Text Processing with Pattern Matching and Regular Expressions

Kevin DelRosso

November 16, 2015

Setting up the workspace

```
rm(list = ls())
setwd("~/Desktop/STA_141/assignment_4/")

require(scales) # for changing alpha level in plots
require(png)
require(grid)
require(magrittr) # for updating row names (while returning object)
require(lattice)
require(stringi)
require(stringr)
require(zoo) # for na.fill
require(cvTools) # for cross-validation
require(MASS)

# loading vposts
load( url( "http://eeyore.ucdavis.edu/stat141/Data/vehicles.rda" ))
```

Functions used throughout

```
trim =
    # trim leading and training whitespace from a character vector
    # also remove punctuation with punct = TRUE
function( char_vector, punct = FALSE)
{
    reg =
    if( punct ) {
        "[[:space:][:punct:]]"
    } else {
        "[[:space:]]"
    }

    rx = sprintf( "^%s+|%s+$", reg, reg )
    gsub( rx, "", char_vector )
}

# remove everything not alphanumeric or space
remove_punct = function(char_vector) {
    gsub( "[^[:alnum:][:space:]]", "", char_vector )
}


get_nearby =
```

```r
    # Extract characters near a matched regular expression (used for testing)
function( text, rx, n_char )
{
    rx = sprintf( ".{0,%s}%s.{0,%s}", n_char, rx, n_char )
    str_extract( text, rx )
}


combine_results =
    # INPUT:
    #   - list_of_results: list of vectors, each with the same length
    #   - elements in list should be in order of importance (most important first)
    #
    # OUTPUT: vector of length list_of_results[[i]]
    # DOC: fill in NA values in list_of_results[[1]] with values in list_of_results[[i]], i > 1
    #
    # Example:
    # list_of_results = list( a = c(1, NA, NA, NA), b = c(2, 3, 6, NA), b = c(1, 2, 3, 4) )
    # output: c(1, 3, 6, 4)
function( list_of_results )
{
    keep = list_of_results[[1]]
    for( results in list_of_results[-1] ) {
        # if keep is missing a value, we look in results to update
        keep = ifelse( !is.na(keep), keep, results )
    }

    keep
}


find_all_patterns =
    # INPUT:
    #   - text: character vector with strings to search with regular expressions
    #   - max_patterns: the largest numbers of patterns to search in helper_function
    #   - helper_function: function to call
    #   - print_output: T/F to print the number of results found with each pattern
    #
    # OUTPUT:
    #   - vector of length text containing all values extracted from text
    # DOC: Common code by many functions that follow
function( text, patterns, helper_function, print_output = TRUE) {

    # convert all text to lower case
    text = tolower(text)
    results = lapply( patterns, function(pat) helper_function( text, pat ) )

    if( print_output ) print( sapply( results, function(vect) sum( !is.na(vect) ) ) )

    combine_results( results )
}
```

Overall strategies. I found the str_extract and str_extract_all functions from the stringr package very useful. Since they don't have arguments for ignore.case, I chose to convert all text to lowercase to help create more positive matches. For each part, I wrote several regular expressions to find matches, which were placed in decreasing order of correct match likelihood, i.e. (best matches, next best, . . . , worst). Using this configuration, I was able to compute the number of positive matches, while also combing the results in a way that kept all the more likely matches, using the combine_results function above.

Part 1 - Price

Let's begin by exploring the data for prices. The dataset contains 34,677 rows, with 20,694 containing common characters often associated with price: $, price, or asking. We can also find several hundred posts which likely won't contain prices since they say "call/text/email for price."

```
# exploring the data for prices
length( !is.na(vposts$price) )
```

```
## [1] 34677
```

```
# sample size of potential prices
dollar_sign = grepl( "\\$", vposts$body )
contain_price = grepl( "price", vposts$body, ignore.case = TRUE )
contain_ask = grepl( "asking", vposts$body, ignore.case = TRUE )
sum( dollar_sign | contain_price | contain_ask )
```

```
## [1] 20694
```

```
# call for price
rx = "(text|call|email)[[:punct:][:space:]]{0,3}(for|4)[[:punct:][:space:]]{0,3}(price|pricing)"
sum( grepl( rx, vposts$body, ignore.case = TRUE ) )
```

```
## [1] 286
```

Function for finding price.

```
##################
# Step 1 - Price #
##################

combine_lists =
    # combine a list_of_lists into one list, concatenation corresponding vectors in each sublist
function( list_of_lists )
{
    # adding arguments for mapply
    list_of_lists$FUN = c
    list_of_lists$SIMPLIFY = FALSE

    do.call( mapply, list_of_lists )
}


clean_prices =
    # INPUT: a list of vectors, each containing prices as strings
    # OUTPUT: a list of the same length containing numeric vectors
```

```r
    # DOC: converts k to thousands and converts prices to numeric
function( price_list )
{
    price_list = lapply( price_list, function(prices) {
        # in case no prices found
        if( length(prices) == 0) return( NA )

        # convert k to thousands
        prices = gsub( "(^|[^[:digit:]])([0-9]{1,3}) ?[kK].?$", "\\2000", prices )

        # remove anything not a digit and conver to numeric
        as.numeric( gsub( "[^[:digit:]]", "", prices ) )
    })
    price_list
}


find_price =
    # INPUT: vector of string
    # OUTPUT: numeric vector of length text, containing all prices found
    # DOC: calls find_price_helper with each pattern
function(text)
{
    # Found several examples like: "$4900100% Guaranteed" so let's just remove these
    text = gsub( "[1-9][05]{1,2}%", " ", text )
    text = tolower(text)

    # numbers for the price, includes for example 3000, 3,000, and 3k
    price_number = "[1-9][0-9]{0,2}(,? ?[0-9]{3}){0,3}"
    price_k = "[1-9][0-9]{0,2} ?[kK][^[:alpha:]]"

    # different patterns for matching price, with and without $
    pat_1 = sprintf( "\\$ ?%s", price_number )
    pat_2 = sprintf( "asking( price):? ?%s", price_number )
    pat_3 = sprintf( "(price|cost):? ?%s", price_number )
    pat_4 = sprintf( "\\$ ?%s", price_k )

    pat = c(pat_1, pat_2, pat_3, pat_4)

    results = lapply( pat, function(pattern) find_price_helper( text, pattern ) )

    # combine and clean prices
    clean_prices( combine_lists(results) )
}


find_price_helper =
    # INPUT: vector of strings and a pattern, i.e. regular expression to use
    # OUTPUT: list of length text, containing character vectors of prices found
function( text, pat )
{
    # the price ends at the first non-digit character or the end of the line.
    rx = sprintf( "%s([^[:digit:]]|$)", pat)
```

```
    # we'll extract all matching prices
    str_extract_all( text, rx)
}


percent_agreement =
    # INPUT:
    #   - price_text: character vector to extract prices, either vposts$body or vposts$title
    #   - true_price: a vector of prices to compare with, likely the listed price: vposts$price
    #
    # OUTPUT: 4 element list
    #   - the percent correct
    #   - the number of prices found
    #   - the number of correct prices
    #   - logical vector if price column verified
function( price_text, true_price = vposts$price, output_given = FALSE, output = NULL )
{
    # output is list of vectors containing all prices found in price_text
    output =
    if( output_given ) {
        output
    } else {
        find_price( price_text )
    }

    # subset on the texts where we found at least one price,
    # check if true_price contained in these prices
    mask = !is.na(output)
    correct_price_found = mapply( function( found_p, true_p ) true_p %in% found_p,
                                  output[mask], true_price[mask] )

    # create logical vector the same length as output
    tf_prices = rep( FALSE, length(output) )
    tf_prices[mask] = correct_price_found

    # return the percent correct and number of texts in which we extracted a price
    percent_correct = mean(correct_price_found)
    setNames( list( round(percent_correct, 2),
                    length(correct_price_found),
                    sum(correct_price_found),
                    tf_prices ),
              c("percent_correct", "prices_found", "number_correct", "verified_price") )
}

fill_na_price =
    # INPUT:
    #   - new_prices: list of price vectors
    #   - true_price: vector of prices, NA values to be filled with new_prices
    # OUTPUT: vector of length true_price with NA values filled
function( new_prices, true_price )
{
    # NA locations to fill
    na_price = is.na(true_price)
```

```r
    new_prices = sapply(new_prices, function(price) {
        if( length(price) == 1 ) return( price )

        # all NA values
        if( sum( is.na(price) ) == length(price) ) return( NA )

        max(price, na.rm = TRUE)
    })

    true_price[ na_price ] = new_prices[ na_price ]
    true_price
}
```

Prices were found using the dollar sign ($) and combinations of: asking price, price, and cost. We also attempted to find prices which use "k" to signify thousands, and then convert those values to the correct numeric value. We also searched for thousands, but got too many false positives with numbers related to odometer and miles. Price were found in the body, description, and title columns. Posts often contained many prices, so we saved all the extracted prices and compare them all against the price column. Below is the summary of the accuracy of the prices extracted from each column, the total number correct, etc. We also added a logical column for prices that were verified using the combination of body and description. When the original price were NA but price were found in either the title, body, or description, we updated to original price. We were able to verify 12,367 prices.

```r
# extract prices from text
prices_body = find_price(vposts$body)
prices_description = find_price(vposts$description)
prices_title = find_price(vposts$title)

tmp_b = percent_agreement( NULL, vposts$price, TRUE, prices_body )
tmp_d = percent_agreement( NULL, vposts$price, TRUE, prices_description )
tmp_t = percent_agreement( NULL, vposts$price, TRUE, prices_title )

# combining prices from body and description
bd_prices = combine_results( list( prices_body, prices_description ) )
tmp_bd = percent_agreement( NULL, vposts$price, TRUE, bd_prices )

set_rownames( rbind( tmp_b, tmp_d, tmp_t, tmp_bd )[ ,1:3],
              c("body", "description", "title", "body_description") )
```

```
##                  percent_correct prices_found number_correct
## body             0.71            15218        10744
## description      0.37            1929         713
## title            0.98            31944        31349
## body_description 0.7             15804        11051
```

```r
vposts$verified_price = as.integer( tmp_bd$verified_price )

# fill NA price values using title, then body, then description
na_price = is.na(vposts$price)
updated_price = vposts$price
new_prices = list( prices_title, prices_body, prices_description )
```

```r
for( prices in new_prices ) {
    updated_price = fill_na_price( prices, updated_price )
}

list_names = c("original_NA", "updated_NA")
setNames( c( sum(na_price), sum( is.na(updated_price) ) ), list_names )
```

```
## original_NA  updated_NA
##        3328        2008
```

```r
# update vposts
vposts$price = updated_price

# updated verified price where original price was NA, but updated price has value
na_mask = na_price & !is.na(updated_price)
vposts$verified_price[na_mask] = 1

table(vposts$verified_price)
```

```
##
##     0     1
## 22310 12367
```

Part 2 - VIN

We extracted VIN numbers by searching for 17 character strings containing only uppercase letters and digits. The characters I, O, and Q were excluded since they are not included in VIN numbers to avoid confusion with 1 and 0. We added the assumption that VIN numbers will include at least 2 letters and 2 digits. Some posts contain a lot of uppercase text, so this assumption is to avoid false positives in those situations. We found 8570 VIN numbers which were added to vposts.

```r
################
# Step 2 - VIN #
################

find_VIN =
    # INPUT: vector of string
    # OUTPUT: character vector of vin numbers
    # DOC: assumptions for VINs
    #   - combination of 17 uppercase letters and digits
    #   - doesn't include I, O, or Q to avoid confusion with 1 and 0
    #   - must have at least 2 letters and 2 numbers (i.e. can't be just LETTERS or digits)
function( text )
{
    # all uppercase letters except I, O, and Q
    most_letters = LETTERS[ !(LETTERS %in% c("I", "O", "Q")) ]
    most_letters = paste( most_letters, collapse = "" )

    rx = sprintf( "[%s[:digit:]]{17}", most_letters )

    vins = str_extract_all(text, rx)
```

```
    # uppercase followed by digit, or digit followed by uppercase
    # must match twice to ensure at least 2 letters and 2 numbers
    # assuming we won't see all letters followed by all digits
    pat = "[[:upper:]][[:digit:]]|[[:digit:]][[:upper:]]"
    pat = sprintf( ".*%s.*%s.*", pat, pat )

    vins = sapply( vins, function(v) {
        # in case no vin numbers found
        if( length(v) == 0) return( NA )

        is_vin = grepl( pat, v )

        if( sum(is_vin) == 0 ) {
            # didn't find any vins
            return( NA )
        } else if ( sum(is_vin) == 1 ) {
            # found only 1 vin, return it
            return( v[is_vin] )
        } else {
            # we found more than 1 vin, return the first one
            return( v[is_vin][1] )
        }
    })

    vins
}

vin_numbers = find_VIN( vposts$body )
sum( !is.na(vin_numbers) )
```

```
## [1] 8570
```

```
vposts$vin = vin_numbers
```

Part 3 - Phone Number

We found that people use various anti-parsing techniques to hide their phone numbers. For this reason we first replaced the words zero, one, … nine, and "o" with the corresponding digit before searching for phone numbers. We created four patterns of phone numbers, variations include: (xxx) xxx-xxxx, xxx.xxx.xxxx, xxx xxx xxxx, and call xxxxxxxxxx. The output prints the number of matches found with each pattern. We found a total of 16,278 phone numbers which we added to vposts.

```
#########################
# Step 3 - phone number #
#########################

find_phone =
    # INPUT: vector of strings
    # OUTPUT: phone numbers extracted from each string
function(text, print_output = TRUE)
{
    # some people write out their phone numbers with words, we'll try to find these
    # o can also be used for zero
```

```
    nums = c("zero", "one", "two", "three", "four", "five", "six", "seven", "eight", "nine", "o")
    nums = sprintf( " ?%s ?", nums ) # optional space before/after words
    num_replace = c(0:9, 0)
    for( i in 1:length(nums) ){
        text = gsub( nums[i], num_replace[i], text )
    }

    # different patterns for phone numbers, in decreasing likelihood of being a phone number
    pat_1 = "(\\([[:digit:]]{3}\\) ?|[[:digit:]]{3}[- ])[[:digit:]]{3}[- ]?[[:digit:]]{4}"
    pat_2 = "[[:digit:]]{3}\\.[[:digit:]]{3}\\.[[:digit:]]{4}"
    pat_3 = "[[:digit:]]{3}[[:space:]]+[[:digit:]]{3}[[:space:]]+[[:digit:]]{4}"
    pat_4 = "[Cc]all[[:punct:][:space:]]{0,2}[[:digit:]]{10}"

    pat = c(pat_1, pat_2, pat_3, pat_4)

    find_all_patterns( text, pat, find_phone_helper, print_output )
}

find_phone_helper =
    # INPUT: vector of strings and a pattern, i.e. regular expression to use
    # OUTPUT: character vector of length text containing phone numbers
function(text, pat)
{
    # starts with either beginning of line or non-digit character and
    # end with end of line or non-digit character
    rx = sprintf( "(^|[^[:digit:]])%s([^[:digit:]]|$)", pat)

    text = str_extract( text, rx)
    gsub( "[^[:digit:]]", "", text )
}

phone = find_phone(vposts$body)
```

```
## [1] 15888   702   969    72
```

```
sum( !is.na(phone) )
```

```
## [1] 16278
```

```
vposts$phone = phone
```

Part 4 - Email / Website

After first extracting emails, we found that very few posts included an email address (likely to avoid spamming). However, in the case that a post doesn't include an email, we'll attempt to extract a website. Emails were found using primarily the "@", while websites were found looking for: http, www., website: / link:, and text ending in .com / .net / . . . . We found 105 emails and 13,823 websites. We also found the breakdown of email / website top-level domains.

```
############################
# Step 4 - email / website #
############################
```

```r
# find email address in vectors of strings
find_email = function(text) {
    str_extract( text, "[[:alnum:]\\._-]+@[[:alnum:]\\.-]+\\.(com|net|gov|org|io|edu)")
}

find_website =
    # INPUT: vector of strings
    # OUTPUT: websites extracted from each string
function(text, print_output = TRUE)
{
    # removing various combination of ... which people sometimes write
    text = gsub( "\\. ?\\. ?\\. ?", " ", text )

    # website patterns
    pat_1 = "https?://(www\\.)?"
    pat_2 = "www\\."
    pat_3 = "([Ww]ebsite|[Ll]ink):? ?"
    pat_4 = "[[:space:][:punct:]]"

    pat = c(pat_1, pat_2, pat_3, pat_4)

    find_all_patterns( text, pat, find_website_helper, print_output )
}

find_website_helper =
    # INPUT: vector of strings and a pattern, i.e. regular expression to use
    # OUTPUT: character vector of length text containing websites
function(text, pat)
{
    rx = sprintf( "%s[^[:space:]!@#$&:,;]*?\\.(com|net|gov|org|io|edu)", pat)

    # for the fourth pattern must end with space of punctuation
    if( pat == "[[:space:][:punct:]]" ) rx = paste0( rx, pat )

    text = str_extract( text, rx)

    # remove website, link, and trim white space and punctuation from the results
    text = gsub("^([Ww]ebsite|[Ll]ink)", "", text)
    trim( text, punct = TRUE )
}

email = find_email( vposts$body )
website = find_website( vposts$body )
```

```
## [1]  4295  9391  3197 12486
```

```r
tmp_e = sum( !is.na( email ) )
tmp_w = sum( !is.na( website ) )

setNames( c(tmp_e, tmp_w), c("email", "website") )
```

```
##    email website
##      105   13823
```

```r
# the websites might include the email
# we'll put email first so it takes preference when combining results
email_website = combine_results( list( email, website ) )
sum( !is.na( email_website ) )
```

```
## [1] 13827
```

```r
table( str_extract( email_website, "\\.[:alpha:]{3}$" ) )
```

```
##
##  .com  .gov  .net  .org
## 12631     1  1119    72
```

```r
vposts$email_website = email_website
```

Part 5 - Year

We found years by searching for 4 digit (xxxx) and 2 digit ('xx) year patterns in both body and description. We first compared the original year with the year_body, and any years that disagreed we then compared with the year_description. We then added a logical column to vposts indicated that the year was verified, with the values:
- 3: the year was verified
- 2: both year_body and year_description are NA (could not be verified)
- 1: both year_body and year_description agree, different from original year
- 0: values in year_body and year_description disagree with original year

In the case of a 1, we'll update the original year with the value from year_body / year_description. From the summary table of year_factor, we see that in most cases the year was either verified, or no year was found. We updated 171 years, leaving only ~700 unreliable.

```r
#################
# Step 5 - Year #
#################

fix_short_year =
    # INPUT: a character vector of years
    # DOC: converts 2 digits years to 4 digit years
function( year_vect )
{
    year_vect = gsub( "^(0[0-9]|1[0-6])", "20\\1", year_vect )
    gsub( "^(9[0-9])", "19\\1", year_vect )
}


find_year =
    # INPUT: vector of strings
    # OUTPUT: year extracted from each string
function(text, print_output = TRUE)
{
    # pattern for full year, or 'yr
    pat_1 = "%s19[0-9]{2}|200[0-9]|201[0-6]%s"
    pat_2 = "%s'(9[0-9]|0[0-9]|1[0-6])%s"
```

```r
    pat = c(pat_1, pat_2)

    find_all_patterns( text, pat, find_year_helper, print_output )
}


find_year_helper =
    # INPUT: vector of strings and a pattern, i.e. regular expression to use
    # OUTPUT: character vector of length text containing years
function( text, pat )
{
    reg = "[[:punct:][:space:]]"

    # year must be found between either punctuation or spaces
    rx = sprintf( pat, reg, reg )

    text = str_extract( text, rx)
    fix_short_year( gsub( "[^[:digit:]]", "", text ) )
}

get_year_factor =
    # INPUT: a data.frame (vposts)
    # OUTPUT: 2 element list
    #   - a factor variable for year indicating the verification status
    #   - the updated year values
function(df)
{
    year_factor = integer( nrow(df) )

    # extract years from body and description
    year_body = find_year( df$body )
    year_description = find_year( df$description )

    # body or description match year
    body_match = year_body == df$year
    description_math = year_description == df$year

    # both body and description NA
    both_NA = is.na(year_body) & is.na(year_description)

    # update year_factor
    year_factor[body_match] = 3
    year_factor[description_math] = 3

    year_factor[both_NA] = 2

    # year from body and description agree
    agree = year_body == year_description

    # body and description agree and different from year
    mask = (year_factor == 0) & agree
    mask[ is.na(mask) ] = FALSE
    year_factor[ mask ] = 1
```

```
    # update original years
    updated_years = df$year
    updated_years[mask] = year_body[mask]

    list_names = c("year_factor", "updated_years")
    setNames( list(year_factor, updated_years), list_names )
}

year_output = get_year_factor(vposts)
```

```
## [1] 24968   134
## [1] 25725    13
```

```
# update vposts
vposts$year = year_output$updated_years
vposts$year_factor = year_output$year_factor

table( year_output$year_factor )
```

```
##
##     0     1     2     3
##   704   171  4449 29353
```

Part 6 - Model

Extracting the model was definitely the most involved task. First we remove hyphens since many model names contain them. For each vehicle maker, we then searched for the pattern "maker model" in the description, but only kept the 90th quantile of models as our common models. We then re-searched the description and body with those common models, in hopes of finding additional matches. We extracted additional models from description and body text, but only kept them if both description and body agreed. Finally, we performed fuzzy matching to look for misspellings and close matches on any remaining missing models.

We can look at the top five models in the data set. We get: civic, accord, grand, camry, and altima, however grand is used by many car companies, i.e. "grand something". We have only 2,522 NA values and extracted over 900 different models. We can also look at the breakdown of the models found by car maker (15 are shown below).

```
#######################
# Step 6 - Find model #
#######################

fix_makers =
    # INPUT: character vector of maker names
    # OUTPUT: character vector with strings fixed
function(makers)
{
    # makers with space in name, making the space optional and adding hyphen
    contains_space = grepl( " ", makers )
    makers[contains_space] = gsub( " ", "[ -]?", makers[contains_space] )

    # adding other common names to some makers
    makers = gsub( "chevrolet", "chevrolet|chevy", makers )
    makers = gsub( "mercedes", "mercedes|mercedes[ -]?benz", makers )
```

```r
    # wrapping all maker names in parenthesis, so we can select \\2 later on
    gsub( "(.*)", "(\\1)", makers )
}

get_common_models =
    # INPUT: a vector of strings and a vector of maker names
    # OUTPUT: the models names in the 90th quantile
function( text, maker )
{
    maker = fix_makers(maker)

    # search for pattern: "maker model"
    rx = sprintf( ".*?%s ([[:alnum:]]+)([^[:alnum:]]|$).*", maker )
    mask = grepl( rx, text )
    matches = gsub( rx, "\\2", text[mask] )

    # only take 90th quantile for most common models
    ninty_percentile = quantile( sort( table(matches) ), probs = seq(0, 1, 0.1) )[10]
    common_models = names( table(matches)[ table(matches) > ninty_percentile ] )

    # remove common models with only a single character
    common_models = common_models[ nchar(common_models) > 1 ]

    common_models
}

# change text to lower case and remove hyphen
clean_model_text = function(text) {
    text = tolower( text )
    gsub( "-", "", text )
}

extract_most_common =
    # INPUT: common model names, and text for description and body
    # OUTPUT: vectors of common models, taken from description and body
function( common_models, description_text, body_text )
{
    # extracts common models from the provided text
    find_common = function(text, rx) {
        trim( str_extract( text, sprintf( "(^|[[:space:]])%s", rx ) ) )
    }

    # short model names must end with a space or punctuation (to avoid false positives)
    common_models = unname( sapply( common_models, function(model) {
        model =
        if( nchar(model) <= 2 ) {
            paste0( model, "([[:space:][:punct:]]|$)" )
        } else {
            model
        }
        model
    }))
```

```r
    rx = paste(common_models, collapse = "|")

    # found common models in description and body
    found_description = find_common( description_text, rx )
    found_body = find_common( body_text, rx )

    # combine the found models, giving preference to description
    combine_results( list(found_description, found_body) )
}


get_model_match =
    # INPUT: text to search, maker name, and pattern to use (either 1 or 2)
    # OUTPUT: character vector of models found
function( text, maker, pattern = 1 )
{
    maker = fix_makers(maker)

    rx =
    if( pattern == 1 ) {
        # finds "maker word_1 word_2"
        sprintf( "%s[[:space:]]+[[:alnum:]]+ [[:alnum:]]+([^[:alnum:]]|$)", maker )
    } else {
        # finds "maker model"
        sprintf( "%s[[:space:]]+[[:alnum:]]+([^[:alnum:]]|$)", maker )
    }

    # remove space and maker from match
    matches = gsub( " ", "", str_extract( text, rx ) )
    trim( gsub( maker, "", matches ), punct = TRUE )
}


extract_two_words =
    # INPUT: text to search, models already found, and maker
    # OUTPUT: vector of found models in most_common
function(text, most_common, maker)
{
    # get all matches with pattern "model word_1 word_2"
    match_using_two = get_model_match( text[ is.na(most_common) ], maker, 1 )

    # replace NA values with ""
    match_using_two[ is.na(match_using_two) ] = ""

    # return values that matched, all other values get NA
    match_using_two[ !(match_using_two %in% unique(most_common)) ] = NA

    match_using_two
}


fuzzy_match =
    # INPUT:
```

```r
    #   - found_matches: potential models
    #   - all_models: all models we've already found
    #   - distance: max distance potential to found model
    #
    # OUTPUT: the model values to use for the found matches
    #
    # DOC:
    #   - find the distance from our potential models to the ones we've already found
    #   - for those less than distance, return the closest match from all_models
function(found_matches, all_models, distance = 1)
{
    # distance from all found_matches to all_models ( a matrix )
    dist_to_match = adist( found_matches, all_models )

    # return the index of the closest match, provided it's <= distance
    closest_match = apply( dist_to_match, 1, function(row) {
        # if entire row is NA, return NA
        if( length(row) == sum( is.na(row) ) ) return( NA )

        # otherwise return the closest match less than distance
        if( min(row, na.rm = TRUE) <= distance ) {
            return( which.min(row) )
        }
        NA
    })

    all_models[ closest_match ]
}


match_remaining =
    # INPUT:
    #   - most_common: vector of found models
    #   - description_text / body_text: text to search
    #   - maker: vehicle maker
    #
    # OUTPUT: updated most_common with additional found models
function( most_common, description_text, body_text, maker )
{
    common_models = unique(most_common)

    # find matches by extracted two words after maker
    match_description_two = extract_two_words( description_text, most_common, maker )
    match_body_two = extract_two_words( body_text, most_common, maker )

    # update most common
    most_common = combine_results( list(most_common, match_description_two, match_body_two) )

    # find matches by extracted one word after maker
    match_description = get_model_match( description_text[ is.na(most_common) ], maker, 2 )
    match_body = get_model_match( body_text[ is.na(most_common) ], maker, 2 )

    # keep all model where description and body match
```

```r
    more_models = match_description[ match_description == match_body ]

    all_models = c( common_models, unique(more_models) )

    # finally use fuzzy matching to extract any remaining results
    final_match_descrip = fuzzy_match( match_description, all_models, 1 )
    final_match_body = fuzzy_match( match_body, all_models, 1 )

    combine_results( list(most_common, final_match_descrip, final_match_body) )
}

models_by_maker =
    # INPUT: data.frame and name of a vehicle maker
    # OUTPUT: vector of found model names for maker
function(df, maker)
{
    maker_tf = grepl( maker, df$maker, ignore.case = TRUE )

    # clean text and subset by the maker
    text_descr = clean_model_text( df$description )[ maker_tf ]
    text_body = clean_model_text( df$body )[ maker_tf ]

    # find the most common models
    common_models = get_common_models( text_descr, maker )

    # search for most common models in description and body
    most_common = extract_most_common( common_models, text_descr, text_body )

    # extract remaining (less common) models
    match_remaining( most_common, text_descr, text_body, maker )
}

find_model =
    # INPUT: data.frame (vposts)
    # OUTPUT: models for every vehicle maker in df
function(df, print_progress = FALSE)
{
    # all makers
    makers = na.omit( unique( df$maker ) )

    # empty vector to store found model results
    models = character( nrow(df) )

    # loop over all makers and extract vehicle models
    for( m in makers ) {
        # to monitor progress
        if( print_progress ) print( m )

        found_models = models_by_maker( vposts, m )

        # making sure makers and models still line up
        maker_tf = grepl( m, vposts$maker, ignore.case = TRUE )
        stopifnot( sum(maker_tf) == length( found_models ) )
```

```r
        models[ maker_tf ] = found_models
    }

    # remove empty model names
    models[ nchar(models) == 0 ] = NA
    models
}


# summary of models found
model_summary = function(vect) {
    na_total = sum( is.na(vect) )
    model_total = sum( !is.na(vect) )
    different_models = length( unique(vect) )

    vect_names = c( "NA_total", "model_total", "different_models" )
    setNames( c(na_total, model_total, different_models), vect_names )
}

models = find_model(vposts)

# updating vposts
vposts$model = models

# top five models
sort( table( models ), decreasing = TRUE )[1:5]
```

```
## models
##  civic accord  grand  camry altima
##    895    878    828    759    666
```

```r
model_summary( models )
```

```
##         NA_total      model_total different_models
##             2522            32155              933
```

```r
makers = na.omit( unique( vposts$maker ) )
maker_models = lapply( makers, function(m) {
    maker_tf = grepl( m, vposts$maker, ignore.case = TRUE )
    model_summary( models[maker_tf] )
})

# summary table for 15 vehicle makers
n = 15
t( as.data.frame( setNames( maker_models, makers )[1:n] ) )
```

```
##           NA_total model_total different_models
## chevrolet      240        3154               88
## nissan         154        2319               30
## infiniti        50         509               29
## acura           22         675               20
```

```
## toyota          88       3244              46
## lexus            7        779              35
## honda           39       2611              25
## bmw             52       1605              77
## dodge           69       1772              38
## ford           213       4053              83
## chrysler        23        812              16
## mazda           38        512              29
## jeep            19       1003              18
## subaru          16        515              13
## mercedes       197       1086              70
```

Modeling

Let's first decide which vehicle models and makers to use for your price prediction model. Since more data is better, we'll look for the models with the highest number of prices. We found the Honda Civic and Toyota Camry have among the most prices. We'll fit a linear model and use Kfold cross validation see how our model performs on new observations. The main functions used are:

```
get_model_df =
    # INPUT: data and a vehicle model
    # OUTPUT: data frame ready for modeling
    # DOC:
    #   - convert year to numeric and add age column
    #   - remove years / ages not verified by the text
    #   - remove rows with price of NA
function(df, model)
{
    df = df[ (!is.na(df$model)) & (df$model == model), ]
    df$year = as.numeric( df$year )

    # only keep verified years
    df = df[ df$year_factor != 0, ]

    # add age column
    df$age = 2016 - df$year

    # only row where price is not NA
    df[ !is.na(df$price), ]
}


fill_col_means =
    # INPUT:
    #   - df: data frame with NA values
    #   - cols: columns in df to fill with column mean
    #   - means_given: logical
    #   - means: list with names corresponding to cols
function(df, cols, means_given = FALSE, means = NULL)
{
    for( c in cols ) {
        col_vals = df[[c]]
        na_rows = is.na( col_vals )
```

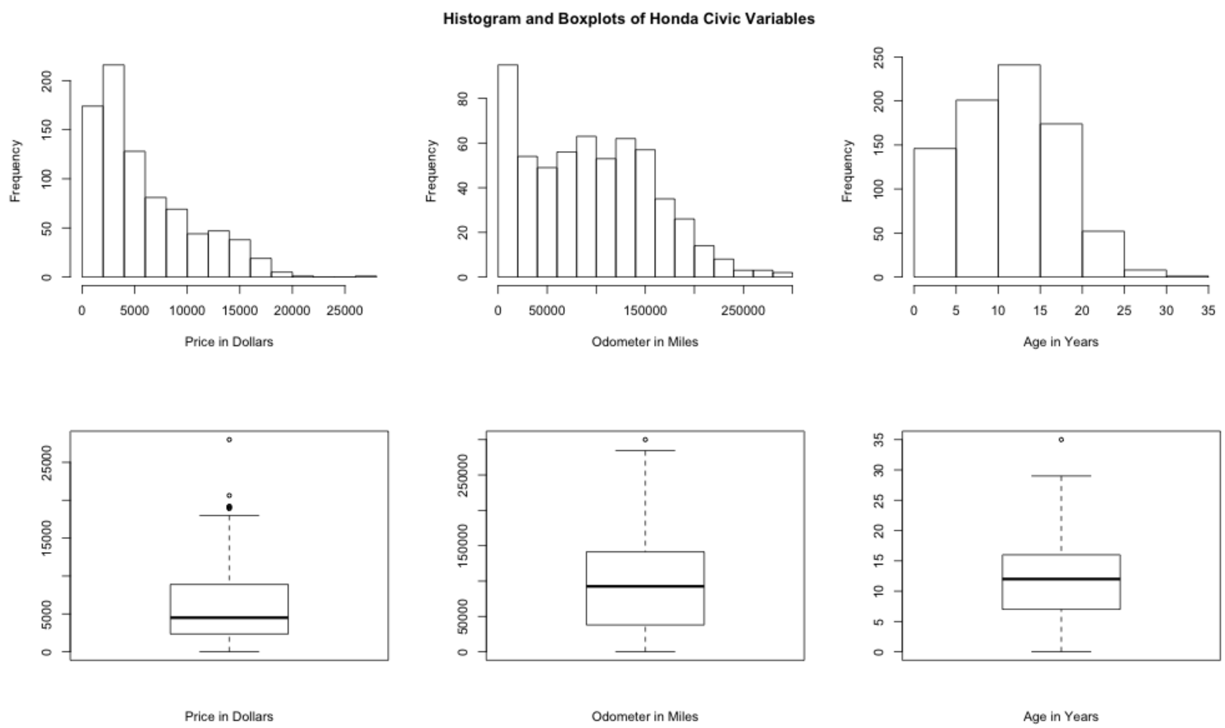
```r
        update_mean =
            if( means_given ) {
                means[[c]]
            } else {
                mean(col_vals, na.rm = TRUE)
            }

        df[ na_rows, c ] = update_mean
    }

    df
}

lm_cv =
    # DOC: fit a linear model of price ~ odometer + age + condition using cross-validation
function(df, k = 5, with_city = FALSE)
{
    # fixing condition
    condition = as.character( df$condition )
    cond = c("excellent", "fair", "good", "like new", "new", "used")
    mask = condition %in% cond
    condition[ !mask ] = "other"
    df$condition = as.factor(condition)

    mse = function(fit) {
        # mse = mean of squared residuals
        mean( summary(fit)$residuals^2 )
    }

    n = nrow(df)
    cv = cvFolds( n, k )
    cv_splits = split( cv$subsets, cv$which )

    cv_mse = sapply( cv_splits, function(holdout) {
        all_index = 1:n

        training_set = df[-holdout, ]

        # fill missing values with column means
        mean_cols = c("odometer", "age")
        training_set = fill_col_means( training_set, mean_cols )
        training_means = lapply( mean_cols, function(col) mean( training_set[[col]] ) )
        names(training_means) = mean_cols

        lm_fit =
            if( with_city ) {
                lm( sqrt(price) ~ odometer + I(odometer^2) + age + I(age^2) + condition + city,
                     data = df, subset = all_index[-holdout] )
            } else {
                lm( sqrt(price) ~ odometer + I(odometer^2) + age + I(age^2) + condition,
                     data = df, subset = all_index[-holdout] )
            }
```

```
        # fill in test data missing values with training value means
        new_data = df[holdout, ]
        new_data = fill_col_means(new_data, mean_cols, TRUE, training_means)

        # square prediction since model fits sqrt(price)
        y_hat = ( predict.lm( lm_fit, new_data) )^2

        sqrt( mean( (new_data$price - y_hat)^2 ) )
    })

    vect_names = c("mse", "average_price")
    setNames( c( mean(cv_mse), mean(df$price) ), vect_names )
}
```

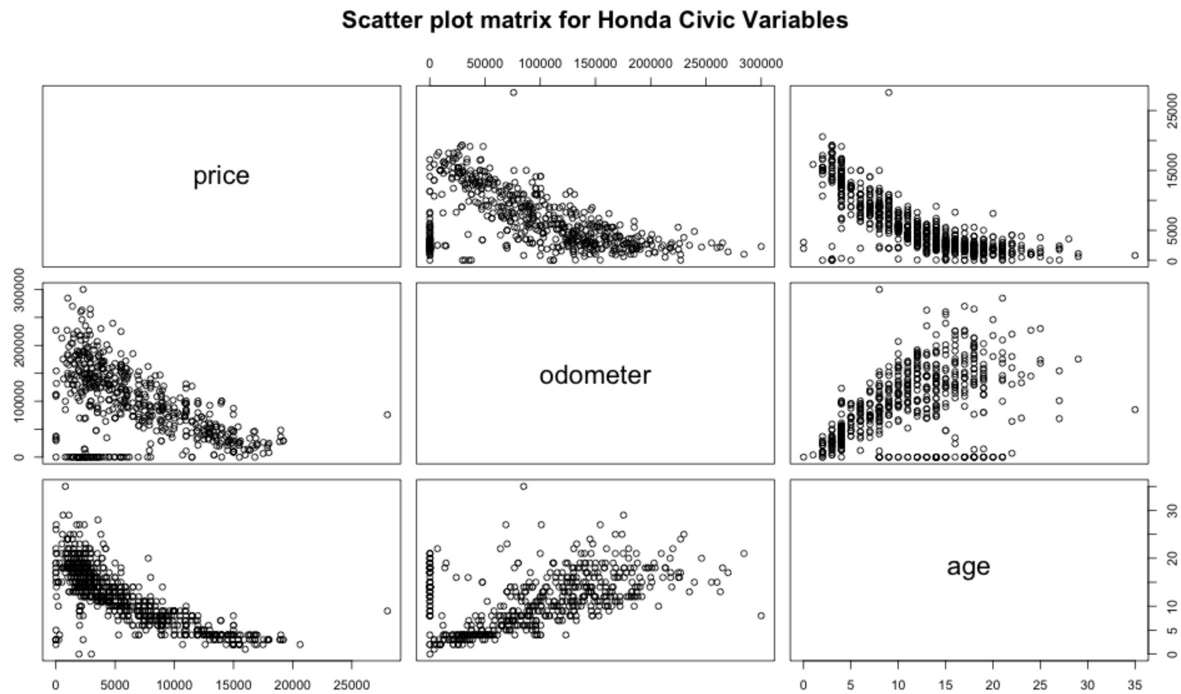
Let's first look at the distribution of price, odometer, and age for both Civic and Camry. First we'll remove some obvious odometer outliers by looking at the body of the text. Looking at the histogram and boxplots we notice that all the variables are right skewed. This is particularly important for the response variable (price) and we'll likely need to perform a transformation. Looking the scatterplot matrix, there appears to be a quadratic relationship between odometer and price, and age and price. We also observe that age and odometer are highly correlated. Note, only plots for Civic are shown to save space, though Camry plots were very similar.

```
# the data for each model
civic_df = get_model_df( vposts, "civic" )
camry_df = get_model_df( vposts, "camry" )

# fixing odometer outliers (verified in body)
civic_df$odometer[ !is.na(civic_df$odometer) & civic_df$odometer == 2630000 ] = 263000
civic_df$odometer[ !is.na(civic_df$odometer) & civic_df$odometer == 580000 ] = 58000

camry_df$odometer[ !is.na(camry_df$odometer) & camry_df$odometer == 2120000 ] = 212000
camry_df$odometer[ !is.na(camry_df$odometer) & camry_df$odometer == 1490000 ] = 149000


# specs for saving plots as png
save_plot = function(file_name, ht = 600) {
    png(file_name, width = 1000, height = ht, pointsize = 16)
    file_name
}

plot_hist_box =
    # for plotting histogram and boxplots for each car model
function( df, title, filename )
{
    plot_hist = function(vect, name) {
        hist(vect, main = "", xlab = name)
    }

    plot_box = function(vect, name) {
        boxplot(vect, main = "", xlab = name)
    }

    variables = c("price", "odometer", "age")
    var_lab = c("Price in Dollars", "Odometer in Miles", "Age in Years")
```

```
    image = save_plot( filename )
    par(mfrow = c(2,3))
    invisible( mapply( plot_hist, df[ ,variables ], var_lab ) )
    invisible( mapply( plot_box, df[ ,variables ], var_lab ) )
    title( main = title, outer = TRUE, line = -2 )

    invisible( dev.off() )
    grid.raster( readPNG(image) )
}

title = "Histogram and Boxplots of Honda Civic Variables"
plot_hist_box( civic_df, title, "./images/civic_1.png" )
```

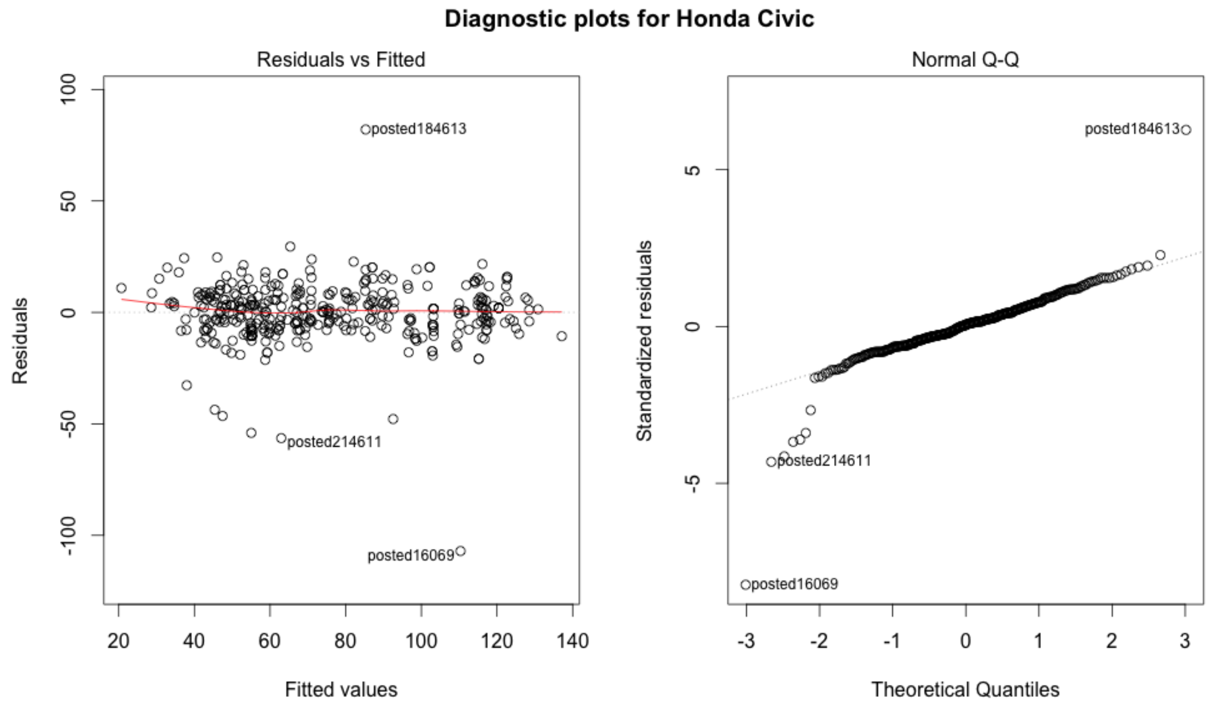



```
plot_pairs =
    # plot scatterplot matrix for different models
function( df, title, filename )
{
    image = save_plot( filename )
    variables = c("price", "odometer", "age")
    pairs( df[ ,variables], main = "Scatter plot matrix for Honda Civic Variables" )
    invisible( dev.off() )
    grid.raster( readPNG(image) )
}

plot_pairs( civic_df, "Scatter plot matrix for Honda Civic Variables", "./images/civic_2.png" )
```



We performed the boxcox procedure to determine that $\sqrt{Y_{price}}$ was the best transformation. The best model used a quadratic term for odometer, quadratic for age, and included condition. We then fit another model including city and found city to be a significant predictor.

```
diagnostic_plots =
    # diagnostic plots for different models
function( df, title, filename )
{
    boxcox( price ~ odometer + I(odometer^2) + age + I(age^2) + condition, data = df)

    lm_fit = lm( sqrt(price) ~ odometer + I(odometer^2) + age + I(age^2) + condition, data = df )
    summary(lm_fit)

    image = save_plot( filename )
    par(mfrow = c(1, 2))
    plot(lm_fit, which = 1)
    plot(lm_fit, which = 2)
    title( main = title, outer = TRUE, line = -2 )
    invisible( dev.off() )
    grid.raster( readPNG(image) )
}

diagnostic_plots( civic_df, "Diagnostic plots for Honda Civic", "./images/civic_3.png")
```




```r
lm_fit = lm( sqrt(price) ~ odometer + I(odometer^2) + age + I(age^2) + condition + city,
             data = civic_df )
```

Finally, using both of the best models found above, we used cross validation to determine the mean squared error of our model to make new predictions. We found that the model which included city was an improvement, however neither model performed particularly well. For example, the average selling price of a Civic was ~\$5,900 but an average the model was off by an average of ~\$2,700.

```r
summary_mat =
    # summarize mse results with and without city in the model
function(df)
{
    list_names = c("no_city", "city")

    output = setNames( list( lm_cv( df ), lm_cv( df, with_city = TRUE ) ), list_names )
    t( do.call(rbind, output) )
}

summary_mat( civic_df )
```

```
##                no_city     city
## mse           2483.307 2483.845
## average_price 5944.537 5944.537
```

```r
summary_mat( camry_df )
```

```
##                no_city     city
## mse           2901.815 2699.917
## average_price 7508.060 7508.060
```