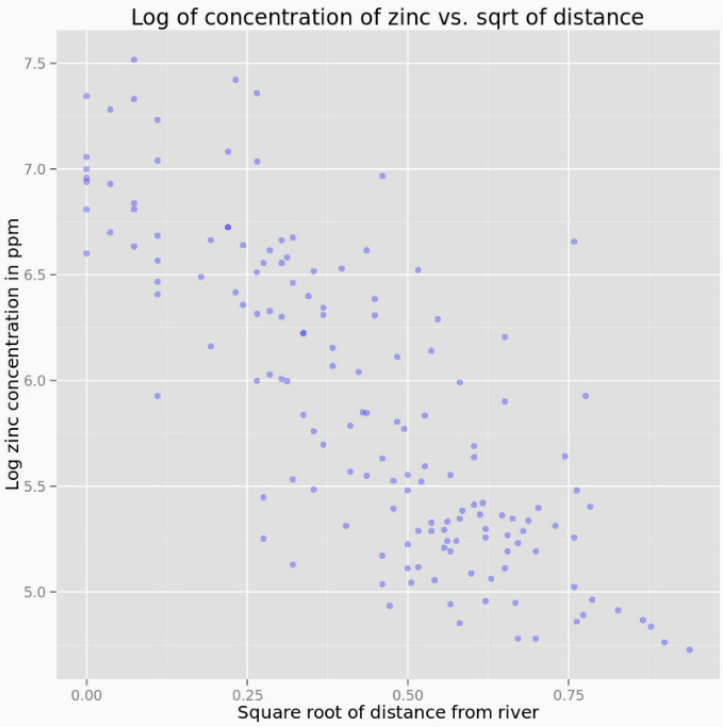
It is noted that the fit remains constant at **0** and **45°** but degrades at **90** and **135°**. The semivariance will proceed to be mapped using the **`map = TRUE`** argument:

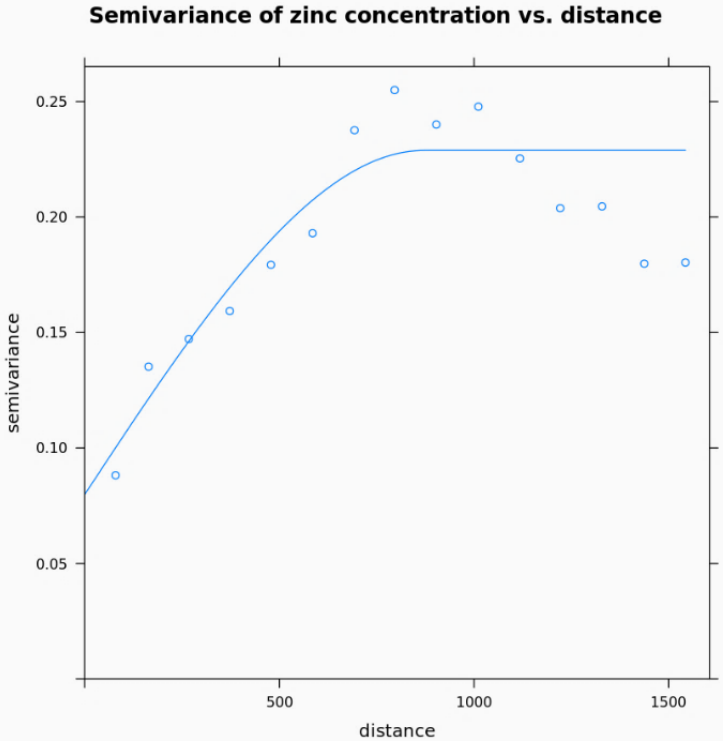



The map illustrates a majority of low semivariance areas with small amounts of high semivariance.

The Variogram analysis is applied equally to the Zinc Concentration of the Meuse River (ex):



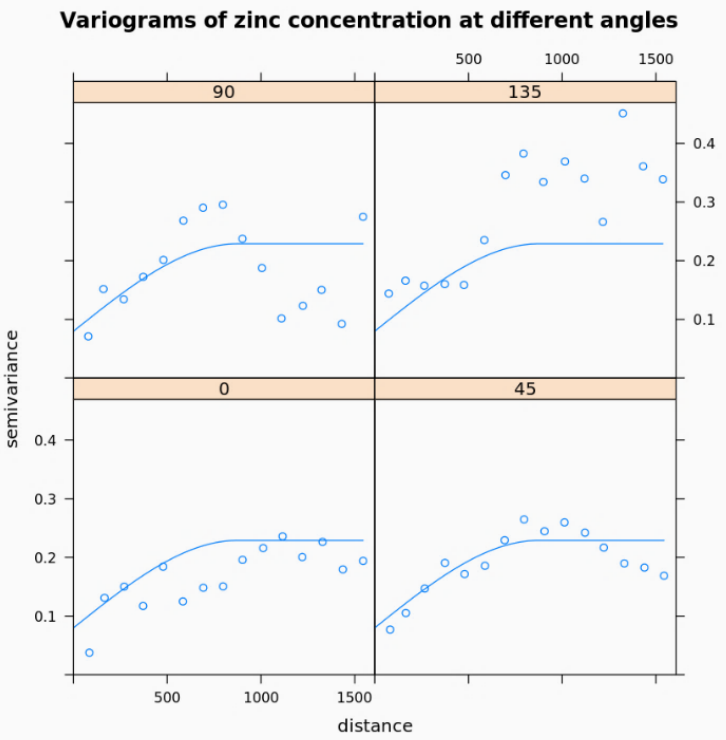



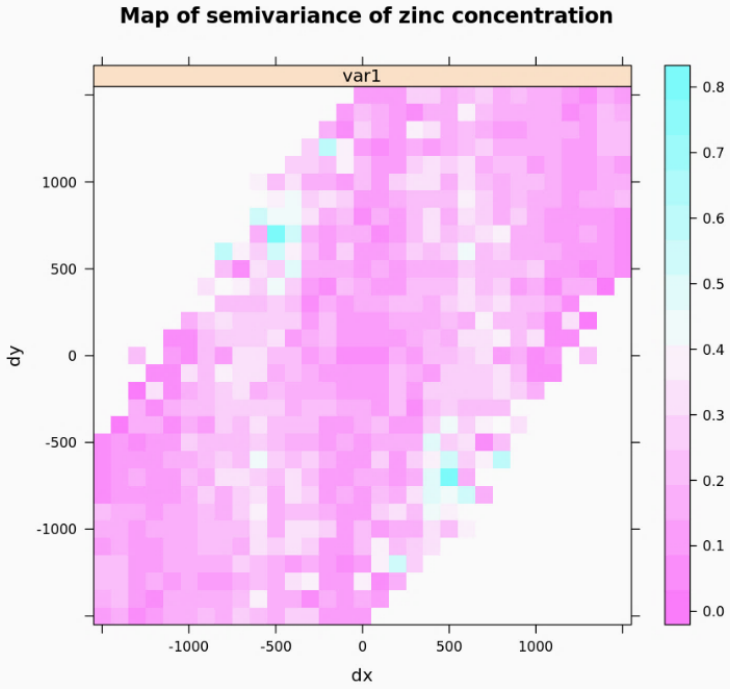

The **semivariance** versus **distance** relationships shows a slightly poorer fit than that of copper.

The **Semivariogram** was computed using the **orthogonal distances** from the river.

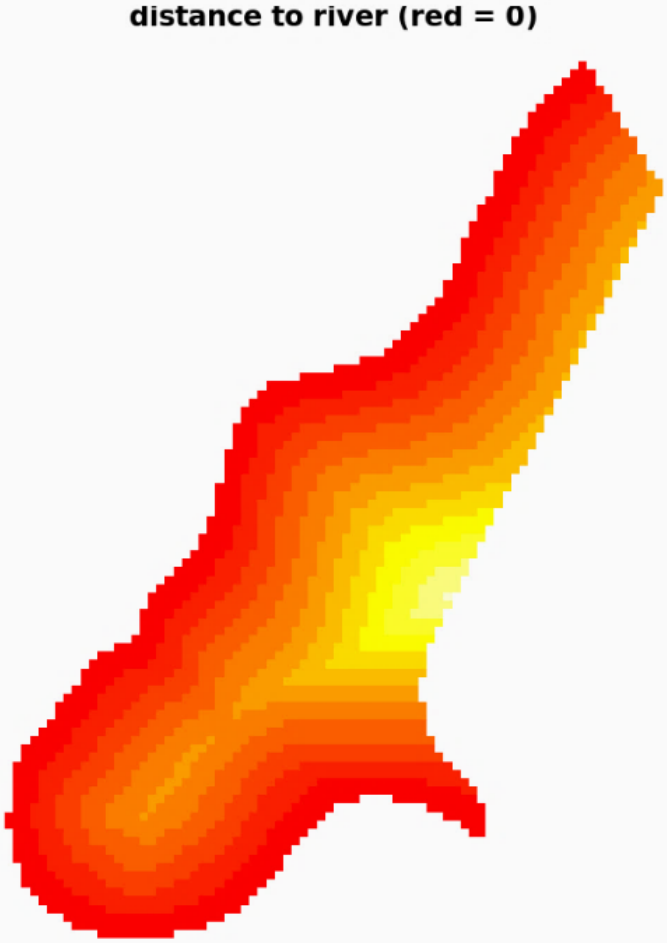
However, there can be dependency structures in a spatial diagram in any direction.

It is noted that the fit remains constant at **0** and **45°** but degrades at **90** and **135°**. The semivariance will proceed to be mapped using the **`map = TRUE`** argument:




# Interpolation of the Copper Concentration



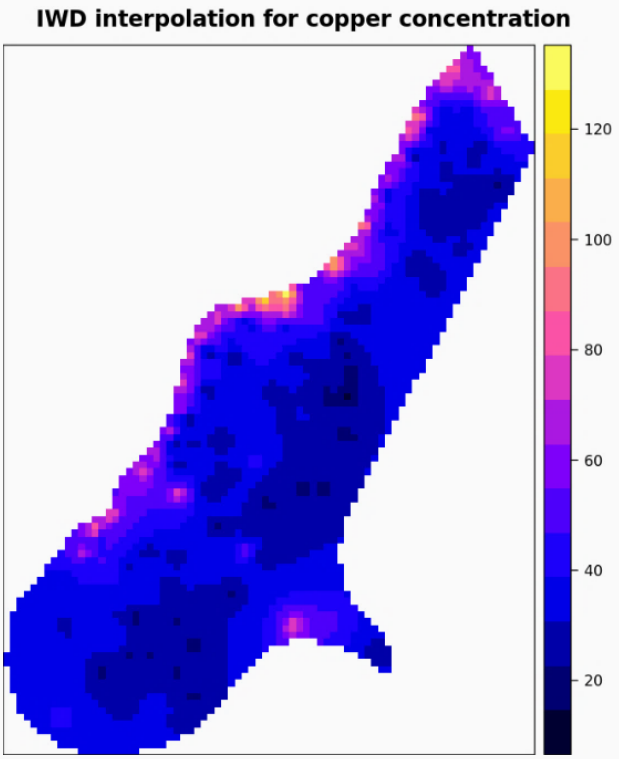

Subsequent to obtaining a valid Variogram, the values can be interpolated. Using the Kriging method, irregularly sampled cooper concentrations can be interpolated to a regular grid.

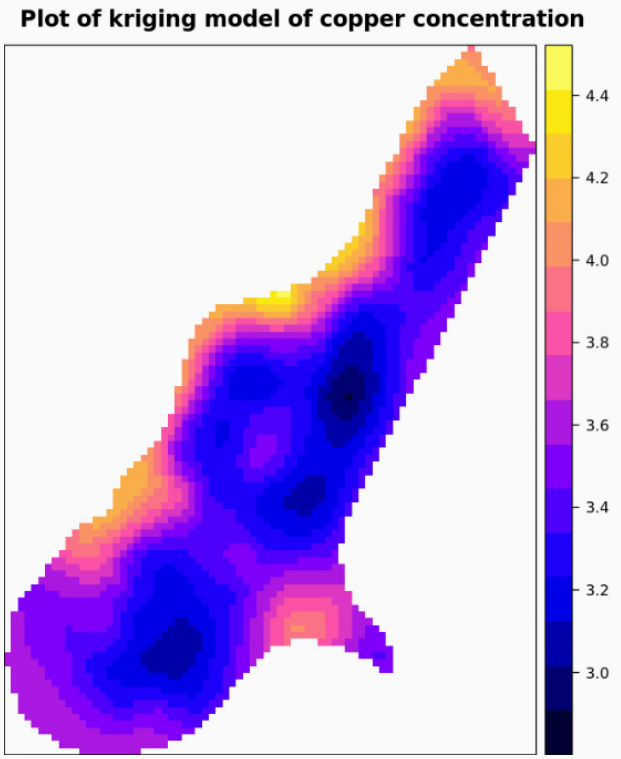
The initial step would be to create a map of the grid used for interpolation (distance from river):

The Meuse River runs from upper right to the lower left of the diagram, curves toward the upper right, and finally toward the lower right. As expected, the lowest distance is near the curve of the river and the greatest distances are near the center (yellow).

Using the distance on the grid the **iwd** function computes the inverse weighted distance of copper concentration. Two new columns are created: **var1.pred** and **var1.var** representing the inverse weighted distance and the variance of computed **IWD**.

The **IWD** is mapped on the following page.



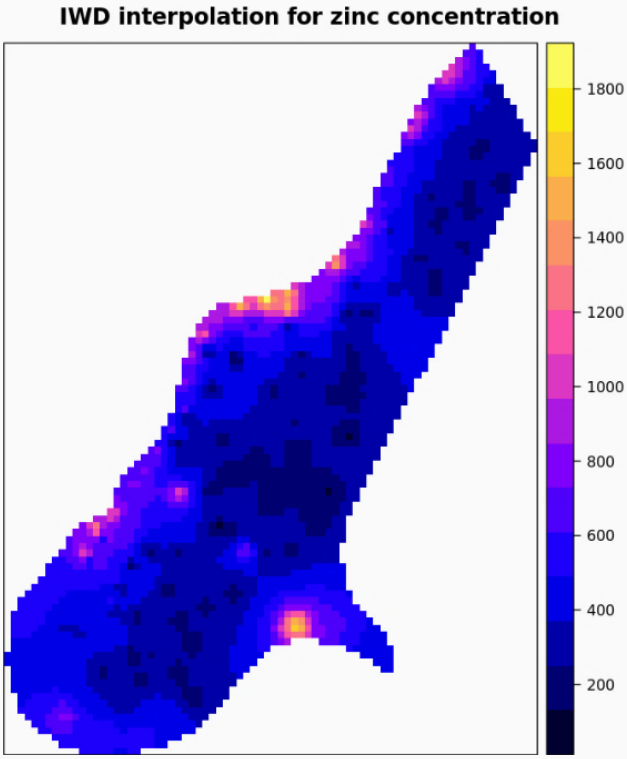



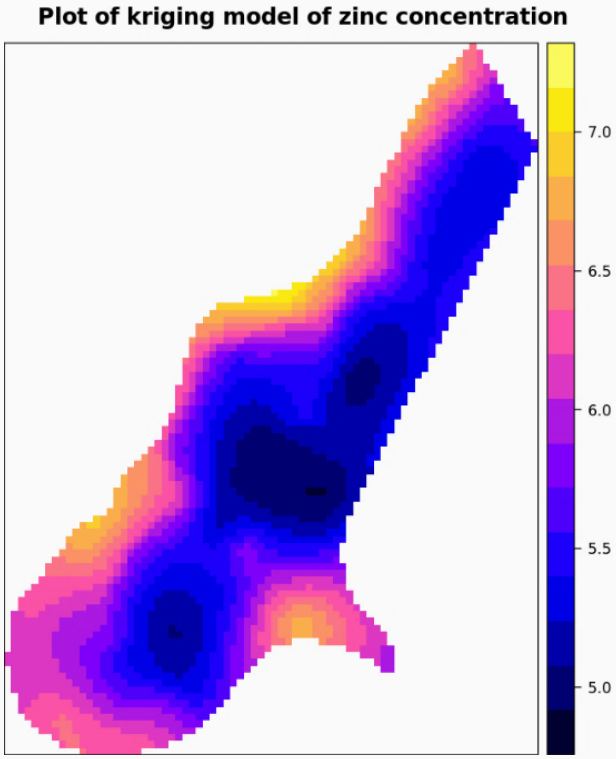

**The Inverse Weighted Distance** of Copper Concentration contains “hotspots” or high values along the river path; with lower concentrations being found further away from the river. Regardless of valuable insights, the graph does not represent an interpolation.

**Kriging** uses the **Spatial Dependency Structure** of the copper Variogram computed earlier.

Examining the map of kriged copper concentration. As expected, the highest concentrations of copper pollution are near the Meuse River. Areas of low concentration are further from the river.

## Interpolation of the Zinc Concentration



# module3 · text analytics

## introduction to 🅃 text analytics

### Introduction

Where text data sources from:

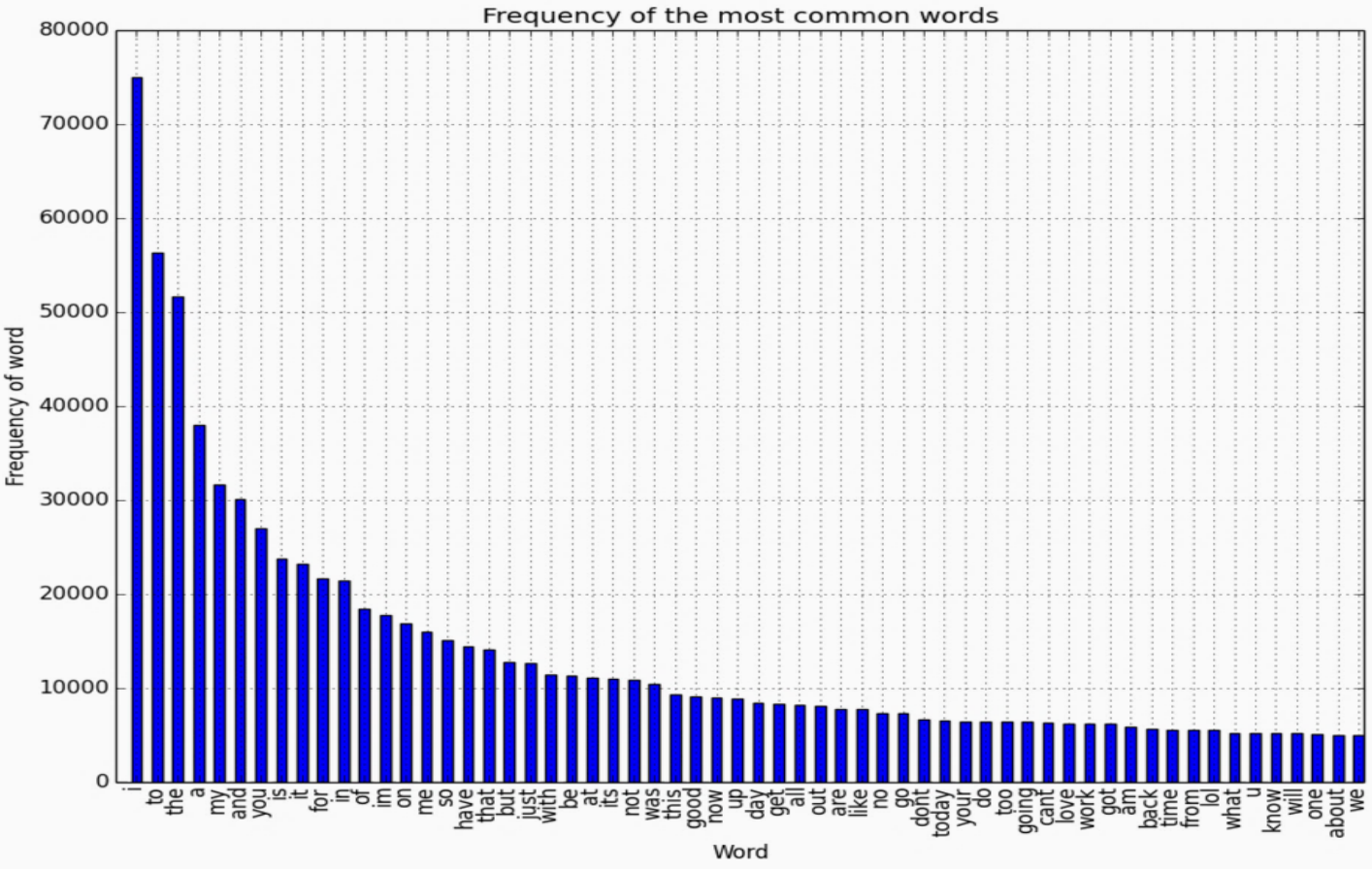
- Almost everywhere on the internet: webpages, social media pages, tweets, messaging, email, product reviews
- News articles, magazine articles
- Books, reference manuals
- Product descriptions, maintenance logs, customer service logs

What can be done with text data:

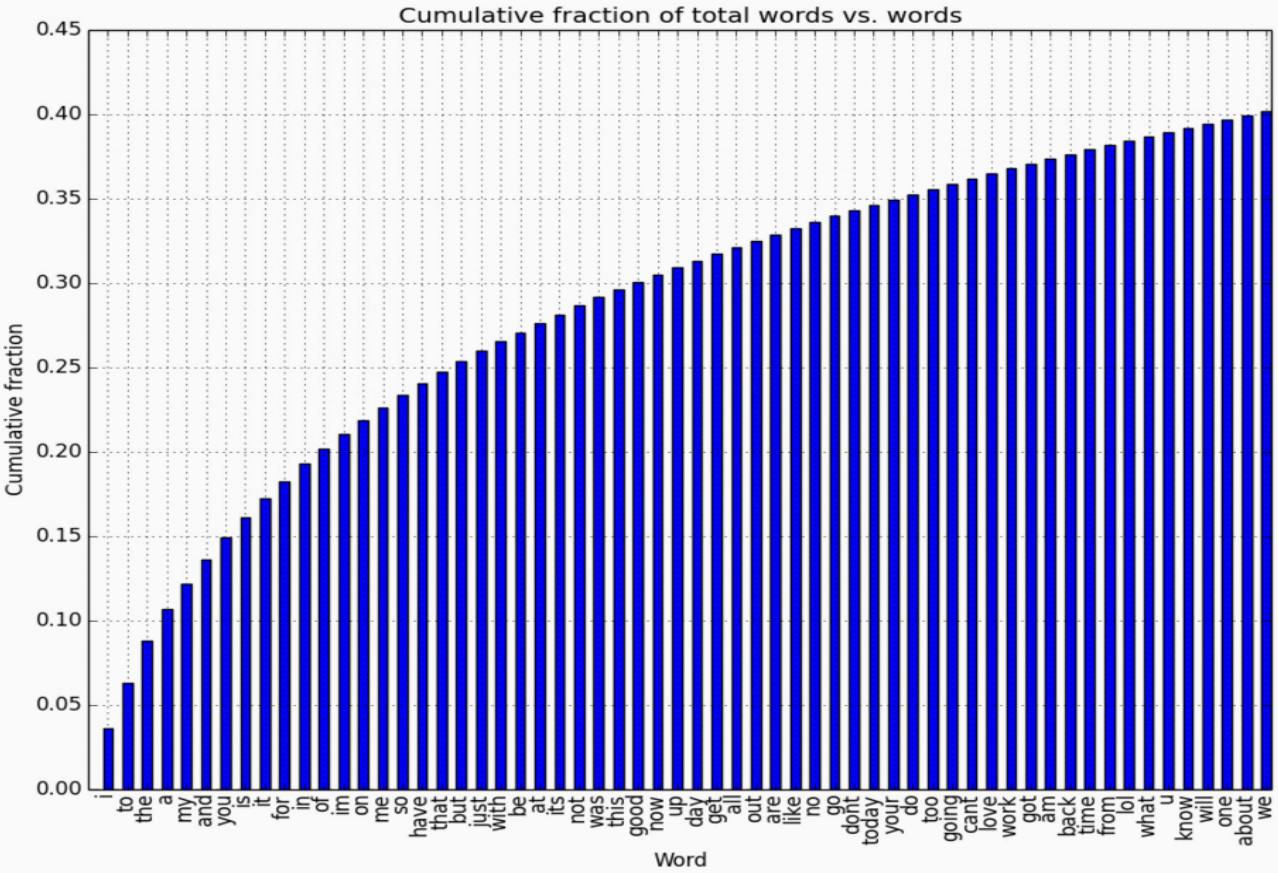
- Summarize it, or represent it in a useful way
- Classify it
- Search for things within it: information retrieval

### Word Frequency

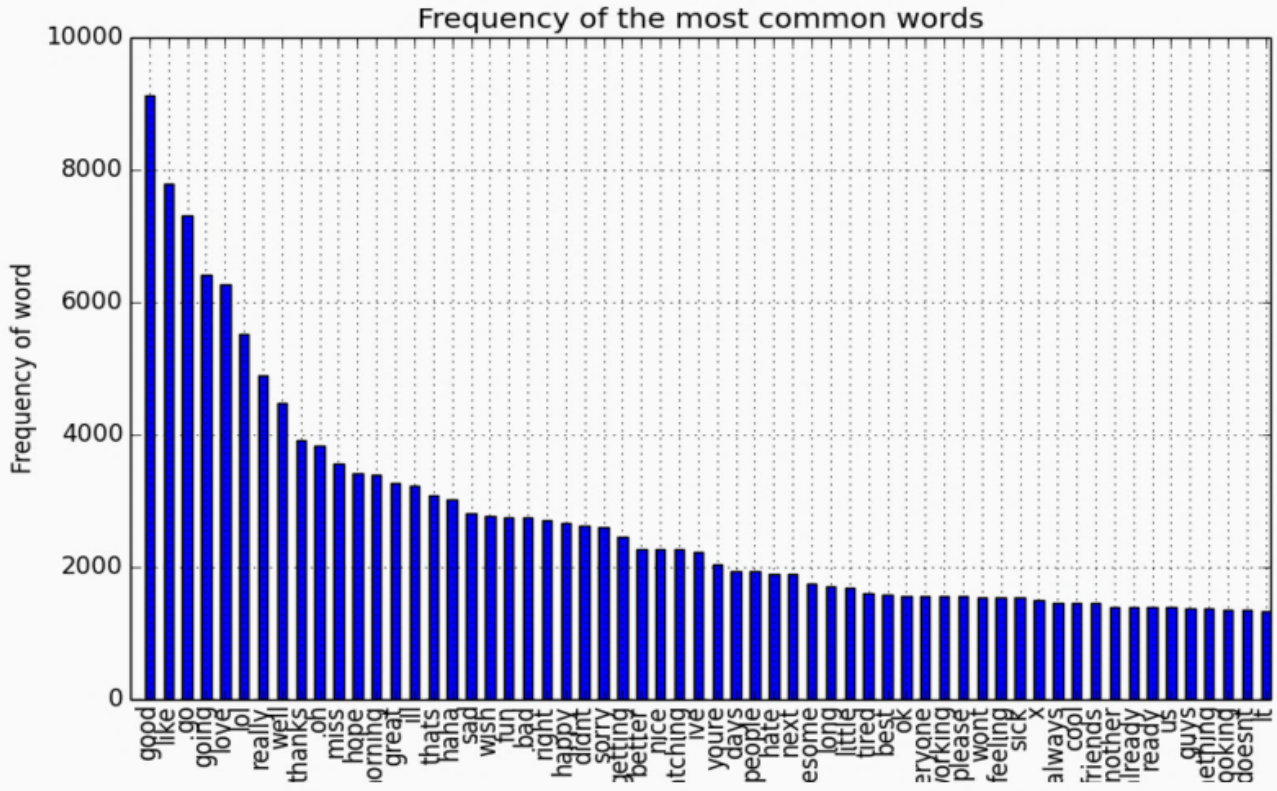
Pareto charts are simple bar charts that have ordered bins amongst the categories:



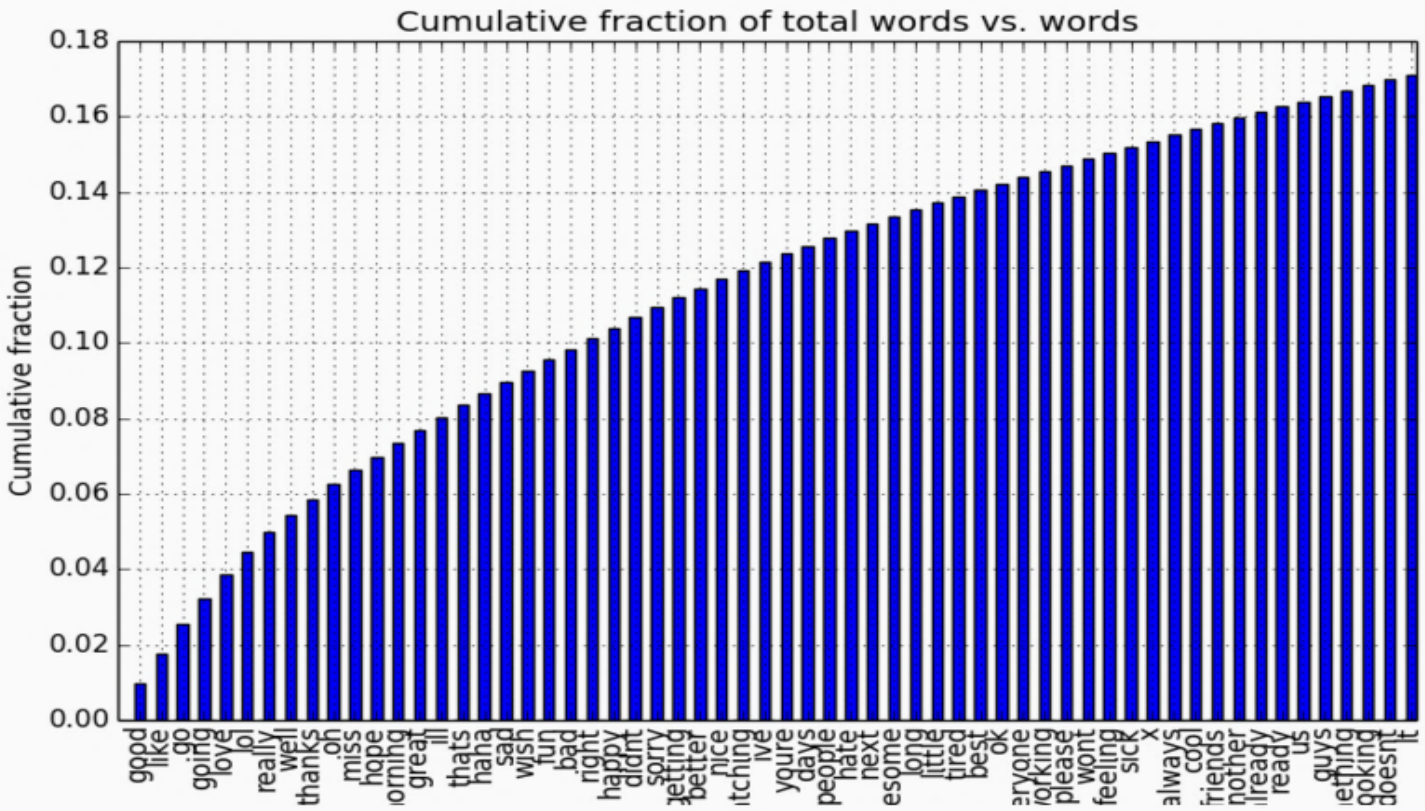

The Pareto chart above illustrates the frequency of most common used works from a Twitter Feed.




A cumulative plot of word frequency demonstrates how common words sum up the total amount of words used. For example, “I” represents ~4% with “I” and “The” representing ~9% cumulatively.

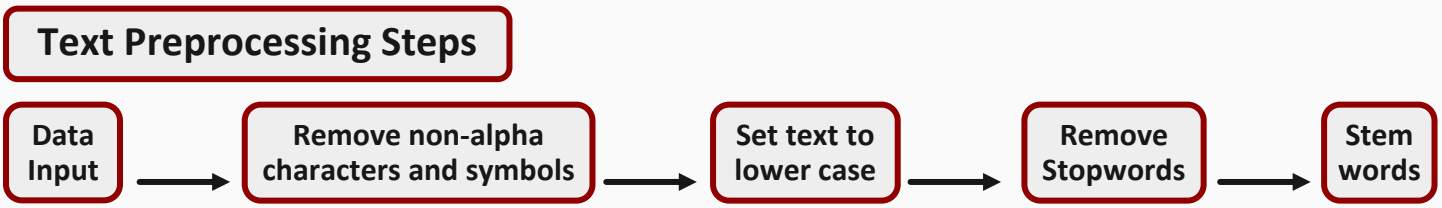



After removing the **stop words** (*the*, *is*, *at*, *which*, that, and *on*). **Stop words** are those that are not very useful for most **natural language processing applications**.




A cumulative Pareto Chart of word frequency after the **stop words** have been removed should words like *good*, *like, love, thanks, and wish* make up the cumulative total of all sampled Twitter words.

When obtaining raw text data, there are certain preprocessing steps to prepare raw data for input:



# stemming

Stemming refers to the reduction of words to their root stems, which is helpful in information retrieval applications. For example:

| | | |
|---|---|---|
| connection, connected, connective, connecting | ➔ | connect |
| train, trains, train's, trains' | ➔ | train |
| the child's trains are connected | ➔ | the child train be connect |

It is important to note that this analysis is entirely limited to English.

**The Xiapian Project - Stemming**

## Porter's Stemming Algorithm (1980)

The algorithm comes from a central idea beginning with breaking each word into a group of vowels (a, e, i, o, u) and consonants (everything else). **V** is a group of vowels that are adjacent and **C** is a group of adjacent consonants:

V is one or more vowels (A, E, I, O, U)

C is one or more consonants

For example, if there are 2 vowels together, they form a **V** group; if there are 2 consonants together, then a **C** group is formed.

The idea behind Porter's algorithm is that all words are of the following form:

**[C]VCVCVC…[V]**

The above demonstrates an optional consonant, then alternating groups of vowels and consonants, followed by an optional vowel group at the end. The above is written alternatively as follows:

**[C](VC){m}[V]** ← with **m** copies of **VC**

The same optional consonant with m copies of alternating groups of vowels and consonants, ending with an optional vowel as before. For example:

m=0 KN, EE, KNEE, Y, BY ← m=0 because there are no V groups followed by C groups.

m=1 BUBBLE, OAKS, KNEES, IVY ←

m=2 BUBBLES, PRIVATE ←

The algorithm works with many rules that are manually scribed. Each rule checks whether a rule obeys a condition and depending on the outcome, the algorithm lengthens or shortens the word:

For each word, check whether it obeys a condition and shorten or lengthen it accordingly.

## Porter's Algorithm, Step 1:

- SSES ➔ SS (e.g., caresses ➔ caress)
- IES ➔ I (e.g., ties ➔ ti, ponies ➔ poni)
- S ➔ <remove> (e.g, potatoes ➔ potato)
- IF m>0 Then EED ➔ EE (e. feed ➔ fee)
- If stem contains vowel and has ED ➔ <remove> (e.g, turned ➔ turn)
- If stem contains vowel and has ING ➔ <remove> (e.g, turning ➔ turn)
- If either of the green rules are successful,
  - AT ➔ ATE (conflat ➔ conflate)
  - BL ➔ BLE (troubl(ed) ➔ trouble)
  - IZ ➔ IZE (siz ➔ size)
- <vowel>Y ➔ I (e.g, happy ➔ happi , whereas sky ➔ sky not ski)

## Porter's Algorithm: Step 2

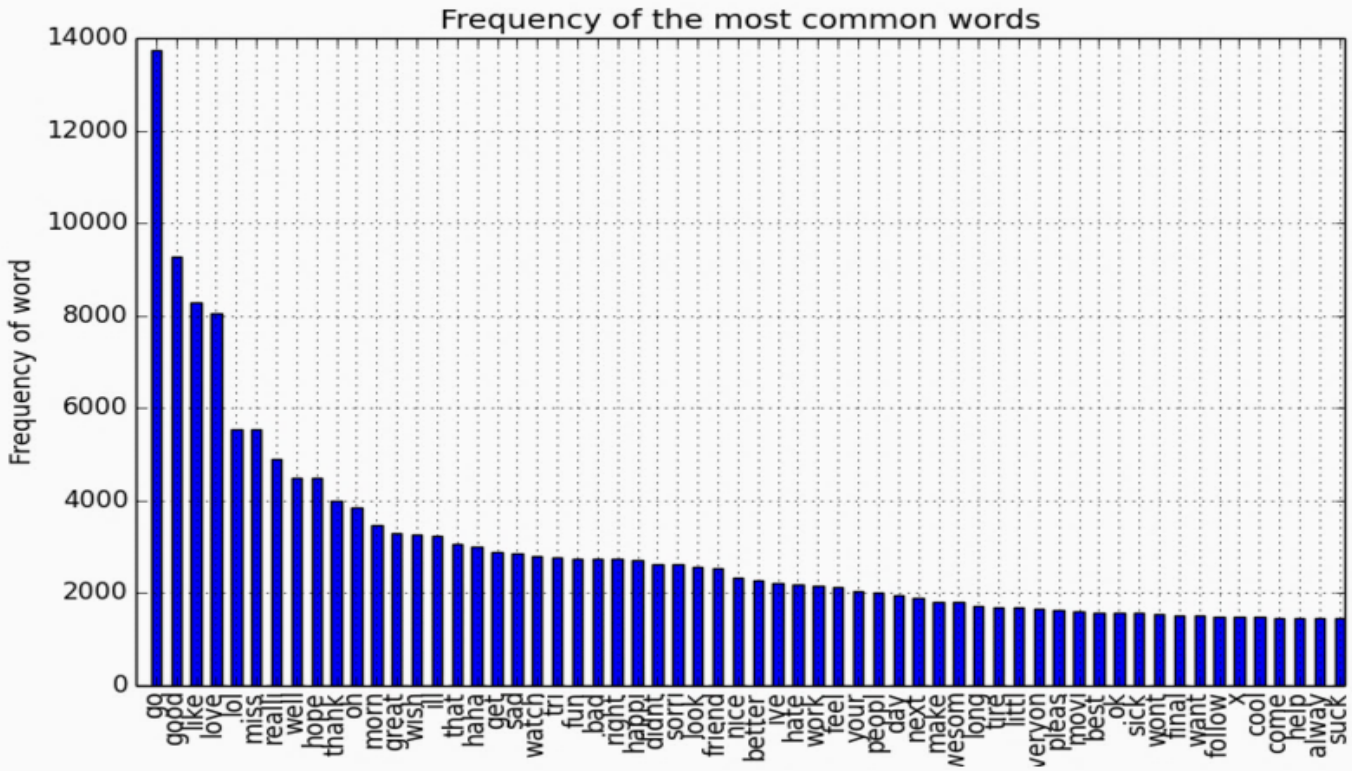
| | | | | | |
|---|---|---|---|---|---|
| (m>0) ATIONAL | ➔ | ATE | relational | ➔ | relate |
| (m>0) TIONAL | ➔ | TION | conditional | ➔ | condition |
| (m>0) ENCI | ➔ | ENCE | valenci | ➔ | valence |
| (m>0) ANCI | ➔ | ANCE | hesitanci | ➔ | hesitance |
| (m>0) IZER | ➔ | IZE | digitizer | ➔ | digitize |
| (m>0) ABLI | ➔ | ABLE | conformabli | ➔ | conformable |
| (m>0) ALLI | ➔ | AL | radicalli | ➔ | radical |
| (m>0) ENTLI | ➔ | ENT | differentli | ➔ | different |
| (m>0) ELI | ➔ | E | vileli | ➔ | vile |
| (m>0) OUSLI | ➔ | OUS | analogousli | ➔ | analogous |
| (m>0) IZATION | ➔ | IZE | vietnamization | ➔ | vietnamize |
| (m>0) ATION | ➔ | ATE | predication | ➔ | predicate |
| (m>0) ATOR | ➔ | ATE | operator | ➔ | operate |
| (m>0) ALISM | ➔ | AL | feudalism | ➔ | feudal |
| (m>0) IVENESS | ➔ | IVE | decisiveness | ➔ | decisive |
| (m>0) FULNESS | ➔ | FUL | hopefulness | ➔ | hopeful |
| (m>0) OUSNESS | ➔ | OUS | callousness | ➔ | callous |
| (m>0) ALITI | ➔ | AL | formaliti | ➔ | formal |
| (m>0) IVITI | ➔ | IVE | sensitiviti | ➔ | sensitive |
| (m>0) BILITI | ➔ | BLE | sensibiliti | ➔ | sensible |

Steps 3, 4, and 5 are similar sets of rules as illustrated above.
Porter's Algorithm is not perfect but often improves information retrieval performance.

The following are examples are errors the Porter's Algorithm is known for:

- severing vs. several ➔ sever
- university vs. universe ➔ univers
- iron vs. ironic ➔ iron

## Most Popular Words




After the raw text data has been preprocessed to remove stopwords, non-alpha characters, converted text to lowercase, removed white spaces, and finally stemmed (etc.), a Pareto chart will plot the result.

# working with text

## Calculating Word Importance

**TF-IDF** answers the question: How important is each word? (or how important is each word stem).

**TF-IDF** is a key factor utilized in search engines.

**TF** is **Term Frequency**; the number of times the word occurs.

- It is logical to draw correlation between frequency and importance.
- However, this is not the case → "The" is a very frequent word, but holds little importance.
- Term frequency cannot be used on its own to determine word importance

**IDF** is **Inverse Document Frequency**:

- log(1/(fraction of documents the word appears in))

**TF_IDF = Term Frequency X Inverse Document Frequency**

Therefore, when is TF_IDF **high**?

- When the term appears **many** times in **few** documents

When is TF_IDF **low**?

- When the term appears in almost all documents
- When the term does not appear often

$$\text{TF} \cdot \log\left(\frac{\#Documents}{\#Documents\ the\ Word\ Appears}\right)$$

## Natural Language Processing

There are many subfields in NLP, some major ones are discussed as follows:

### Named Entity Recognition

The goal of NER is to label words that are names of things:

people, organizations, locations, gene proteins, etc.

For example, take the following sentence:

" Cynthia and Steve worked for Duke and Quantia Analytics in 2016 in Niagara .

**Person**
**Organization**
**Location**

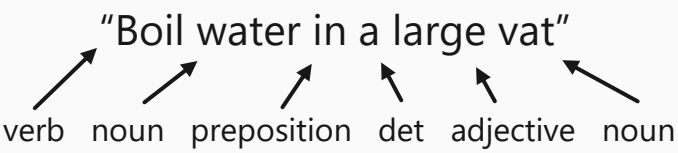
NER analysis is useful for answering questions such as:
What are all organizations mentioned in a set of legal documents?
What places were involved in articles about a military group?

## Part of Speech Tagging

Another subfield of NLP is Part of Speech Tagging, where the goal is to assign each word in a sentence to a part of speech:

noun, adjective, verb, adverb

For example, take the following sentence:




The technique is difficult in applying to machine learning because of the double entendre in English:

For example, *water* is a **noun** in the sentence above but could also be used as **verb** as follows:

> "**Water** the plant"

The importance behind this distinction is illustrated while building a speech synthesizer that needs to sound natural:

When text becomes speech, the noun version of a word is sometimes pronounced differently than the verb version.

Take for example:

> "I **object** to your taking this class"
>
> "What is that **object** you have in your hand?"

Probabilistic machine learning models are used to do the tagging:

> Sentence $W = w_1, w_2, w_3, \dots, w_n$ "Boil water in a large vat"
>
> Assign a sequence of tags $T = t_1, t_2, t_3, \dots, t_n$

Choose $T$ to maximize $P(T|W)$: Maximize the probability that the words arise from a tag.

# working with text

## Parsing

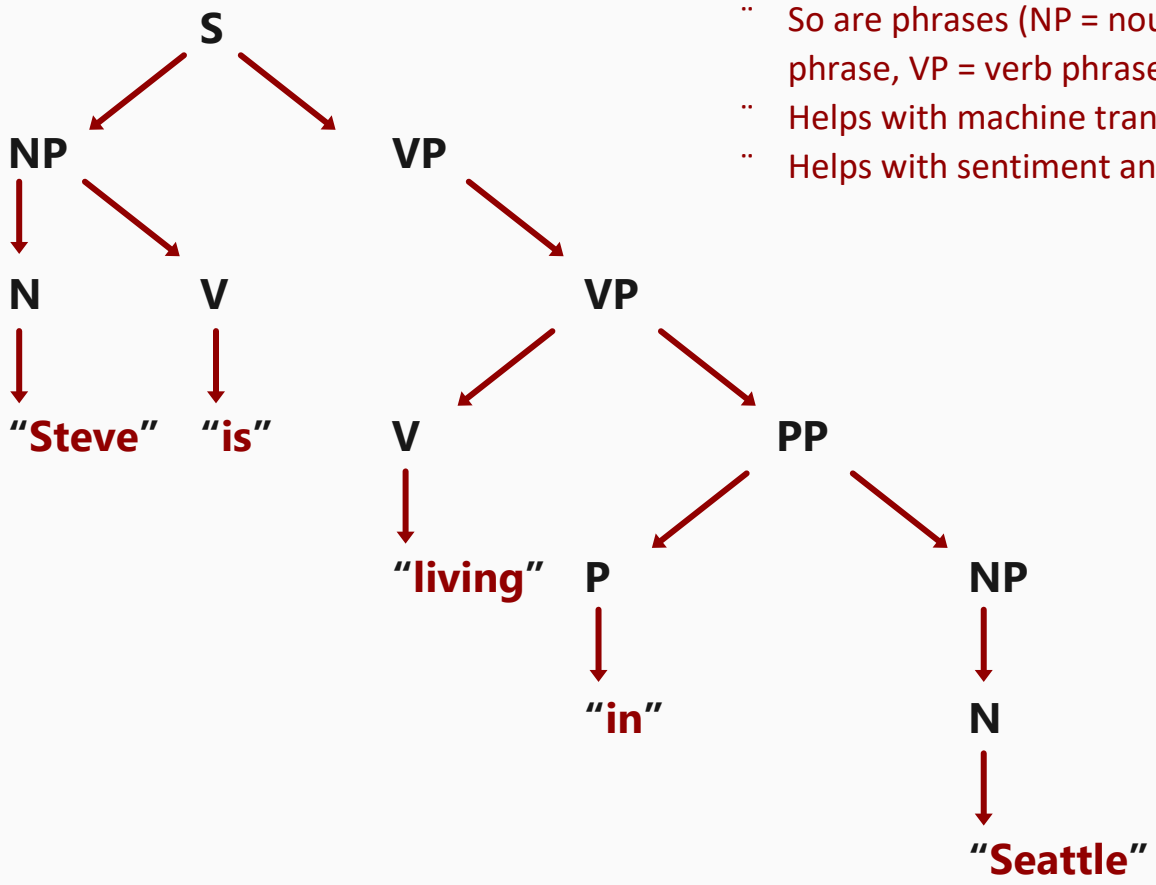
Parsing is another subfield of Natural Language Processing (NLP)

The goal in Parsing is to produce a **Parse Tree**, which provides a structure to the sentence.

A **Parse Tree** includes the information about parts of speech:

### "Steve is living in Seattle"

- Parts of Speech are included
- So are phrases (NP = noun phrase, VP = verb phrase)
- Helps with machine translation
- Helps with sentiment analysis




The above **Parse Tree** contains phrases in the form of:

- noun phases → "Steve"
- verb phrases → "living in Seattle"

Parsing is helpful in machine translation between languages by helping in determining order.
Parsing also helps with **Sentiment Analysis** by adding context to the words in a sentence.

# Sentiment Analysis

**Sentiment Analysis** is another subfield of NLP attributing positive and negative attributes to examples. Take the following sentence for example:

- "Why don't customers like our product?"

Tweets utilizes words such as "slow, hideous, lousy, awful, and wasteful"
Competitors' Tweets use words such as: "awesome, best, cool, fun, loved it"
Although seemingly obvious, **Sentiment Analysis** is difficult to apply in practice:

5 stars: "This product is a crazy idea. I can't believe I'm actually writing a review of it, I originally thought the idea was terrible, but it's actually very nice. It allows you to blow lots of bubbles really slowly, so the whole yard is filled with bubbles of all sizes. I ended up totally going for it!

1 star: "I tried this and didn't have much luck getting it to work. Yesterday I bought a different bubble machine, and it was great, the bubbles came out really fast, and the setup was easy and fun. I loved that one, so I don't recommend purchasing this."

Various usage of positive and negative words could contradict the actual sentiment as seen above.

**Sentiment Analysis** can use **supervised learning** if a labeled corpus of text is available to learn from.

A classification of positive and negative sentiment might compose a feature such as:

- Average sentiment of the words in the full review.

  "I tried this a couple of times and really liked it. The setup was difficult and annoying, but when I finally got it assembled the bubbles were totally awesome. I would definitely recommend this product."

- Average sentiment of single words that reference the product.

  "The bubble blower is so awesome…"

  "…nevermind that. Let's discuss the bubble blower, which has worked out great so far…"

**Parse Trees** can handle the features illustrated above.

# text analytics lab

The lab uses R programming to work with text data. Programs are coded to clean text, remove stopwords, and apply Porter Stemming to the remaining words. An Azure ML web service will be deployed to classify Tweets based on Sentiment analysis.

The lab works with a set of 160,000 tweets, which include sentiment labels.

Social media sentiment is an important indicator of public opinion. Determining sentiment can be valuable in a number of applications including brand awareness, product launches, and detecting political trends.

Raw text is inherently messy. Machine understanding and analysis is inhibited by the presence of extraneous symbols and words that clutter the text. The exact nature of the required text cleaning depends on the application. In this case, the lab focuses on text cleaning to facilitate sentiment classification. The presence of certain words determines the sentiment of the tweet. Words and symbols which are extraneous to this purpose are distractions at best, and a likely source of noise in the analysis. The following steps prepare the tweet text for analysis:

- Symbols and unnecessary white space which do not convey sentiment are removed, leaving only alphabetic characters.
- There is no difference in the sentiment conveyed by a word in upper case or lower case, so all case is set to lower.
- Stop words are words that occur with high frequency in text, but do not have any particular meaning. Examples include word like “the”, “and”, and “this”. Since these words are relatively common, yet communicate no particular sentiment, they can bias analytics. Therefore, stopwords which do not convey sentiment are therefore removed from the tweet text.
- A stem is a root word. For example, “go” is the root word of conjugated verbs, “going”, “goes”, and “gone”. The meaning of these words is the same in terms of analysis. A process known as stemming is applied to transform words to their roots, before analysis.

The **tweets.csv** and **stopwords.csv** files are loaded into the Azure Machine Learning Studio.

## Load and transform the tweet data

The data is then loaded and column names are set to convenient values in R in a Jupyter Notebook.

| | sentiment | tweets |
|---|---|---|
| 1 | 4 | @elephantbird Hey dear, Happy Friday to You Already had your rice's bowl for lunch ? |
| 2 | 4 | Ughhh layin downnnn Waiting for zeina to cook breakfast |
| 3 | 0 | @greeniebach I reckon he'll play, even if he's not 100%...but i know nothing!! ;) It won't be the same without him. |
| 4 | 0 | @vaLewee I know! Saw it on the news! |
| 5 | 0 | very sad that http://www.fabchannel.com/ has closed down. One of the few web services that I've used for over 5 years |
| 6 | 0 | @Fearnecotton who sings 'I Remember'? i alwaysss hear it on Radio 1 but never catch the artist |

- The **Sentiment** column contains a sentiment score {0,4} for **-/+** sentiment of the tweet.
- The **Tweets** column contains the actual text of the tweet.

In order to work with the text in the tweets using the tools in the R tm package, they must be converted to a **corpus object**. A **tm vector corpus** is a vector of corpus objects. In this case, the text of each tweet is a single corpus.

## Clean and Lower Case Symbols with R

The R Text Mining package **tm** is used to perform some basic filtering and cleaning of the tweet text

- Remove numbers.
- Remove punctuation.
- Remove excess white space.
- Convert to lower case.

The code uses the **tm_map** function to apply the specified transformation to the text.

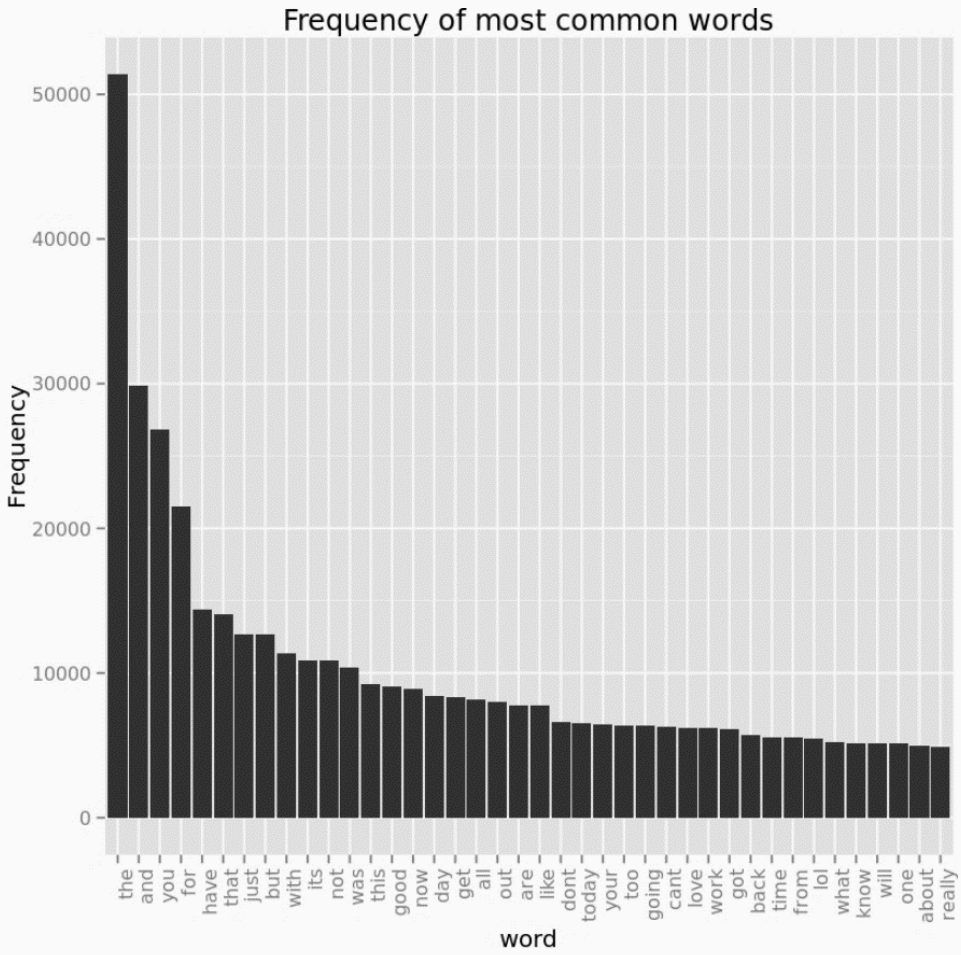
A **term-document matrix (TDM)** of the tweets is a sparse matrix structure with the words (terms) in the rows and documents which may or may not contain that word in the columns. The count of word occurrence of the document is contained in the cells.

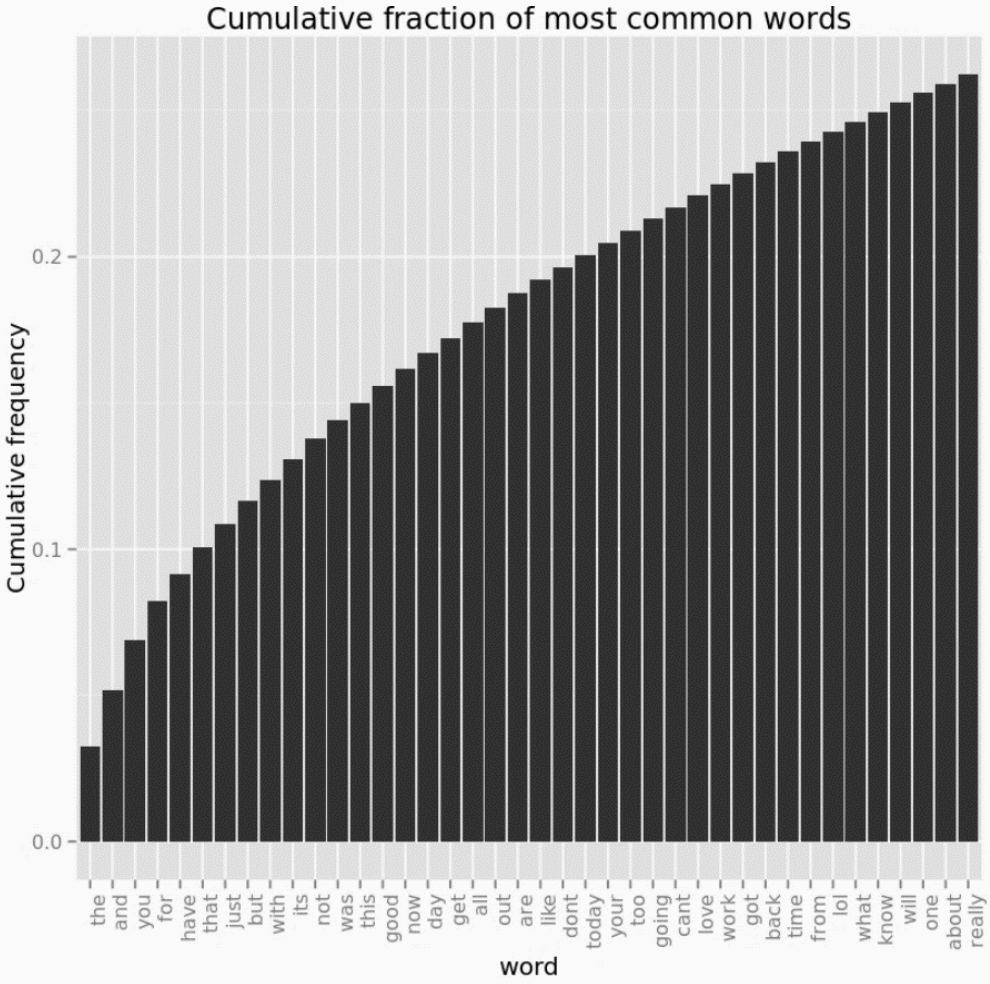
The frequency of words is simply computed by summing the values in the rows. Note that the **row_sums** function from the **slam** package is used to deal with the sparse matrix structure. A **dataframe** is then constructed with the words and their frequencies in descending order.

| | word | freq | Cum |
|---|---|---|---|
| **the** | the | 51366 | 0.03266961 |
| **and** | and | 29900 | 0.05168649 |
| **you** | you | 26847 | 0.06876162 |
| **for** | for | 21508 | 0.08244106 |
| **have** | have | 14357 | 0.09157234 |
| **that** | that | 14059 | 0.1005141 |
| **just** | just | 12674 | 0.108575 |
| **but** | but | 12651 | 0.1166212 |
| **with** | with | 11401 | 0.1238724 |
| **its** | its | 10912 | 0.1308126 |
| **not** | not | 10835 | 0.1377039 |
| **was** | was | 10348 | 0.1442854 |
| **this** | this | 9234 | 0.1501583 |
| **good** | good | 9076 | 0.1559308 |
| **now** | now | 8935 | 0.1616136 |
| **day** | day | 8416 | 0.1669663 |
| **get** | get | 8296 | 0.1722427 |
| **all** | all | 8148 | 0.177425 |
| **out** | out | 8013 | 0.1825214 |
| **are** | are | 7777 | 0.1874677 |

Notice that the most frequent words are in the head of this data frame. Of these 20 most frequent words, only one, 'good', is likely to convey much information on sentiment.

Many of the most frequent words are stop words, such as “the”, “and”, and “you”, which are not likely to be helpful in determining sentiment. Also, the frequency of the words drops off fairly quickly to less than 500 out of the 160,000 tweets.




Another tool for examining the frequency of words in a corpus of documents is the **cumulative distribution frequency (CDF)** plot:

The conclusions one can draw from the second chart are largely the same as the first. The most frequent words are stop words and the frequency of words drops off rather quickly. Also notice, that the frequency of the words becomes uniform fairly quickly.

Finally, the normalized text in the processed tweets is illustrated as follows:

| | sentiment | text |
|---|---|---|
| **tweets.content1** | 4 | elephantbird hey dear happy friday to you already had your rices bowl for lunch |
| **tweets.content2** | 4 | ughhh layin downnnn waiting for zeina to cook breakfast |
| **tweets.content3** | 0 | greeniebach i reckon hell play even if hes not but i know nothing it wont be the same without him |
| **tweets.content4** | 0 | valewee i know saw it on the news |
| **tweets.content5** | 0 | very sad that httpwwwfabchannelcom has closed down one of the few web services that ive used for over years |
| **tweets.content6** | 0 | fearnecotton who sings i remember i alwaysss hear it on radio but never catch the artist |

All text is lower case and there are no numbers, punctuation or special characters.

Examine the head of the resulting word frequency data frame to determine the following:

- What is the percentage of all words for these first 20 words?
- Of these 20 words, how many are likely to contribute sentiment information?
- Are these 20 words different from the words seen for the raw text?

## Remove Stopwords

The extraneous characters and whitespace have been removed from the tweet text. The results show that the most frequent words do not communicate much sentiment information. These frequent words, which are largely extraneous, are known as stop words and should be removed from the text before further analysis:

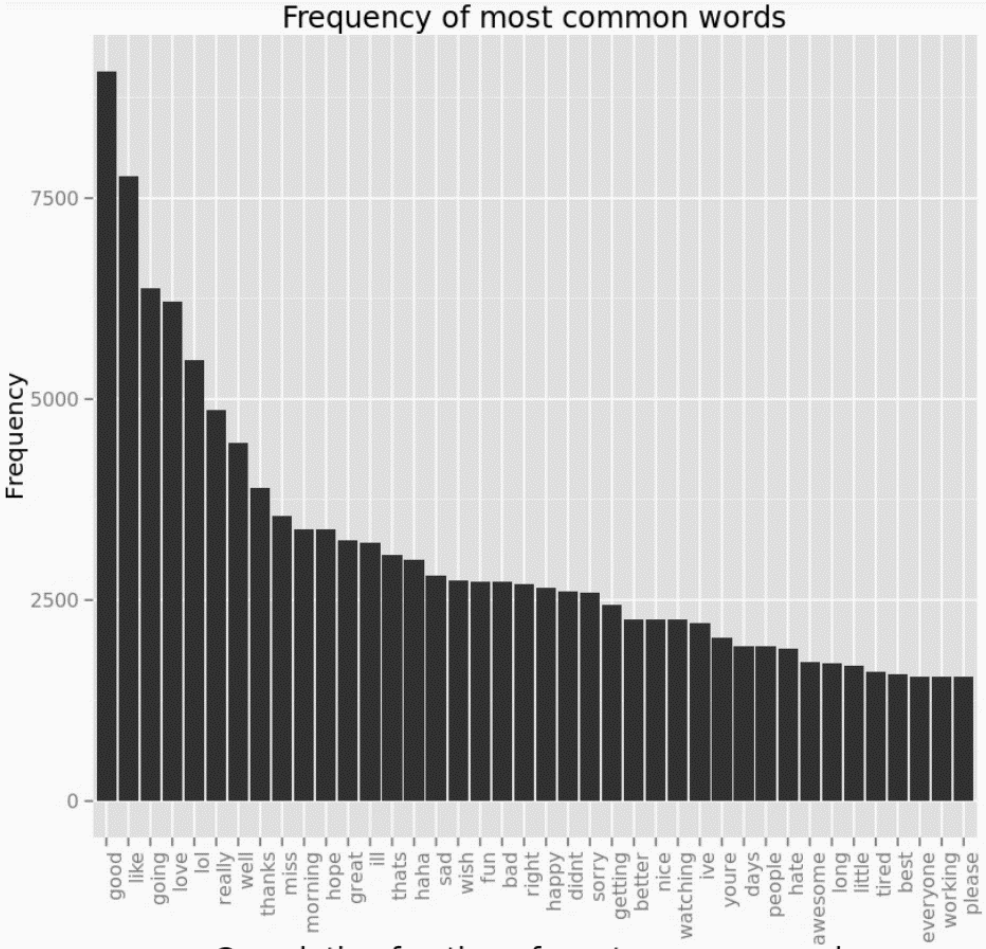
"a" "about" "above" "actual" "after" "again" "against" "all" "alreadi" "also" "alway" "am" "amp" "an" "and" "ani" "anoth" "any" "anyth" "are" "aren't" "around" "as" "at" "aww" "babi" "back" "be" "becaus" "because" "bed" "been" "befor" "before" "being" "below" "between" "birthday" "bit" "book" "both" "boy" "but" "by" "call" "can" "can't" "cannot" "cant" "car" "check" "com" "come" "could" "couldn't" "day" "did" "didn" "didn't" "dinner" "do" "doe" "does" "doesn" "doesn't" "doing" "don" "don't" "done" "dont" "down" "during" "each" "eat" "end" "even" "ever" "everyon" "exam" "famili" "feel" "few" "final" "find" "first" "follow" "for" "for." "found" "friday" "from" "further" "game" "get" "girl" "give" "gone" "gonna" "got" "gotta"

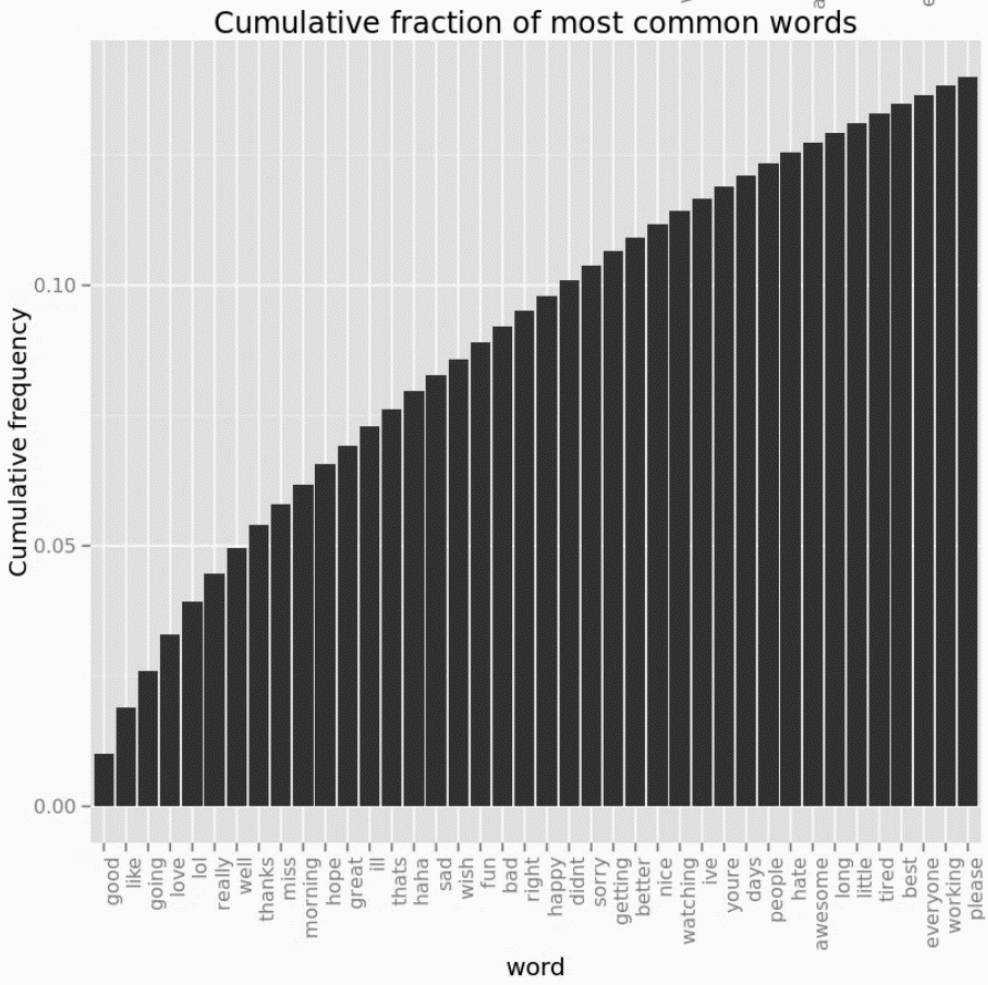
- These words are generally common in English language text.
- None of these words seem likely to indicate any particular sentiment.
- Some of these words, like 'aww', are specialized to this application of analyzing tweets.

The next code applies the **removeWords** operation to the tweet text:

The distribution of word frequency is not quite different. Note that many of the most frequent words are now likely to convey some sentiment, such as “good”, “like”, and “love”. Evidently, removing stop words has had the desired effect.

The **CDF** of the tweets with the stopwords removed is illustrated on the following page:




As before, this chart shows a number of frequent words which are likely to convey sentiment. However, note that these 40 most frequent words only make up about 15% or the total.

The head of the resulting word frequency data frame to determine is as follows:

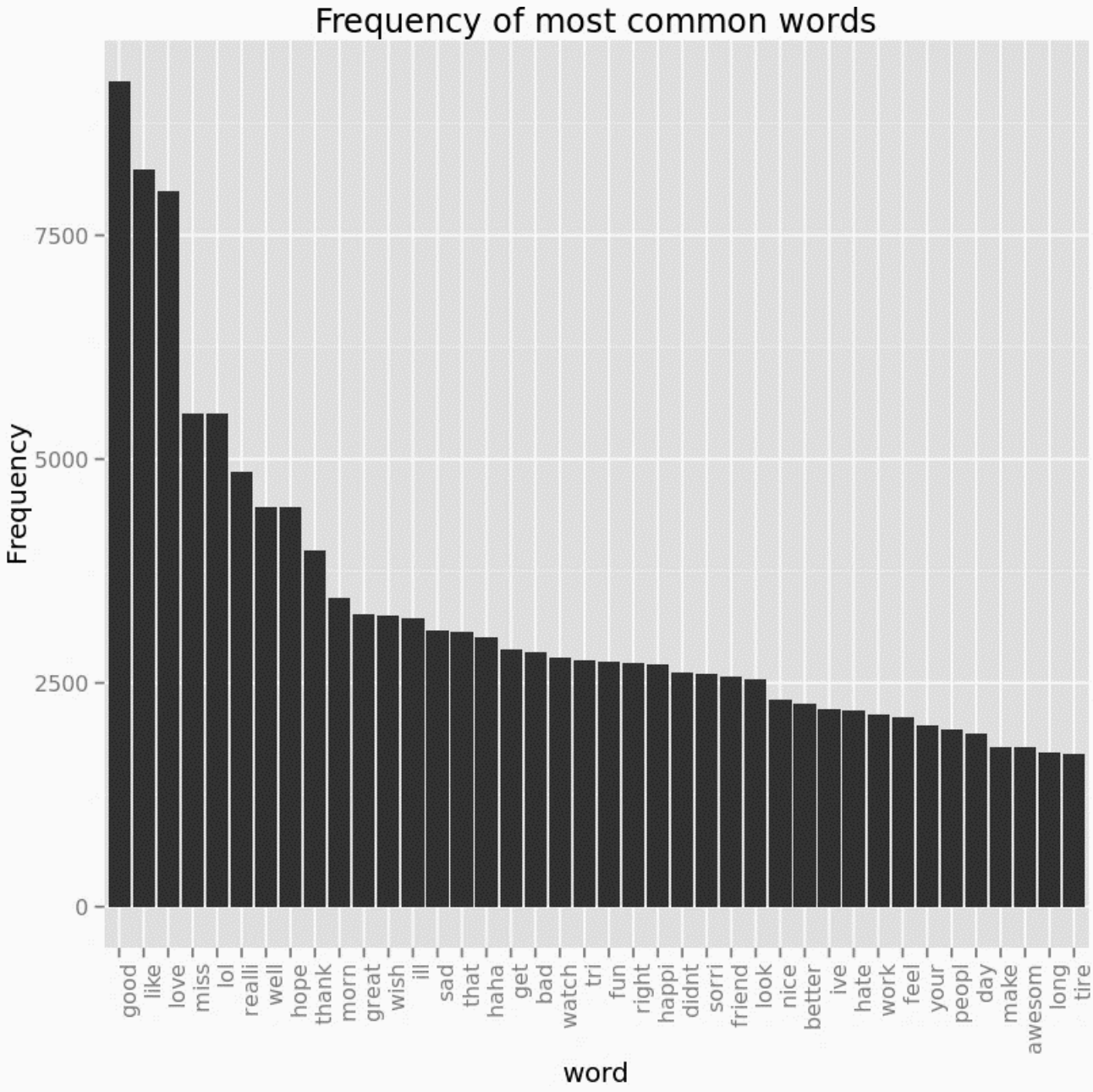
- What is the percentage of all words for these first 20 words?
- Of these 20 words, how many are likely to contribute sentiment information?
- Are these 20 words different from the words seen for the normalized text?

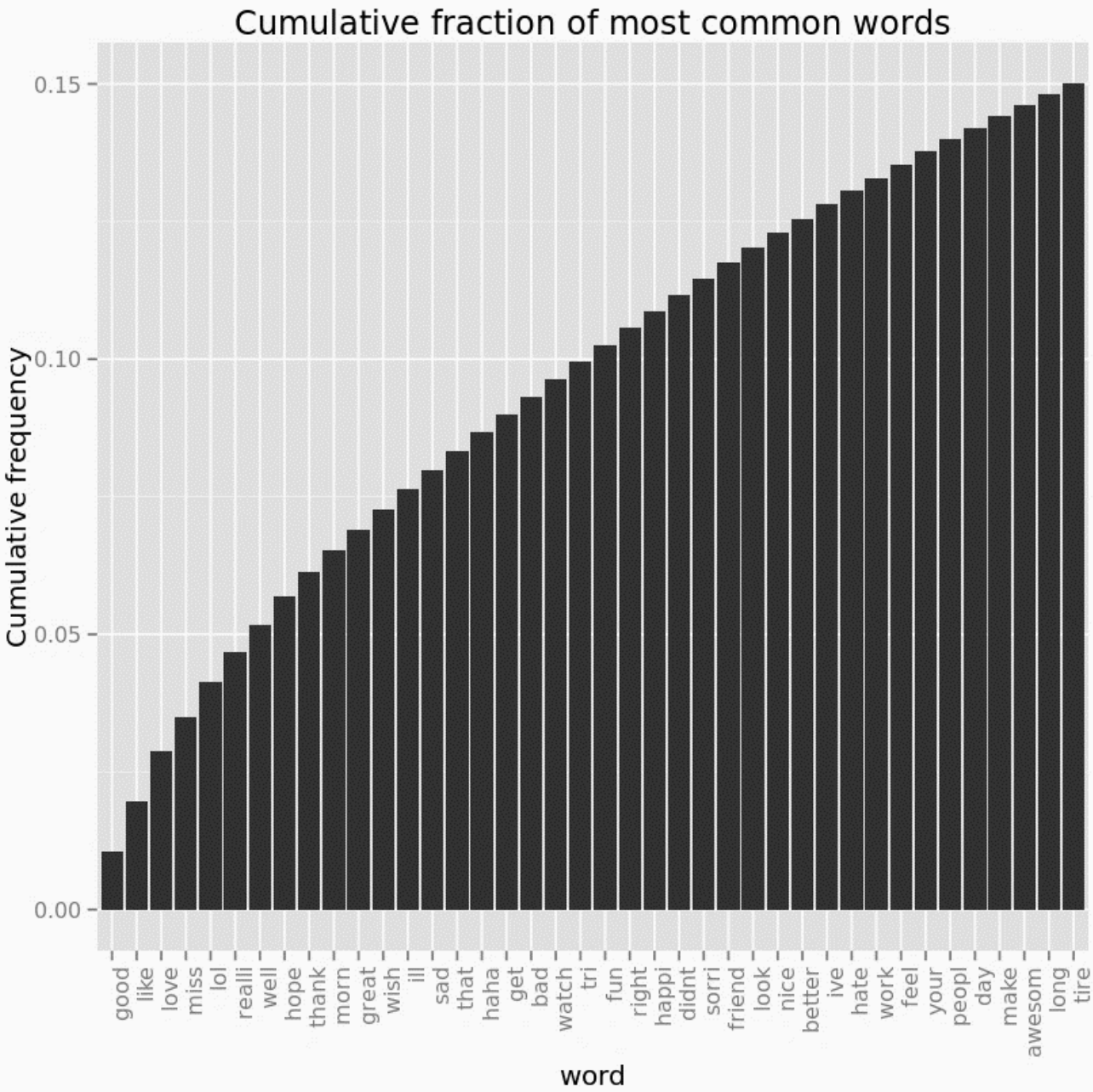
| | word | freq | Cum |
|---|---|---|---|
| **good** | good | 9076 | 0.01019084 |
| **like** | like | 7767 | 0.01891188 |
| **going** | going | 6384 | 0.02608005 |
| **love** | love | 6217 | 0.03306071 |
| **lol** | lol | 5489 | 0.03922394 |
| **really** | really | 4859 | 0.04467979 |
| **well** | well | 4451 | 0.04967752 |
| **thanks** | thanks | 3899 | 0.05405545 |
| **miss** | miss | 3548 | 0.05803926 |
| **morning** | morning | 3386 | 0.06184118 |
| **hope** | hope | 3383 | 0.06563972 |
| **great** | great | 3243 | 0.06928107 |
| **ill** | ill | 3207 | 0.072882 |
| **thats** | thats | 3066 | 0.07632461 |
| **haha** | haha | 3005 | 0.07969872 |
| **sad** | sad | 2800 | 0.08284266 |
| **wish** | wish | 2748 | 0.0859282 |
| **fun** | fun | 2734 | 0.08899803 |
| **bad** | bad | 2730 | 0.09206336 |
| **right** | right | 2699 | 0.09509389 |

## Stem the Words

The tweet text has been cleaned and stop words removed. There is one last data preparation step required, stemming the words. Stemming is a process of reducing words to their stems or roots. For example, conjugated verbs such as “goes”, ”going”, and ”gone” are stemmed to the word “go”. Both Python and R offer a choice of stemmers. Depending on this choice, the results can be more or less suitable for the application. In this case, the popular Porter stemmer is applied.

The Porter stemmer is in the R **SnowBallC** library and the word frequency bar chart and **CDF** follow:






| | sentiment | text |
|---|---|---|
| **tweets.content1** | 4 | elephantbird dear happi alreadi rice bowl |
| **tweets.content2** | 4 | ughhh layin downnnn wait zeina cook breakfast |
| **tweets.content3** | 0 | greeniebach reckon hell hes noth wont without |
| **tweets.content4** | 0 | valewe news |
| **tweets.content5** | 0 | sad httpwwwfabchannelcom close web servic ive use year |
| **tweets.content6** | 0 | fearnecotton sing rememb alwaysss radio catch artist |

The two charts using the stemmed words to the charts created with just stop word filtering have notable differences. For example, some words like “good”, and “like” have moved higher in the order of most frequent words, while some other words like “going” have moved down.

| | word | freq | Cum |
|---|---|---|---|
| **good** | good | 9214 | 0.01042899 |
| **like** | like | 8241 | 0.01975667 |
| **love** | love | 7991 | 0.02880139 |
| **miss** | miss | 5512 | 0.03504022 |
| **lol** | lol | 5508 | 0.04127452 |

| | | | |
|---|---|---|---|
| **realli** | realli | 4859 | 0.04677425 |
| **well** | well | 4469 | 0.05183254 |
| **hope** | hope | 4464 | 0.05688518 |
| **thank** | thank | 3982 | 0.06139226 |
| **morn** | morn | 3451 | 0.06529832 |

| | | | |
|---|---|---|---|
| **great** | great | 3275 | 0.06900517 |
| **wish** | wish | 3258 | 0.07269278 |
| **ill** | ill | 3227 | 0.0763453 |
| **sad** | sad | 3079 | 0.07983031 |
| **that** | that | 3066 | 0.0833006 |

| | | | |
|---|---|---|---|
| **haha** | haha | 3005 | 0.08670185 |
| **get** | get | 2871 | 0.08995143 |
| **bad** | bad | 2848 | 0.09317498 |
| **watch** | watch | 2787 | 0.09632948 |
| **tri** | tri | 2757 | 0.09945003 |

## Transformations Code (R)

```
dataset <- maml.mapInputPort(1)

library(tm) ## Text mining library

## Set the column names
colnames(dataset) <- c("sentiment", "tweets")

## Extract text data and coerce the vector to a tm corpus
tweet.text <- Corpus(VectorSource(dataset['tweets']))

## Apply transformations to the corpus
tweet.text <- tm_map(tweet.text, content_transformer(removeNumbers))
tweet.text <- tm_map(tweet.text, content_transformer(removePunctuation))
tweet.text <- tm_map(tweet.text, content_transformer(stripWhitespace))
tweet.text <- tm_map(tweet.text, content_transformer(tolower))

## Transform the processed corpus back to a vector of
## character strings in a dataframe
tweet_content <- unlist(sapply(tweet.text, '[', "content"))
outframe <- data.frame(tweets = enc2utf8(tweet_content),
                       sentiment = dataset$sentiment / 2 - 1,
                       stringsAsFactors = F,
                       row.names = NULL)

## Output the result
maml.mapOutputPort("outframe")
```

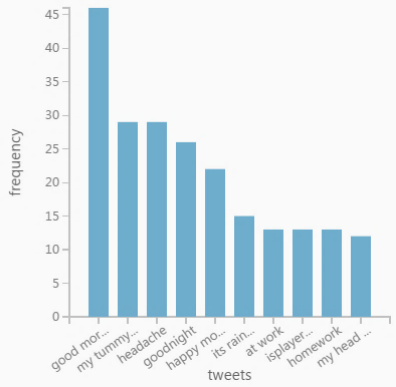
Statistics

| | |
|---|---|
| Unique Values | 158761 |
| Missing Values | 0 |
| Feature Type | String Feature |

Visualizations

tweets

Histogram




## Stopwords Code (R)

```
dataset <- maml.mapInputPort(1)
stop.words <- maml.mapInputPort(2)

library(tm) ## Text mining library

## Extract text data and coerce the vector to a tm corpus
tweet.text <- Corpus(VectorSource(dataset['tweets']))

## Remove the stopwords
stop.words['words'] <- unique(stop.words['words'])
tweet.text <- tm_map(tweet.text, removeWords, stop.words[, 'words'])

## Transform the processed corpus back to a vector of
## character strings in a dataframe
dataset['tweets'] <- data.frame(text =
enc2utf8(unlist(sapply(tweet.text, `[`, "content"))),
                                stringsAsFactors = F)

## Output the result
maml.mapOutputPort("dataset")
```

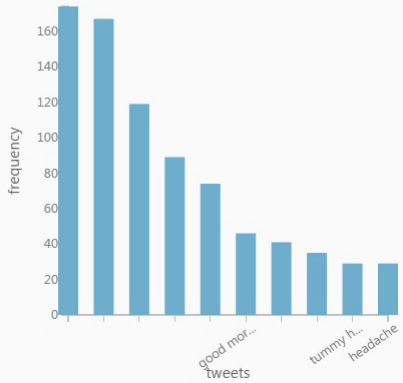
Statistics

| | |
|---|---|
| Unique Values | 157475 |
| Missing Values | 0 |
| Feature Type | String Feature |

Visualizations

tweets

Histogram




## Stemming Code (R)

```
dataset <- maml.mapInputPort(1)

library(tm) ## Text mining library
library(SnowballC) ## For stemming words

## Extract text data and create a tm corpus
tweet.text <- Corpus(VectorSource(dataset['tweets']))

## Stem the words in the tweets
tweet.text <- tm_map(tweet.text, stemDocument)

## Transform the processed corpus back to a vector of
## character strings in a dataframe
dataset['tweets'] <- data.frame(text =
enc2utf8(unlist(sapply(tweet.text, `[`, "content"))),
                                stringsAsFactors = F)
## Output the result
maml.mapOutputPort("dataset")
```

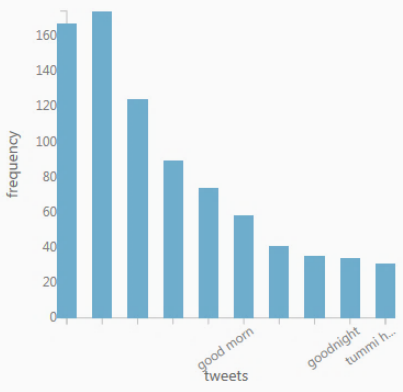
Statistics

| | |
|---|---|
| Unique Values | 157158 |
| Missing Values | 0 |
| Feature Type | String Feature |

Visualizations

tweets

Histogram



## Feature Hashing

Target column(s)

**Selected columns:**
**Column names**: tweets

Launch column selector

Hashing bitsize

15

N-grams

2

| | |
|---|---|
| START TIME | 9/11/2016 12:56:25 PM |
| END TIME | 9/11/2016 12:56:25 PM |
| ELAPSED TIME | 0:00:00.000 |
| STATUS CODE | Finished |
| STATUS DETAILS | Task output was present in output cache |

## Two-Class Logistic Regression

Create trainer mode

Parameter Range

Optimization tolerance

Use Range Builder

0.0001, 0.0000001

L1 regularization weight

Use Range Builder

0.0, 0.01, 0.1, 1.0

L2 regularization weight

Use Range Builder

0.01, 0.1, 1.0

Memory size for L-BFGS

Use Range Builder

5, 20, 50

Random number seed

1234

Allow unknown categorical levels

| | |
|---|---|
| START TIME | 9/11/2016 12:56:24 PM |
| END TIME | 9/11/2016 12:56:24 PM |
| ELAPSED TIME | 0:00:00.000 |
| STATUS CODE | Finished |
| STATUS DETAILS | Task output was present in output cache |

## Logistic Regression Classifier

### Settings

| Setting | Value |
|---|---|
| Optimization Tolerance | 7.219577E-05 |
| L1 Weight | 0.9873694 |
| L2 Weight | 0.8192218 |
| Memory Size | 37 |
| Quiet | True |
| Use Threads | True |
| Allow Unknown Levels | True |
| Random Number Seed | 1234 |

## DAT203.3x: Tweet Sentiment › Feature Hashing › Transformed dataset

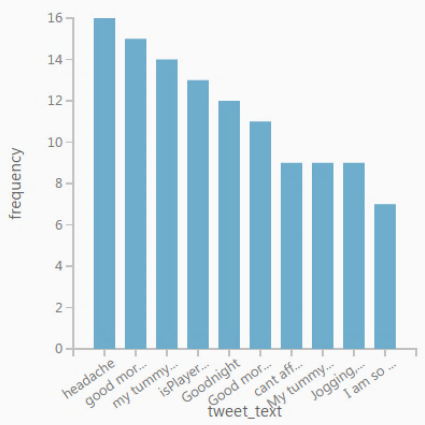
| rows | columns |
|---|---|
| 160000 | 32770 |

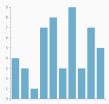
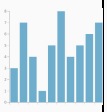
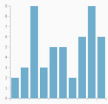
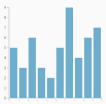
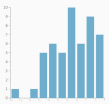
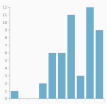
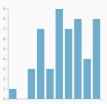
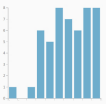
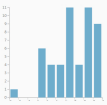
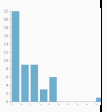
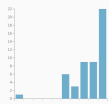
### Statistics

| | |
|---|---|
| Unique Values | 159384 |
| Missing Values | 0 |
| Feature Type | String Feature |

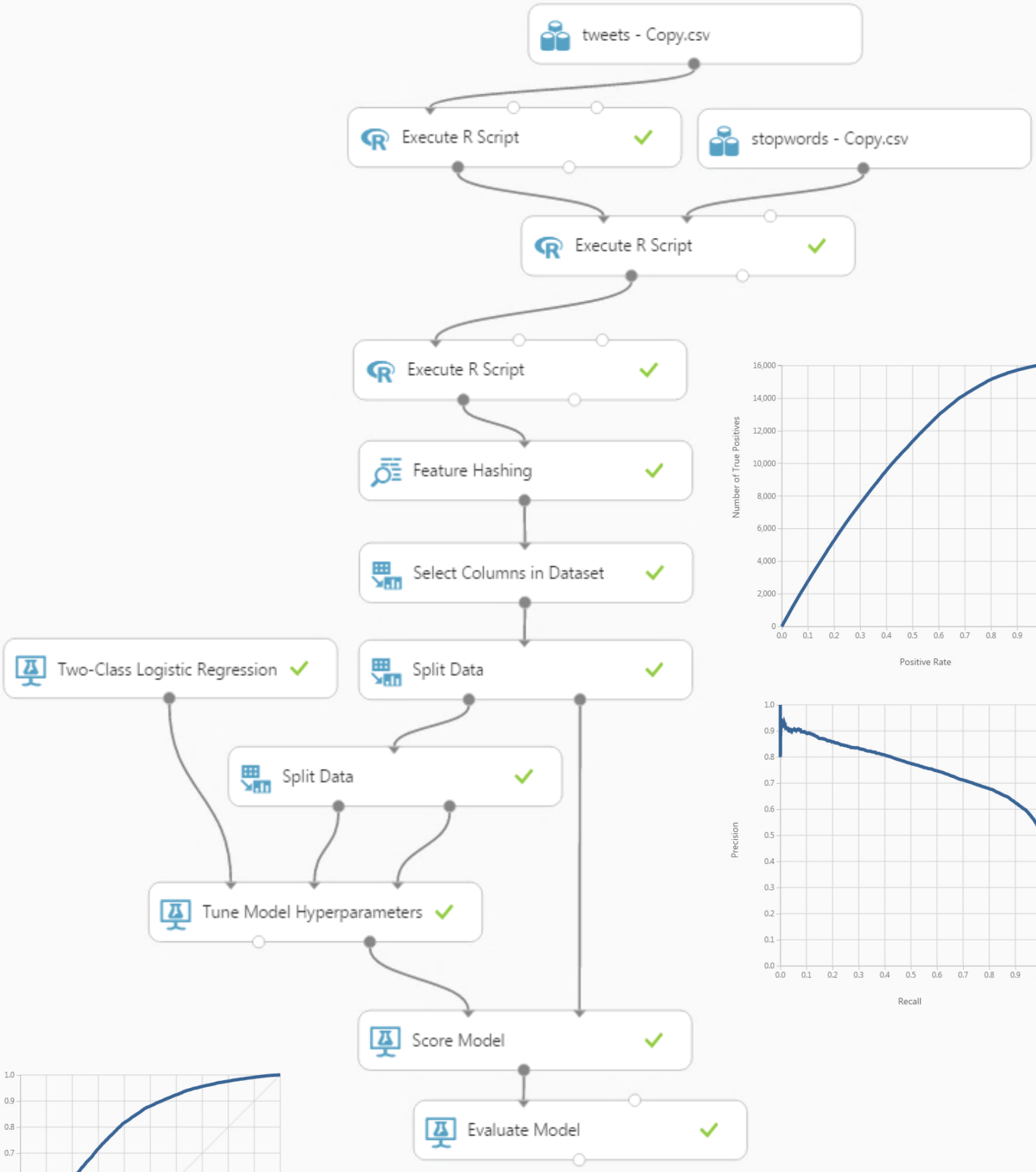
### Visualizations

**tweet_text**
Histogram



| Optimization Tolerance | L1Weight | L2Weight | Memory Size | Accuracy | Precision | Recall | F-Score | AUC | Average Log Loss | Training Log Loss |
|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  |  |  |  |  |
| 0.000072 | 0.987369 | 0.819222 | 37 | 0.715938 | 0.706338 | 0.737015 | 0.721351 | 0.786657 | 0.562243 | 18.885164 |
| 0.000062 | 0.965212 | 0.84226 | 47 | 0.715703 | 0.706289 | 0.736337 | 0.721 | 0.786473 | 0.562561 | 18.839362 |
| 0.000042 | 0.966165 | 0.837019 | 33 | 0.715417 | 0.705509 | 0.737328 | 0.721068 | 0.786168 | 0.562846 | 18.79819 |
| 0.000037 | 0.949337 | 0.744665 | 49 | 0.714688 | 0.70522 | 0.735554 | 0.720067 | 0.785144 | 0.565134 | 18.468124 |
| 0.000071 | 0.863326 | 0.680521 | 17 | 0.714219 | 0.705531 | 0.733152 | 0.719076 | 0.784801 | 0.56727 | 18.159966 |
| 0.000078 | 0.890538 | 0.592252 | 32 | 0.712708 | 0.704053 | 0.731691 | 0.717606 | 0.782702 | 0.570259 | 17.728712 |
| 0.000066 | 0.983824 | 0.427967 | 34 | 0.7125 | 0.702834 | 0.734092 | 0.718123 | 0.782088 | 0.572611 | 17.389419 |
| 0.00006 | 0.792988 | 0.656329 | 7 | 0.711224 | 0.702287 | 0.731064 | 0.716387 | 0.781737 | 0.572943 | 17.341528 |
| 0.000042 | 0.797965 | 0.74529 | 12 | 0.711016 | 0.7029 | 0.728768 | 0.7156 | 0.781616 | 0.57176 | 17.512114 |
| 0.000059 | 0.630235 | 0.81784 | 15 | 0.709818 | 0.702742 | 0.725009 | 0.713702 | 0.780277 | 0.575658 | 16.949781 |
| 0.000075 | 0.902023 | 0.278784 | 16 | 0.709271 | 0.701041 | 0.727463 | 0.714008 | 0.779992 | 0.577874 | 16.630165 |
| 0.000086 | 0.822914 | 0.557223 | 37 | 0.709219 | 0.700769 | 0.727985 | 0.714117 | 0.780054 | 0.576258 | 16.863311 |
| 0.000095 | 0.599584 | 0.961183 | 44 | 0.708255 | 0.701673 | 0.722295 | 0.711835 | 0.77894 | 0.57573 | 16.939368 |
| 0.000048 | 0.62759 | 0.886608 | 43 | 0.708021 | 0.701109 | 0.722921 | 0.711848 | 0.778503 | 0.577022 | 16.75308 |
| 0.000092 | 0.702919 | 0.693006 | 18 | 0.707552 | 0.700612 | 0.722556 | 0.711415 | 0.778491 | 0.578702 | 16.51068 |
| 0.000004 | 0.855496 | 0.300866 | 34 | 0.706875 | 0.699369 | 0.723391 | 0.711177 | 0.775861 | 0.587661 | 15.218137 |
| 0.000008 | 0.579902 | 0.899284 | 43 | 0.706823 | 0.700574 | 0.720102 | 0.710204 | 0.777338 | 0.579268 | 16.429014 |
| 0.000087 | 0.902168 | 0.234615 | 44 | 0.706667 | 0.699147 | 0.723234 | 0.710987 | 0.776459 | 0.587072 | 15.303101 |
| 0.000055 | 0.810852 | 0.296853 | 17 | 0.706224 | 0.699453 | 0.720885 | 0.710007 | 0.77638 | 0.587958 | 15.175238 |
| 0.000059 | 0.538525 | 0.913186 | 49 | 0.706146 | 0.700198 | 0.718693 | 0.709325 | 0.776486 | 0.581103 | 16.164302 |
| 0.000082 | 0.487715 | 0.852109 | 7 | 0.704766 | 0.699469 | 0.715717 | 0.7075 | 0.775048 | 0.584891 | 15.617813 |
| 0.000082 | 0.517855 | 0.801053 | 18 | 0.704271 | 0.698448 | 0.716605 | 0.70741 | 0.774875 | 0.586322 | 15.411283 |
| 0.000042 | 0.438004 | 0.766232 | 12 | 0.702083 | 0.696973 | 0.71269 | 0.704744 | 0.771902 | 0.594117 | 14.286758 |
| 0.000049 | 0.44319 | 0.803723 | 30 | 0.701771 | 0.697049 | 0.711385 | 0.704144 | 0.771883 | 0.59286 | 14.4681 |
| 0.000062 | 0.76727 | 0.171883 | 34 | 0.699792 | 0.693935 | 0.712481 | 0.703086 | 0.76969 | 0.60724 | 12.393441 |
| 0.000057 | 0.73465 | 0.144597 | 29 | 0.698828 | 0.692047 | 0.714047 | 0.702875 | 0.768911 | 0.611077 | 11.839867 |
| 0.000035 | 0.282475 | 0.931563 | 26 | 0.698776 | 0.694516 | 0.707313 | 0.700856 | 0.768482 | 0.600342 | 13.388639 |
| 0.000048 | 0.69215 | 0.219875 | 27 | 0.698568 | 0.693735 | 0.708618 | 0.701097 | 0.768936 | 0.610133 | 11.976103 |
| 0.000015 | 0.440053 | 0.578547 | 49 | 0.698255 | 0.694063 | 0.706635 | 0.700292 | 0.767434 | 0.607729 | 12.322923 |
| 0.000009 | 0.28278 | 0.904311 | 29 | 0.698255 | 0.694103 | 0.70653 | 0.700261 | 0.76795 | 0.601917 | 13.161364 |
| 0.000021 | 0.787116 | 0.05818 | 41 | 0.698099 | 0.692175 | 0.711072 | 0.701496 | 0.766425 | 0.620188 | 10.525497 |
| 0.000029 | 0.683195 | 0.220871 | 22 | 0.697812 | 0.692846 | 0.708253 | 0.700465 | 0.76784 | 0.611684 | 11.752335 |
| 0.000035 | 0.510715 | 0.362778 | 45 | 0.696927 | 0.692647 | 0.705591 | 0.699059 | 0.765129 | 0.619046 | 10.690283 |
| 0.000079 | 0.475943 | 0.400795 | 28 | 0.69651 | 0.692493 | 0.704494 | 0.698442 | 0.764749 | 0.619777 | 10.584724 |
| 0.000088 | 0.559042 | 0.24014 | 19 | 0.69599 | 0.691862 | 0.704286 | 0.698019 | 0.764667 | 0.626174 | 9.66191 |
| 0.000061 | 0.537679 | 0.236857 | 32 | 0.695286 | 0.689489 | 0.708096 | 0.698669 | 0.762416 | 0.63113 | 8.946835 |
| 0.000078 | 0.219613 | 0.793519 | 25 | 0.694193 | 0.690689 | 0.700893 | 0.695754 | 0.763573 | 0.616491 | 11.058797 |
| 0.000044 | 0.291094 | 0.528909 | 35 | 0.692917 | 0.689908 | 0.698335 | 0.694096 | 0.760212 | 0.632603 | 8.734293 |
| 0.00008 | 0.122658 | 0.808293 | 35 | 0.691901 | 0.689361 | 0.69609 | 0.692709 | 0.759991 | 0.627567 | 9.460836 |
| 0.000007 | 0.128374 | 0.749161 | 48 | 0.691458 | 0.68901 | 0.695412 | 0.692196 | 0.758809 | 0.632897 | 8.691962 |
| 0.000019 | 0.265951 | 0.419201 | 5 | 0.69125 | 0.687465 | 0.698805 | 0.693088 | 0.757015 | 0.651799 | 5.964985 |
| 0.000048 | 0.204696 | 0.574287 | 13 | 0.690964 | 0.688841 | 0.694054 | 0.691438 | 0.757935 | 0.642081 | 7.366973 |
| 0.000066 | 0.381252 | 0.186516 | 21 | 0.688906 | 0.686071 | 0.69395 | 0.689988 | 0.752958 | 0.677696 | 2.228742 |
| 0.000032 | 0.204689 | 0.471536 | 49 | 0.688516 | 0.685886 | 0.69301 | 0.68943 | 0.754481 | 0.657154 | 5.192418 |
| 0.000034 | 0.541512 | 0.02431 | 43 | 0.688151 | 0.684836 | 0.694524 | 0.689646 | 0.751224 | 0.688735 | 0.636169 |
| 0.000039 | 0.207929 | 0.386836 | 8 | 0.6875 | 0.685232 | 0.691027 | 0.688117 | 0.752809 | 0.667518 | 3.697091 |
| 0.000079 | 0.130017 | 0.371928 | 6 | 0.687109 | 0.687277 | 0.684084 | 0.685677 | 0.752501 | 0.674533 | 2.685072 |
| 0.000065 | 0.127815 | 0.443888 | 39 | 0.684818 | 0.682784 | 0.687738 | 0.685252 | 0.749667 | 0.680648 | 1.802955 |
| 0.000032 | 0.069577 | 0.497344 | 32 | 0.684036 | 0.682404 | 0.685859 | 0.684127 | 0.748955 | 0.683001 | 1.463398 |
| 0.000057 | 0.03185 | 0.21697 | 38 | 0.670521 | 0.669923 | 0.669364 | 0.669643 | 0.731614 | 0.819961 | 18.295771 |

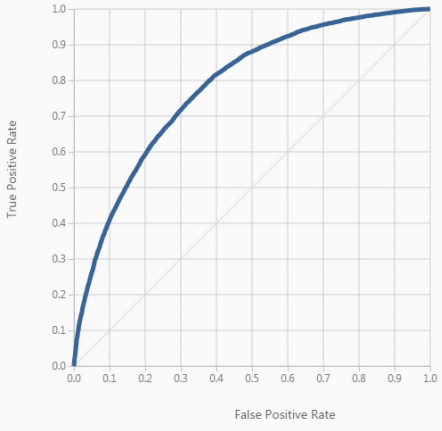
# Tweet Sentiment









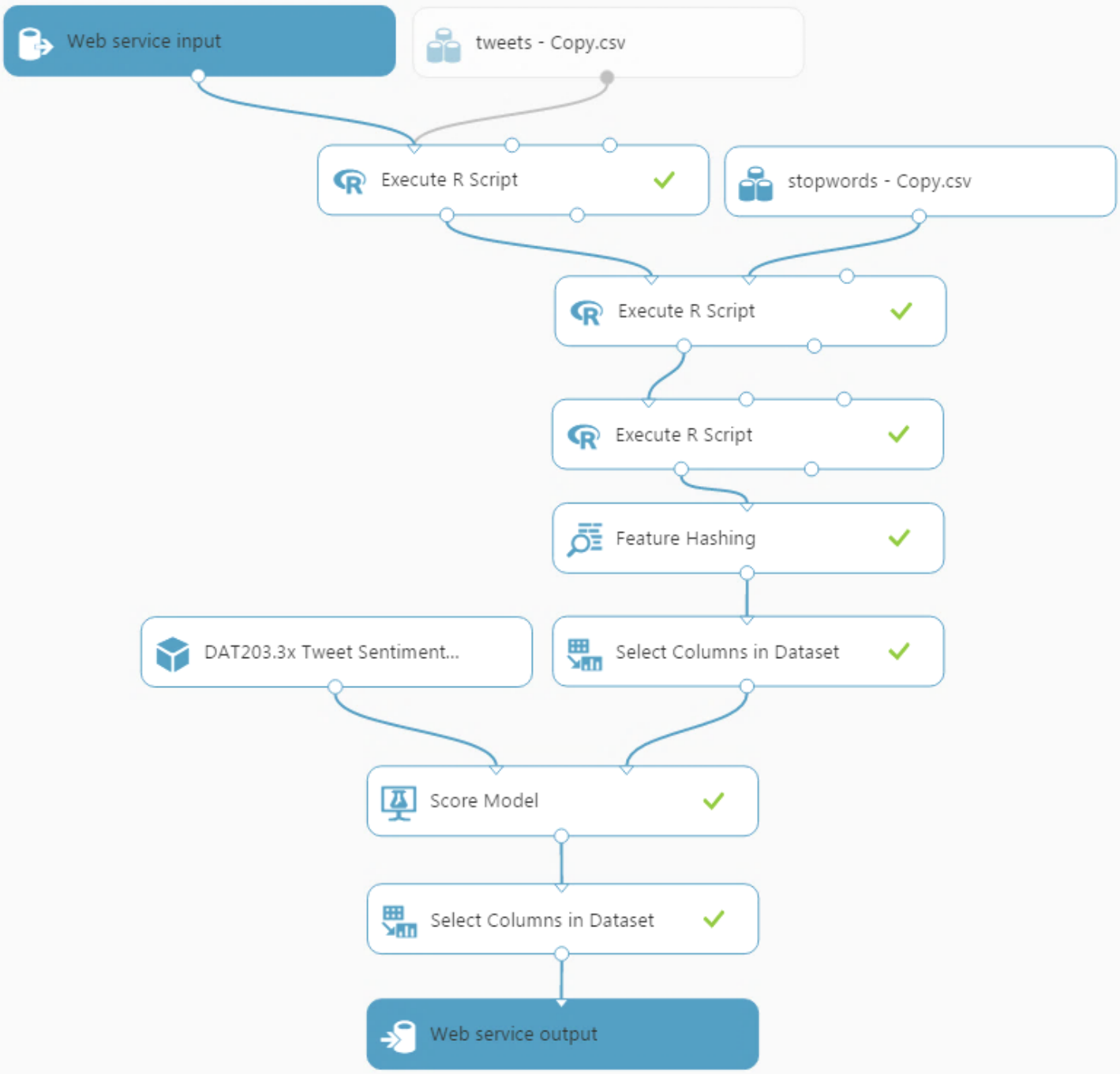

| True Positive | False Negative | Accuracy | Precision | Threshold | AUC |
|---|---|---|---|---|---|
| 11764 | 4290 | 0.710 | 0.702 | 0.5 | 0.782 |

| False Positive | True Negative | Recall | F1 Score |
|---|---|---|---|
| 5002 | 10944 | 0.733 | 0.717 |

| Positive Label | Negative Label |
|---|---|
| 1 | -1 |

| Score Bin | Positive Examples | Negative Examples | Fraction Above Threshold | Accuracy | F1 Score | Precision | Recall | Negative Precision | Negative Recall | Cumulative AUC |
|---|---|---|---|---|---|---|---|---|---|---|
| (0.900,1.000] | 2006 | 260 | 0.071 | 0.553 | 0.219 | 0.885 | 0.125 | 0.528 | 0.984 | 0.001 |
| (0.800,0.900] | 2404 | 600 | 0.165 | 0.609 | 0.414 | 0.837 | 0.275 | 0.564 | 0.946 | 0.009 |
| (0.700,0.800] | 2520 | 903 | 0.272 | 0.660 | 0.560 | 0.797 | 0.432 | 0.609 | 0.889 | 0.029 |
| (0.600,0.700] | 2313 | 1241 | 0.383 | 0.693 | 0.653 | 0.755 | 0.576 | 0.655 | 0.812 | 0.069 |
| (0.500,0.600] | 2502 | 1982 | 0.523 | 0.710 | 0.716 | 0.702 | 0.732 | 0.718 | 0.687 | 0.150 |
| (0.400,0.500] | 1672 | 1823 | 0.632 | 0.705 | 0.740 | 0.663 | 0.836 | 0.776 | 0.573 | 0.240 |
| (0.300,0.400] | 1048 | 1874 | 0.723 | 0.679 | 0.738 | 0.625 | 0.901 | 0.820 | 0.455 | 0.343 |
| (0.200,0.300] | 776 | 2045 | 0.812 | 0.639 | 0.725 | 0.587 | 0.949 | 0.865 | 0.327 | 0.462 |
| (0.100,0.200] | 516 | 2505 | 0.906 | 0.577 | 0.700 | 0.544 | 0.981 | 0.901 | 0.170 | 0.614 |
| (0.000,0.100] | 297 | 2713 | 1.000 | 0.502 | 0.668 | 0.502 | 1.000 | 1.000 | 0.000 | 0.782 |

# Tweet Sentiment [Predictive Exp.]



# module4 · image analysis

## introduction to image analysis

### Introduction

Image data can be sourced from various places including:

- Photographs
- Security cameras, smart phone cameras
- Scanned documents (e.g., bank checks)
- Medical images
- Artwork
- Science (e.g., biology)
- Mechanical processes (e.g., flow through a duct)

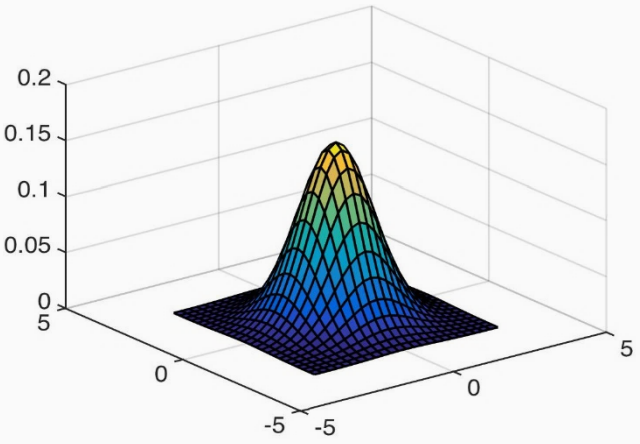
Image data can be transformed and manipulated in many ways including:

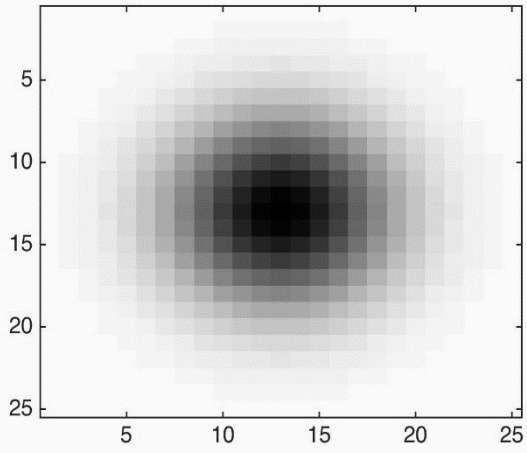
- Reduce noise
  - Not just to make images look nicer visually, but also for detecting what objects are inside the images. Imagine a self-driving car trying to reduce noise from the rain on its camera to identify where the center line of the road and the curb is.
- Find interesting features (edges, corners)
  - The interesting features try to help discern what the essence of the image is; a way to summarize the image in the way needed for some other purpose.
- Use the features for classifying images
  - Finding out what is inside the image

### Blurring and Denoising

Image processing is a well-studied and highly mature field. **Blurring** for the purpose of **Denoising** is one of the most popular techniques applied to images. The latter involves convolution with what is called a **filter**; the **filter** is typically **Gaussian**.

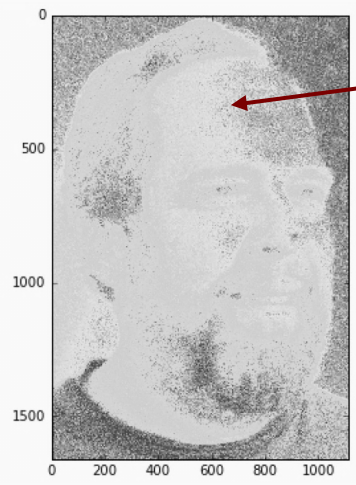
The following is a 2-dimensional plotted Gaussian (left) and a top view of the Gaussian (right). The right image displays larger values in the middle and smaller values on the outside:



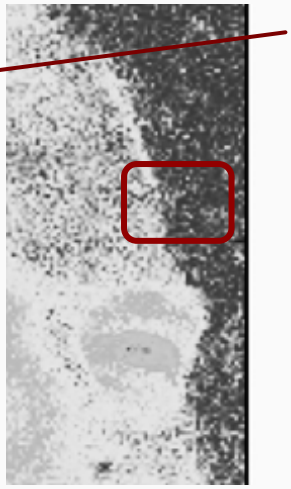

The process will take the latter Gaussian and **'convolve'** it with an image.

The following illustrates **convolution** of an image with a **filter** (typically Gaussian):




The pixel intensity of the original image (left) is between 0 and 255.

In order to **denoise** the original image, a chosen area of noise is zoomed in on the image (right).



The simplest way to smooth out the original image pane would be to replace the center of each entire with the average of pixel intensities in the square:

The following measure takes the simple average of the pixel intensities:

$$\text{Average}(\ \square\ ) = \sum_{pixels} \square \bullet \square$$

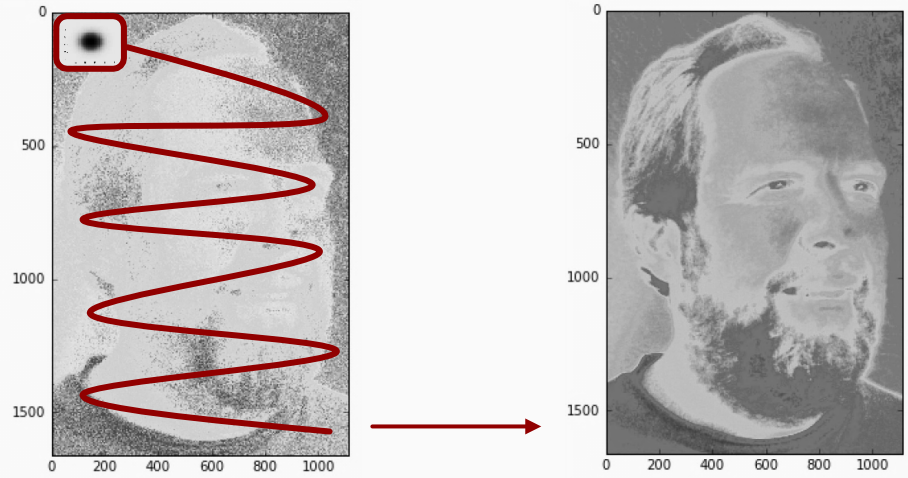
The average pixel intensity of the pane can be computed by multiplying each pixel intensity by $\frac{1}{\text{the number of pixels}}$ and sum all of the resulting values. The average value computed in the pane can replace the middle pixel in the image by its average.

A disadvantage of the above process is trying to estimate what belongs in the middle of the square; it should depend more on the pixels near the middle opposed to the edges.

Rather than an average weight for each individual square, the center pixels are more weighted than the outer. The latter is where Gaussian is applied as a weighting function:

$$\text{Weighted Average}\ (\ \square\ ) = \sum_{pixels} \square \bullet \square$$

The process is repeated over every spot in the image, taking a weighted average of the pixels. The pixels in the nearby area are each weighted according to the height of the Gaussian; the new intensity for that point in the center. The process of computing the latter all over the image is **convolution**:



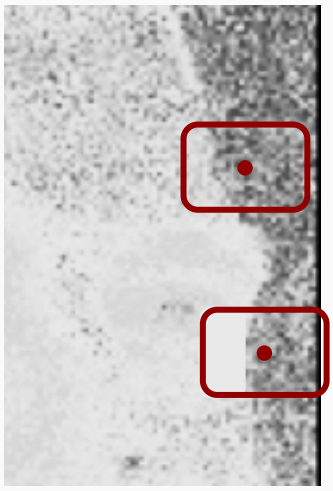

The resulting image has been **denoised**. However, if the Gaussian was too wide, the image would have become too blurred. If the bandwidth was too narrow, the image would remain noisy.

It is not uncommon for this process to blur edges and lines.

# Denoising with a Median Filter

A **Median Filter** computes the median of the intensities as opposed to the weighted average of the Gaussian Filter. The reason for applying a **Median Filter** is due to **preserving edges**. Gaussian Filters are prone to blurring edges in an image.

Median(  )

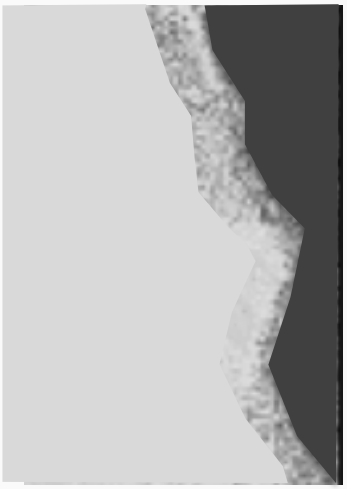


Median filters tend to be better at preserving edges than Gaussian filters.

In the illustrated pane (right), a Gaussian Filter would be influenced by the lighter portion to the left side of the pane. With a **Median Filter**, the pixel in the center will maintain its intensity measurement regardless.

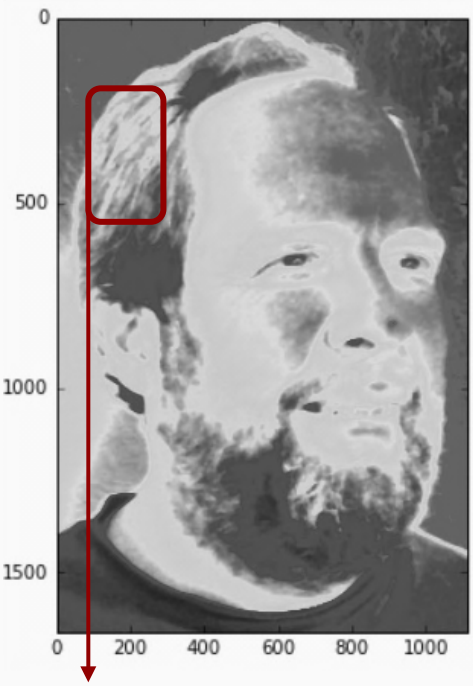
Even if a portion was extremely contrasted within the pain, the median itself would not be affected as seen in the lower pane in the left image.

Therefore, the edges will mostly be preserved in an image and smoothing achieved in the appropriate areas.

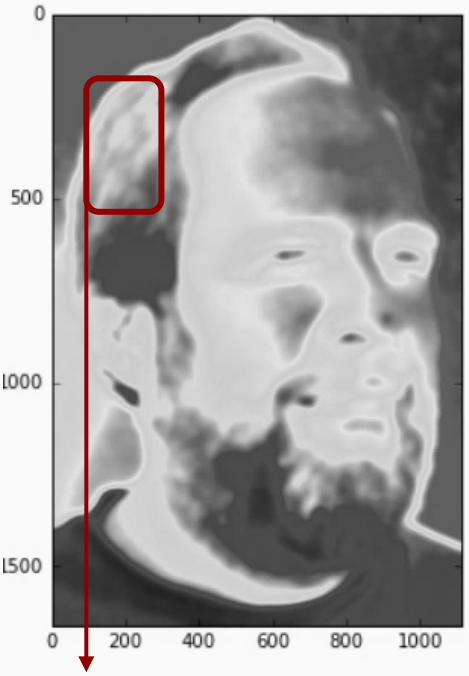


The following images display an image with a Gaussian Filter (left image) and a Median Filter (right image) using the same sized filter.

A panel of the image is zoomed in below to illustrate the preservation of edges that the Median Filter is able to maintain.




**Median Filter**




**Gaussian Filter**

# working with images

## Edge Detection

In the computer vision paper of **Viola and Jones**, a **weak classifier** was used for **edge detection** in images by moving the filter around an image and subtracting what is in the white area from what is in the black area:

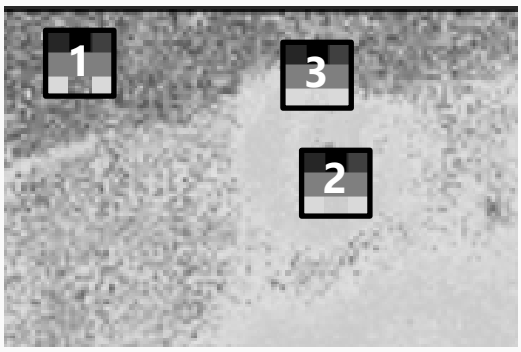
The weak classifier does not find much difference between the areas on the man's forehead, thus no edge is detected. The same result happens when the weak classifier is applied to the man's beard. However, when the weak classifier is applied to the man's eyes, the subtracted areas result in a large number because the man's eyes are much darker than the surrounding area of the man's cheeks. Thus, the weak classifier successfully detects an edge.

## Sobel Edge Detector

A more powerful **edge detector** is the **Sobel Edge Detector**, which is slightly different than the Viola and Jones weak classifier. The **Sobel Edge Detector** puts more weight on the center gradients of the image than on the outer edges.

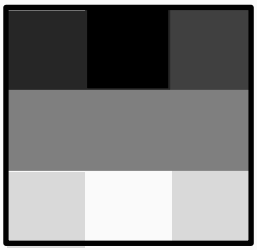
Weak Classifier

In the example below, a Sobel Edge Detector is applied to a section of a noisy image. The detector appropriately does not register anything in areas **1** and **2**; the value of the gradients are $\approx 0$. The detector does, however, register a large gradient (change) when the pixel intensities are subtracted in area **3** and determines the presence of an edge.



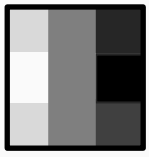

Translating the Sobel Edge Detector from greyscale into numerical value produces the resulting matrices:

Sobel Edge Detector



$$S_{horizontal} = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$S_{vertical} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix}$$



In the above matrices, 0 represents neutral. The outer pixel are weighted accordingly, with the middle values of the outer pixels weighted most heavily. In the horizontal edge detector, the top intensities will be subtracted from the bottom. In the vertical edge detector, the right intensities will be subtracted from the left. Note that the inclusion of +1s and -1s on the matrix edges are for smoothing purposes. The model would be unstable in application if just the +2s and -2s were applied

To illustrate, the vertical edge detector matrix is decomposed to show that actual vertical edge detector and the corresponding smoother which is similar to a Gaussian (normalizes the edge):

$$S_{vertical} = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \times \begin{bmatrix} -1 & 0 & +1 \end{bmatrix}$$

$y$ axis smoothing (Gaussian-like)

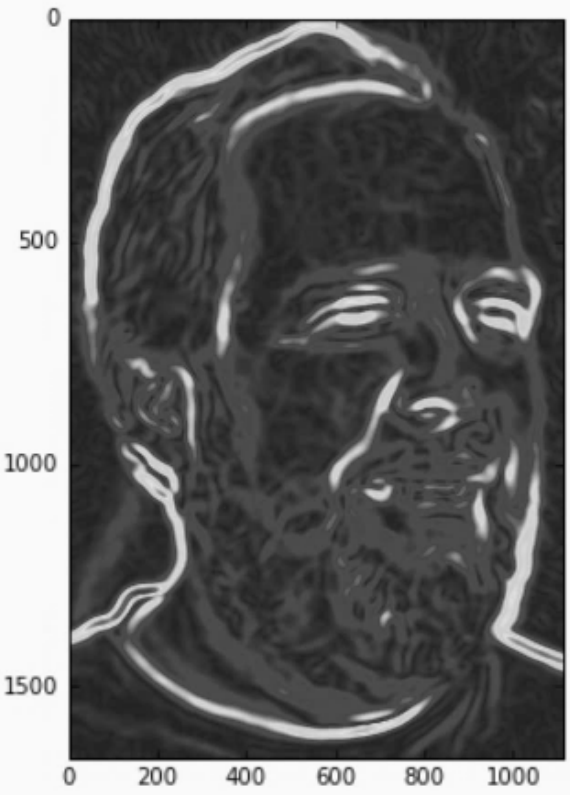
$x$ axis derivative (vertical detector)

$G_{horizontal} = S_{horizontal} \times A$ → Image with edges detected in the **horizontal** direction.

$G_{vertical} = S_{vertical} \times A$ → Image with edges detected in the **vertical** direction.

$$G = \sqrt{G_{vertical}^2 + G_{horizontal}^2} \qquad \theta = \text{atan}\left(\frac{G_{horizontal}}{G_{vertical}}\right)$$

The above notation illustrates the convolving of an edge detector with an image. The **Sobel Edge Detector** will register edges in both the **horizontal** and **vertical** directions. The **Pythagorean Theorem** is applied to compute the total magnitude of the edge. The angle of the edge is obtained through the **tangent** of the angle being the change in $y \rightarrow \Delta y$ over the change in $x \rightarrow \Delta x$. As seen in the above notation: $\text{edge angle} = \frac{\Delta y = \text{horizontal detector gradient}}{\Delta x = \text{vertical detector gradient}}$. The results are plotted below:



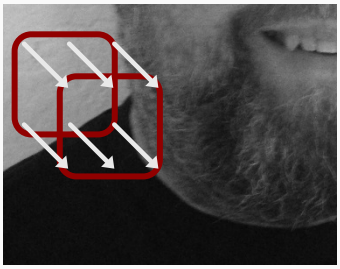
# working with images

## Corner Detection

Edge detectors do not typically classify corners well because they register linear gradient changes. In corners, pixel intensities occur in multiple directions at a time. The **Harris Corner Detector** registers changes in pixel intensities as it travels around the image.

The following function neasures the change in intensity across the region when $\delta_x$ and $\delta_y$ are shifted:

$$E(\delta_x, \delta_y) = \sum_{x,y} w(x,y)\left[I(x+\delta_x, y+\delta_y) - I(x,y)\right]^2$$



The $x$ and $y$ represents the directions within the image. The subtraction within the brackets represents the intensity from shifting. Each starting point $(x,y)$ is shifted by $(\delta_x, \delta_y)$. The function is squared because the significance of differences in any direction is relevant. Approaching a corner will output a high intensity, regardless of the direction approached. If the intensity is ~constant then the function $\left[I(x+\delta_x, y+\delta_y) - I(x,y)\right]^2$ will output ~0. Each pixel will then be weighted by $w(x,y)$ as chosen by the scientist. $w(x,y) = 1$ inside the box and 0 outside. $w(x,y) =$ larger value near the center of the box (similar to a **Gaussian Kernel**). The weighted differences are then summed within the entire box, computing function $E(\delta_x, \delta_y)$.

A first order **Taylor Expansion** is use to approximate the intensity at the shifted position:

$$I(x+\delta_x, y+\delta_y) \approx I(x,y) + I_x(x,y)\delta_x + I_y(x,y)\delta_y$$ *use discrete derivatives

Plug the **Taylor Expansion** into the orginal function:

$$E(\delta_x, \delta_y) \approx \sum_{x,y} w(x,y)\left[I_x(x,y)\delta_x + I_y(x,y)\delta_y\right]^2$$

Explaining the intution behind the above expressions:

$$E(\delta_x, \delta_y) \approx \sum_{x,y} w(x,y)\left[I_x(x,y)\delta_x + I_y(x,y)\delta_y\right]^2$$

$$= \begin{bmatrix}\delta_x & \delta_y\end{bmatrix} Q \begin{bmatrix}\delta_x \\ \delta_y\end{bmatrix}$$

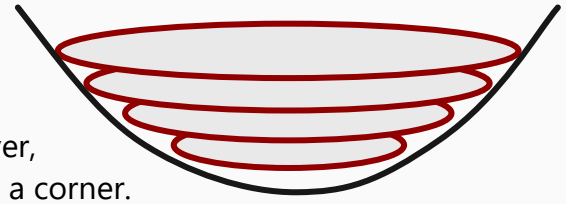
$$\text{where Q} = \sum_{\delta_x}\sum_{\delta_y} w(\delta_x, \delta_y) \begin{bmatrix} I_x^2(x,y) & I_x(x,y)I_y(x,y) \\ I_x(x,y)I_y(x,y) & I_y^2(x,y) \end{bmatrix}$$

$$Q = \begin{bmatrix} \sum_{\delta_x}\sum_{\delta_y} w(\delta_x, \delta_y)\, I_x^2(x,y) & \sum_{\delta_x}\sum_{\delta_y} w(\delta_x, \delta_y)\, I_x(x,y) I_y(x,y) \\ \sum_{\delta_x}\sum_{\delta_y} w(\delta_x, \delta_y)\, I_x(x,y) I_y(x,y) & \sum_{\delta_x}\sum_{\delta_y} w(\delta_x, \delta_y)\, I_y^2(x,y) \end{bmatrix}$$

The expression can be written out as the vector of deltas, times a matrix, times the vector of deltas.

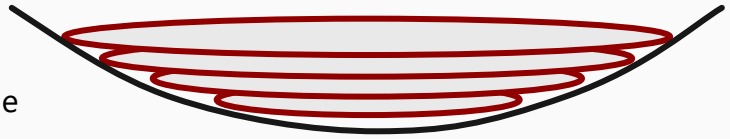
The summations are distributed within the matrix to illustrate 4 seperately derivable numbers. If the function $E(\delta_x, \delta_y)$ is equal to a constant, the result is just the formula for an ellipse:

$$= [\delta_x \quad \delta_y]\, Q \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix} = \text{Constant} \quad \rightarrow \quad \text{formula for an ellipse}$$



Therefore, the level sets of the function $E(\delta_x, \delta_y)$ are ellipses.

A notable curve in the ellipses indicate an approaching corner in the image. When the ellipses are broad as the image is passed over, the function $E(\delta_x, \delta_y)$ remains stationary in value not indicicating a corner.



In constrast, if the function $E(\delta_x, \delta_y)$ experiences drastic changes, meaning the intensities are varying significantly as the image is passed over, corners in the image are being detected throughout.

The determination of whether or not the function $E(\delta_x, \delta_y)$ is changing can be seen through examining the ellispes. However, determining whether or not the ellipses are wide (narrow) is seen through evaluating the **eigenvalues** of the $Q$ matrix. The Eigenvalues of $Q \begin{bmatrix} \delta_x \\ \delta_y \end{bmatrix}$ are related to $\frac{1}{\text{principal axes of the ellipse}}$. Therefore, if both **eigenvalues** are large, there exists intensity variance in multiple directions ($\geq$ two directions), indicating a corner.

In summary, compute (approximate) $\boldsymbol{Q}$ for each $(\boldsymbol{x}, \boldsymbol{y})$. If $\boldsymbol{Q}$ has **2** large **eigenvalues** → **corner**.

The right image illustrates the **Harris Corner Detector's** output of the image.

# working with images

## Mathematical Morphology

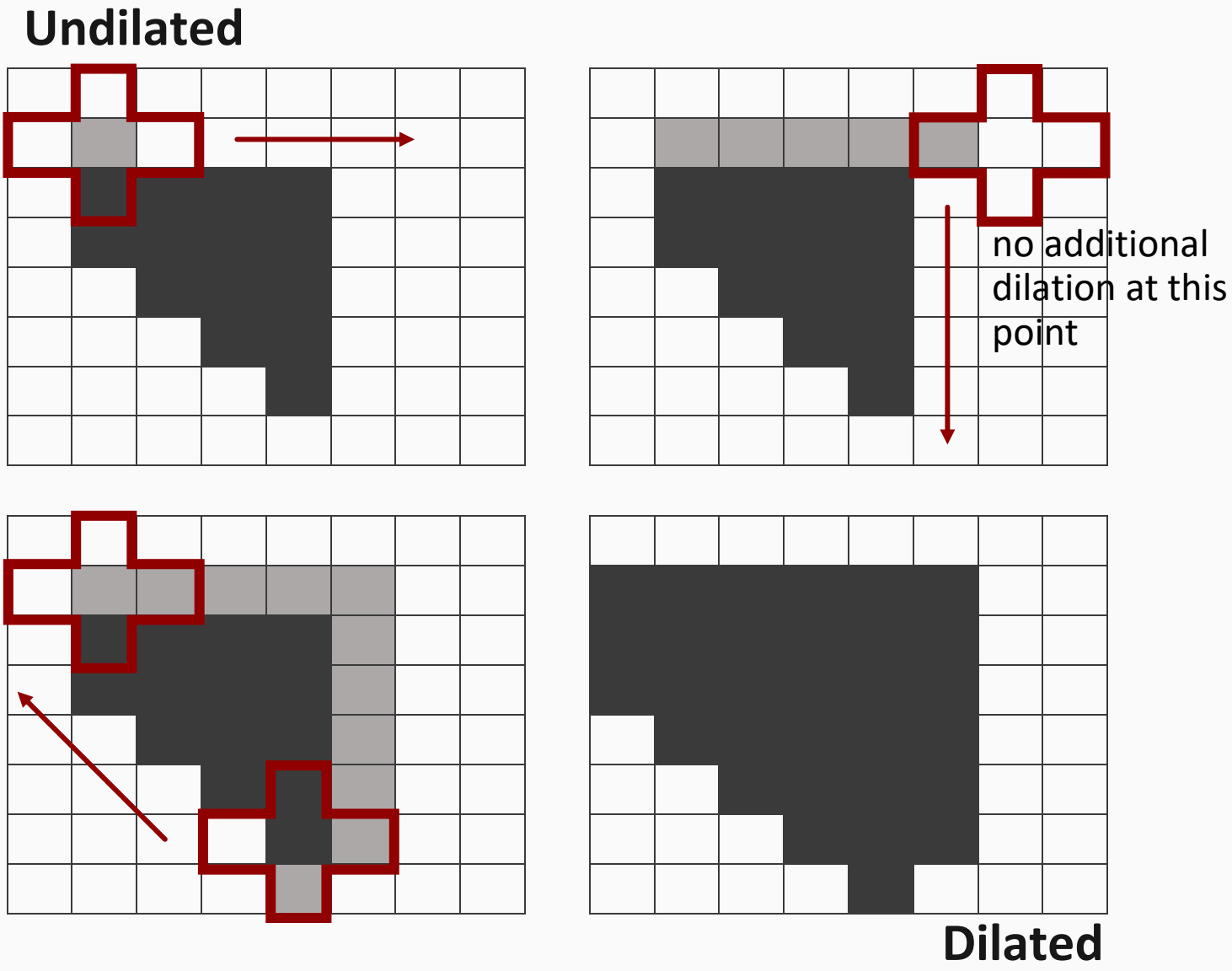
**Dilation** is a technique that **adds** a boundary of pixels around a bright object within an image.

**Erosion** is a technique that **removes** a boundary around a bright object within an image.

The above technique require an applied **structuring element**; a shape applied to the object:

The chosen **structuring element** traces the image to perform **dilation**:

The **center pixel** of the structuring element will be colored by the darkest (**minimum**) of the neighbors within the structuring element. The process is repeated as the **structuring element traces the image**. Note in the **second quadrant** in the illustration below that no neighbors will be colored dark; the only dark-colored pixel within the neighbors' proximity is one that was colored by the operation and *not* from the original image. The process continues around the entire image:

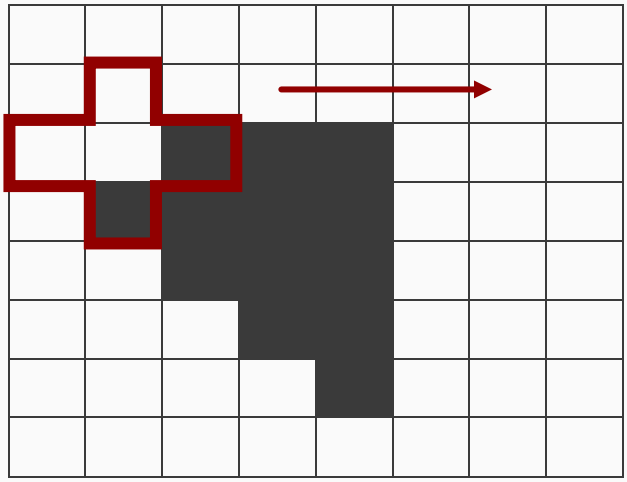


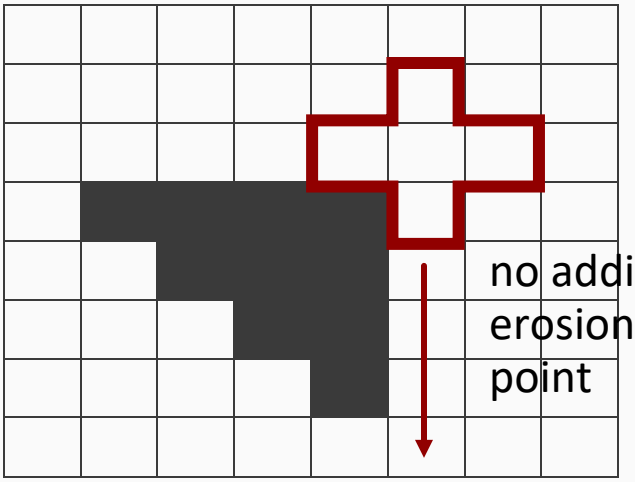

The edges of an image are the focal point within the operation above; the result is a **dilated** image.

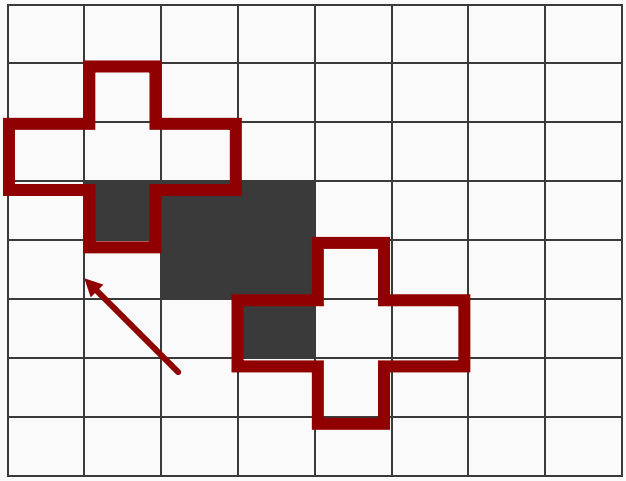
**Erosion** is simply the opposite operation of **dilation**. It is important to note however, that performing **Erosion** on an object subsequent to **dilation** will not necessarily reconstruct the object as before. The same property applies when **dilating** an object that was previously **Eroded**.

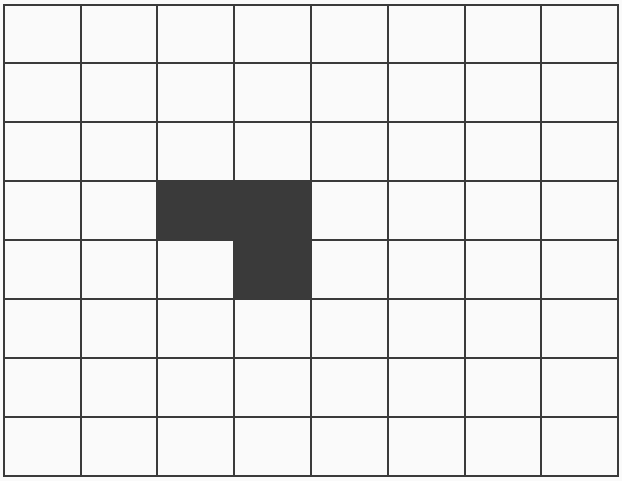
**Erosion** performs the operation of coloring the center pixel the **lightest** (**maximum**) of the neighbors within the **structuring element**. The object proceeds to be traced as before until convergence:

## Uneroded





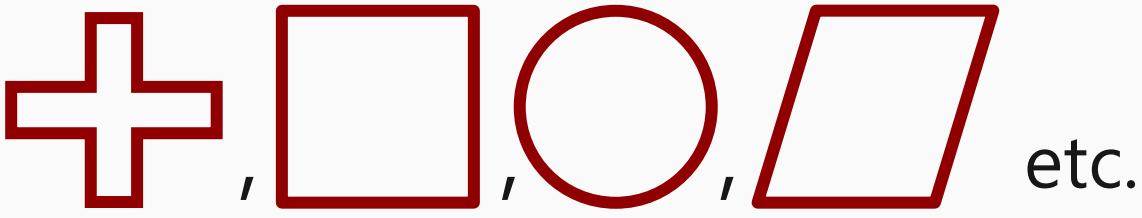



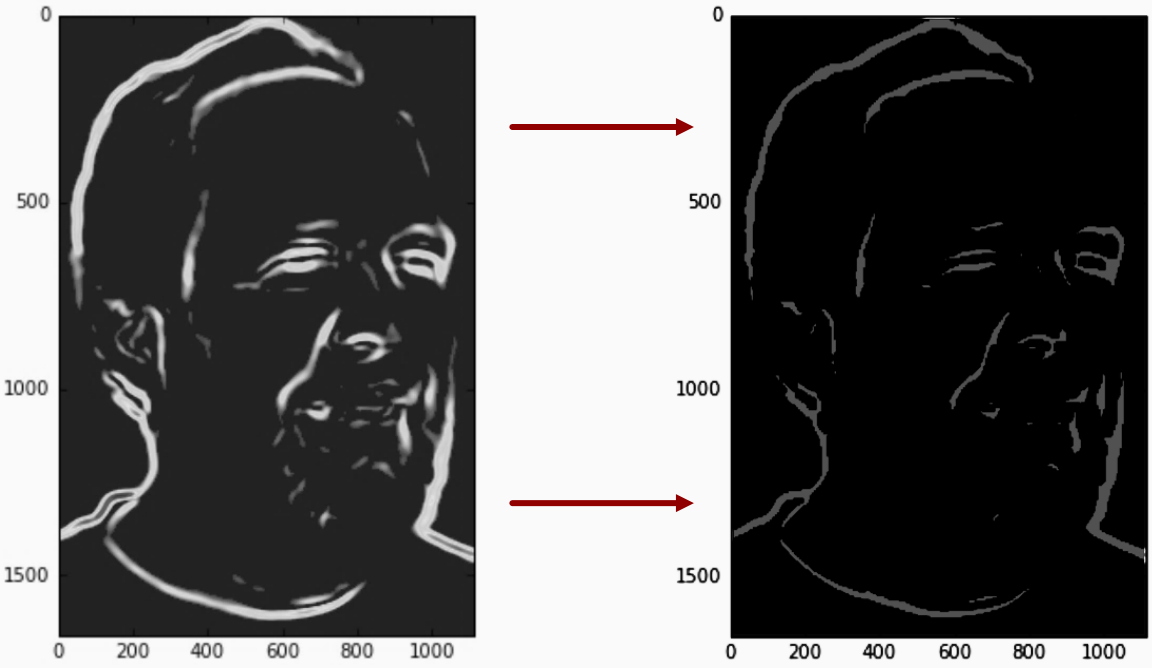


## Eroded

The edges of an image are the focal point within the operation above; the result is an **eroded** image.

Depending on the object and the desired application of **Dilation** and **Erosion**, multiple structuring elements are available for the above operations:






The left image is an example of an image that has undergone **Sobel Edge Detection**. The resulting image has been **Eroded** using the techniques above. Note the brightness contrast is *not* relevant to the operation; the refinement of excess edges are a direct product of **Erosion**.

https://robhentac.wordpress.com/2010/09/21/morphological-operations/
http://homepages.inf.ed.ac.uk/rbf/HIPR2/open.htm

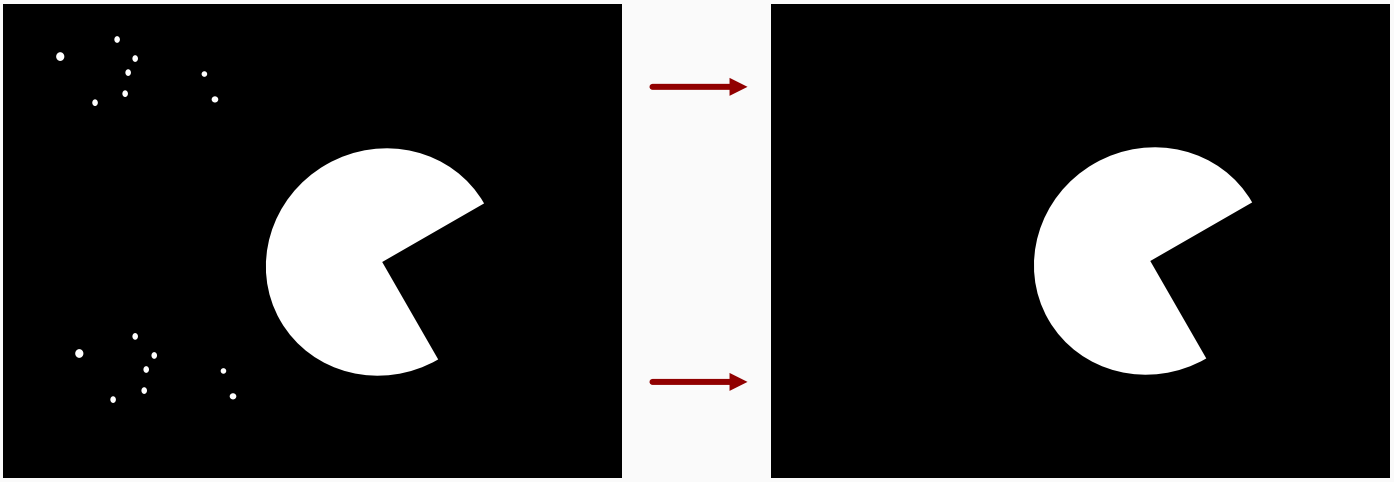
# working with images

## Opening and Closing

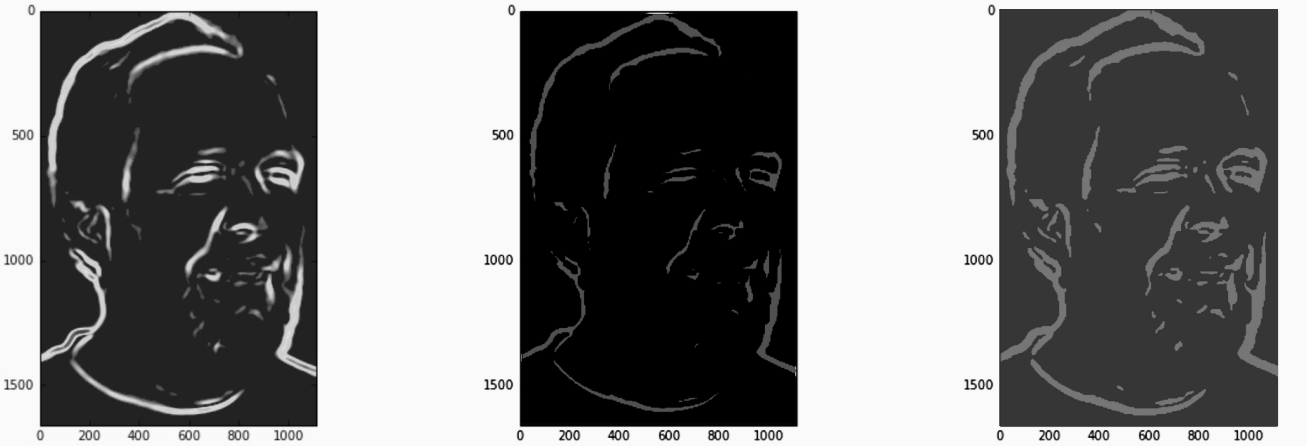
The operation of **Opening** is simply **erosion** followed by **dilation**.

**Opening** and **Closing** are useful techniques to eliminate certain types of noise or certain types of shapes depending on the **structuring element** chosen in application.

In theory, **erosion** will effectively remove the white noise seen in the image below. The outer layer of Pacman' will equally be removed. However, when the operation is followed by **dilation**, the outer layer of Pacman' will be restored without any of the previously removed white noise:



The following illustrates the **opening** operation with the working subject image:

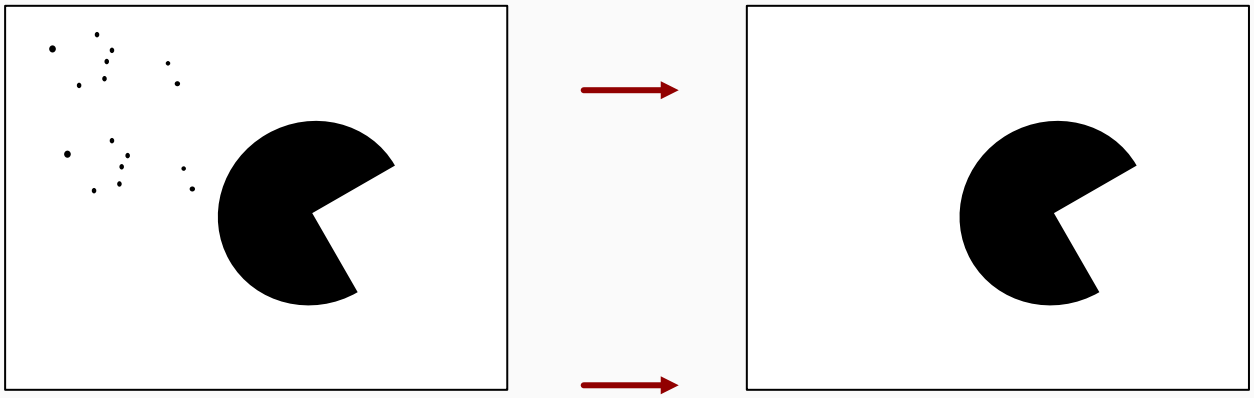



**Sorbel Edge Detector** **Erosion** **Dilation**

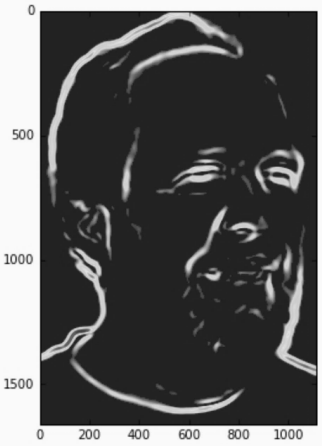
**Opening**

The operation of **Closing** is simply the opposite of **opening**; **dilation** followed by **erosion**.

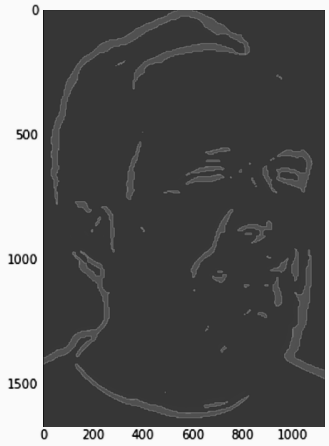
In theory, **dilation** will effectively and the white space in place of the pepper noise seen in the image below. The outer layer of Pacman' will equally be expanded, opposite when **opening** will remove the edge. When the operation of **dilation** is followed by **erosion**, the outer layer of Pacman' will be restored without any of the previously additional pepper noise:



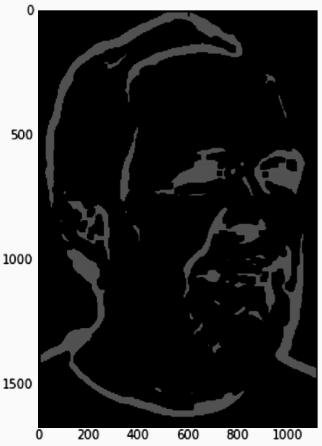
The following illustrates the **opening** operation with the working subject image:




**Sorbel Edge Detector**




**Dilation**




**Erosion**

**Closing**

# image analysis lab

### Image analysis applications in Data Science

Images exists in all areas of workspace, personal life and social context. Images are a widely used unstructured data type, and analysis and preparation of images is a common data science task.

### Explore an image

In exploring the properties of a greyscale image, images are cached as files in the working directory:

```
import os
import urllib
import urllib2
url = "https://github.com/MicrosoftLearning/Applied-Machine-
Learning/raw/master/Labs/Faces/Steve.jpg"
fileobject = urllib2.urlopen(url)

from scipy import misc
steve = misc.imread(fileobject, mode = 'L')
```

The image will be stored as ordinary Numpy array of unsigned 8-bit integers of dimension 1661x1113.

```
type(steve)
```

numpy.ndarray

```
steve.dtype
```

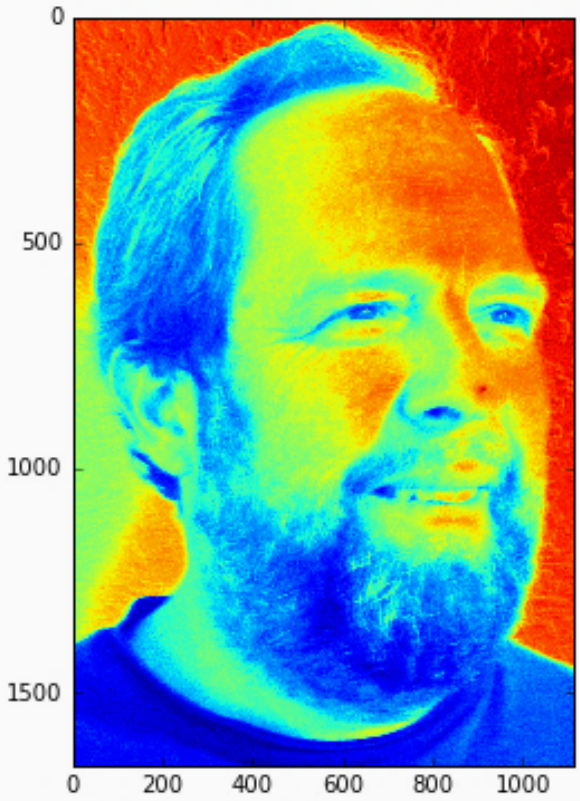
dtype('uint8)

```
steve.shape
```

(1661,1113)

Plotting a Numpy array image using the imshow function from the matplotlib package:

```
%matplotlib inline
def plot_im(im):
    import matplotlib.pyplot as plt
    import numpy as np
    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.imshow(np.array(im).astype(float))
    return 'Done'
plot_im(steve)
```

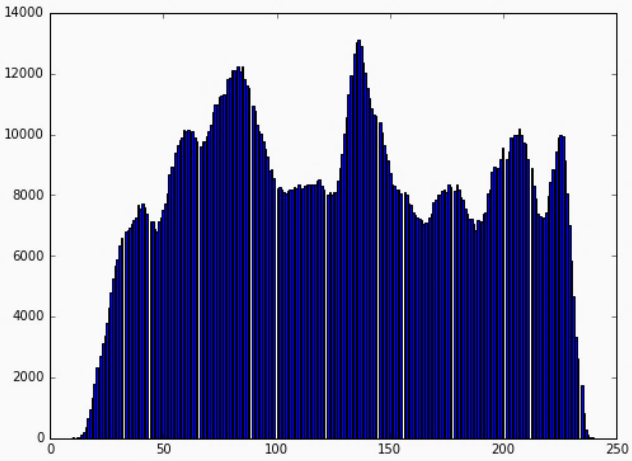


### Histograms and Equalization

It is often the case that a raw image does not have the favorable statistical properties required for further analysis. For example, poor contrast in the image can make it difficult to detect features. Histogram equalization is a widely used method for improving the properties of an image.

Ideally, the histogram of the image should be close to a uniform distribution. that the Numpy **flatten** method is applied to the array. This method removes the dimension attribute, creating a 1-D array:

```
def hist_im(im, bins = 256):
    """ Display histogram of flattened image"""
    import matplotlib.pyplot as plt
    import numpy as np
    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.hist(np.array(im).flatten(), bins = bins)
    return 'Done'
hist_im(steve)
```

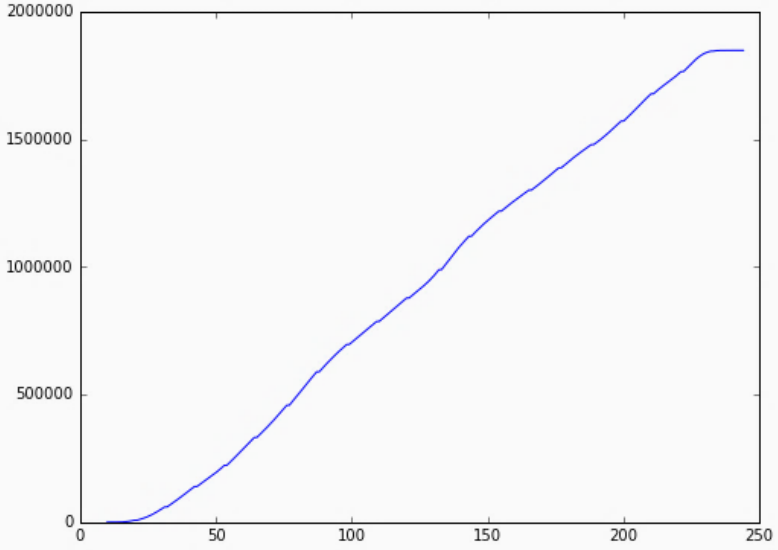



Examining the **Histogram** illustrates a **non-uniform distribution** of the image above.

Another tool for visualizing image statistics is the **cumulative distribution function** (**CDF**) plot:

```
def cdf_im(im, bins = 256):
    """Display cumulative distirbution of flattened image"""
    import matplotlib.pyplot as plt
    import numpy as np

    y, x = np.histogram(np.array(im).flatten(), bins = bins)
    y = y.cumsum()

    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.plot(x[:256], y)
    return 'Done'
cdf_im(steve)
```

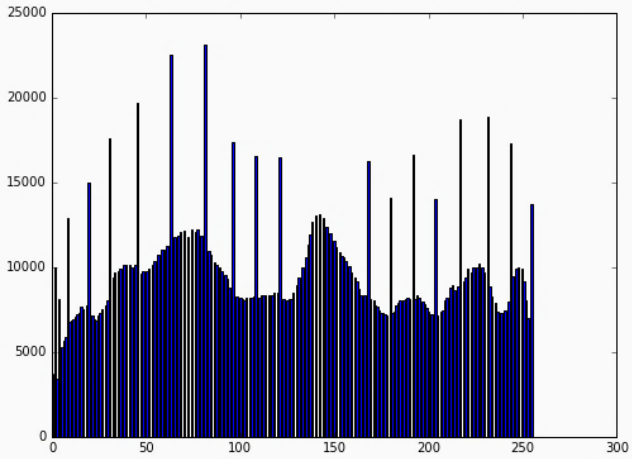
The **CDF** of an image with uniformly distributed pixel value is a straight line; the above **CDF** is curved, particularly at the ends.



**Histogram equalization** is often used to improve the statistics of images. In simple terms, the histogram equalization algorithm attempts to adjust the pixel values in the image to create a more uniform distribution. The code below uses a simple **linear interpolation** method on the image histogram to equalize the image:

**Note:** The histogram will appear to have spikes, which are the result of the binning, and not a problem with the equalization.

```
def image_equalize(im, num_bins = 256):
    """Function to equalize the image"""
    import numpy as np
    ## Compute the histogram of flattened image
    imhist, bins = np.histogram(im.flatten(), num_bins, normed=True)

    cdf = imhist.cumsum() #cumulative distribution function
    cdf = 255 * cdf / cdf[-1]  # normalize

    ## Interpolate to equalize the image
    out = np.interp(im.flatten(), bins[:-1], cdf)
    return out.reshape(im.shape)

steve_eq = image_equalize(steve)
hist_im(steve_eq)
cdf_im(steve_eq)
```

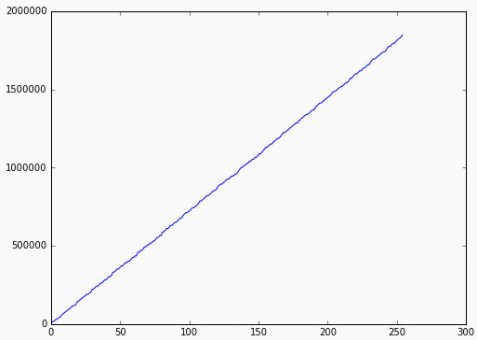



The histogram of the equalized image is more uniform in appearance.

Also displayed is the **CDF** of the equalized image:

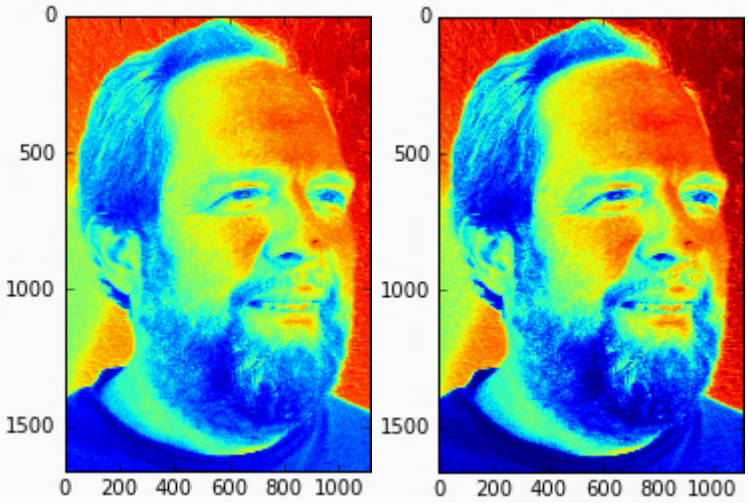
The **CDF** is more linear than before. Both the histogram and CDF indicate that the statistics of the image have improved.

A comparison of the actual images are illustrated below between the **unequalized → equalized** image respectively:




```
def plot_im2(im1, im2):
    import matplotlib.pyplot as plt
    import numpy as np
    fig, ax = plt.subplots(1, 2, figsize = (6,12))
    ax[0].imshow(im1)
    ax[1].imshow(im2)
    return 'Done'
plot_im2(steve, steve_eq)
```

The **original image** is on the **left** and the **equalized image** is on the **right**. Notice the improved contrast in the equalized image.

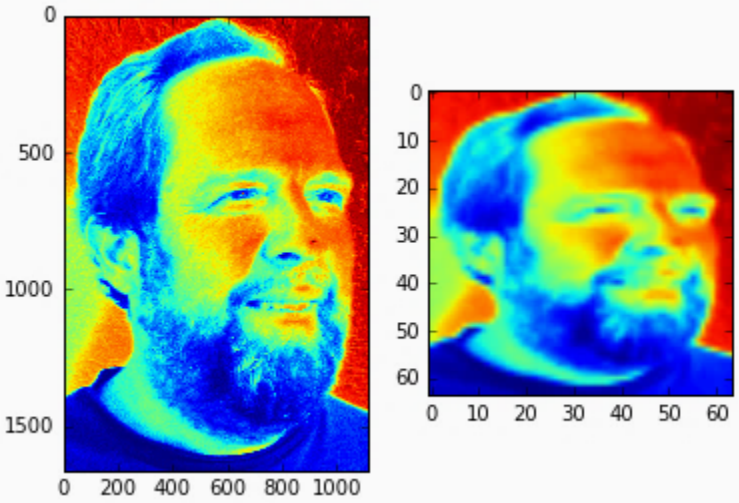


### Image Manipulation

After examining the properties of images, basic image manipulation is applied where appropriate.

**Resizing** is a common form of image manipulation. Images are often resampled to a smaller size to reduce the amount of data which must be processed. The **scipy.misc.imresize** method provides a convenient way to resize an image:

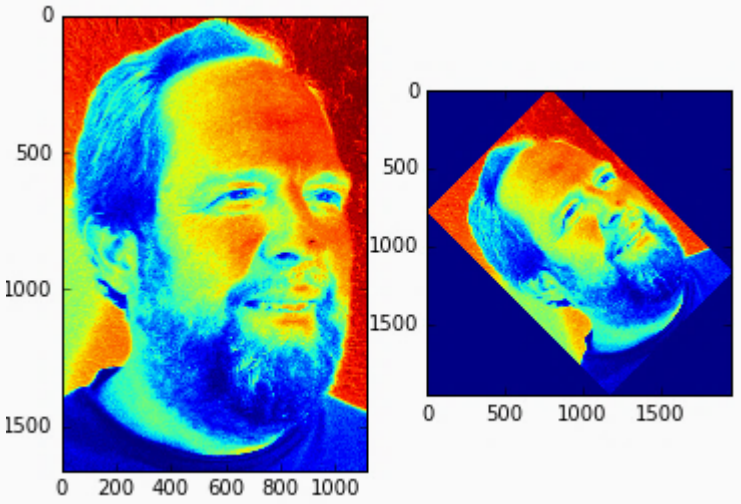
```
def resize(im, size = (64, 64)):
    import scipy.misc as mc
    return mc.imresize(im, size)
plot_im2(steve_eq, resize(steve_eq))
```




The reduced image output is more course and granular than the original. However, the integrity of the original image is maintained considering it was reduced by a factor of >500.

**Rotation** is performed by **pixel interpolation**.
The **scipy.ndimage.interpolation.rotate** method is to rotate the image by 45°:

```
def rotate(im, angle = 45):
    from scipy.ndimage import interpolation
    return interpolation.rotate(im, angle)
plot_im2(steve_eq, rotate(steve_eq))
```




The rotated image on the right is a duplicate of the image on the left. Notice that the area around the rotated image has been backfilled with zero values.

### Working with a list of images

Typically multiple images are analyzed as a group. This group of images can be stored as a list object in Python:

```
import os
import urllib
import urllib2

baseUrl = "https://github.com/MicrosoftLearning/Applied-Machine-
Learning/raw/master/Labs/CarrotImages/Carrot"

image_list = []
for i in range(1,10):
    url = baseUrl + str(i) + ".JPG"
    fileobject = urllib2.urlopen(url)
    im = misc.imread(fileobject)
image_list.append(im)
```

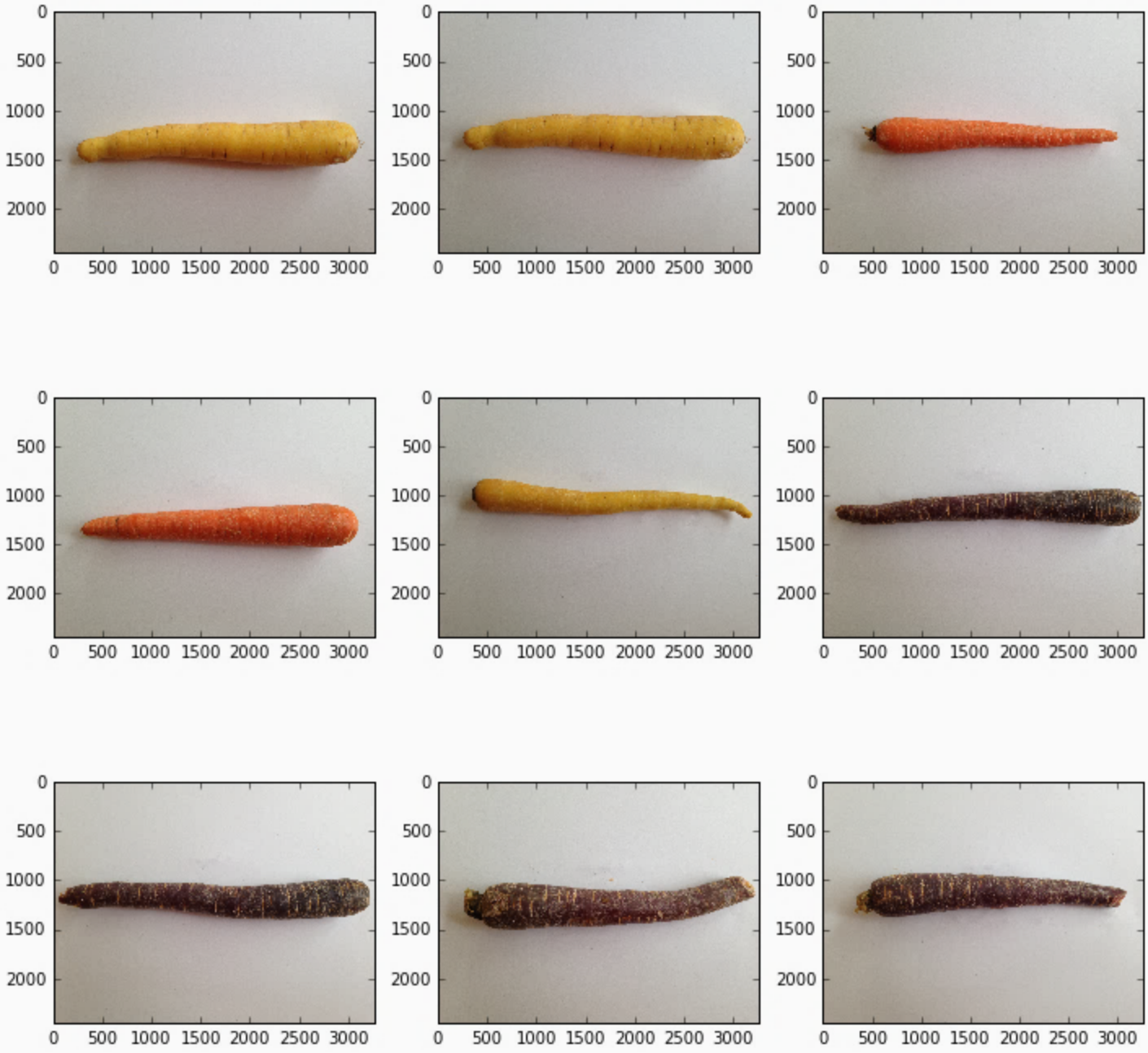
The list contains 9 images and the numpy image array has dimensions of (2448, 3264, 3):

```
len(image_list)
image_list[0].shape
```

The Numpy image array has three dimensions. These are color images, with red, green and blue layers, each stored as 2x2 arrays.

The code in the function below integrates over the images in the list and displays each one into an array of axes (images of carrots):

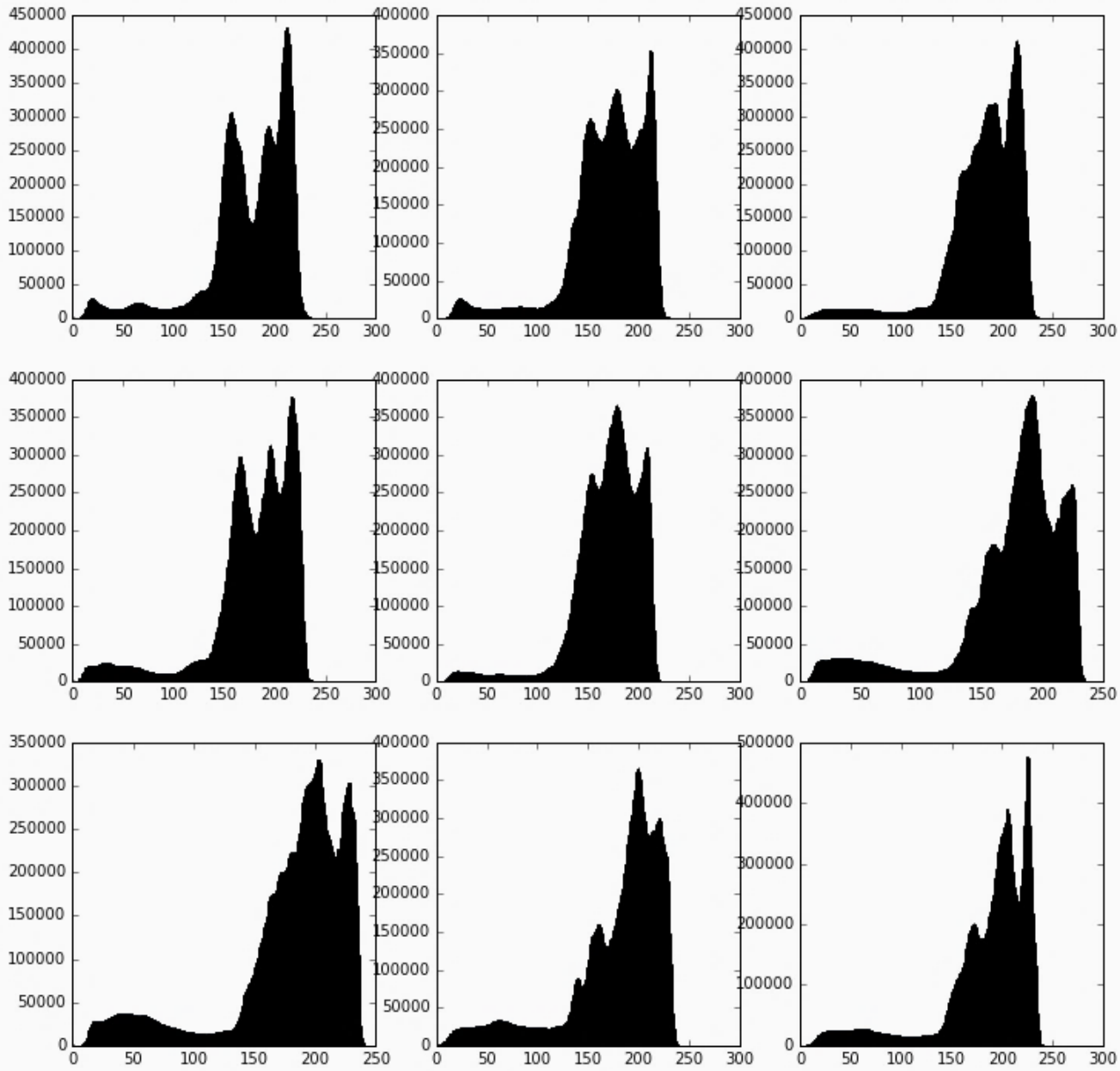
```
def plot_carrot(im_list):
    import matplotlib.pyplot as plt
    fig, ax = plt.subplots(3, 3, figsize = (12,12))
    j = -1
    for i, image in enumerate(im_list):
        k = i % 3
        if k == 0: j += 1
        ax[j,k].imshow(image)
    return 'Done'
plot_carrot(image_list)
```




The 9 images of carrots which have different shapes, several colors and different orientations.

The code below loops over the list of images and plots each histogram into an array of axes:

```
def hist_carrot(im_list, bins = 256):
    """ Display histogram of flattened image"""
    import matplotlib.pyplot as plt
    import numpy as np
    fig, ax = plt.subplots(3, 3, figsize = (12,12))
    j = -1
    for i, image in enumerate(im_list):
        k = i % 3
        if k == 0: j += 1
        ax[j,k].hist(np.array(image).flatten(), bins = bins)
    return 'Done'
hist_carrot(image_list)
```



Each image has a long left tail, arising from the background.

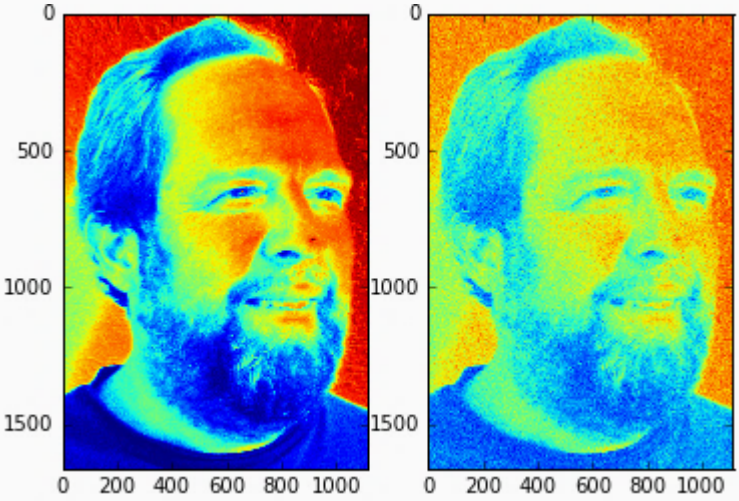
### Image Filtering

Images have been loaded, properties explored, and basic processing performed. Next will apply specific filters to the images. **Filters** are applied to images in order to either enhance aspects of the image or to remove undesired properties of the image (such as noise).

Gaussian or white noise is added to the face image. The code below does the following:

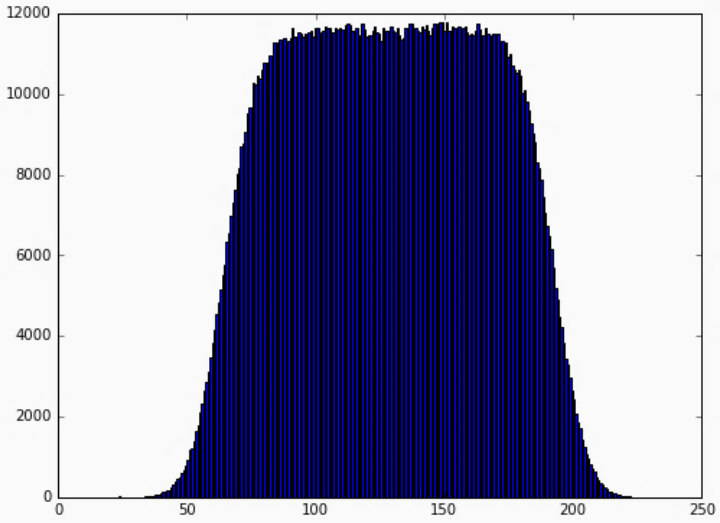
- Generate a one dimensional array of Gaussian noise.
- Shape the noise array to match the image.
- Add the noise array to the image.
- Ensure there are no image values less than zero.

```
def add_noise(im, mean = 128, sd = 20):
    """Adds Gausian noise to the image"""
    from numpy.random import normal
    import numpy as np
    im_a = np.array(im)
    shape = im_a.shape
ng = normal(loc = mean, scale = sd, size
= shape[0] * shape[1])
    ng.shape = shape
    ng = np.divide( np.add(ng, im_a), 2.0)
    ng[np.where( ng < 0 )] = 0
    return ng
steve_n = add_noise(steve_eq)
plot_im2(steve_eq, steve_n)
```




The original image in on the left and the noisy image is less distinct on the right.

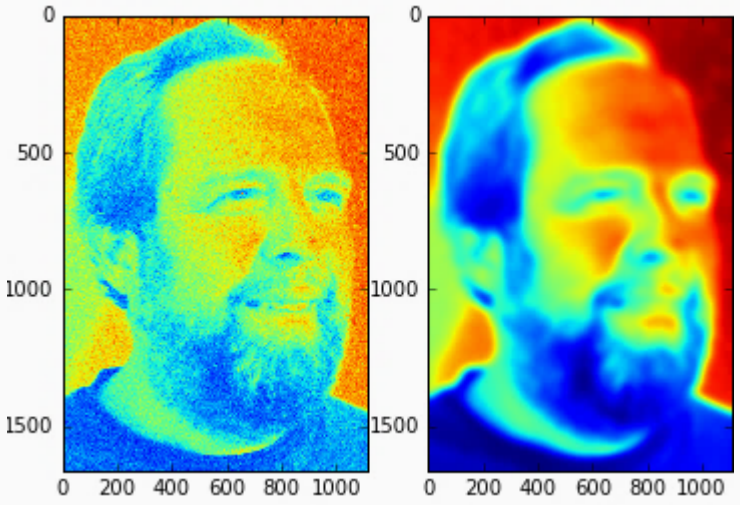
The Histogram (`hist_im(steve_n)`) of the noisy image is a smoother and more uniform. The range of pixel values is also limited. These changes are the result of adding white noise to the image, a process often referred to as 'pre-whitening':




Next applies a Gaussian filter. A **Gaussian filter** is a two dimensional filter using a Gaussian or bell-shaped curve kernel. In effect, the **Gaussian filter** is a smoothing filter. The span of the filter determines the degree of smoothing of the filter.

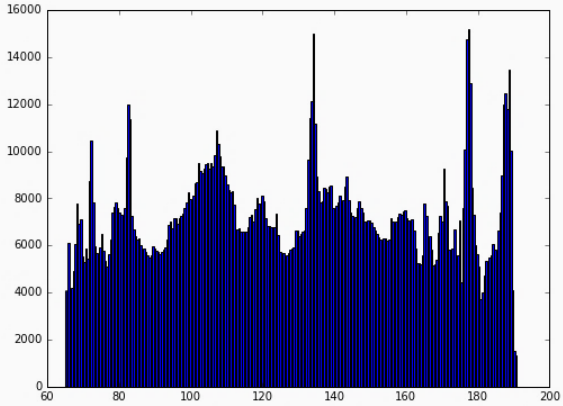
```
def gauss_filter(im, sigma = 16):
    from scipy.ndimage.filters import
gaussian_filter as gf
    import numpy as np
    im_a = np.array(im)
    return gf(im_a, sigma = sigma)
steve_g = gauss_filter(steve_n)
plot_im2(steve_n, steve_g)
```

The 2-d span of the filter is specified in pixels:




- The filtered image on the right has smoother or blurred features. For this reason, Gaussian filters are often called blurring filters.
- The image on the right does not exhibit the 'salt and pepper' noise of the image on the left.

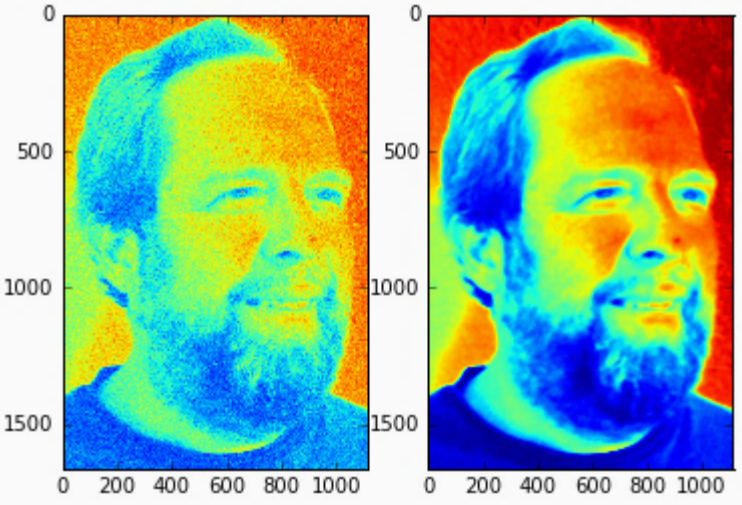
The Histogram (`hist_im(steve_n)`) of the filtered image is jagged when compared to the noisy image. This texture arises from removing the 'whitening' from the Gaussian noise.




Next applys a **median filter** on the image. Whereas, a Gaussian filter acts as a smoother, a median filter enhances edges and transitions.

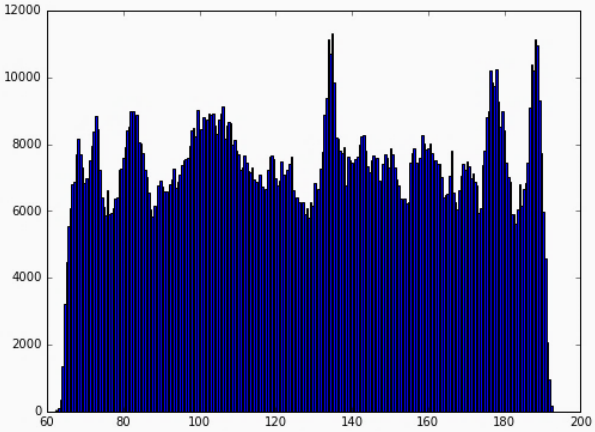
The code below applies a **median filter** to the image. The median filter kernel is a rectangular patch with a span specified as the filter size:

```
def med_filter(im, size = 16):
    from scipy.ndimage.filters import
median_filter as mf
    import numpy as np
    im_a = np.array(im)
    return mf(im_a, size = size)
steve_m = med_filter(steve_n)
plot_im2(steve_n, steve_m)
hist_im(steve_m)
```




Comparing the filtered image on the right to the original noisy image on the left and to the Gaussian filtered image on the right:

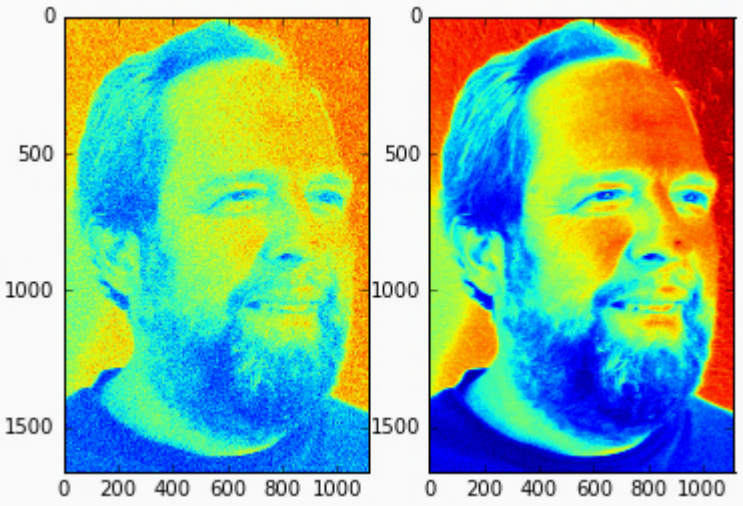
- The edges and transitions in the median filtered image are sharper and more distinct when compared to the noisy image or the Gaussian filtered image on the left.
- Regions between edges in the median filtered image are more uniform looking and do not have the 'salt and pepper' look of the noisy image on the right.
- The filtering has smoothed the histogram slightly.



### Using a Different Span

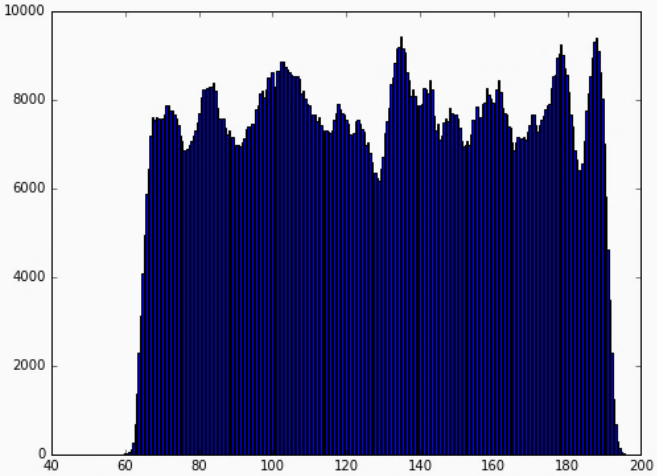
Applying a Gaussian filter to the noisy image with a different span (**sigma value = 2**) and visualizing the result:

```
def gauss_filter2(im, sigma = 2):
    from scipy.ndimage.filters import
gaussian_filter as gf
    import numpy as np
    im_a = np.array(im)
    return gf(im_a, sigma = sigma)
steve_g = gauss_filter2(steve_n)
plot_im2(steve_n, steve_g)
```




The original noisy image and the Gaussian filtered reveal:

- The filtered image shows less salt and pepper noise than the original.
- Compared to the first filtered image, the features in the new filtered image are less blurred.




Next will **pre-whiten** the carrot images by adding Gaussian noise. The process is the same as before.

The code below iterates over the images in the list and adds Gaussian noise to each image in turn:

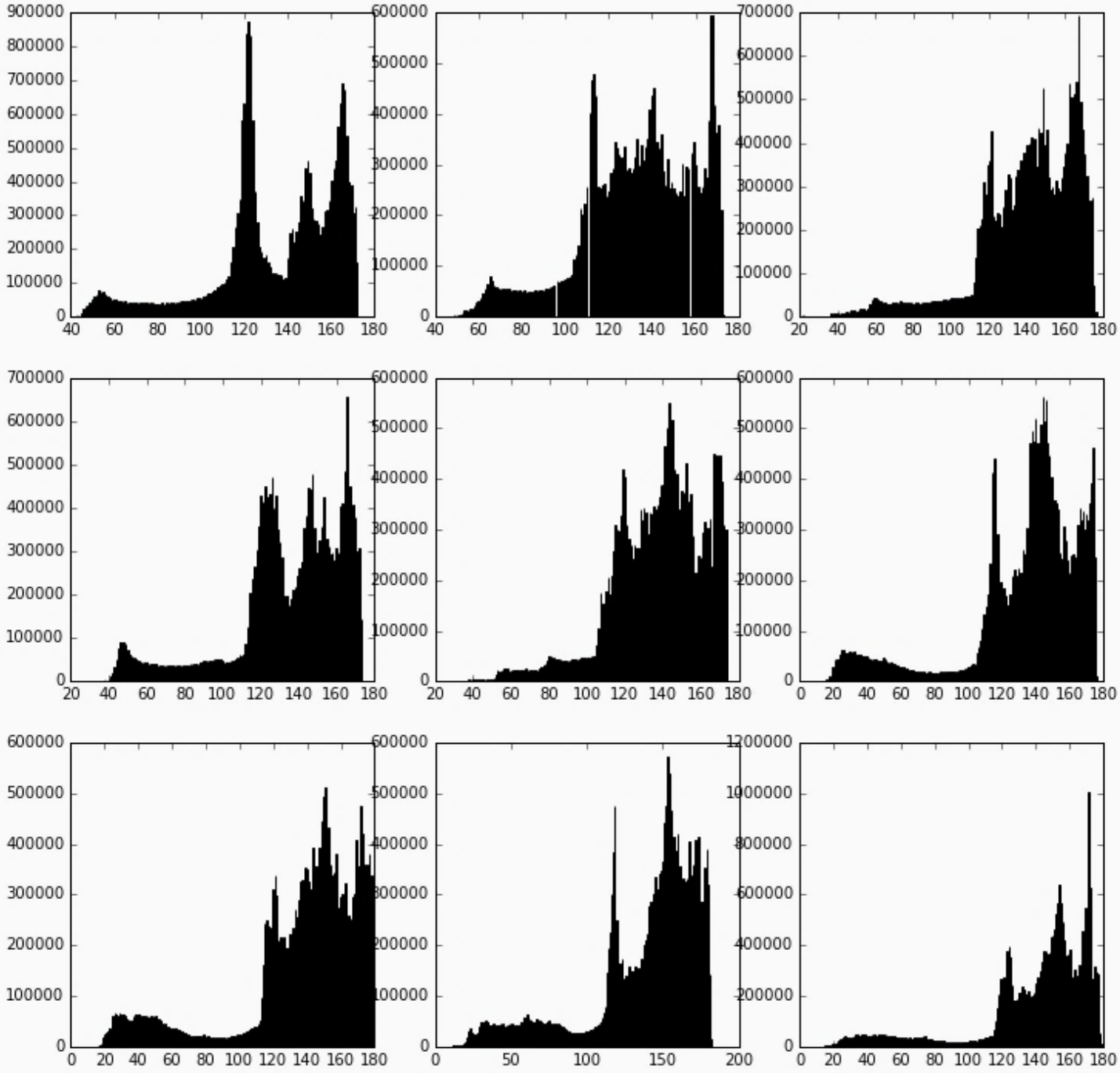
```
def pre_white(im_list, mean = 0, sd = 20):
    """Adds Gausian noise to the image"""
    from numpy.random import normal
    import numpy as np
    out = []
    for image in im_list:
        shape = image.shape
        ng = normal(loc = mean, scale = sd, size = shape[0] * shape[1] * shape[2])
        ng.shape = shape
        ng = np.add(ng, image)
        ng[np.where( ng < 0 )] = 0
        ng *= 255.0 / np.amax(ng)  # normalize
        ng = ng.astype(np.uint8)
        out.append(ng)
    return out
carrot_filter = pre_white(image_list)
```

Next applies a Gaussian filter to the images. The code below iterates over the list of images and applies the filter to each image:

```
def gauss_filter(im_list, sigma = 20):
    from scipy.ndimage.filters import gaussian_filter as gf
    out = []
    for image in im_list:
        out.append(gf(image, sigma = sigma))
    return out
carrot_filter = gauss_filter(carrot_filter)
```

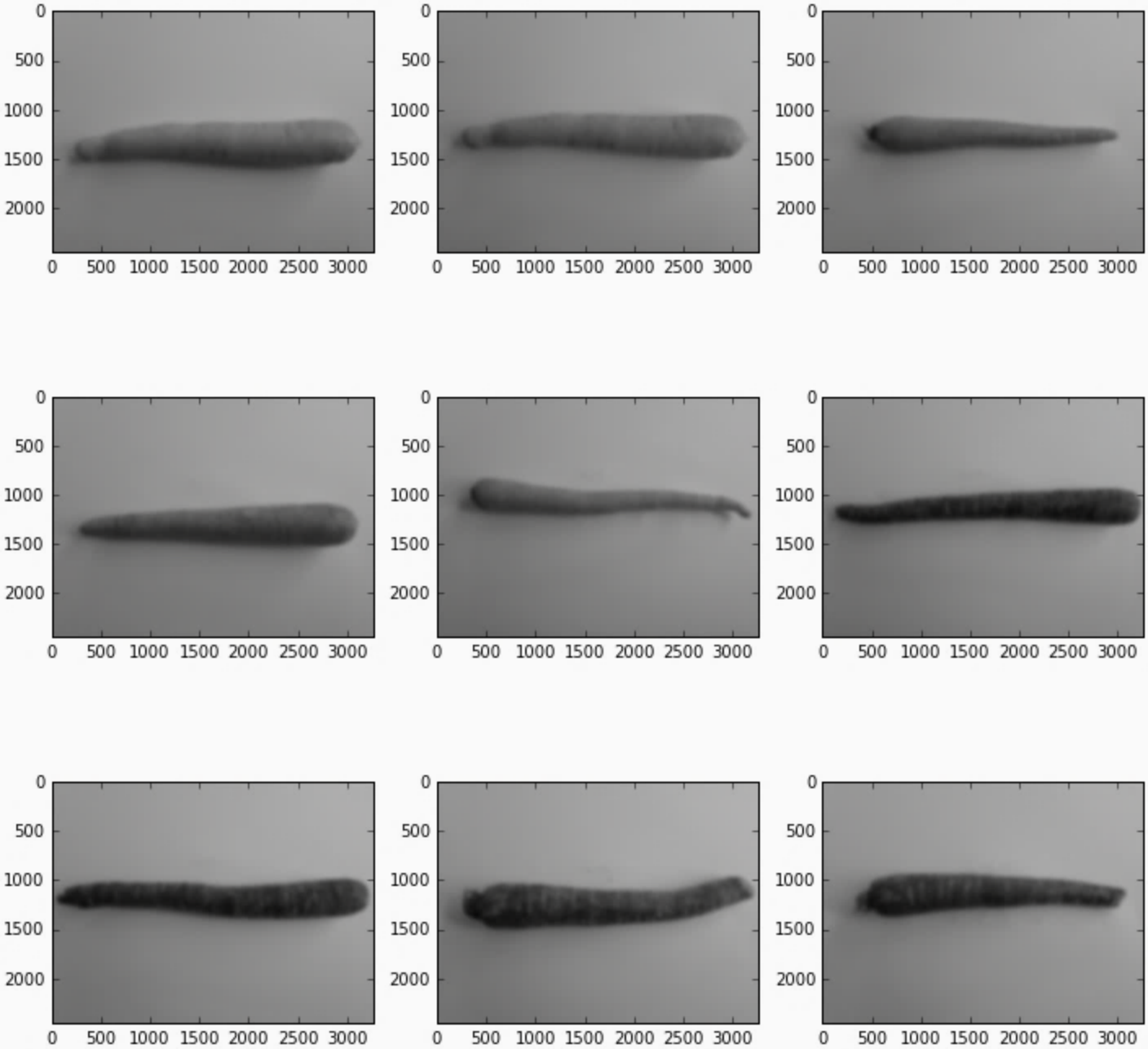
The **pre-whitened histograms** and images are displayed by executing the code below:

```
hist_carrot(carrot_filter)
plot_carrot(carrot_filter)
```



Comparing the histograms and images to the original histograms and images, note the following:

- The histograms of the pre-whitened images have a more jagged appearance.
- The pre-whitened images are blurred or softer and have lost considerable color contrast when compared to the original images.


0 500 1000 1500 2000 2500 3000

**Feature Extraction**

Images have now been explored, manipulated and filtered. Next extracts features from the processed images. **Feature extraction** is an indispensable step in preparing image data for machine learning.

First applies the **Sobel Edge Detection Algorithm**. The Sobel edge detection algorithm finds regions of the image with large gradient values in multiple directions. Regions with **high omnidirectional gradient** are likely to be **edges** or **transitions** in the pixel values.

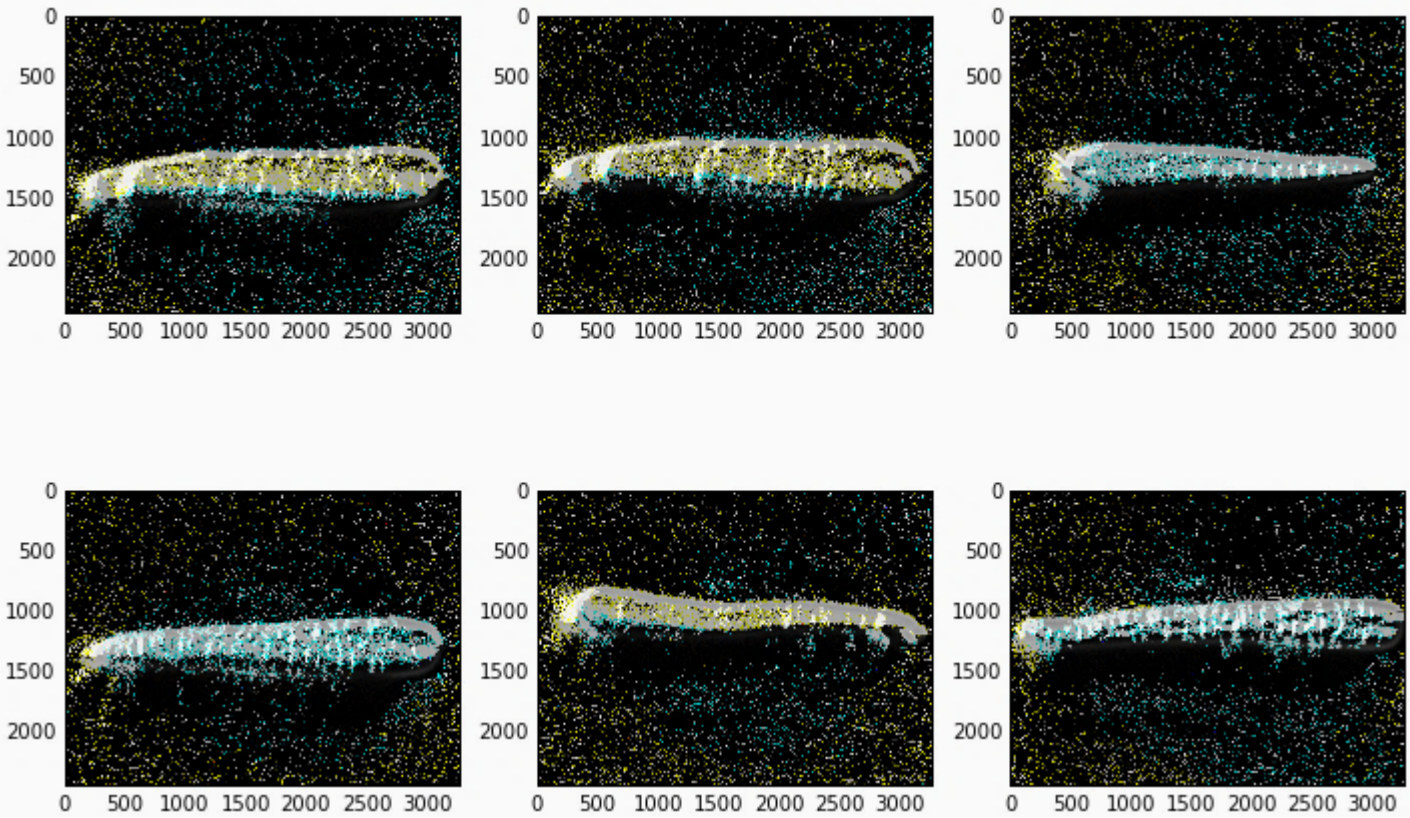
The code below applies the Sobel algorithm to a list of images, using these steps:

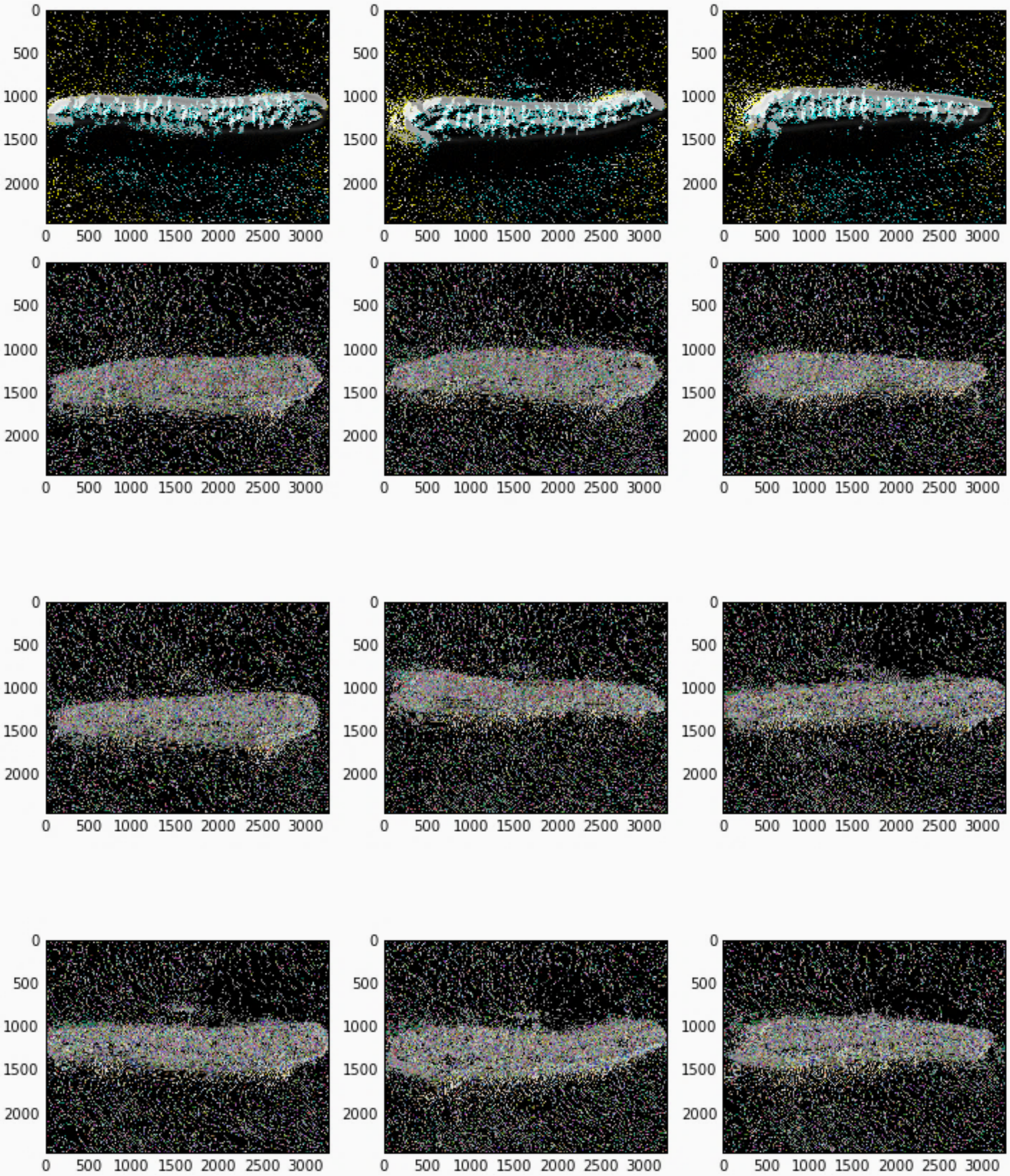
- The gradient in the x and y (horizontal and vertical) directions are computed.
- The magnitude of the gradient is computed.
- The gradient values are normalized.

```
def edge_sobel(im_list):
    from scipy import ndimage
    import numpy as np
    out = []
    for image in im_list:
        dx = ndimage.sobel(image, 1)  # horizontal derivative
        dy = ndimage.sobel(image, 0)  # vertical derivative
        mag = np.hypot(dx, dy)  # magnitude
        mag *= 255.0 / np.amax(mag)  # normalize (Q&D)
        mag = mag.astype(np.uint8)
        out.append(mag)
    return out
```

```
carrot_filter = edge_sobel(carrot_filter)#
```

```
plot_carrot(carrot_filter)
```

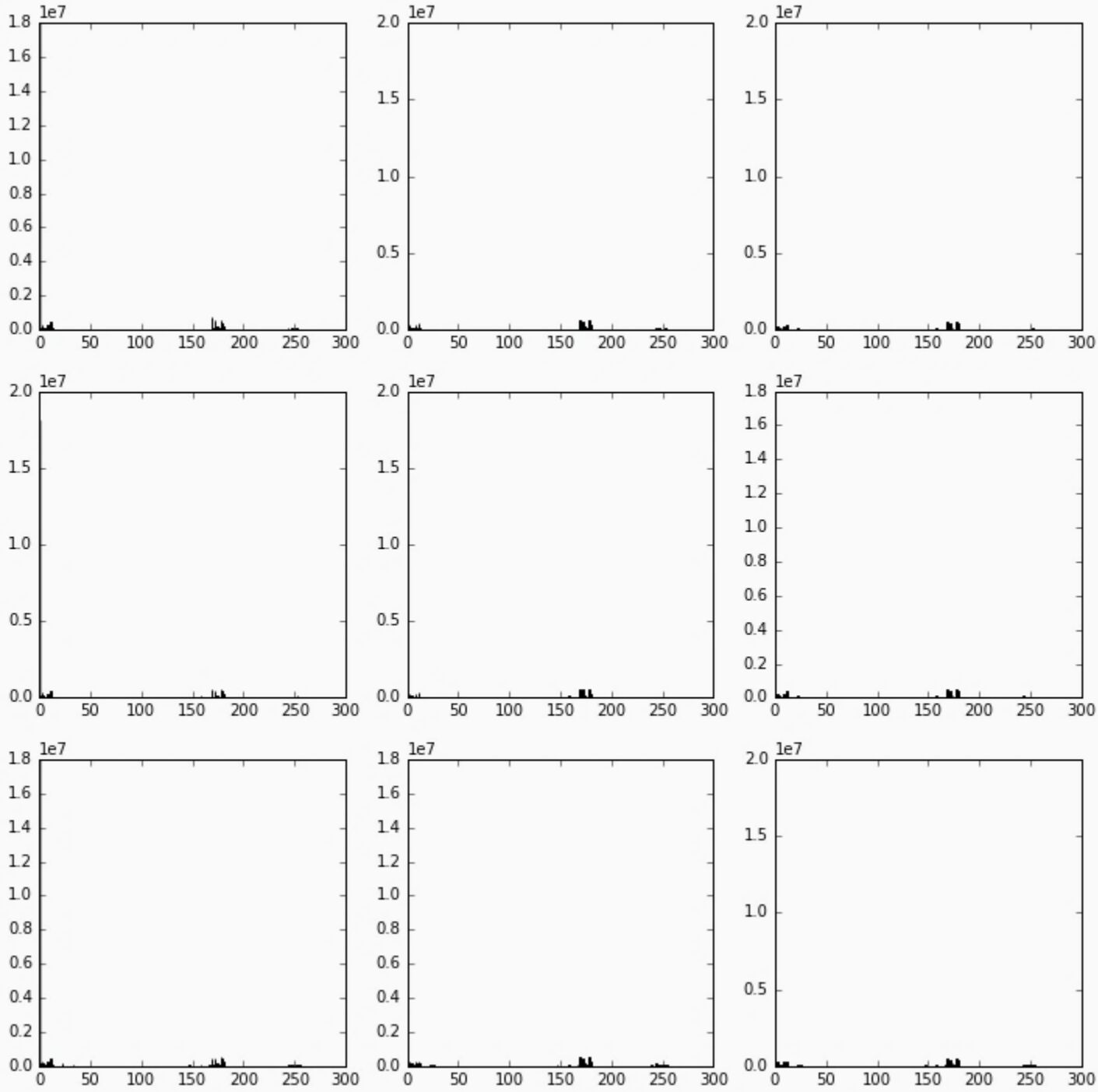





The carrots now appear in a skeletal form. The edges of the carrots are clear, along with the numerous cross marks on each carrot. These regions of the carrot image have high pixel gradient values.

The code in the cell below plots the histograms of the gradients of the images:

```
hist_carrot(carrot_filter)
```



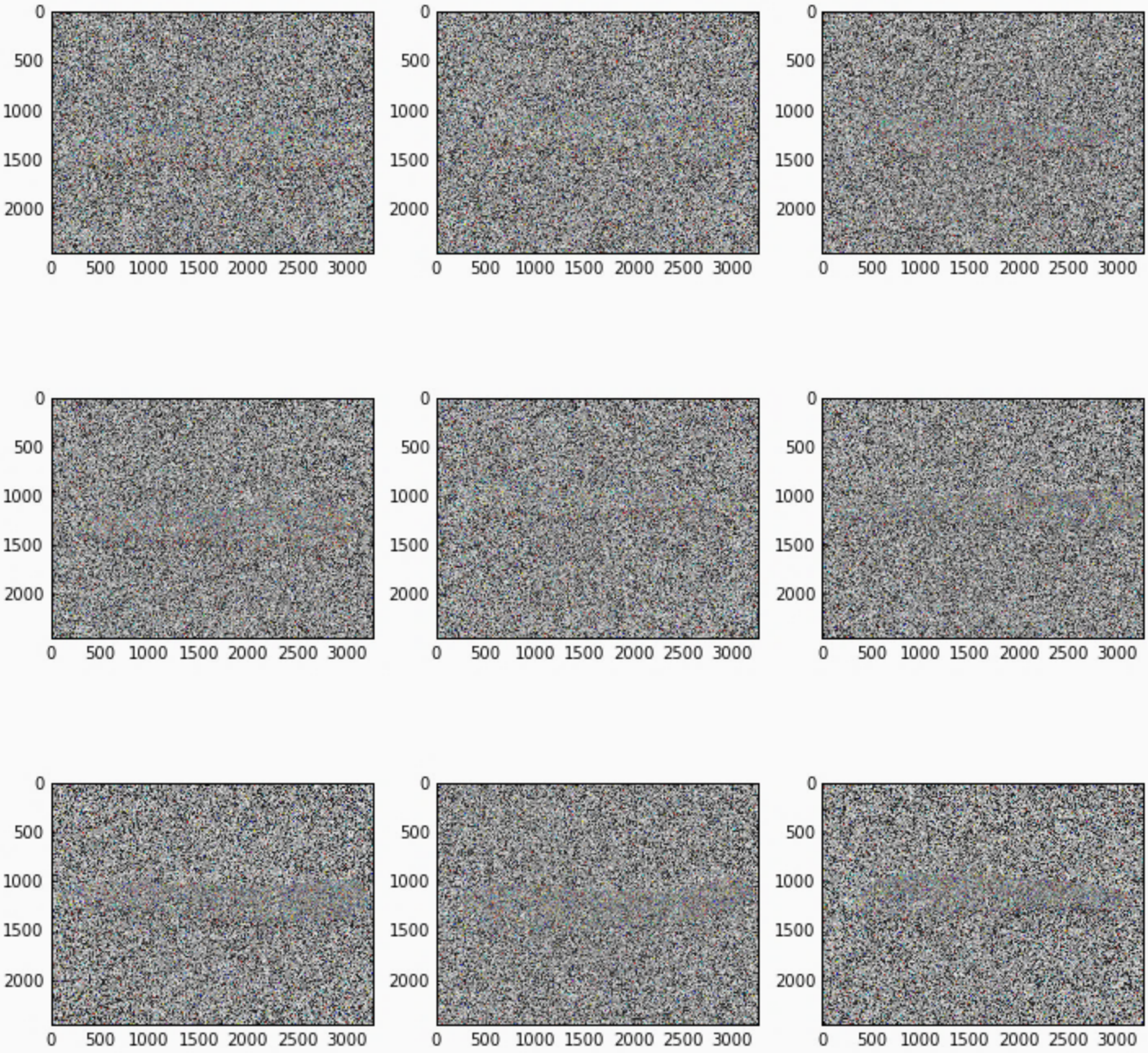

The pixel values in each histogram are in three groups:

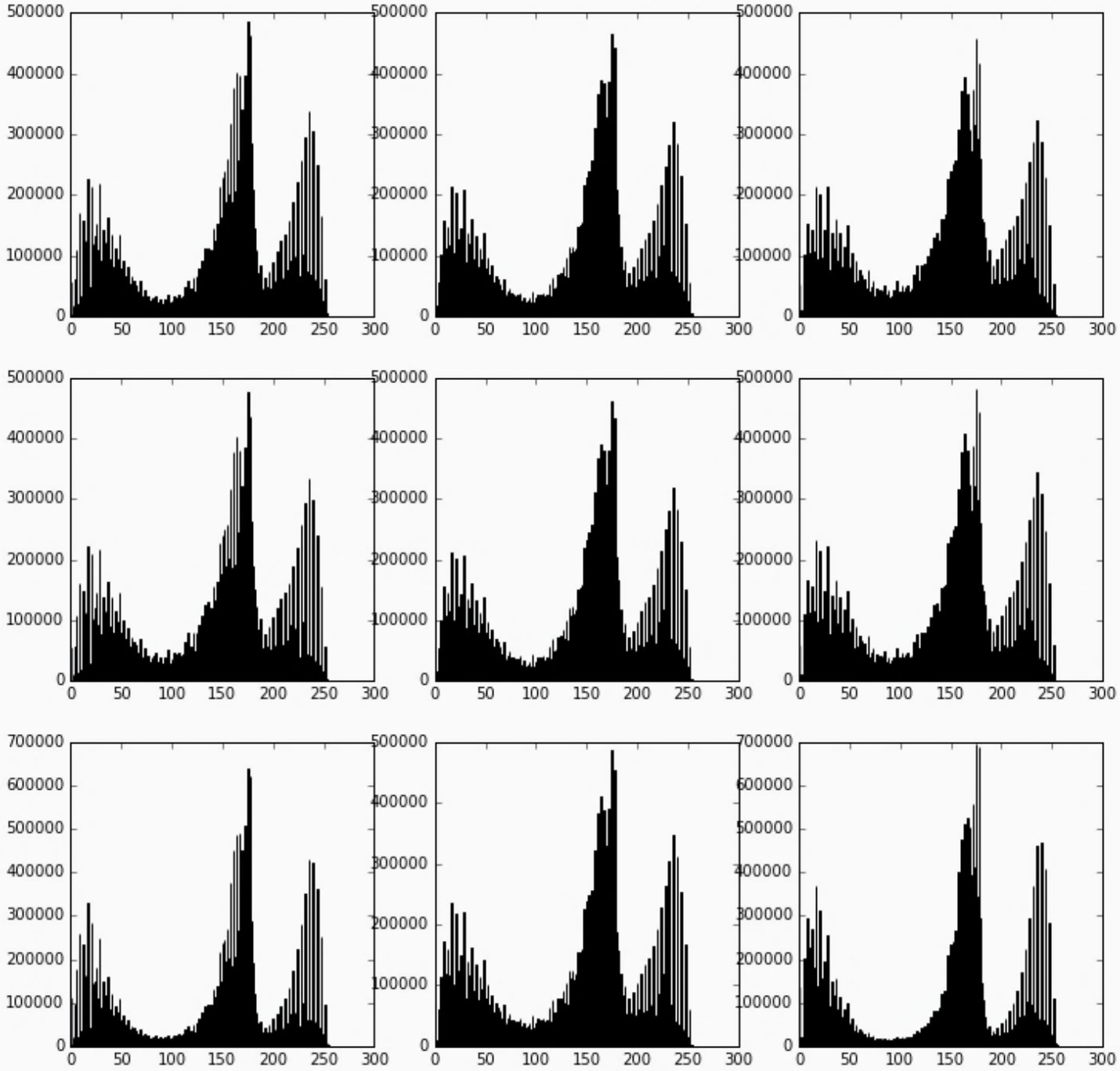
- The majority of the pixels have values at or near zero. These values represent regions of the image with little or no gradient.
- There is a group of pixels with values between 150 and 180, which likely represent the noise seen in the images of the gradient.
- The highest pixel values, above 220, are the edges of the image.

### Applying the Sobel Edge Detection Algorithm

To illustrate this difference pre-processing has on the algorithm output values, the Sobel edge detection algorithm is applied to the original carrot images:

- When compared to the edges computed from the prepared images, the edges of the carrots from the raw images are less distinct.
- When compared to the edges computed from the prepared images the edges of the carrots from the raw images are more noisy.




500000
400000
300000
200000
100000
0
0 50 100 150 200 250 300
700000
600000
500000
400000
300000
200000
100000
0

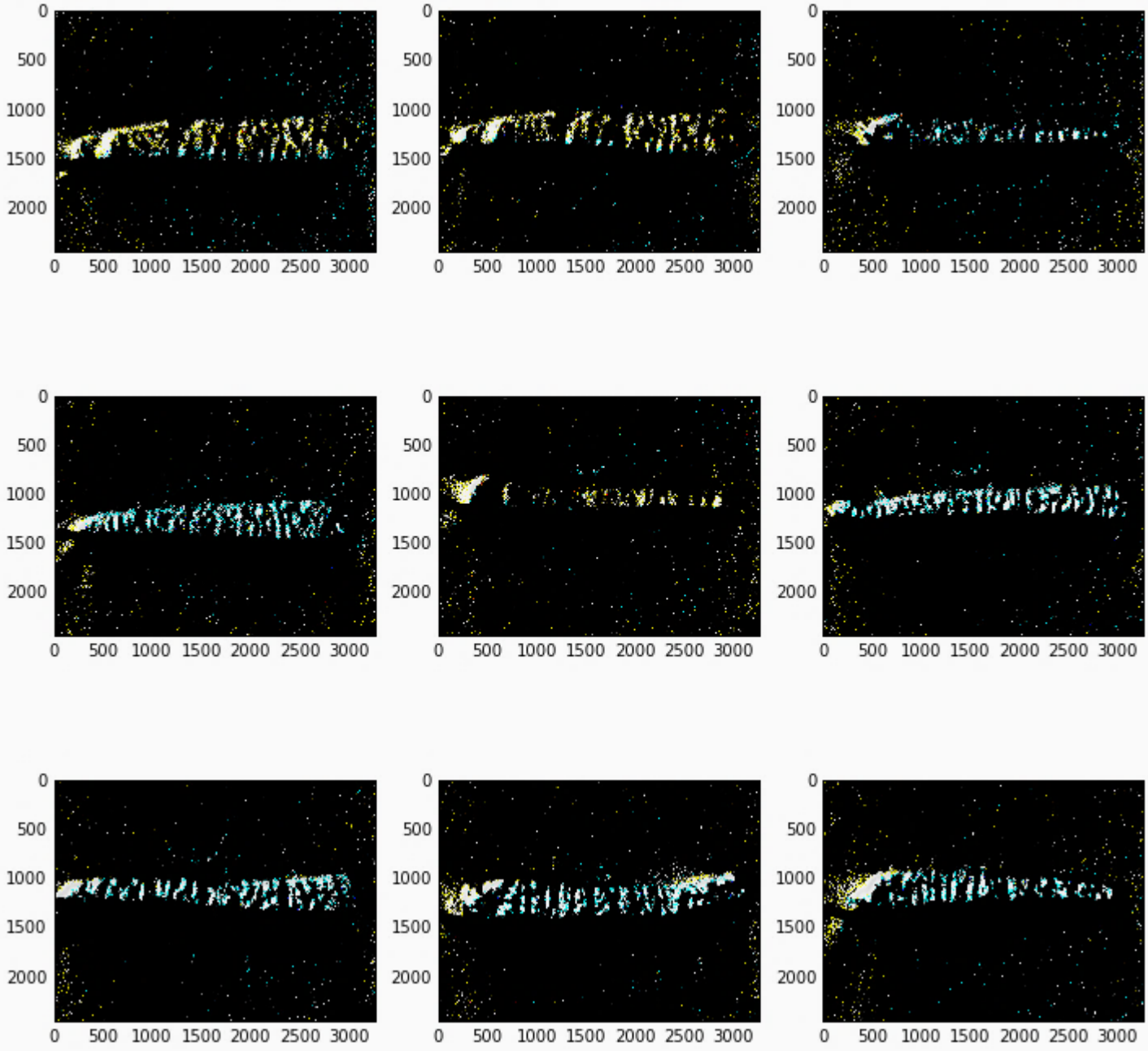
### Segmentation

**Image segmentation** is a process of segregating certain regions or segments. **Thresholding** of pixel values on the image is a simple segmentation algorithm.

The code below thresholds the pixel values in each image in a list. All pixel values below the threshold are set to zero. The code executes an image threshold at a pixel value of 200.

```
def threshold(im_list, thresh = 200):
    import numpy as np
    out = []
    for image in im_list:
        image[np.where( image < thresh )] = 0
        out.append(image)
    return out
carrot_filter = threshold(carrot_filter)
plot_carrot(carrot_filter)
```

The edges of the carrots remain and are more distinctive than before **thresholding**. However, there are shadows in the image which have also been enhanced.

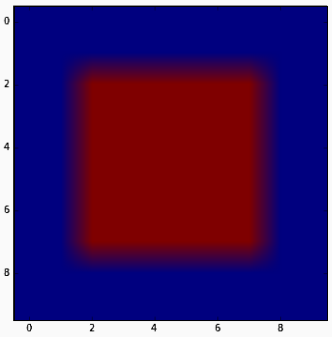


### Corner Detection

Another example of a feature extraction algorithm is **corner detection**. In simple terms, the **Harris Corner Detection Algorithm** locates regions of the image with large changes in pixel values in all directions. These regions are said to be corners. The Harris corner detector is paired with the **corner_peaks** method. This operator filters the output of the Harris algorithm, over a patch of the image defined by the span of the filters, for the most likely corners.

As a simple example of corner detection, the algorithm is applied to a square shape. The code below creates and displays a matrix containing a square:

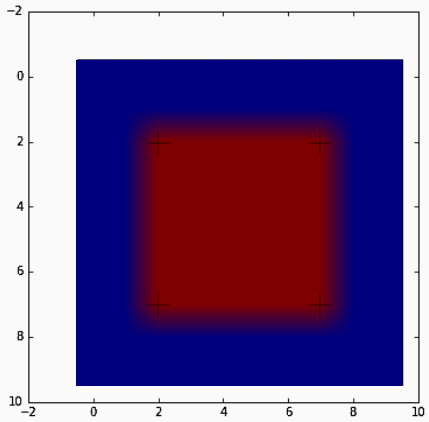
```
import numpy as np
square = np.zeros([10, 10])
square[2:8, 2:8] = 1
plot_im(square)
```



The code below, applies the Harris corner detector and the corner_peaks algorithm to the square:

```
from skimage.feature import corner_harris, corner_peaks
crn = corner_peaks(corner_harris(square), min_distance=1)
crn
```

```
Out[138]: array([[2, 2],
                 [2, 7],
                 [7, 2],
                 [7, 7]])
```



The corner detector has located four corners.

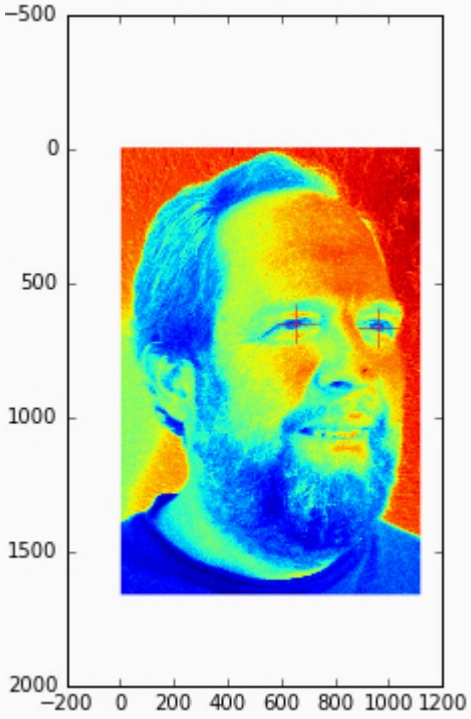
The code below plots the corners on top of the image:

```
def plot_harris(im, harris, markersize = 20, color = 'red'):
    import matplotlib.pyplot as plt
    import numpy as np
    fig = plt.figure(figsize=(8, 6))
    fig.clf()
    ax = fig.gca()
    ax.imshow(np.array(im).astype(float))
ax.plot(harris[:, 1], harris[:, 0], 'r+', color
 = color, markersize=markersize)
    return 'Done'
plot_harris(square, crn, color = 'black')
```

The four corners of the square have been correctly detected.

Next, applies the Harris corner detection algorithm to detect corners in the face image. Notice the span of the corner-peaks filter is now set to 4 pixels:

```
def corner_harr(im, min_distance = 4):
    from skimage.feature import corner_harris, corner_peaks
    mag = corner_harris(im)
    return corner_peaks(mag, min_distance = min_distance)
harris = corner_harr(steve, min_distance = 4)
plot_harris(steve, harris)
```



The corner detector has located the two eyes.

### Image Morphology

Morphology methods are often used to enhance image features. Using these methods, a set of features can be cleaned and made more consistent.

### Erosion

The first morphology method applied is erosion. **Erosion** is a fundamental operation and forms the basis of some other morphology operators. An erosion operator removes or erodes pixels at the edge of objects, by setting the values to zero.

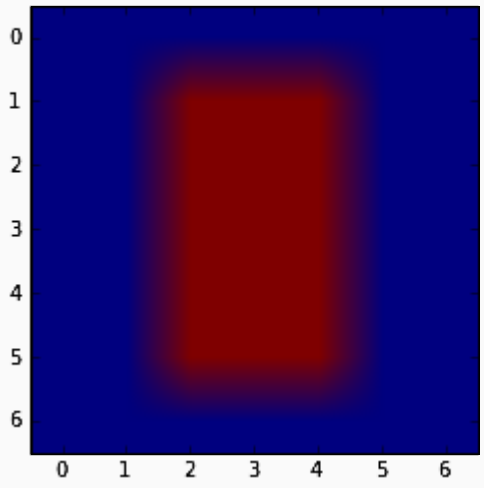
The code below creates a simple image containing a rectangle:

```
import numpy as np
from skimage import morphology
import matplotlib.pyplot as plt
a = np.zeros((7,7), dtype=np.int)
a[1:6, 2:5] = 1
a
```

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 1, 1, 1, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

```
plt.imshow(a)
```

```
<matplotlib.image.AxesImage at 0x7f774254c190>
```

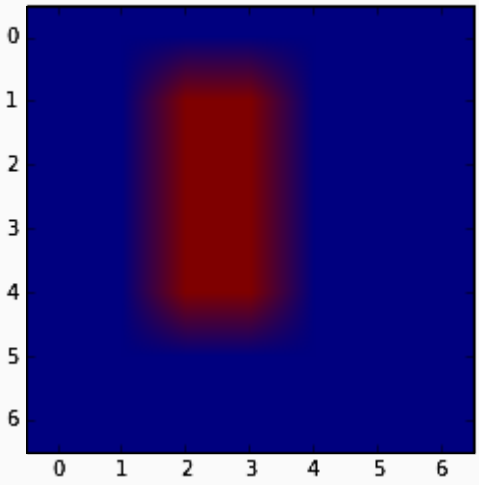



The code below erodes the image using a square shaped 2x2 operator:

```
a_erosion = morphology.binary_erosion(a, np.ones((2,2))).astype(np.uint8)
print(a_erosion)
plt.imshow(a_erosion)
```

```
[[0 0 0 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

```
<matplotlib.image.AxesImage at 0x7f77415d3f90>
```



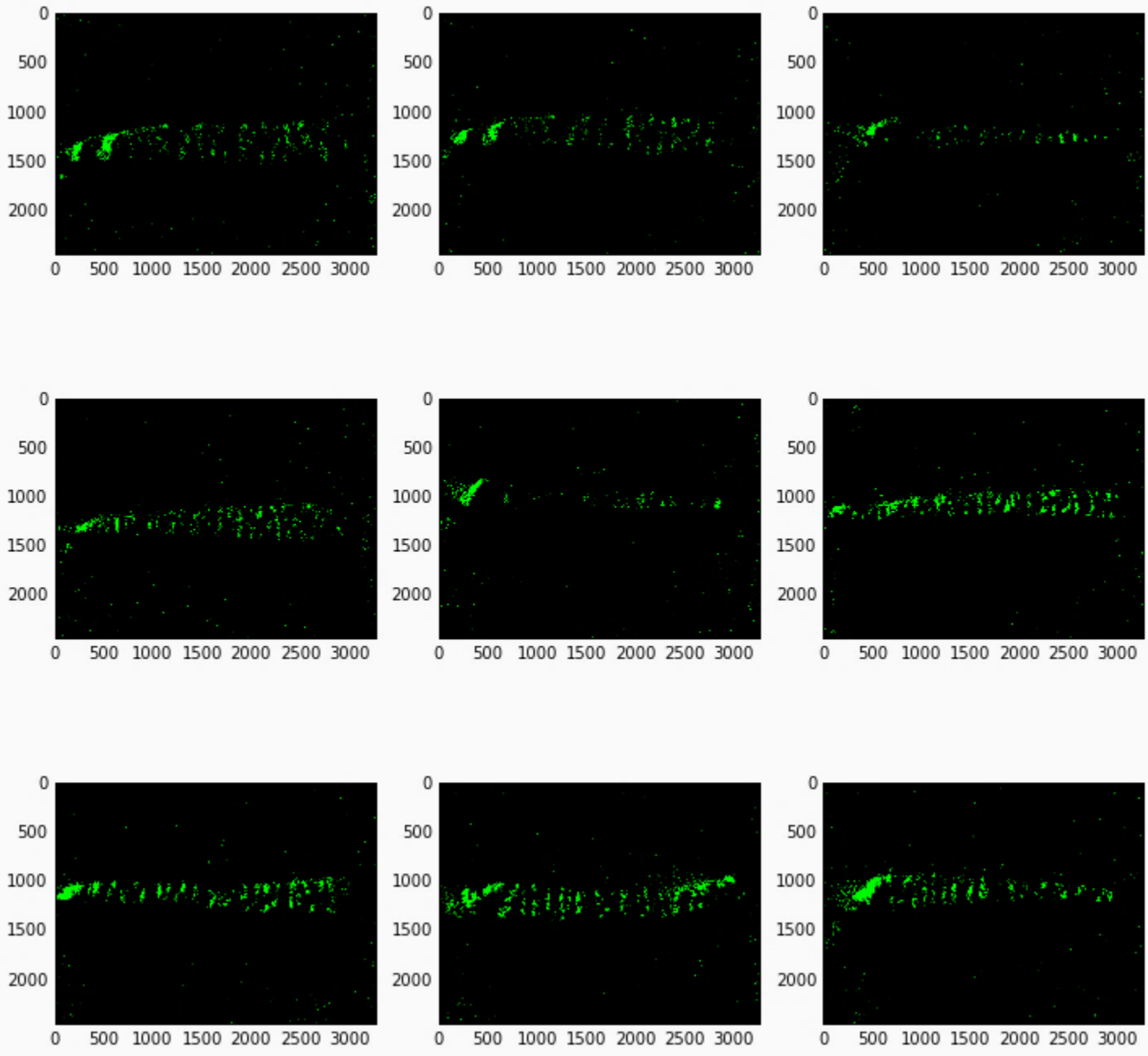

The erosion operator has reduced the size of the rectangle by one pixel in each dimension.

Now, run the erosion operator against the results of the edge detection applied to the carrot images. Since the carrot image has color, represented by a n x m x 3-d array, a 3-d operator is used. This is opposed to the gray scale face image which is represented by a n x m x 1-d array.

The code in the cell below iterates over the list of images, applying the erosion operator to each:

```
def im_erosion(im_list, structure = (4,4,3)):
    from scipy.ndimage import morphology
    import numpy as np
    out = []
    for image in im_list:
        out.append(morphology.binary_erosion(image,structure=np.ones(structure)))
    return out
carrot_erosion = im_erosion(carrot_filter, structure = (2,2,3))
plot_carrot(carrot_erosion)
```

Comparing images to the images resulting from applying edge detection and segmentation. The features of the carrots themselves are thinner. Additionally, the noise is less noticeable.



### Dilation

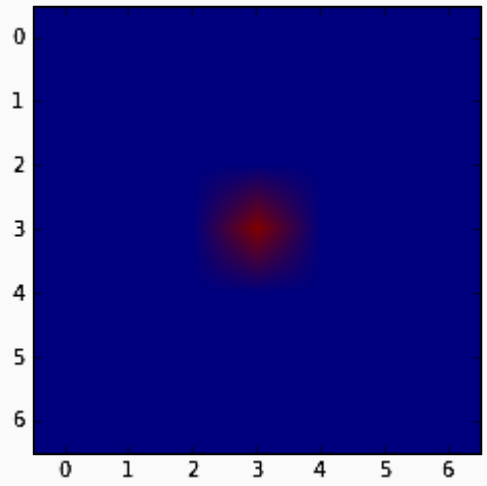
Dilation is another fundamental morphology operator. A **dilation** operator accretes pixels to image features. In effect, the image features are expanded or thickened.

The code below creates an image with a single positive pixel:

```
a = np.zeros((7,7), dtype=np.int)
a[3, 3] = 1
a
```

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 1, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

```
plt.imshow(a)
```

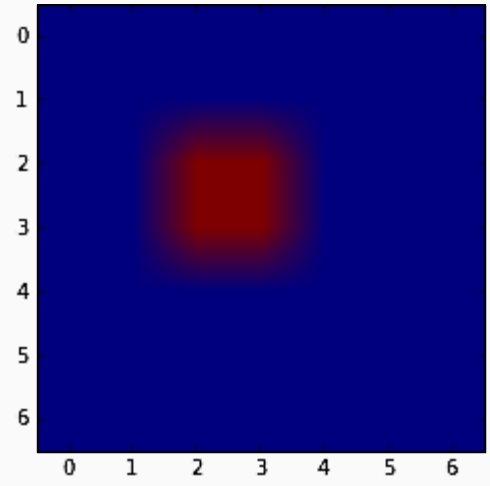



As before, a 2x2 square operator is used and applied to a diamond shaped dilation operator against the simple image created by executing the code below:

```
a_dilation = morphology.binary_dilation(a,
np.ones((2,2))).astype(np.uint8)
print(a_dilation)
plt.imshow(a_dilation)
```

```
[[0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 1 1 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

The image has accreted a row and column of non-zero pixels on the upper and left edges.



### Opening

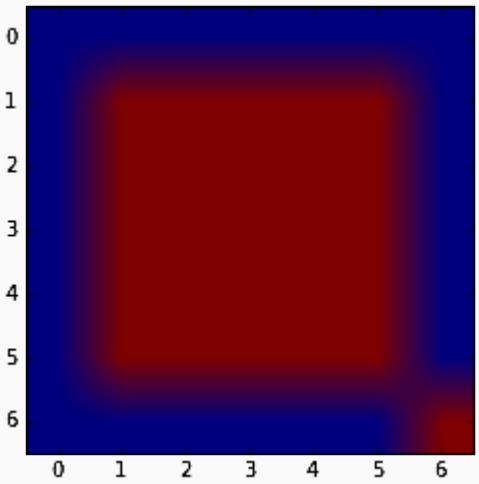
**Opening** is a morphological operation comprised of an erosion operator followed by a dilation operator. Opening is useful in cleaning certain types of noise from images. For example, opening tends to create better separated features in an image.

The code below creates an image with a square and a single non-zero pixel in the lower right corner:

```
a = np.zeros((7,7), dtype=np.int)
a[1:6, 1:6] = 1
a[6, 6] = 1
a
```

```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 1, 1, 1, 1, 1, 0],
       [0, 0, 0, 0, 0, 0, 1]])
```

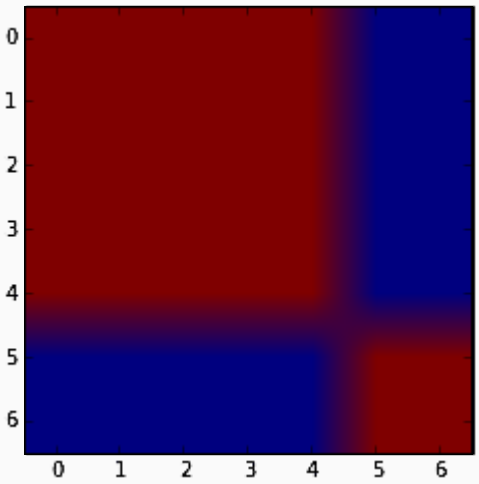
```
plt.imshow(a)
```




Apply the morphological opening operator to the image:

```
a_open = morphology.binary_opening(a, np.ones((2,2))).astype(np.uint8)
print(a_open)
plt.imshow(a_open)
```

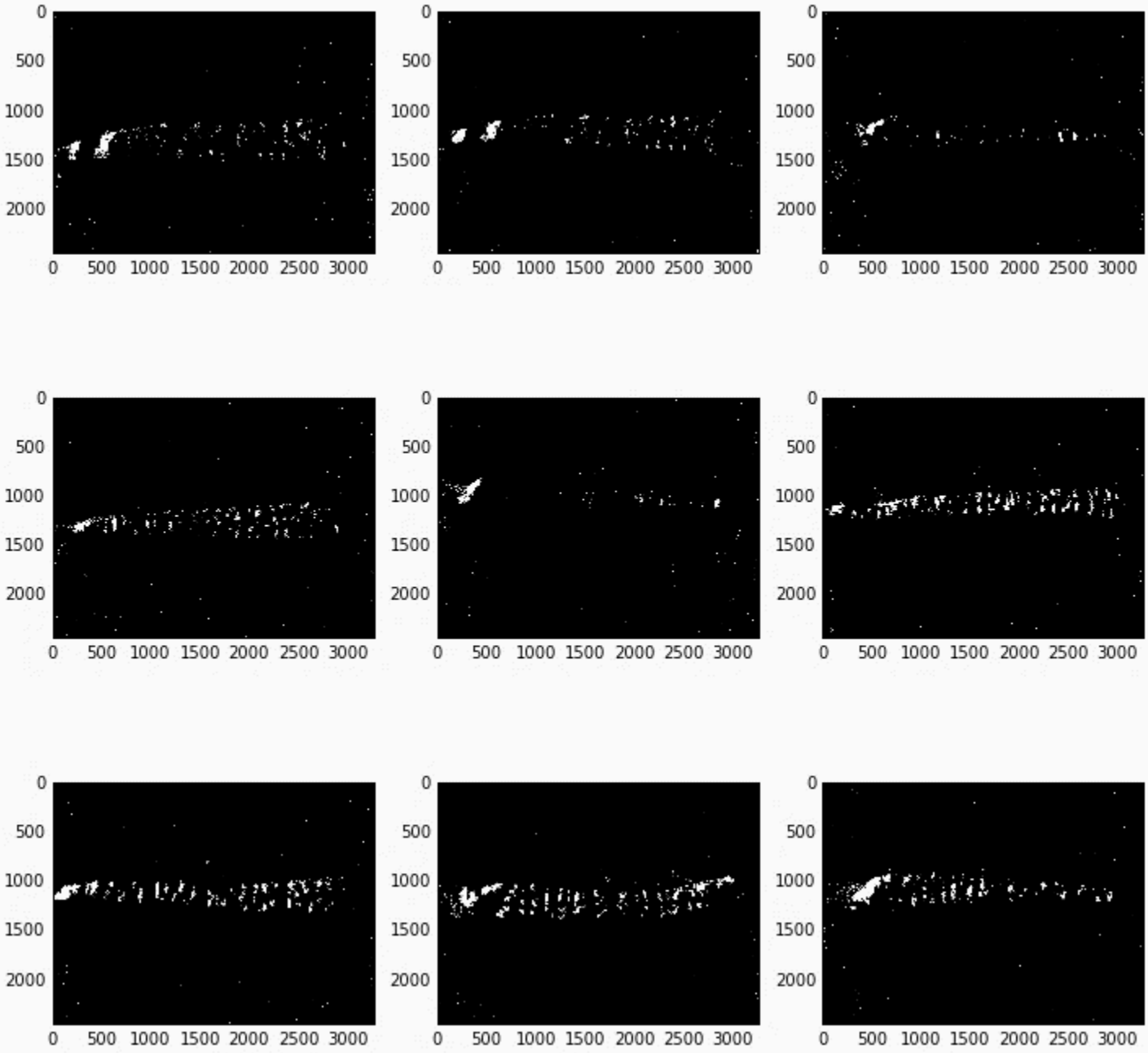
```
[[1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [0 0 0 0 0 1 1]
 [0 0 0 0 0 1 1]]
```




Compare the result to the original image. The pixel in the lower right has been set to zero. The image of the square retains its original dimensions. In effect, the non-zero pixel in the lower right corner has been filtered as though it was noise. Retaining the dimensions of the square is the result of following the opening operator with a dilation operator.

The code below applies the opening operator to the list of carrot edge images. The operator shape is asymmetric. Since the carrot edge features to be enhanced are elongated, mostly in the vertical (y) direction, an operator longer than it is wide is chosen. Apply the opening operator as follows:

```
def im_open(im_list, structure = (2,2,1)):
    from scipy.ndimage import morphology
    import numpy as np
    out = []
    for image in im_list:
        out.append(morphology.binary_opening(image,structure=np.ones(structure)))
    return out
carrot_open = im_open(carrot_filter, structure = (2,4,3))
plot_carrot(carrot_open)
```

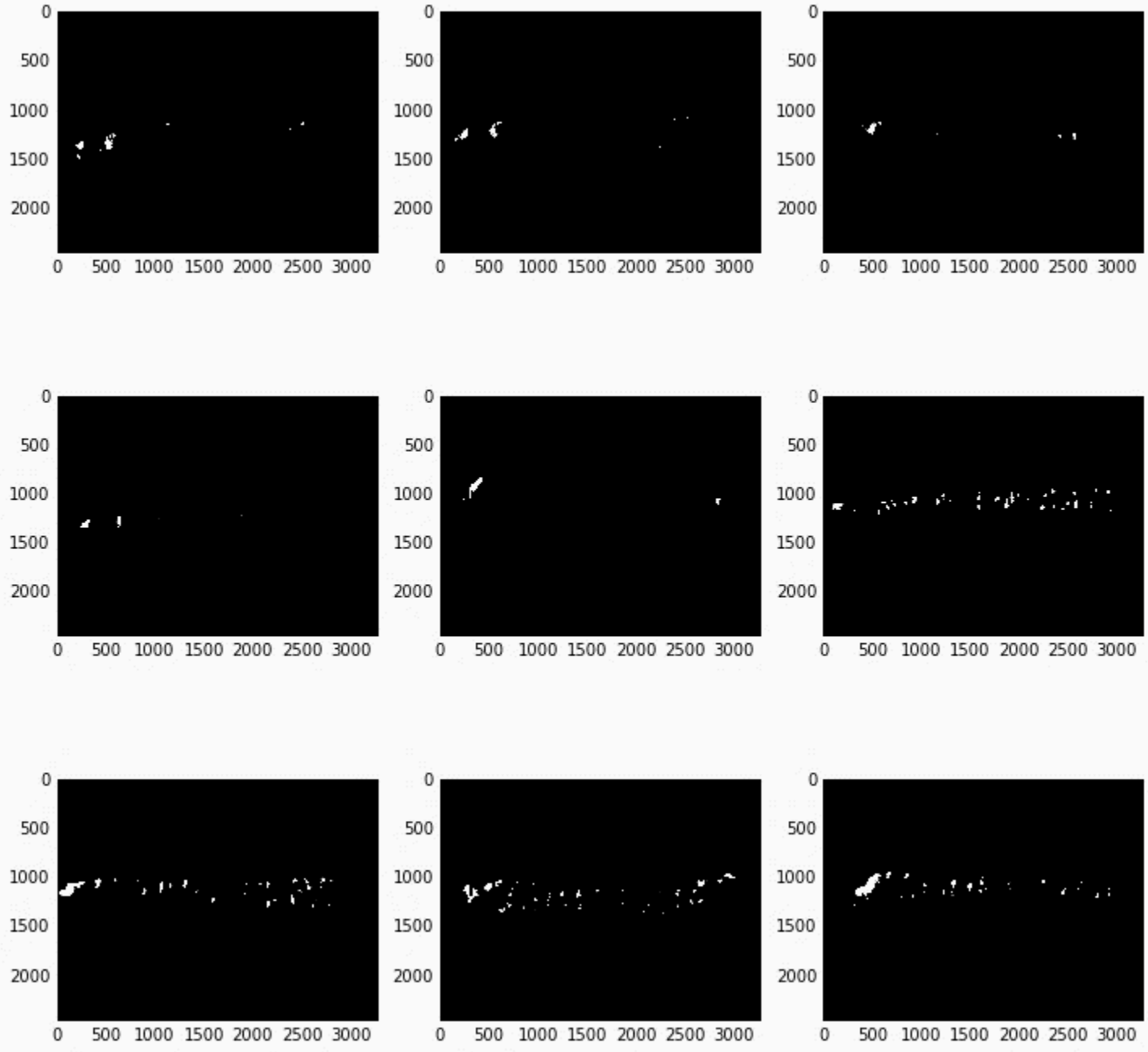



Notice much of the noise has been eliminated by applying the opening operator. Despite the reduction in noise, many essential features of the carrot edges have been retained. However, the features are thinner.

### Changing the Operator Shape for Opening

To see this difference in operator effects, apply the opening operator with a shape defined by **(8,8,3)**. Compare the result to the carrot edge images created with an operator shape of **(2,4,3)**:

```
carrot_open = im_open(carrot_filter, structure = (8,8,3))
plot_carrot(carrot_open)
```




- The carrot edge features are better preserved by the smaller opening operator.
- The carrot edge feature created by the larger opening operator has less noise.

### Closing

**Closing** is a morphological operation comprised of a dilation operator followed by an erosion operator, the opposite order of an opening operation. Like the opening operator, the closing operator can be useful in reducing noise when extracting features from images.

The code below creates a square with a hole in the middle:

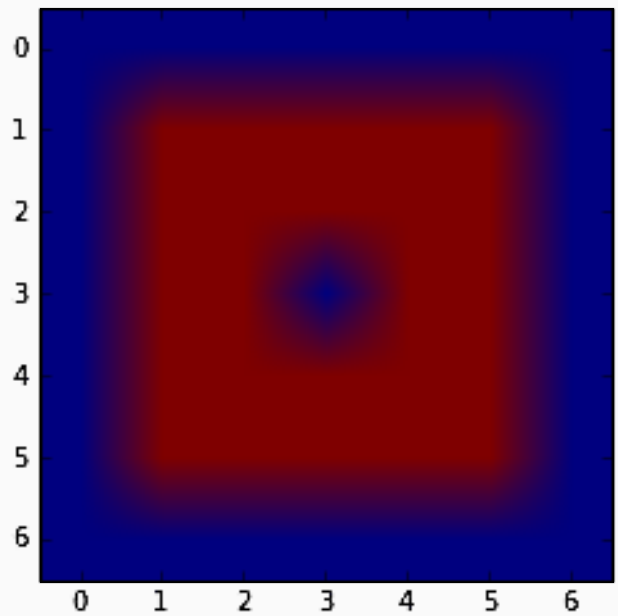
```
a = np.zeros((7,7), dtype=np.int)
a[1:6, 1:6] = 1
a[3,3] = 0
plt.imshow(a)
```

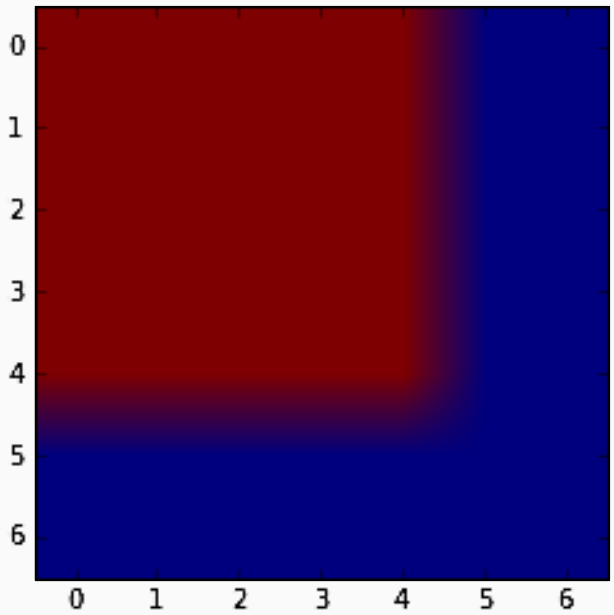
The code below applies the morphological closing operator to the image:

```
a_close = morphology.binary_closing(a, np.ones((2,2))).astype(np.uint8)
print(a_close)
plt.imshow(a_close)
```

```
[[1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [1 1 1 1 1 0 0]
 [0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0]]
```

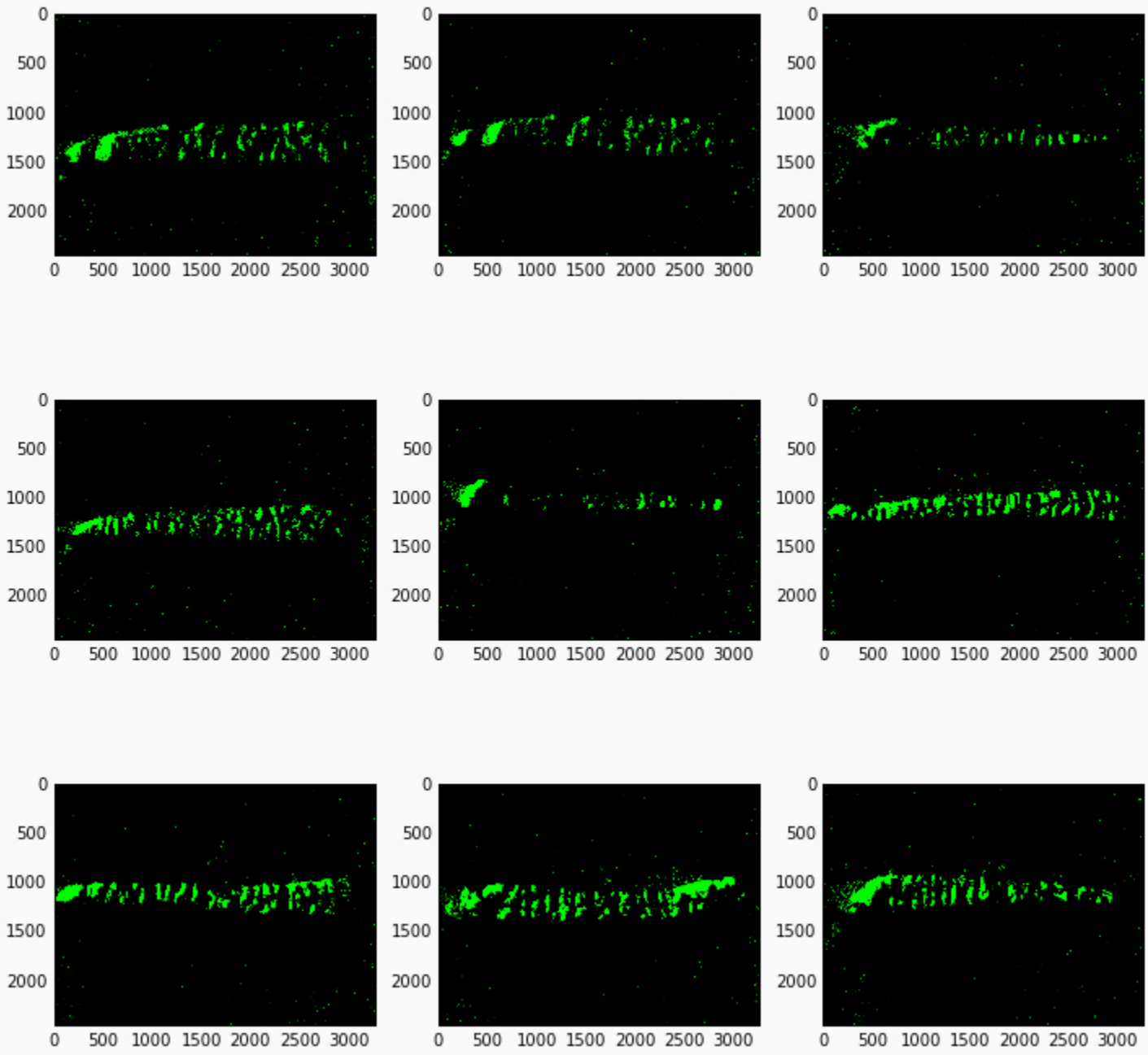
The hole in the square has been healed. The image of the square retains its original dimensions. Retaining the dimensions of the square results from the dilation operator with an erosion operator.



The code below applies the closing operator to the list of carrot images. The operator shape is asymmetric. Since the carrot edge features to be enhanced are elongated mostly in the vertical (y) direction, an operator longer than it is wide is chosen. The code applies the closing operator results:

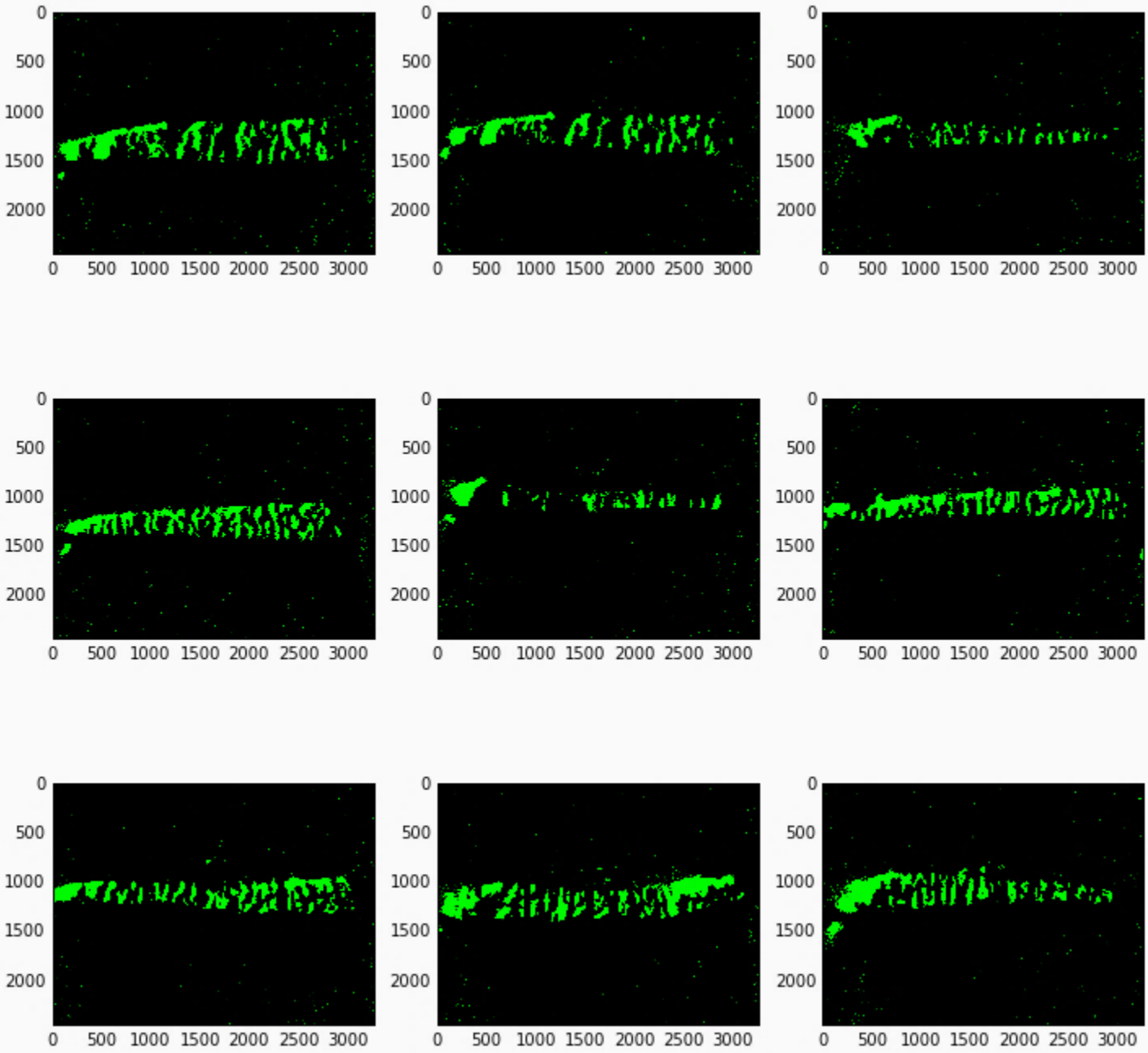
```
def im_close(im_list, structure = (2,2,3)):
    from scipy.ndimage import morphology
    import numpy as np
    out = []
    for image in im_list:
out.append(morphology.binary_closing(image,structure=np.ones(structure)))
    return out
carrot_close = im_close(carrot_erosion, structure = (2,6,3))
plot_carrot(carrot_close)
```




Comparing the images to the original images of the carrot edge features after segmentation. Much of the noise has been reduced by applying the closing operator. Despite the reduction in noise nearly all the essential features of the carrot edges are retained.

### Change the Operator Shape for Closing

Changing the operator shape affects the results of the closing operation. To see this difference, the closing operator with a shape defined by **(6,12,3)** is applied to the carrot edge images. The result is compared to the result created with an operator shape of **(2,6,3)**:




- The carrot edge features better preserved by the larger closing operator.
- The carrot edge feature created by the larger closing operator has less noise.
- Of the four operators you have tried, a) **opening (2,4,3)**, b) **opening (8,8,3)**, c) **closing (2,6,3)** and **closing (6,12,3)**, it appears the first **opening (2,4,3)** did the best job reducing noise while preserving the carrot edge features.

**Summary**

This lab performed a steps to create images with features used in machine learning models:

- Loaded and explored the properties of the images.
- Equalized the image histograms.
- Plotted images.
- Added noise (pre-whitened) and filtered images.
- Extracted features from the images.
- Applied morphological operators to the image features.