Home Depot 2015

February 16, 2016

1 Home Depot Product Search Relevance

The goal of this analysis is to determine how to predict relevance of a search on Home Depot's website. The training data were labelled by crowdsourcing humans, but the hope is that the text and numerical features will be enough to predict relevance via machine learning

```
In [17]: # Import my library stack
         import pandas as pd, numpy as np, matplotlib.pyplot as plt
         import os
         import pprint
         import copy
         import gc
         %matplotlib inline

         # Some nice display tools for ipython
         from IPython.display import display, HTML

         # There are several files that the Kaggle competition included for this analysis
         DATADIR = "%s/home_depot_2015/"%os.environ["KAGGLE_DATA_DIR"]
```

2 1: Overview of the data

There are three files I will take a peek at here:

- **train.csv** – The training set, which contains products, searches, and relevance scores
- **test.csv** – The test set, which contains products and searches –> I am to predict relevance scores
- **product_descriptions.csv** – Contains product id and a plain text description of the product
- **attributes.csv** – Contains product id and several attributes, but for only a subset of products

I'm just going to preview the first few rows of each.

```
In [18]: # Preview the first 5 rows of a csv file given the path to it
         def preview_data(file, name):
             print display(HTML("<h3>First three rows of %s</h3>"%name))
             preview_df = pd.read_csv(file)
             print display(preview_df.head(1))

         def get_path(file):
             return "%s%s"%(DATADIR, file)

         # Define the files for later
```

```
f_train = get_path('train.csv')
f_test = get_path('test.csv')
f_desc = get_path('product_descriptions.csv')
f_attr = get_path('attributes.csv')

# Do all four
files = [(f_train, 'train'), (f_test, 'test'), (f_desc, 'descriptions'), (f_attr, 'attributes')
map(lambda x: preview_data(x[0], x[1]), files)
```

```
<IPython.core.display.HTML object>


None

   id  product_uid                      product_title    search_term  \
0   2       100001  Simpson Strong-Tie 12-Gauge Angle  angle bracket

   relevance
0          3


None

<IPython.core.display.HTML object>


None

   id  product_uid                      product_title        search_term
0   1       100001  Simpson Strong-Tie 12-Gauge Angle  90 degree bracket


None

<IPython.core.display.HTML object>


None

   product_uid                                product_description
0       100001  Not only do angles make joints stronger, they ...


None

<IPython.core.display.HTML object>


None

   product_uid      name                                              value
0       100001  Bullet01  Versatile connector for various 90° connection...


None
```

Out[18]: [None, None, None, None]

3 2: Feature engineering

After looking at these spreadsheets, I realize there isn't a ton of information with which to work. My initial thought is to do some sort of a word matching procedure (e.g. see if one of the search words matches one of the words in the title (or description, or attributes). Better still, I could take the individual letters in each word of the search query and try to see if they appear consecutively in the raw string of the title, description, or attributes.

Let's give that a try.

In [19]:
```
# I will definitely want to use multiprocessing in this one
from multiprocessing import Pool
```

In [20]:
```
# The attributes are a little trickier. There may be 0 or many attributes per 1 product_uid
# I want to concatenate all strings belonging to a particular product_uid
def collapse_attr(data):
    attr = {}
    for d in data:
        # d is an array of form [product_uid, name, value]
        if not np.isnan(d[0]):
            # Concatenate the attribute if it exists, otherwise add it
            if str(int(d[0])) in attr:
                attr[str(int(d[0]))] = "%s %s"%( attr[str(int(d[0]))], str(d[2]) )
            else:
                attr[str(int(d[0]))] = str(d[2])

    return attr
```

In [21]:
```
# Create the training attribute array, which we will append to the dataframe in the pipeline
ATTR_ARR = collapse_attr(np.array(pd.read_csv(f_attr)))
```

3.0.1 Functions for processing the strings

These will format the strings, add alternate suffixes, and add some common abbreviations if applicable. Since we're dealing with Home Depot data, we have a general idea of what types of abbreviations we might encounter.

In [27]:
```
# Given a word, return all forms of it and its abbreviations
def abbrev(s):
    abrv_groups = [
        ["'", "in", "inches", "inch"],
        ["pounds", "pound", "lbs", "lb"],
        ["sqft", "sq", "square", "foot", "feet", "\"", "ft"],
        ["gal", "gallon", "gallons"],
        ["oz", "ounces", "ounce"],
        ["cm", "centimeters", "centimeter"],
        ["m", "meter", "meters"],
        ["mm", "milimeter", "millimeter", "milimeters", "millimeters"],
        ["a", "amp", "amps", "ampere", "amperes"],
        ["w", "watt", "watts"],
        ["v", "volt", "volts"],
        ["cu", "cubic", "inch", "foot", "feet", "'", "\"", "ft", "in", "inches"],
        ["whirpool","whirlpool", "whirlpoolga", "whirlpoolstainless","stainless"],
        ["and", "&", "+"],
        ["x", "by"]
    ]
```

```
    # If we can match the word in an abbreviation group, return the whole group
    for g in abrv_groups:
        if s in g:
            return g

    # For values like 3x3, we want to also search 3 by 3
    # This is super nasty code but whatever
    s_list = list(s)
    if len(s_list) > 2:
        if s_list[0].isdigit() and s_list[-1].isdigit() and "x" in s_list:
            for i in xrange(25):
                for j in xrange(25):
                    if s=="%sx%s"%(i,j):
                        return [str(i), "by", "xby", str(j), "%sby%s"%(str(i), str(j)), "%sby%

    # If we can't match anything just return an empty array
    return []

# Turn the string into a series of words
def process_string(s):

    # Split into words
    words = s.split(" ")
    # Split by dashes if there are any
    words = np.hstack(np.array(map(lambda x: x.split("-"), words)))
    # Split by x (e.g. 3*3)
    words = np.hstack(np.array(map(lambda x: x.split("*"), words)))
    # Split by /
    words = np.hstack(np.array(map(lambda x: x.split("/"), words)))
    # Get rid of commas
    words = map(lambda x: x.replace(',', ''), words)
    # Get rid of semicolons
    words = map(lambda x: x.replace(';', ''), words)
    # Get rid of colons
    words = map(lambda x: x.replace(':', ''), words)
    # Get rid of periods
    words = map(lambda x: x.replace('.', ''), words)
    # Get rid of blanks
    words = filter(lambda x: x != ' ' and x != '', words)
    return words


def pre_process_strings(query):

    # Lowercase all the things
    query = query.lower()

    # Split the query into an array of char arrays
    query_words = process_string(query)

    return query_words


# This function processes strings like "4x4" or "4'x4'"
```

```
def process_x_by(s):
    if any(i.isdigit() for i in s) and "x" in s:
        new_s = list(s)
        new_s.append(s)
        return new_s
    else:
        return [s]


# Get a list of words similar to the word if applicable
# This will get called with a word in the QUERY
def extension_words(word):

    # Make damn sure everything is lower case
    w = word.lower()

    # 1: Start processing by teasing out AxB strings
    ret_words = process_x_by(w)

    # 2: Add all abbreviations and flatten what was returned in the first step
    abbr = abbrev(w)
    np.hstack([ret_words, abbr])

    # 3a: If the word is small (<4 chars) or contains a number, only add s and return
    if any(i.isdigit() for i in w) or len(list(word)) < 4:
        ret_words.append("%ss"%w)
        return filter(lambda x: x!='' and x!=' ', ret_words)


    # 3b: add suffixed words

    # A list of suffixes
    suffixes = ['s', 'ed', 'ing', 'n', 'en', 'er', 'est', 'ise', 'fy', 'ly',
                'ful', 'able', 'ible', 'hood', 'ess', 'ness', 'less', 'ism',
                'ment', 'ist', 'al', 'ish', 'tion']

    # 4b: If the word ends in one of these suffixes, add the smaller version
    # to strings; otherwise, add this to the end of the word and add that
    for x in xrange(len(suffixes)):
        l = len(suffixes[x])
        if w[-l:] == suffixes[x]:
            ret_words.append(w[0:-l])
        else:
            ret_words.append(w+suffixes[x])

    return filter(lambda x: x!='' and x!=' ', ret_words)
```

In [23]:
```
#_df = pd.read_csv(f_train)
#query_words = _df['search_term'].apply(lambda x: pre_process_strings(x))

#for i in query_words:
#    print i
```

3.0.2 Functions for doing word searches

These will determine if words in the query are in the matching string. We want to know three basic things: * Does the string contain any of the query words? * What fraction of the query words are in the matching words? * What fraction of the chars making up query words are found in the matching words?

In [28]:
```
import sys
"""# Return the size of the matched word (or 0)
# Since the matched word can be one with a suffix,
#   return the length of the original word if the matched word(s) is/are longer
#-----------------------------------------------------------
def match_chars(word, compare):

    strings = extension_words(word)

    # If no words are large enough, return a 0 match rate
    if not strings: return 0

    # Return the sum of characters in the matched words
    # Max that can be returned is the length of the word
    return min(len(word), sum( map(lambda s: len(word) if s in compare else 0, strings) ))
"""

# Determine if the word is in the comparison string
#   @returns 1 or 0
#-----------------------------------------------------------
def match_word(word, compare):

    #strings = filter( lambda x: x!='' and x!=' ', extension_words(word) )
    strings = extension_words(word)

    # If no words are large enough, return a 0 match count
    if not strings: return 0

    # Return the number of words matched
    return min(1, sum( map(lambda s: 1 if s in compare else 0, strings ) ))

# Determine the number of times the word (or any version of it) matches a string
#   @returns array of match counts
def match_word_count(word, compare):

    strings = filter( lambda x: x!='' and x!=' ', extension_words(word) )

    # If no words are large enough, return a 0 match count
    if not strings: return 0

    # Return the number of times any of the words were matched
    return max( map(lambda s: compare.count(s) , strings ) )



## STRING MATCHING FUNCTIONS
##============================================================

## WHOLE WORD matching
```

```
#----------------------------------------------------------------------
# Get the number of unique words that are matched
def matched_words(query, to_match):
    query_words = pre_process_strings(query)
    return sum( map(lambda x: match_word(x, to_match.lower()), query_words) ) if query_words el


# Get the count of all words matched (i.e. if a word is matched more than once, it is counted 
# We pass a minimum length for a given word to be matched
def count_matched_words(query, to_match):
    query_words = pre_process_strings(query)
    return sum( map(lambda x: match_word_count(x, to_match.lower()), query_words) )


## QUERY matching
#----------------------------------------------------------------------
# Whether or not the whole query is in the string
def matched_query(query, to_match):
    return query in to_match

# How many times the whole query is in the string
def count_matched_query(query, to_match):
    return to_match.count(query)



# Return the number of characters in the matched string divided by the size of the query
"""def char_match_fraction(query, to_match):

    # Process the query into words
    query, to_match_words, words = pre_process_strings(query, to_match)

    # Get the number of total matched characters
    #matched_chars = sum( filter(lambda x: x != None, map(lambda x: match_chars(x, to_match), 
    matched_chars = sum( map(lambda x: match_chars(x, to_match), words) )

    # Return the fraction of matched characters / query size
    return float(matched_chars) / len(query)


# Get a fraction of words in the query that match the comparison string
def word_match_fraction(query, to_match):

    # Process the query into words
    query, to_match, words = pre_process_strings(query, to_match)

    # Get the number of matched words
    matched_words = sum( map(lambda x: match_word(x, to_match), words) )

    # Return the fraction of words in the query that matched
    return float(matched_words) / len(query.split(" "))

## BINARY matching
#----------------------------------------------------------------------
```

```
# Return a 1 if the word is matched to a list of strings (words) in the matching string
def word_match(query, to_match):

    # Process the query into words
    query, to_match_words, query_words = pre_process_strings(query, to_match)

    match = min( 1, sum( map(lambda x: match_word(x, to_match_words), query_words) ) )
    return match


# Return a 1 if the word is matched in the to_match string
def char_match(query, to_match):

    # Process the query into words
    query, to_match_words, query_words = pre_process_strings(query, to_match)

    match = min( 1, sum( map(lambda x: match_chars(x, to_match), query_words) ) )
    return match

# Return a 1 if the whole original query is in the matched string
def whole_query_match(query, to_match):
    return 1 if query in to_match else 0

# Return a 1 if all the words in the original query are in the matched string
def whole_query_words_match(query, to_match):
    matches = map(lambda x: x in to_match, query.split(" "))
    return min(matches)
"""
```

```
Out[28]: 'def char_match_fraction(query, to_match):\n    \n    # Process the query into words\n    query
```

4 3: Feature Engineering Pipeline

I will start by engineering new features and removing the long strings in my data set. Specifically, I want to add

- Match rates of query relating to title and description (determined by char_match_fraction function)
- String length columns of query, description, and title columns

I will go ahead and build a new training set based on this.

```
In [29]: import time

         # POOL lambda functions (need to be defined outside the function that calls them)
         # With multiprocessing, we can't use lambda, so I will define some basic functions here
         #==============================================================================
         """
         def lambda_char_match_fraction(a):
             return char_match_fraction( str(a[0]), str(a[1]) )

         def lambda_word_match_fraction(a):
             return word_match_fraction( str(a[0]), str(a[1]) )
```

```
def lambda_word_match_bin(a):
    return word_match( str(a[0]), str(a[1]) )

def lambda_char_match_bin(a):
    return char_match( str(a[0]), str(a[1]) )

def lambda_whole_query_words_match(a):
    return whole_query_words_match( str(a[0]), str(a[1]) )

def lambda_whole_query_match(a):
    return whole_query_match( str(a[0]), str(a[1]) )
"""
def lambda_in_attr(a):
    return ATTR_ARR[str(int(a))] if str(int(a)) in ATTR_ARR else ''

def lambda_char_len(a):
    return len( filter(lambda l: l != " " and l != "-" and l !="*", list( str(a) )) )
    #return 1 if np.isnan(l) or l < 1 else l

def lambda_word_len(a):
    return len( pre_process_strings(str(a)) )

def l_matched_words(a):
    return matched_words( str(a[0]), str(a[1]) )

def l_count_matched_words(a):
    return count_matched_words( str(a[0]), str(a[1]) )

def l_matched_query(a):
    return matched_query( str(a[0]), str(a[1]) )

def l_count_matched_query(a):
    return count_matched_query( str(a[0]), str(a[1]) )

## DATA PIPELINE
#================================================================================
# Given the data (train or test) and description files,
# perform a series of operations to produce a data set on which we can do ML
def feature_pipeline(data_file, **kwargs):

    # Define my multiprocessing pool and start the timer
    POOL = Pool(maxtasksperchild=1000)
    start = time.time()

    ## Read files
    #----------------------------------------------------------

    # Read the initial train.csv and join it to product descriptions
    _df = pd.read_csv(data_file)

    # Add in descriptions because they are 1:1
    df = pd.merge(_df, pd.read_csv(f_desc), how='outer')

    ## ADD LENGTH Columns (filter out whitespace)
```

```
#-------------------------------------------------------
def ln(x): return np.log(x)

# Char lengths
df['desc_char_l'] = pd.Series( POOL.map(lambda_char_len, df['product_description']) )#.app
df['title_char_l'] = pd.Series( POOL.map(lambda_char_len, df['product_title']) )#.apply(ln
df['query_char_l'] =  pd.Series( POOL.map(lambda_char_len, df['search_term']) )#.apply(ln)

# Word lengths
df['desc_word_l'] = pd.Series( POOL.map(lambda_word_len, df['product_description']) )#.app
df['title_word_l'] = pd.Series( POOL.map(lambda_word_len, df['product_title']) )#.apply(ln
df['query_word_l'] = pd.Series( POOL.map(lambda_word_len, df['search_term']) )#.apply(ln)



## ADD MATCH COLUMNS
#-------------------------------------------------------

# Description columns
desc_zip = np.dstack( ( np.array(df['search_term']), np.array(df['product_description']) ))
df['desc_matched'] = pd.Series( POOL.map(l_matched_words, desc_zip ) )
df['desc_count'] = pd.Series( POOL.map(l_count_matched_words, desc_zip ) )
df['desc_query_matched'] = pd.Series( POOL.map(l_matched_query, desc_zip ) )
df['desc_query_count'] = pd.Series( POOL.map(l_count_matched_query, desc_zip ) )
df['desc_frac_matched'] = df['desc_matched'] / df['query_word_l']

#df['desc_char'] = pd.Series( POOL.map(lambda_char_match_fraction, desc_zip ) )
#df['desc_word'] = pd.Series( POOL.map(lambda_word_match_fraction, desc_zip ) )
#df['desc_word_bin'] = pd.Series( POOL.map(lambda_word_match_bin, desc_zip) )
#df['desc_char_bin'] = pd.Series( POOL.map(lambda_char_match_bin, desc_zip) )
#df['desc_whole_query'] = pd.Series( POOL.map(lambda_whole_query_match, desc_zip) )
#df['desc_whole_query_words'] = pd.Series( POOL.map(lambda_whole_query_words_match, desc_z

# Title columns
title_zip = np.dstack( (np.array(df['search_term']), np.array(df['product_title']) ))[0]
df['title_matched'] = pd.Series( POOL.map(l_matched_words, title_zip ) )
df['title_count'] = pd.Series( POOL.map(l_count_matched_words, title_zip ) )
df['title_query_matched'] = pd.Series( POOL.map(l_matched_query, title_zip ) )
df['title_query_count'] = pd.Series( POOL.map(l_count_matched_query, title_zip ) )
df['title_frac_matched'] = df['title_matched'] / df['query_word_l']

#df['title_char'] = pd.Series( POOL.map(lambda_char_match_fraction, title_zip ) )
#df['title_word'] = pd.Series( POOL.map(lambda_word_match_fraction, title_zip ) )
#df['title_word_bin'] = pd.Series( POOL.map(lambda_word_match_bin, title_zip) )
#df['title_char_bin'] = pd.Series( POOL.map(lambda_char_match_bin, title_zip) )
#df['title_whole_query'] = pd.Series( POOL.map(lambda_whole_query_match, title_zip) )
#df['title_whole_query_words'] = pd.Series( POOL.map(lambda_whole_query_words_match, title


# Combo columns
#df['desc_char_word'] = df['desc_char'] * df['desc_word']
#df['title_char_word'] = df['title_char'] * df['title_word']
#df['desc_title_char'] = df['desc_char'] * df['title_char']
#df['desc_title_word'] = df['desc_word'] * df['title_word']
```

```
        ## ADD ATTRIBUTE columns
        #-----------------------------------------------------------
        # First we need the attr column added to the df
        df['attr'] = POOL.map(lambda_in_attr, df['product_uid'])

        df['attr_char_l'] = pd.Series( POOL.map(lambda_char_len, df['attr']) )#.apply(ln)
        df['attr_word_l'] = pd.Series( POOL.map(lambda_word_len, df['attr']) )#.apply(ln)

        # Now build the columns normally
        attr_zip = np.dstack( (np.array(df['search_term']), np.array(df['attr']) ))[0]
        df['attr_matched'] = pd.Series( POOL.map(l_matched_words, attr_zip ) )
        df['attr_count'] = pd.Series( POOL.map(l_count_matched_words, attr_zip ) )
        df['attr_query_matched'] = pd.Series( POOL.map(l_matched_query, attr_zip ) )
        df['attr_query_count'] = pd.Series( POOL.map(l_count_matched_query, attr_zip ) )
        df['attr_frac_matched'] = df['attr_matched'] / df['query_word_l']

        #df['attr_char'] = pd.Series( POOL.map(lambda_char_match_fraction, attr_zip ) )
        #df['attr_word'] = pd.Series( POOL.map(lambda_word_match_fraction, attr_zip ) )
        #df['attr_word_bin'] = pd.Series( POOL.map(lambda_word_match_bin, attr_zip) )
        #df['attr_char_bin'] = pd.Series( POOL.map(lambda_char_match_bin, attr_zip) )
        #df['attr_whole_query'] = pd.Series( POOL.map(lambda_whole_query_match, attr_zip) )
        #df['attr_whole_query_words'] = pd.Series( POOL.map(lambda_whole_query_words_match, attr_z



        ## REMOVE TEXT columns
        #----------------------------------------------------------
        map(lambda x: df.pop(x), ['product_uid', 'search_term', 'product_title', 'product_descript

        print "df size: %s"%str(np.shape(df))
        ## DROP ROWS NaN values (but only if kwargs does not include submission)
        if 'submission' in kwargs:
            clean_df = df.copy()
        else:
            clean_df = df.copy().dropna()

        ## POP OFF THE IDS and return the two dfs
        #----------------------------------------------------------
        ids = clean_df['id']
        clean_df.pop('id')


        print "clean_df size: %s"%str(np.shape(clean_df))
        print display(HTML("<font color='blue'><b>Data pipelined in %s s</b></font>"%(time.time()-
        return clean_df, ids
```

```
In [30]: df, ids = feature_pipeline(f_train)
```

```
df size: (143828, 25)
clean_df size: (74067, 24)
```

```
<IPython.core.display.HTML object>
```

```
None
```

5 4: Plot the Feature Distributions

As a sanity check, it is good to check out the first few lines of my data frame and also to graph the features to make sure there are actual distributions of them.

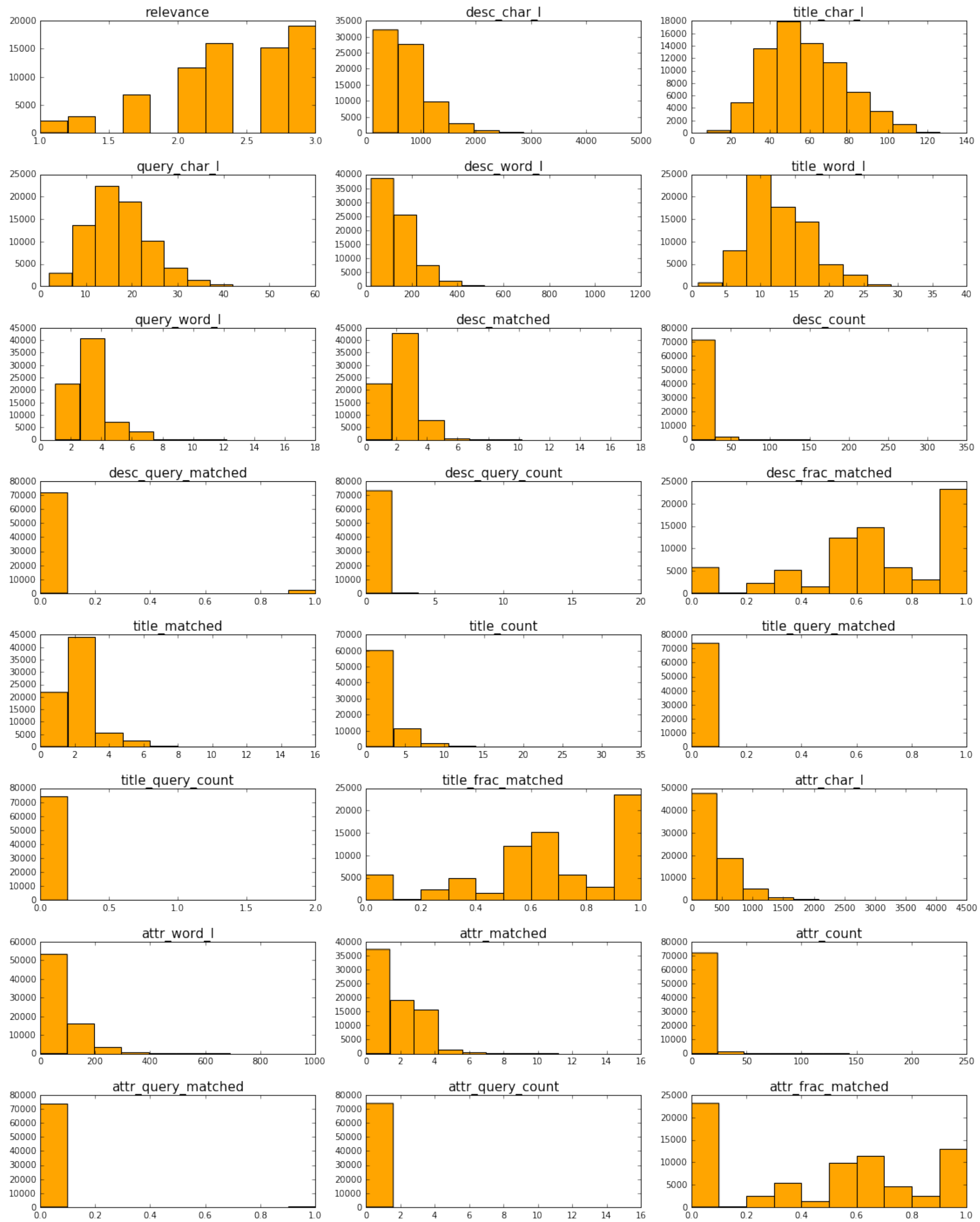
In [31]:
```
# Plot the distribution (histogram) of my features
def plot_hist(col, name):
    fig = plt.figure()
    ax = fig.add_subplot(111)

    plt.title('%s' %name, fontsize=15)
    #fig.colorbar(cax)
    plt.hist(col)
    #plt.xlabel('Value')
    #plt.ylabel('Count')
    plt.show()
```

In [32]:
```
# Feature plots
features = list(df.columns.values)
# Plot a bunch of stuff
dim = len(features)/3 + 1 if len(features)%3 > 0 else len(features)/3

f, axarr = plt.subplots(dim, 3, figsize=(16,20))
plt.tight_layout(pad=3)

for i in range(0, dim):
    # For each row
    for j in range(0, 3):
        # For each element in the row
        if (i*3 + j) < len(features):
            # As long as the chart exists in the tuple
            axarr[i][j].hist( df[ features[i*3+j] ], color='orange' )
            axarr[i][j].set_title( features[i*3+j], fontsize=15 )
```



6 5: Learning

A few notes about the distributions:

- The string length columns look to be distributed pretty nicely

- The description matches are heavily favored to the right (meaning the strings match well); we would expect this from a search engine
- The relevance scores are also heavily favored to the right (again, we expect this engine to work reasonably well, so this makes makes sense)

Everything so far looks reasonable. Now I will go ahead and set up a machine learning pipeline to test some algorithms on the training/test data.

```
In [33]: from sklearn import pipeline, grid_search
```

```
In [34]: from sklearn.ensemble import RandomForestRegressor as RF, BaggingRegressor as BR
```

6.0.3 5.1 Define X and y matrices

```
In [35]: # Define the X and y matrices
         def define(df):
             if 'relevance' in df:
                 y = pd.Series(df['relevance'])
                 df.pop('relevance')
                 X = np.array(df.copy())
                 return X, y
             else:
                 print 'X and y already defined.'
                 return

         train_X, train_y = define(df)
```

6.0.4 5.2 Pipeline Learners

```
In [38]: rfr = RF(n_jobs=-1, n_estimators=400)
         __models = pipeline.Pipeline([('RF',rfr)])
```

6.0.5 5.3 Setup Grid Search

```
In [39]: from sklearn.metrics import mean_squared_error, make_scorer

         ## Define the loss function
         # This is a custom root-MSE (RMSE) function with tighter errors
         def f_mse(y, y_pred):
             return mean_squared_error(y, y_pred)**0.5

         RMSE = make_scorer(f_mse, greater_is_better=False)
```

```
In [41]: param_grid = {
             'RF__max_depth': [None, 10, 15]
         }

         grid_search_args = {
             'estimator': __models,
             'param_grid': param_grid,
             'n_jobs': -1,
             'cv': 5,
             'verbose': 0,
             'scoring': RMSE
         }
```

```
start = time.time()

model = grid_search.GridSearchCV(**grid_search_args)
```

In [46]:
```
gc.collect()
```

Out[46]: 88

6.0.6 5.5 Run Grid Search CV

In [44]:
```
gc.collect()
model.fit(train_X, train_y)
gc.collect()
print "GridSearchCV completed in %s s"%(time.time()-start)
print "Best parameters found by grid search: %s"%model.best_params_
print "Best CV score: %s"%model.best_score_
```

```
GridSearchCV completed in 552.102079153 s
Best parameters found by grid search: {'RF__max_depth': 15}
Best CV score: -0.477999193822
```

7 6: Test Set

Now I will move over to the test set. I will predict based on the model I just generated.

7.0.7 6.1 Pipeline Features

In [45]:
```
df_test, df_test_ids = feature_pipeline(f_test)
```

```
df size: (193661, 24)
clean_df size: (166693, 23)

<IPython.core.display.HTML object>


None
```

7.0.8 6.2 Predict test_y

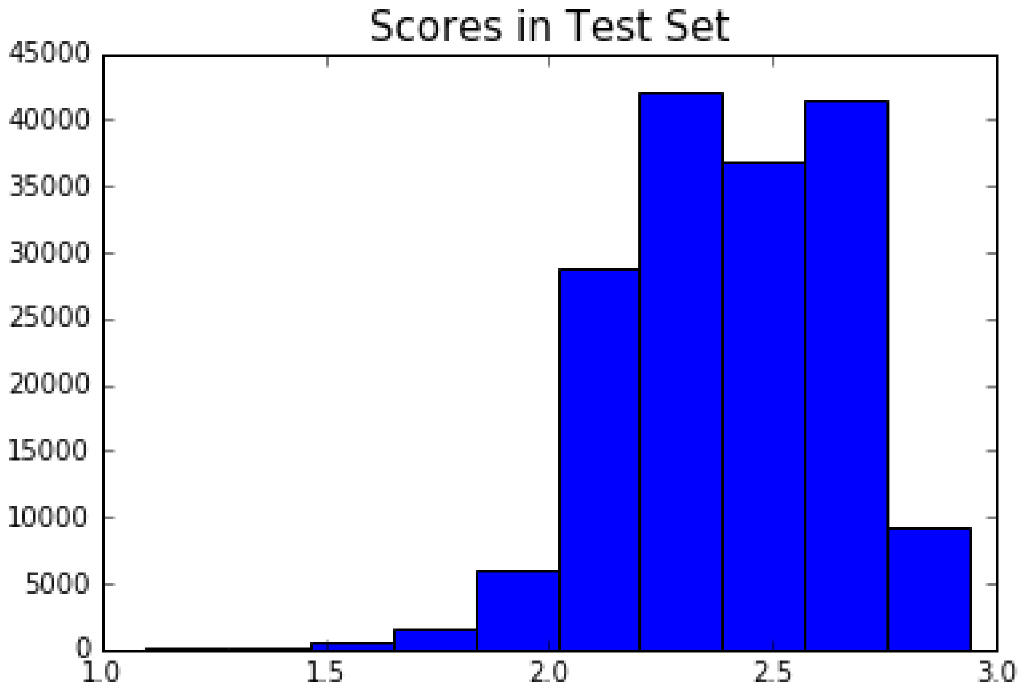
In [47]:
```
# Define a new X and predict y using the model we fit earlier
test_X = np.array(df_test.copy())
test_y = model.predict(test_X)

final_y = map(lambda x: 1 if x < 1. else 3 if x > 3. else x, test_y)
```

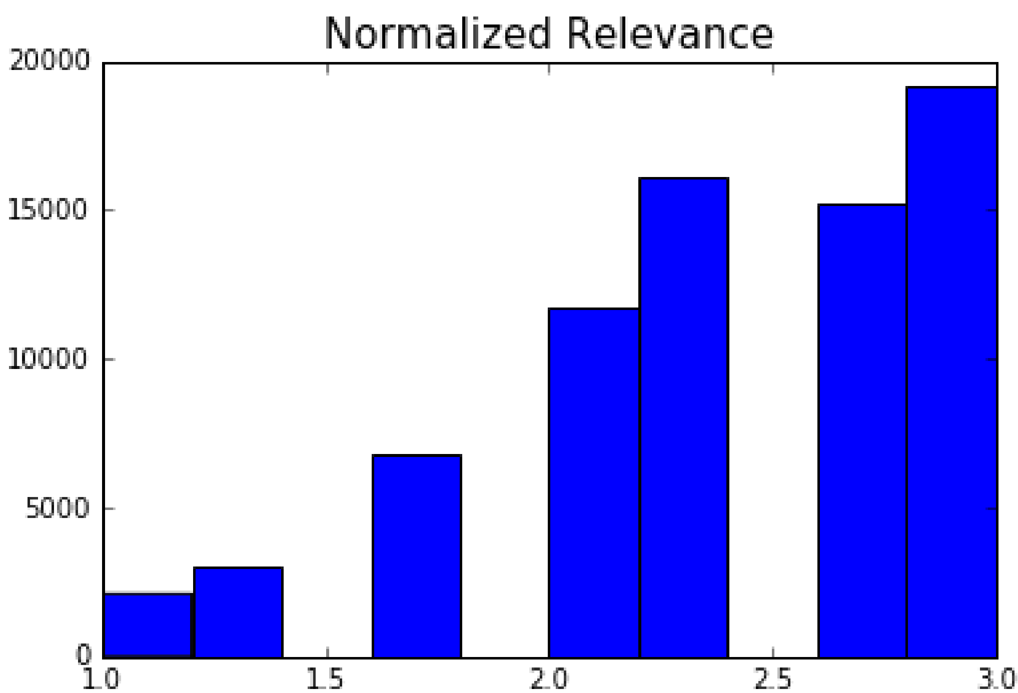
7.0.9 6.3 Look at distributions

After looking at the data, I want to look at the distribution and compare it to the one from the test set.

In [48]:
```
plot_hist(final_y, 'Scores in Test Set')
```




```
In [49]: plot_hist(train_y, 'Normalized Relevance')
```



```
In [50]: print "Test mean: %s, std: %s"%(np.mean(final_y), np.std(final_y))
         print "Training mean: %s, std: %s"%(np.mean(train_y), np.std(train_y))
```

```
Test mean: 2.40364655224, std: 0.246189110697
Training mean: 2.38163379103, std: 0.533980343669
```

8 7: Submission

Now I am finally ready to write the submission file!

```
In [51]: ## Join ids with submission y
         def submit(ids, relevances, file_name):

             # ids need to be integers
             ids = map(lambda x: int(x), ids)

             # Build a dataframe
             submission = pd.DataFrame(index=ids)
             submission.index.name = 'id'
             submission['relevance'] = relevances

             # Print the head just for a sanity check
             submission.head(10)

             # Write the file
             path = "%s/%s.csv"%(DATADIR, file_name)
             submission.to_csv(path, header=True, index=True)
```

```
In [52]: submit(df_test_ids, final_y, 'submission3')
```

```
In [ ]:
```