Home Depot 2015

February 13, 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 [1]: # Import my library stack
        import pandas as pd, numpy as np, matplotlib.pyplot as plt
        import os
        import pprint
        import copy
        %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 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 [2]: # Preview the first 5 rows of a csv file given the path to it
        def preview_data(file, name):
            print display(HTML("<br><h2>First five rows of %s</h2>"%name))
            preview_df = pd.read_csv(file)
            print display(preview_df.head(5))

        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  \
0   2       100001                  Simpson Strong-Tie 12-Gauge Angle
1   3       100001                  Simpson Strong-Tie 12-Gauge Angle
2   9       100002  BEHR Premium Textured DeckOver 1-gal. #SC-141 ...
3  16       100005  Delta Vero 1-Handle Shower Only Faucet Trim Ki...
4  17       100005  Delta Vero 1-Handle Shower Only Faucet Trim Ki...

          search_term  relevance
0       angle bracket       3.00
1           l bracket       2.50
2           deck over       3.00
3    rain shower head       2.33
4  shower only faucet       2.67


None

<IPython.core.display.HTML object>


None

   id  product_uid                      product_title  \
0   1       100001  Simpson Strong-Tie 12-Gauge Angle
1   4       100001  Simpson Strong-Tie 12-Gauge Angle
2   5       100001  Simpson Strong-Tie 12-Gauge Angle
3   6       100001  Simpson Strong-Tie 12-Gauge Angle
4   7       100001  Simpson Strong-Tie 12-Gauge Angle

                search_term
0          90 degree bracket
1           metal l brackets
2           simpson sku able
3       simpson strong  ties
4  simpson strong tie hcc668


None

<IPython.core.display.HTML object>


None
```

```
   product_uid                                product_description
0       100001  Not only do angles make joints stronger, they ...
1       100002  BEHR Premium Textured DECKOVER is an innovativ...
2       100003  Classic architecture meets contemporary design...
3       100004  The Grape Solar 265-Watt Polycrystalline PV So...
4       100005  Update your bathroom with the Delta Vero Singl...


None

<IPython.core.display.HTML object>


None

   product_uid      name                                              value
0       100001  Bullet01  Versatile connector for various 90° connection...
1       100001  Bullet02  Stronger than angled nailing or screw fastenin...
2       100001  Bullet03  Help ensure joints are consistently straight a...
3       100001  Bullet04             Dimensions: 3 in. x 3 in. x 1-1/2 in.
4       100001  Bullet05                          Made from 12-Gauge steel


None

Out[2]: [None, None, None, None]
```

3 Feature engineering: Part 1

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

In [4]: # 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 [5]: # 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)))
```

In [6]:
```
import sys
## UTILITY FUNCTIONS
##==========================================================
# 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)))
    # 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)
    return words


def pre_process_strings(query, to_match):

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

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

    # Split the matching string
    to_match = process_string(to_match)

    # Join words in the query
    for i in xrange(1, len(words)):
        words.append(words[i] + words[i-1])

    return query, to_match, words


# Get a list of words similar to the word if applicable
def extension_words(word):

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

    # Make sure the word is > 4 characters long and doesn't contain numbers
    if any(str(i) in word for i in xrange(10)) or len(list(word)) < 4:
        return [w]

    # Build a list of suitable strings
    strings = [w]

    # 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']

    # 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]:
            strings.append(w[0:-l])
        else:
            strings.append(w+suffixes[x])

    return strings



# Return the highest fraction of consecutive matching characters
# Throw the word out if it's too small or if <1/2 of chars are found
#---------------------------------------------------------
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
    return sum( map(lambda s: len(word) if s in compare else 0, strings ) )


# Determine if the word is in the comparison string
#---------------------------------------------------------
def match_word(word, compare):

    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 sum( map(lambda s: 1 if s in compare else 0, strings ) )


## STRING MATCHING FUNCTIONS
##===========================================================
# This will take the search query as well as a string to match it against.
# The query will be split into words and those will be split into characters.
#   -If any consective number of such characters match the string, it will record that match.
#   -The match will be a percentage of the word matching combined with a percentage of the word
#      that matched.
```

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

    # Process the query into words
    query, to_match, 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), w
    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(" "))


# Return a 1 if the word is matched to a list of strings 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
```

In [7]:
```
word = "zip-tie"
char_match_fraction("another tie", word)
```

Out[7]: 0.2727272727272727

4 Feature Engineering Pipeline: Part 1

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 [8]:
```
import time

## Utility functions
#---------------------------------------------------------

# Normalize the column to unit-length 1
# Input is a datafram column
def norm(col):
    # Get the mean
    mu = col.mean()
    std = col.std()
    # Cut off the tails
    cp = pd.Series(map(lambda x: (mu-3*std) if x < (mu-3*std) else (mu+3*std) if x > (mu+3*std)
    return cp / cp.max()
    #return col / col.max()


# 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_char_len(a):
    l = len( filter(lambda l: l != " ", list( str(a) )) )
    return 0 if np.isnan(l) else l

def lambda_word_len(a):
    l = len(str(a).split(" "))
    return l if l > 1 else 0

def lambda_in_attr(a):
    return ATTR_ARR[str(int(a))] if str(int(a)) in ATTR_ARR else ''

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

def lambda_char_match(a):
    return char_match( 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 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 MATCH COLUMNS
    #--------------------------------------------------------

    # Description columns
    desc_zip = np.dstack( ( np.array(df['search_term']), np.array(df['product_description']) ))
    df['desc_char'] = norm(pd.Series( POOL.map(lambda_char_match_fraction, desc_zip ) ))
    df['desc_word'] = norm(pd.Series( POOL.map(lambda_word_match_fraction, desc_zip ) ))
    df['desc_word_bin'] = pd.Series( POOL.map(lambda_word_match, desc_zip) )
    df['desc_char_bin'] = pd.Series( POOL.map(lambda_char_match, desc_zip) )


    # Title columns
    title_zip = np.dstack( (np.array(df['search_term']), np.array(df['product_title']) ))[0]
    df['title_char'] = norm( pd.Series( POOL.map(lambda_char_match_fraction, title_zip ) ))
    df['title_word'] = norm( pd.Series( POOL.map(lambda_word_match_fraction, title_zip ) ))
    df['title_word_bin'] = pd.Series( POOL.map(lambda_word_match, title_zip) )
    df['title_char_bin'] = pd.Series( POOL.map(lambda_char_match, title_zip) )


    # 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 LENGTH Columns (filter out whitespace)
    #--------------------------------------------------------

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

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



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

    # Now build the columns normally
    attr_zip = np.dstack( (np.array(df['search_term']), np.array(df['attr']) ))[0]
    df['attr_char'] = norm(pd.Series( POOL.map(lambda_char_match_fraction, attr_zip ) ))
    df['attr_word'] = norm(pd.Series( POOL.map(lambda_word_match_fraction, attr_zip ) ))
    df['attr_char_l'] = norm( pd.Series( POOL.map(lambda_char_len, df['attr']) ))
    df['attr_word_l'] = norm( pd.Series( POOL.map(lambda_word_len, df['attr']) ))
    df['attr_word_bin'] = pd.Series( POOL.map(lambda_word_match, attr_zip) )
    df['attr_char_bin'] = pd.Series( POOL.map(lambda_char_match, attr_zip) )

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

    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
    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()-st
    return clean_df, ids

## For submission, we need ids in order
## This function pops off the id column and returns it as a series
def get_id_col(data_file):
    df = pd.read_csv(data_file)
    print "df shape: %s"%str(np.shape(df))
    new_df = df.copy().dropna()
    print "new_df shape: %s"%str(np.shape(new_df))
    return pd.Series(new_df['id'])
```

```
In [9]: df, ids = pipeline(f_train)
```

```
df size: (143828, 26)
clean_df size: (74067, 25)

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

```
None
```

In [10]: `df.head()`

Out[10]:

```
   relevance  desc_char  desc_word  desc_word_bin  desc_char_bin  title_char  \
0       3.00   0.250579   0.293828              1              1    0.224371
1       2.50   0.000000   0.000000              0              1    0.000000
2       3.00   0.579115   0.587657              1              1    0.000000
3       2.33   0.244314   0.195886              1              0    0.218761
4       2.67   0.723894   0.783542              1              1    0.518545

   title_word  title_word_bin  title_char_bin  desc_char_word      ...       \
0    0.292573               1               0        0.073627      ...
1    0.000000               0               1        0.000000      ...
2    0.000000               0               0        0.340321      ...
3    0.195049               1               0        0.047858      ...
4    0.585146               1               0        0.567201      ...

   query_char_l  desc_word_l  title_word_l  query_word_l  attr_char  \
0      0.338912     0.367827      0.156325      0.275363   0.000000
1      0.225941     0.367827      0.156325      0.275363   0.000000
2      0.225941     0.479030      0.429894      0.275363   0.359902
3      0.395398     0.296543      0.508057      0.413044   0.506112
4      0.451883     0.296543      0.508057      0.413044   0.899755

   attr_word  attr_char_l  attr_word_l  attr_word_bin  attr_char_bin
0   0.000000     0.241120     0.252192              0              1
1   0.000000     0.241120     0.252192              0              1
2   0.361877     0.526712     0.557080              1              1
3   0.482503     0.358553     0.380169              1              0
4   0.965007     0.358553     0.380169              1              1

[5 rows x 25 columns]
```

5 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.

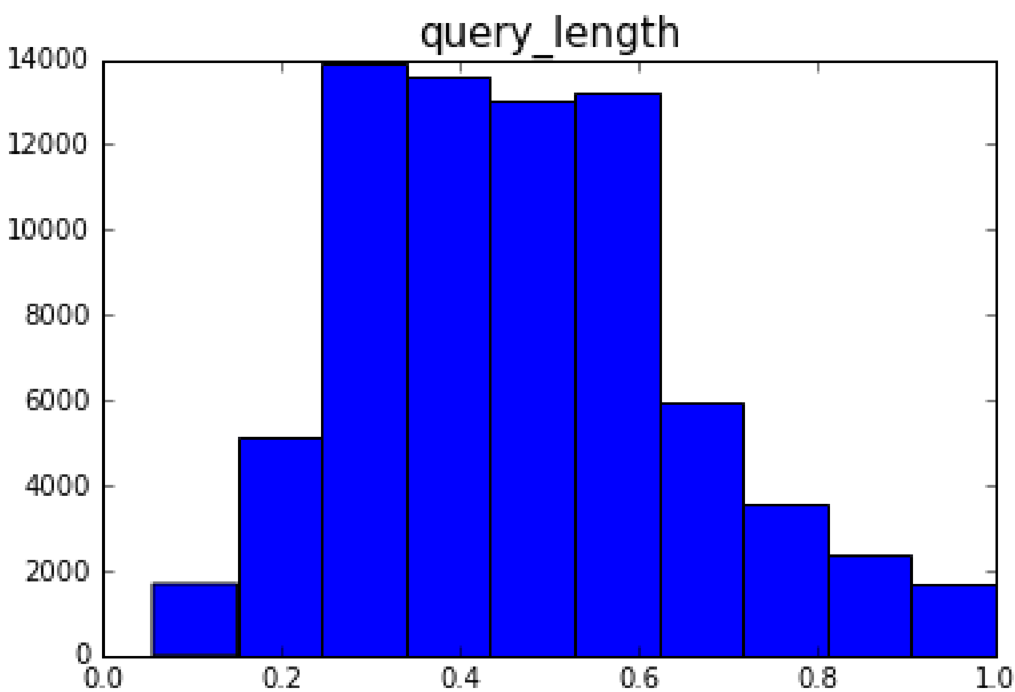
In [11]:

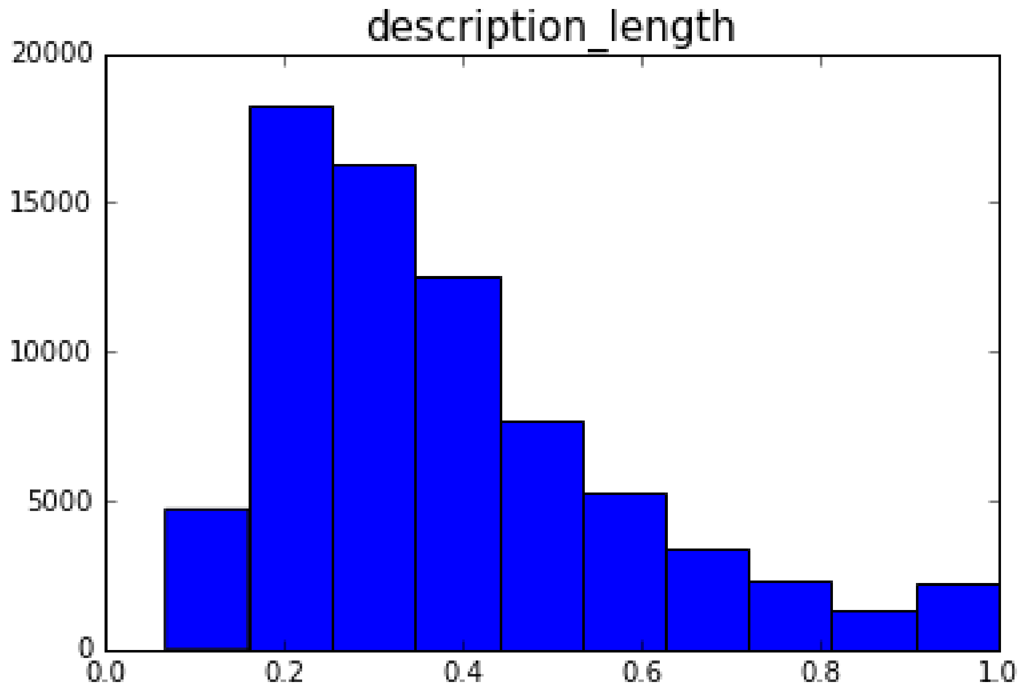
```python
# Plot the distribution (histogram) of my features
def plot_hist(col, name):
    fig = plt.figure()
    ax = fig.add_subplot(111)

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

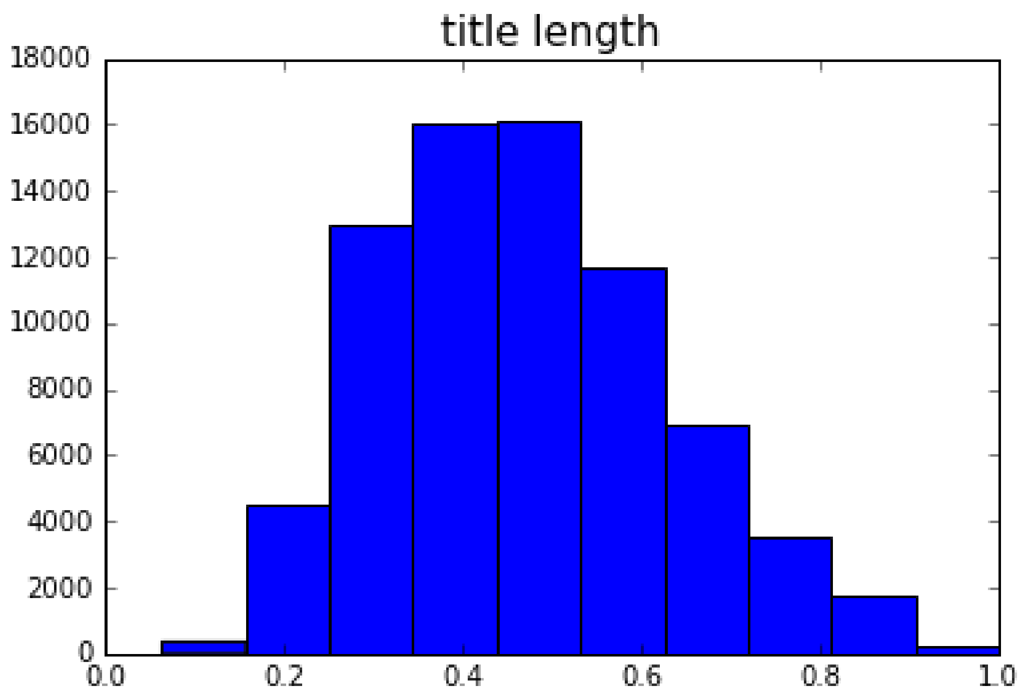
In [12]:

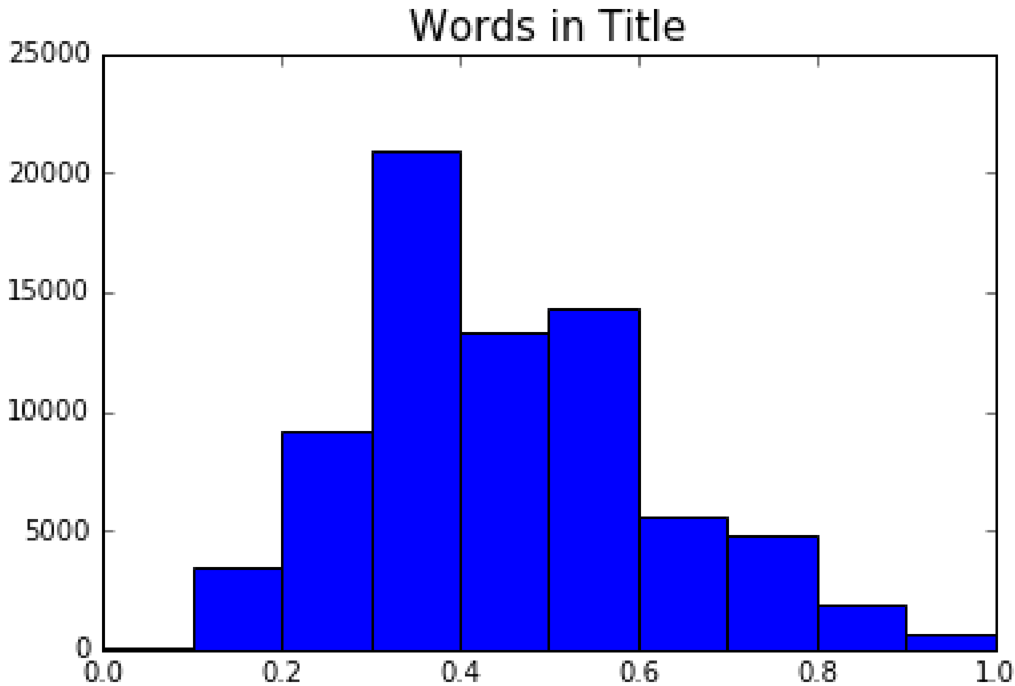
```python
# Feature plots
features = [
```

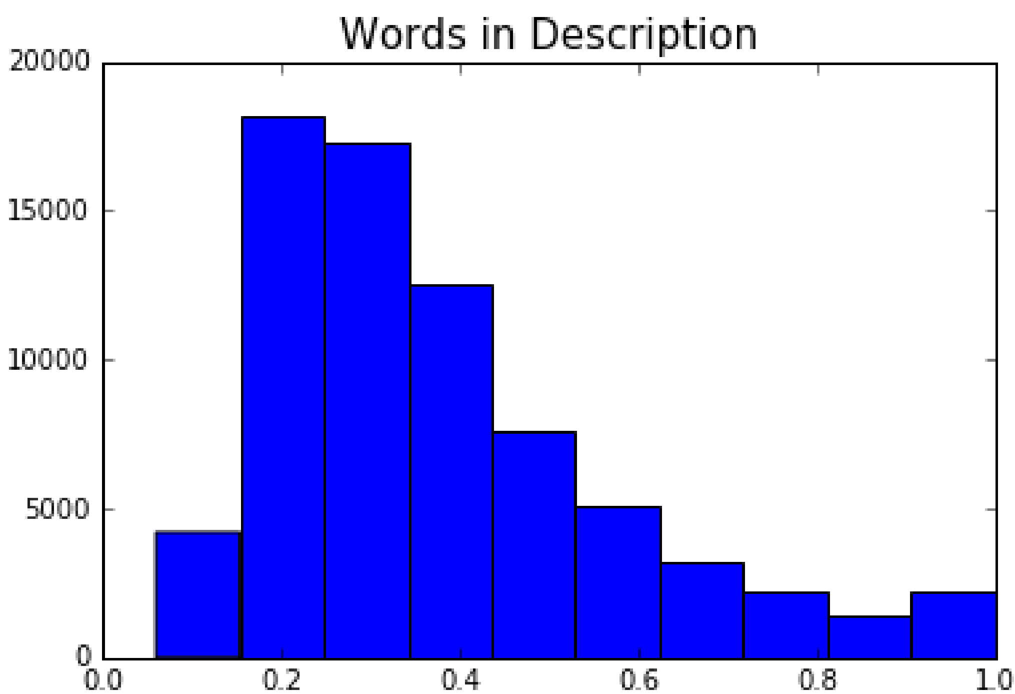
```
    ('query_char_l', 'query_length'),
    ('desc_char_l', 'description_length'),
    ('title_char_l', 'title length'),
    ('title_word_l', 'Words in Title'),
    ('desc_word_l', 'Words in Description'),
    ('query_word_l', 'Words in query'),
    ('desc_word', 'Fraction of Words in Description'),
    ('title_word', 'Fraction of Words in Title'),
    ('desc_char', 'Fraction of Chars in Description'),
    ('title_char', 'Fraction of Chars in Title'),
    ('attr_char', 'Fraction of Chars in Attr'),
    ('attr_char_l', 'Characters in Attr'),
    ('attr_word', 'Fraction of Words in Attr'),
    ('attr_word_l', 'Words in Attr'),
    ('attr_char_bin', 'attr_char_bin'),
    ('attr_word_bin', 'attr_word_bin'),
    ('title_char_bin', 'title_char_bin'),
    ('title_word_bin', 'title_word_bi'),
    ('desc_char_bin', 'desc_char_bin'),
    ('desc_word_bin', 'desc_word_bin')
]

map(lambda x: plot_hist(df[x[0]], x[1]), features)
```

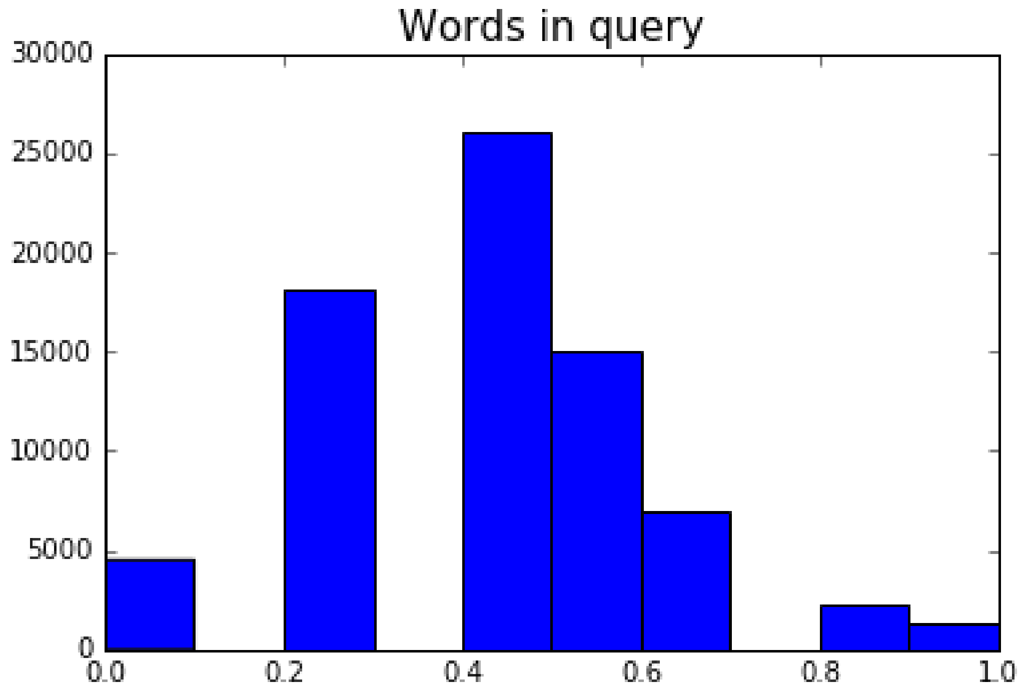


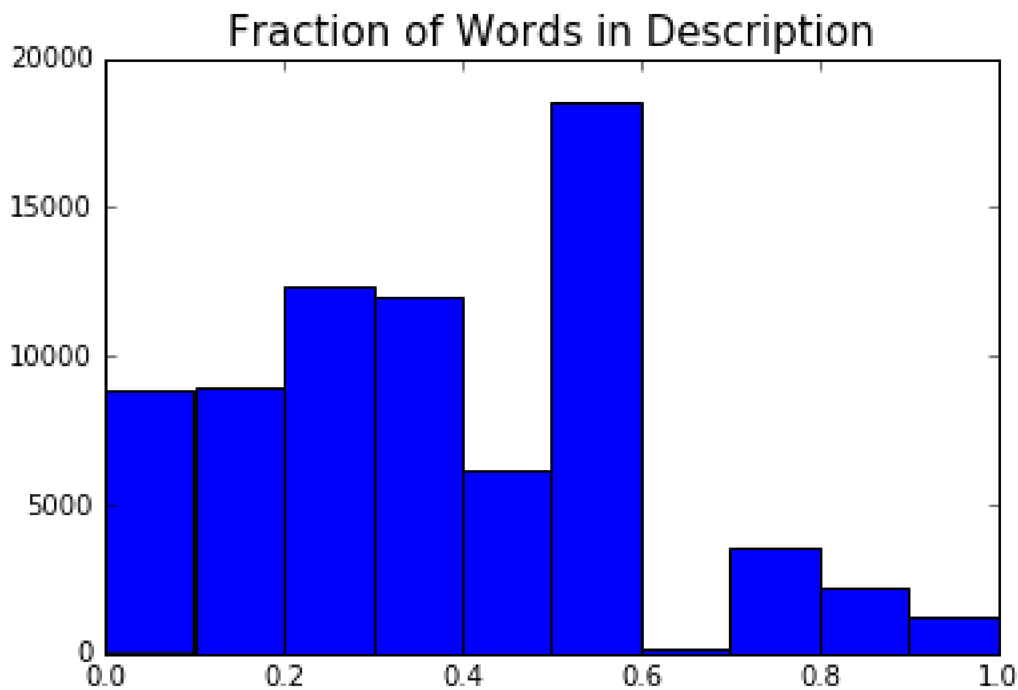

description_length
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

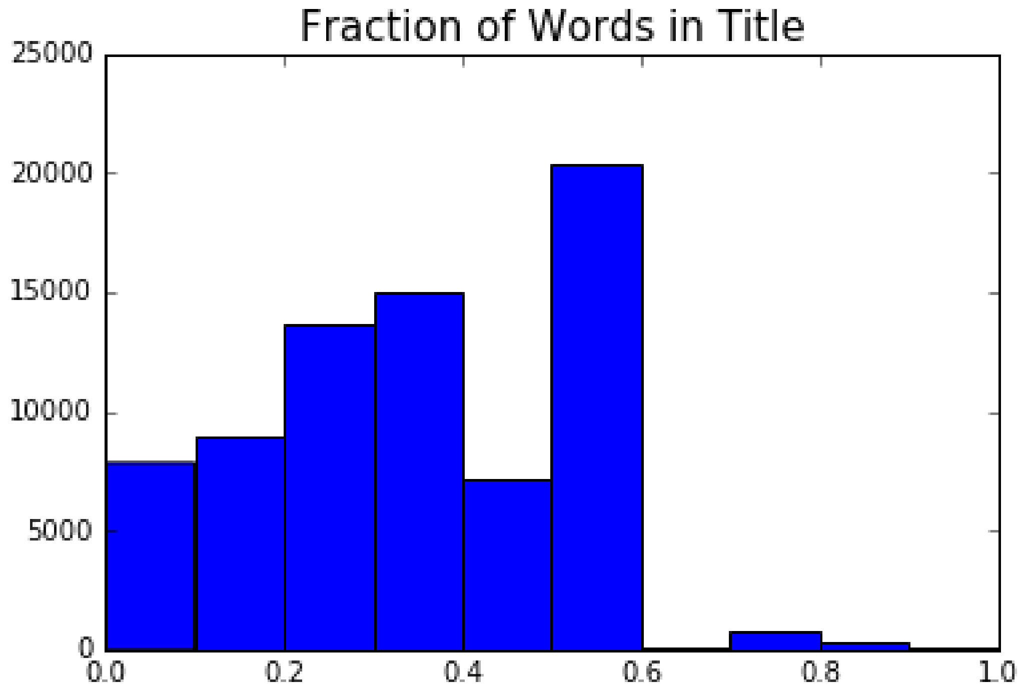

title length
18000
16000
14000
12000
10000
8000
6000
4000
2000
0
0.0
0.2
0.4
0.6
0.8
1.0

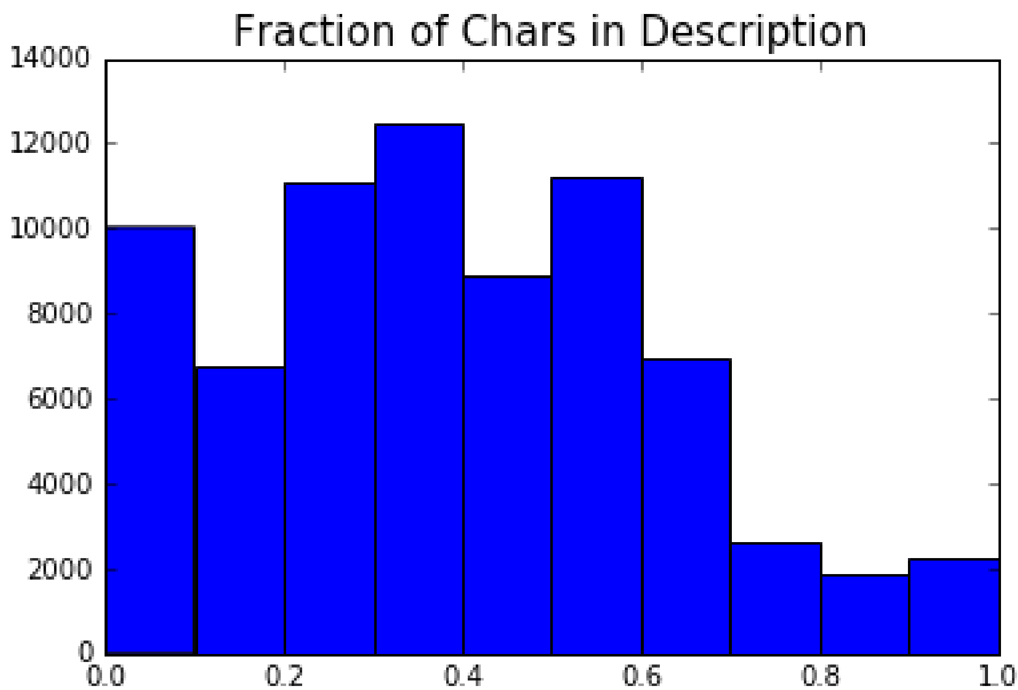

Words in Title
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

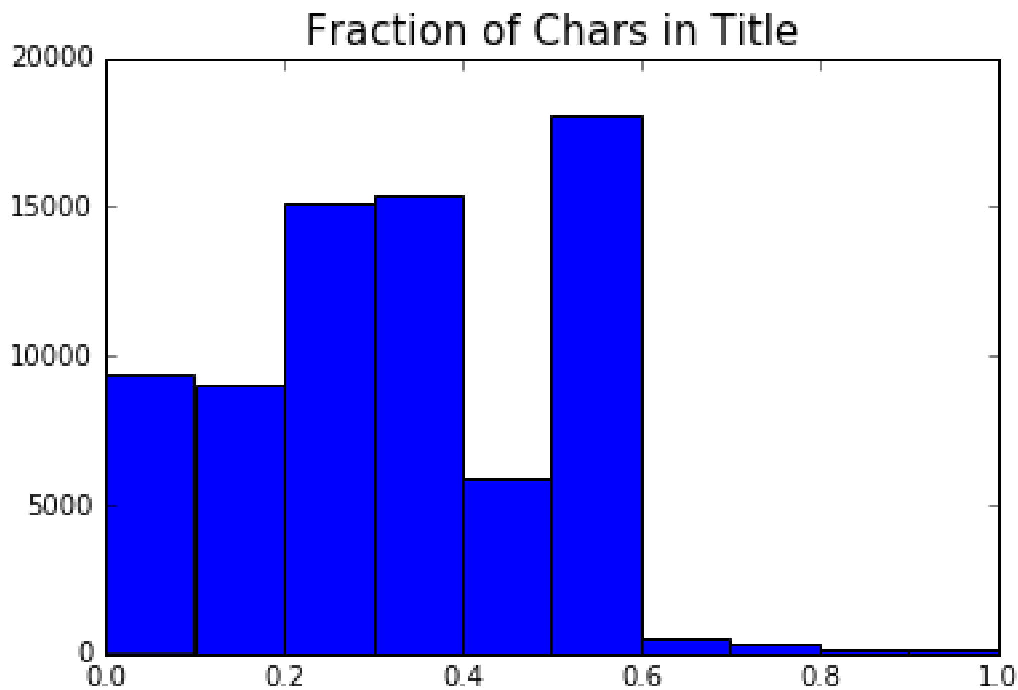

Words in Description
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

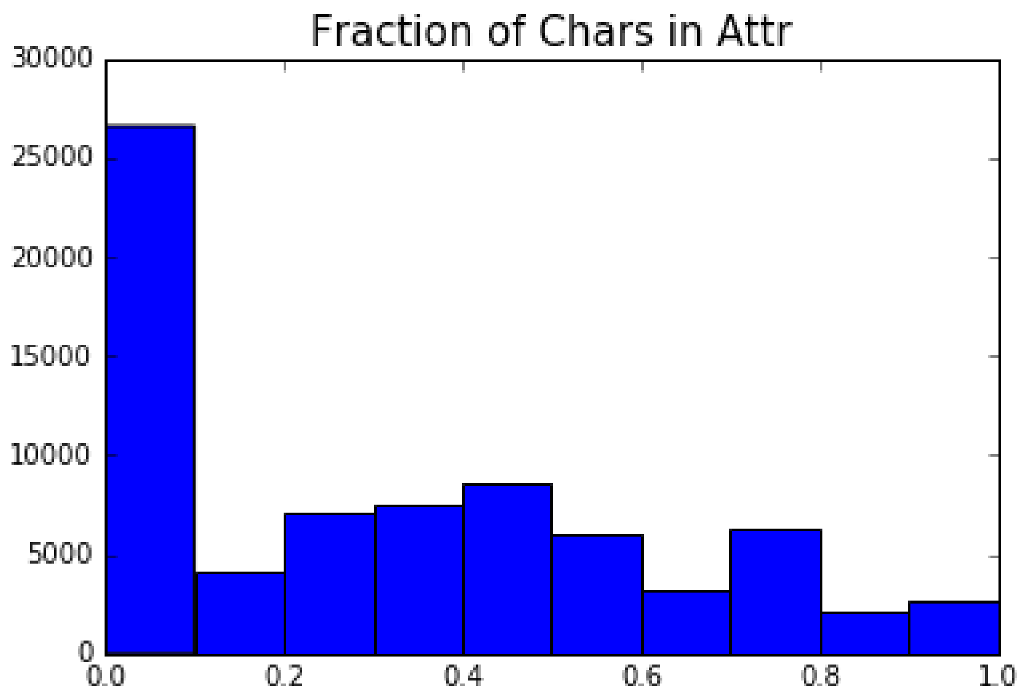

Words in query
30000
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

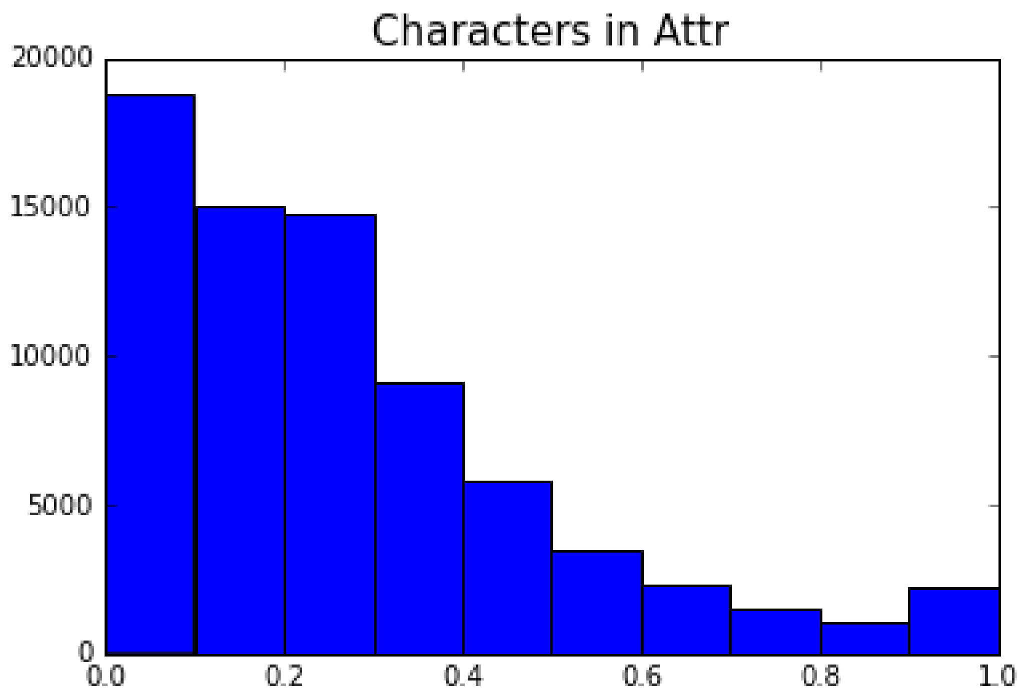

Fraction of Words in Description
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

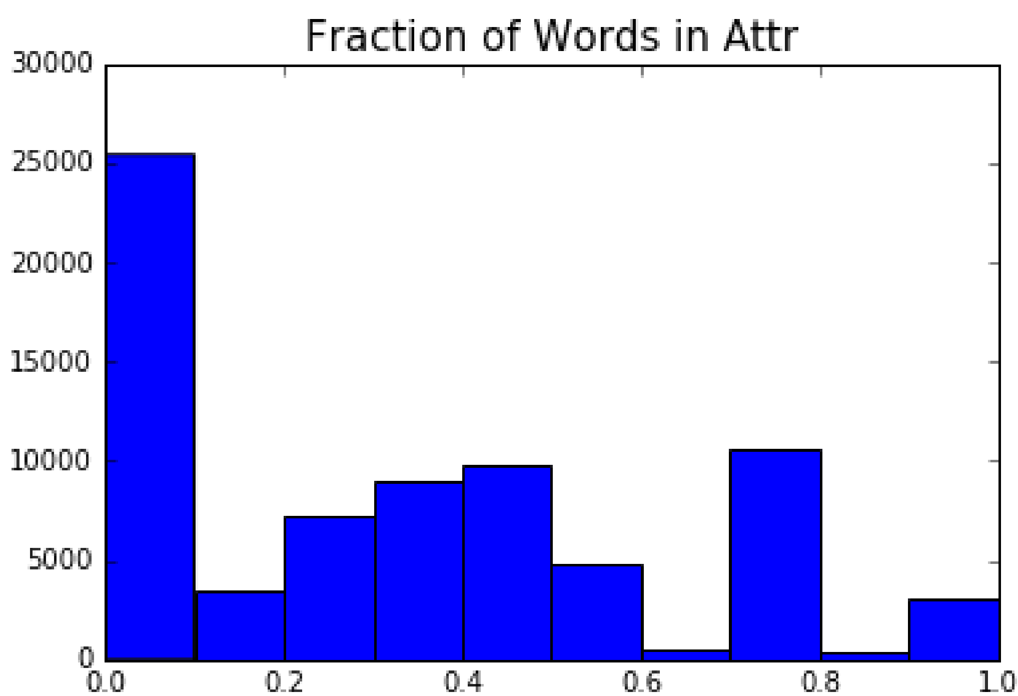

Fraction of Words in Title
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

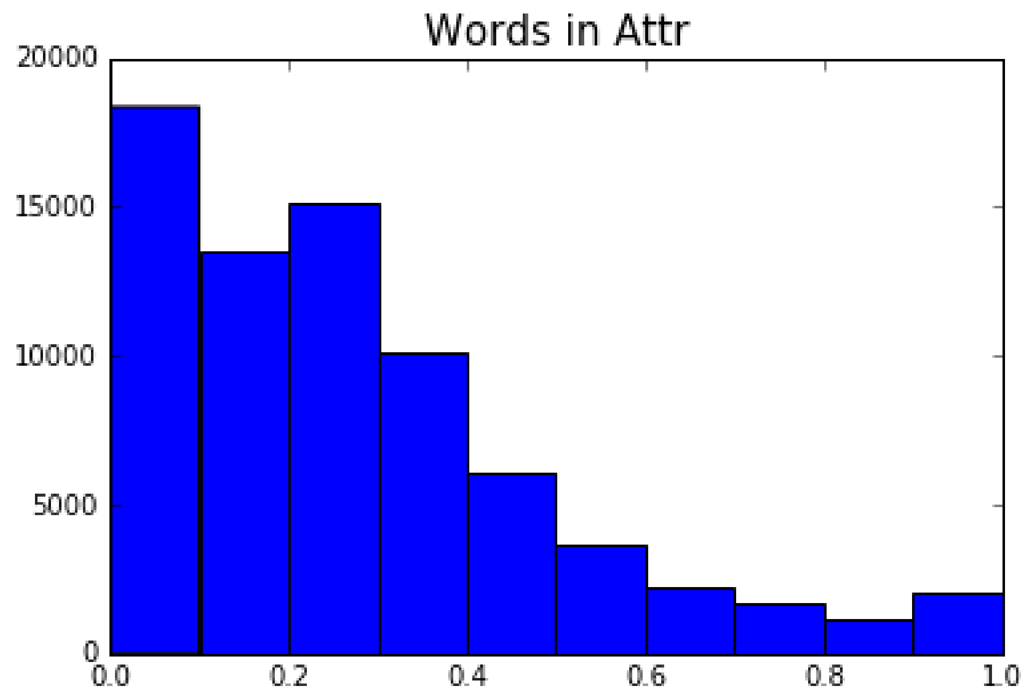

Fraction of Chars in Description
14000
12000
10000
8000
6000
4000
2000
0
0.0
0.2
0.4
0.6
0.8
1.0

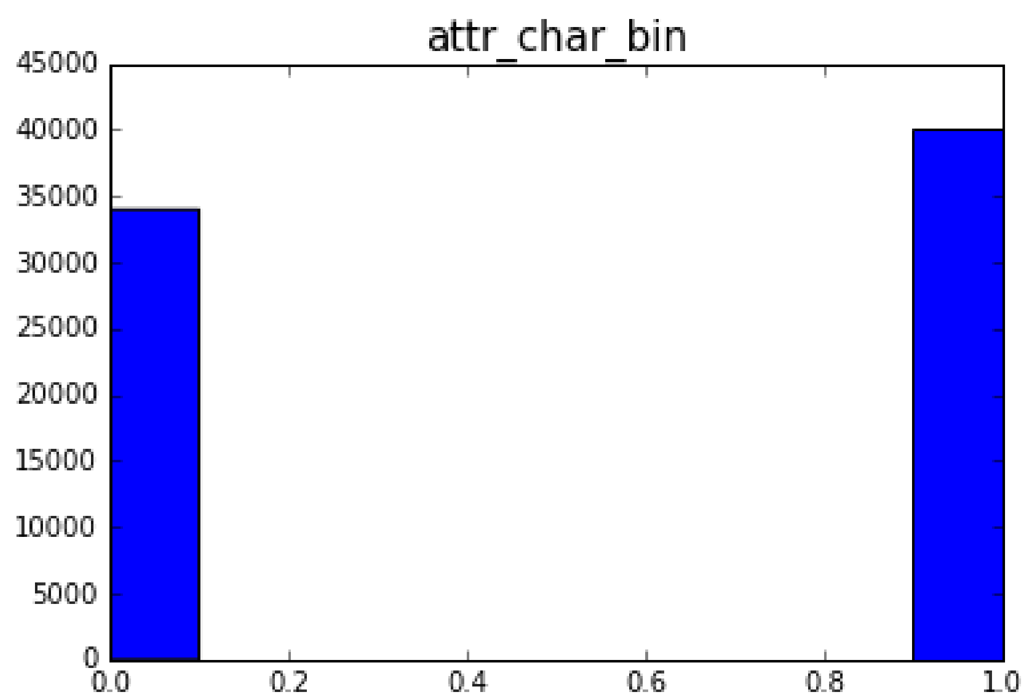

Fraction of Chars in Title
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

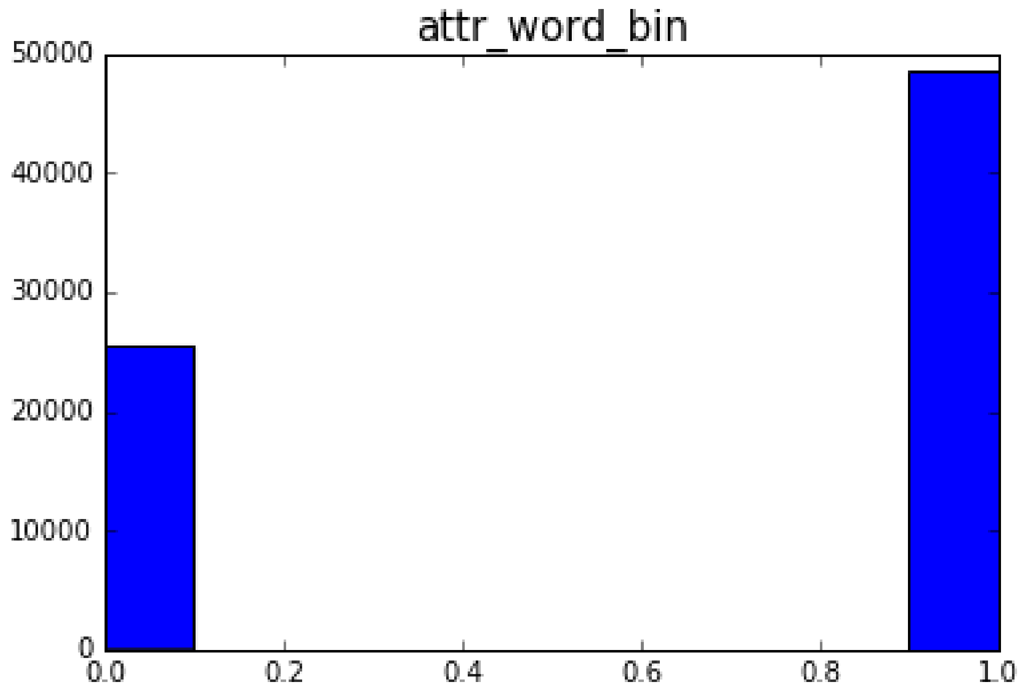

Fraction of Chars in Attr
30000
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

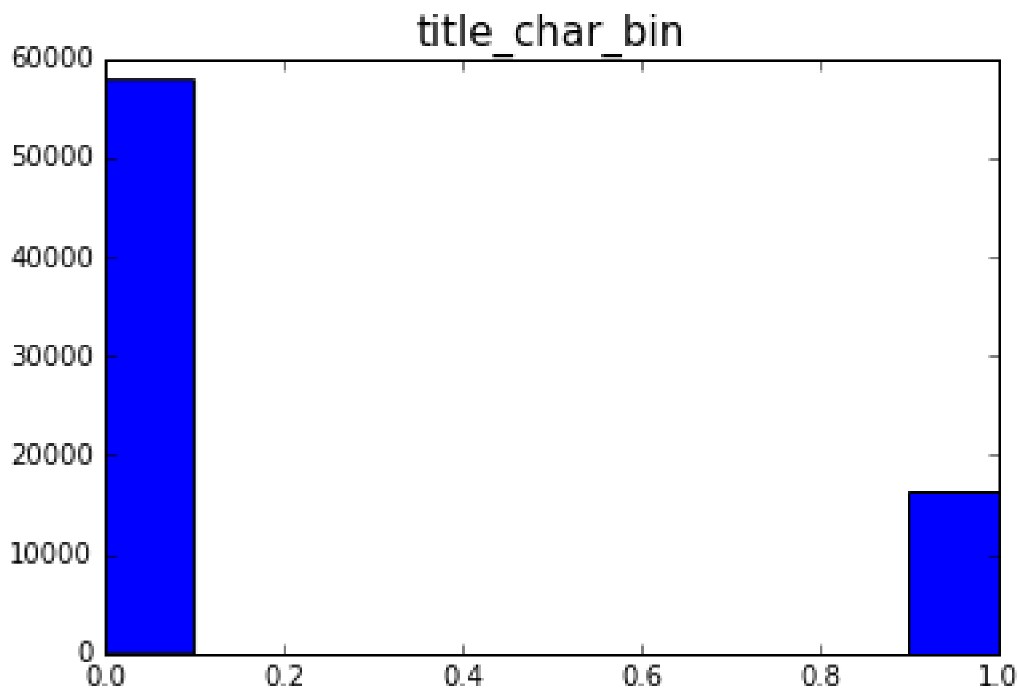

Characters in Attr
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

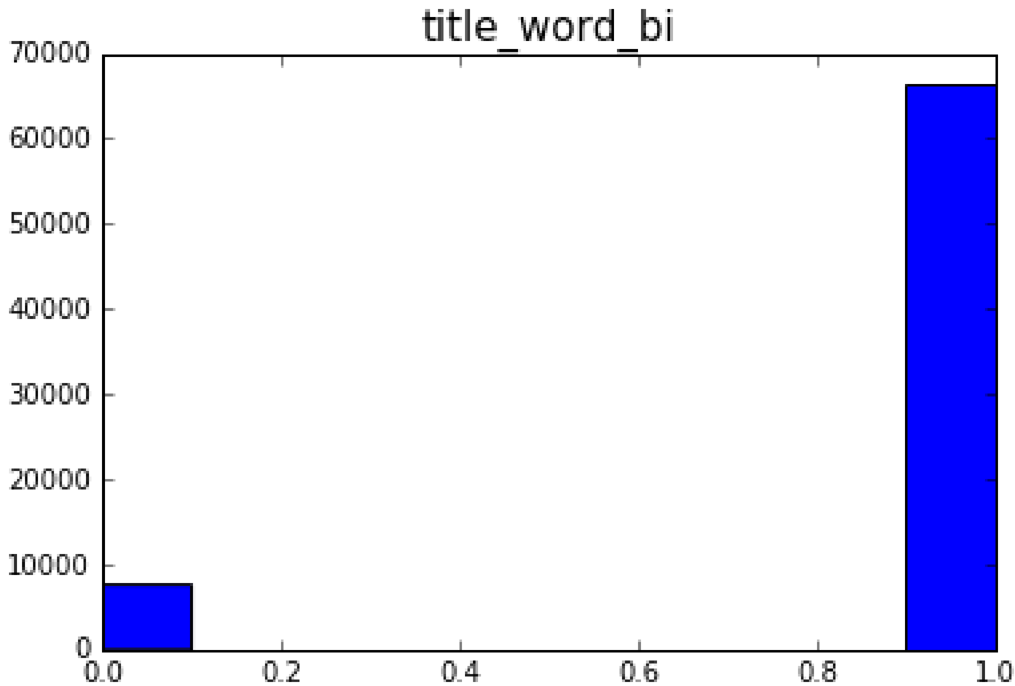

Fraction of Words in Attr
30000
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

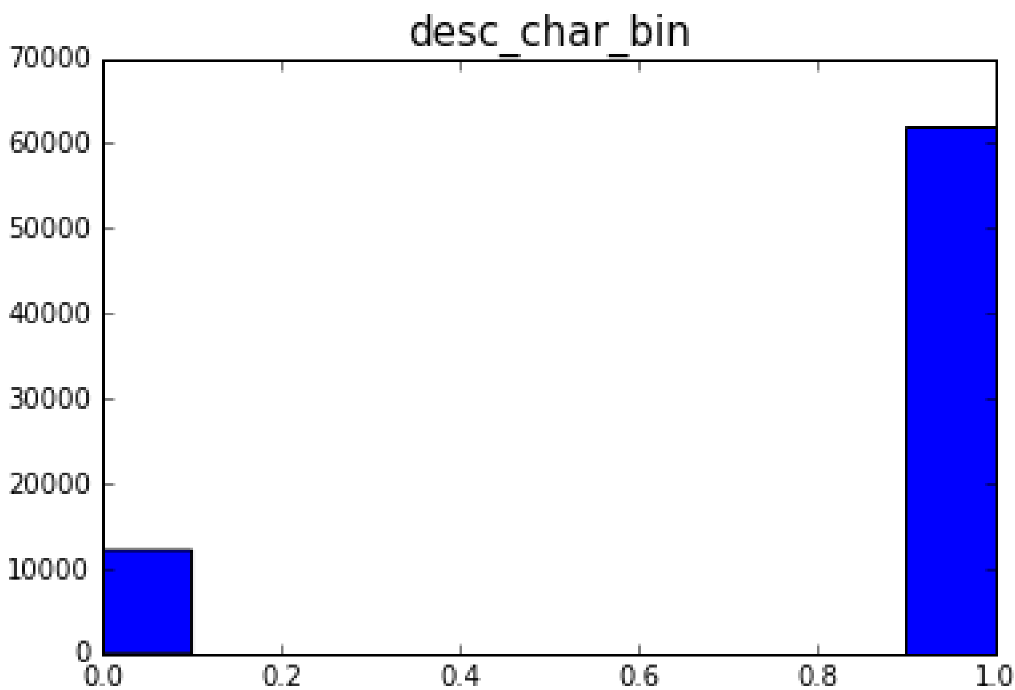

Words in Attr
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0

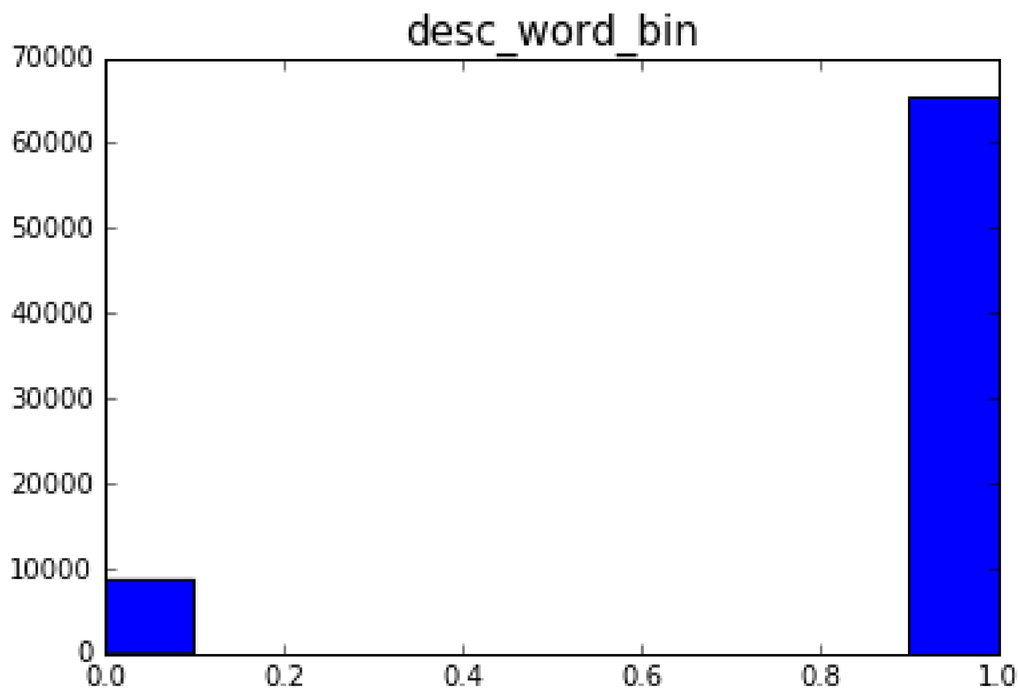

attr_char_bin
45000
40000
35000
30000
25000
20000
15000
10000
5000
0
0.0
0.2
0.4
0.6
0.8
1.0


attr_word_bin
50000
40000
30000
20000
10000
0
0.0
0.2
0.4
0.6
0.8
1.0


title_char_bin
60000
50000
40000
30000
20000
10000
0
0.0
0.2
0.4
0.6
0.8
1.0


title_word_bi
70000
60000
50000
40000
30000
20000
10000
0
0.0
0.2
0.4
0.6
0.8
1.0


desc_char_bin
70000
60000
50000
40000
30000
20000
10000
0
0.0
0.2
0.4
0.6
0.8
1.0

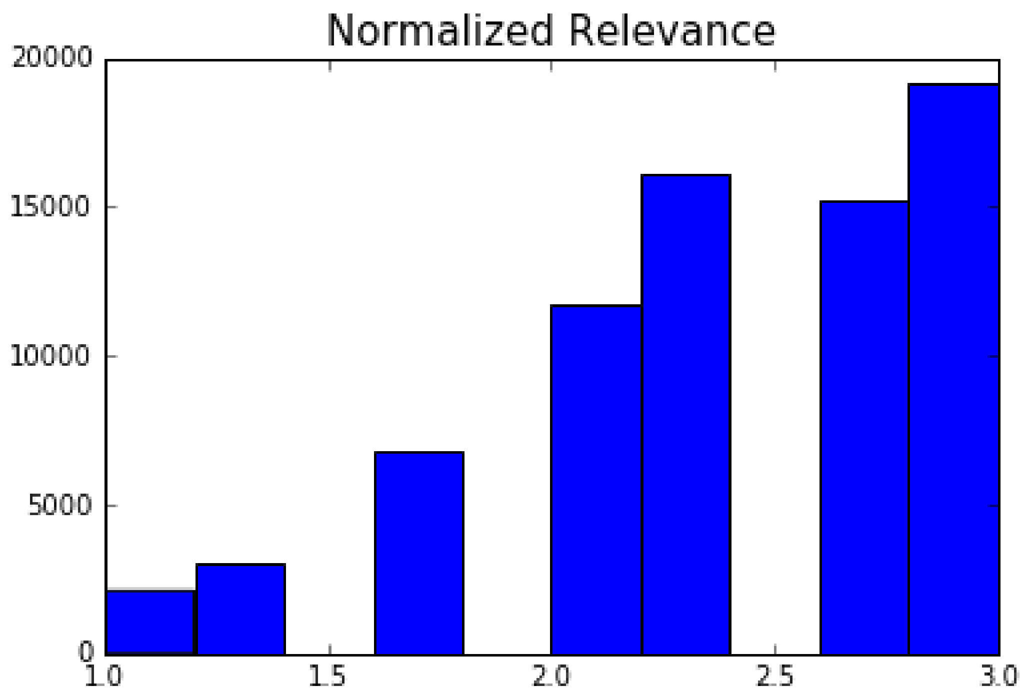



```
Out[12]: [None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None,
          None]
```

5.0.1 Training Set: Distribution of Relevance Scores

I also want to take a look at how the relevance scores are distributed in the training set

```
In [13]: plot_hist(df['relevance'], 'Normalized Relevance')
```



6 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 [14]: # Since relevance scores are R [1,3], we can divide them by 3 to put them in the "norm" range
         def divide_y(y):
             return y/3.

         # Move relevance over to y
         if 'relevance' in df:
             train_y = divide_y(pd.Series(df['relevance']))
             df.pop('relevance')

             # Rename df
             train_X = np.array(df.copy())

In [15]: from sklearn import cross_validation

         ## CROSS VALIDATION
```

```
##================================================================
## Cross-validate and return score
def cv_score(X, y, folds, model):
    # Get an array of scores
    scores = cross_validation.cross_val_score(model, X, y, cv=folds, n_jobs=-1)
    # Return mean and std
    return (abs(np.mean(scores)), np.std(scores))

## Cross-validate and return the a predicted y vector
def cv_fit(X, y, folds, model):
    return cross_validation.cross_val_fit(model, X, y, cv=folds, n_jobs=-1)


## MODEL OPTIMIZATION
##=================================================================
# Optimize the number of CV folds
def tune_folds(X, y, MODEL, **kwargs):

    # Range of folds
    min_i = kwargs['min_i'] if 'min_i' in kwargs else 3
    max_i = kwargs['max_i'] if 'max_i' in kwargs else 10
    f = [i for i in xrange(min_i, max_i+1)]

    # Get the scores
    scores = map(lambda i: {'folds': i, 'score': cv_score(X, y, i, MODEL)}, f)

    # Plot means
    plt.plot(f, map(lambda x: x['score'][0], scores))
    return scores


# Map an array of param values to an array of CV scores and plot it
#    @ models is an L-dimensional list of models instantiated with the param value
#    @ labels is an L-dimensional list of labels corresponding 1:1 with models being tested
#--------------------------------------------------------
def tune_model(X, y, model, **kwargs):

    start = time.time()
    folds = 3

    args = kwargs['args']
    static = kwargs['static']

    # Iterate through the args
    for arg, val in args.iteritems():

        # Copy the static args to a new set of args and add the arg we're optimizing
        def append_arg(static, arg, val):
            static[arg] = val
            return static

        # Init the models with pointers to updated static arguments
        # Note that static arguments can be updated 1 of 2 ways:
        #    1: Before calling this function (tune_model)
```

```
                #    2: By appending a dynamic arg (which we are trying to optimize) using append_arg
                _models = map(lambda x: model( **append_arg(static, arg, x) ), args[arg])

                # Get the scores (CV is itself multi-processed so I won't use a pool here)
                scores = map(lambda m: cv_score(X, y, folds, m), _models)

                # Plot means; plot categorical variables in a bar chart and quantitative ones in a lin
                plt.figure()

                if isinstance( args[arg][0], str):
                    left = [i for i in xrange(len(args[arg]))]
                    plt.bar(left, map(lambda x: x[0], scores), width=0.5, tick_label=args[arg], align=
                else:
                    plt.plot(args[arg], map(lambda x: x[0], scores))

                plt.title(arg, fontsize=16)


            #display(HTML("<font color='blue'>Best MSE: %s</font>"%(global_mse) ))
            #display(HTML("<font color='blue'>Trained model in %s s</font>"%(time.time()-start)))

            return

        # Once all of the dynamic args have been turned into static args, evaluate the model
        # Note: ALL models are trained on 3 CV folds
        def eval_model(X, y, model):
            (mean, std) = cv_score(X, y, 3, model)
            display(HTML("<b>Model optimized with MSE: %s +/- %s</b>"%(mean, std)))
            return
```

7 Regression Methods

Here I will look at some vanilla regression methods

In [ ]:
```
from sklearn.linear_model import Ridge, Lasso
from sklearn.ensemble import GradientBoostingRegressor as GBR
```

In [ ]:
```
R = Ridge()
print 'Ridge Regression Cross Validation Curve'
#tune_folds(train_X, train_y, R, min_i=3, max_i=20)
```

In [ ]:
```
args = {
    'alpha': [i/10. for i in range(0, 10)],
}
r_args = {'args': args, 'static': {}}

tune_model(train_X, train_y, Ridge, **r_args )
```

7.1 Ensemble Methods

Here I will start by exploring a few ensemble methods and see where they take me. Reminder that this is a regression problem.

```
In [16]: from sklearn.ensemble import AdaBoostRegressor as ABR
         from sklearn.ensemble import GradientBoostingRegressor as GBR
         from sklearn.ensemble import RandomForestRegressor as RF
```

7.1.1 AdaBoost

```
In [40]: abr_dynamic_args = {
             #'loss': ['linear', 'exponential', 'square'],
             #'n_estimators': [10, 20, 30, 40, 50, 60],
             #'learning_rate': [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.]
         }

         abr_static_args = {
             'loss': 'linear',
             'n_estimators': 200,
             'learning_rate': 0.8
         }

         abr_args = {'args': abr_dynamic_args, 'static': abr_static_args}
         tune_model(train_X, train_y, ABR, **abr_args)

In [29]: eval_model(train_X, train_y, ABR(**abr_static_args))

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

7.1.2 Gradient Boosting Regressor

```
In [64]: gbr_dynamic_args = {
             #'alpha': [0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
             #'n_estimators': [2,4,6,8,10],
             #'min_samples_split': [4,5,6,7,8,9],
             #'min_samples_leaf': [1, 2, 3, 4],
             #'min_weight_fraction_leaf': [0.0, 0.05 ,0.1, 0.2, 0.3, 0.4, 0.5],
             #'max_depth': [2,5,10],
             #'learning_rate': [0.01, 0.03, 0.05, 0.07, 0.09],
             #'loss': ['ls', 'lad', 'huber', 'quantile']
         }

         # After working through the above, I optimized a few of the params
         gbr_static_args = {
             'n_estimators': 150,
             'learning_rate': 0.8,
             #'min_samples_leaf': 2,
             #'min_samples_split': 5,
             'loss': 'ls',
             'min_weight_fraction_leaf': 0.05,
             'max_depth': 5
         }

         gbr_args = {'args': gbr_dynamic_args, 'static': gbr_static_args}
         tune_model(train_X, train_y, GBR, **gbr_args)

In [49]: eval_model(train_X, train_y, GBR(**gbr_static_args))

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

7.1.3 Random Forest

```
In [ ]: rf_dynamic_args = {
            'n_estimators': [5, 10, 15, 20, 25, 30],
            'max_depth': [None, 2, 3, 4],
            #'min_samples_split': [1, 2, 3, 4, 5],
            'min_samples_leaf': [1, 2, 3, 4],
            #'min_weight_fraction_leaf': [0.1, 0.2, 0.3, 0.4, 0.5],
        }

        rf_static_args = {
            #'min_weight_fraction_leaf': 0.5,
            'min_samples_split': 2,
            'n_jobs': -1

        }

        rf_args = {'args': rf_dynamic_args, 'static': rf_static_args}
        tune_model(train_X, train_y, RF, **rf_args)
```

8 Fitting the Model

Now that I have tested various models with CV, I will fit the best one to the whole training set.

```
In [65]: #M = ABR(**abr_static_args)
         M = GBR(**gbr_static_args)
         M.fit(train_X,train_y)
```

```
Out[65]: GradientBoostingRegressor(alpha=0.9, init=None, learning_rate=0.8, loss='ls',
                      max_depth=5, max_features=None, max_leaf_nodes=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.05, n_estimators=150,
                      presort='auto', random_state=None, subsample=1.0, verbose=0,
                      warm_start=False)
```

9 Test Set

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

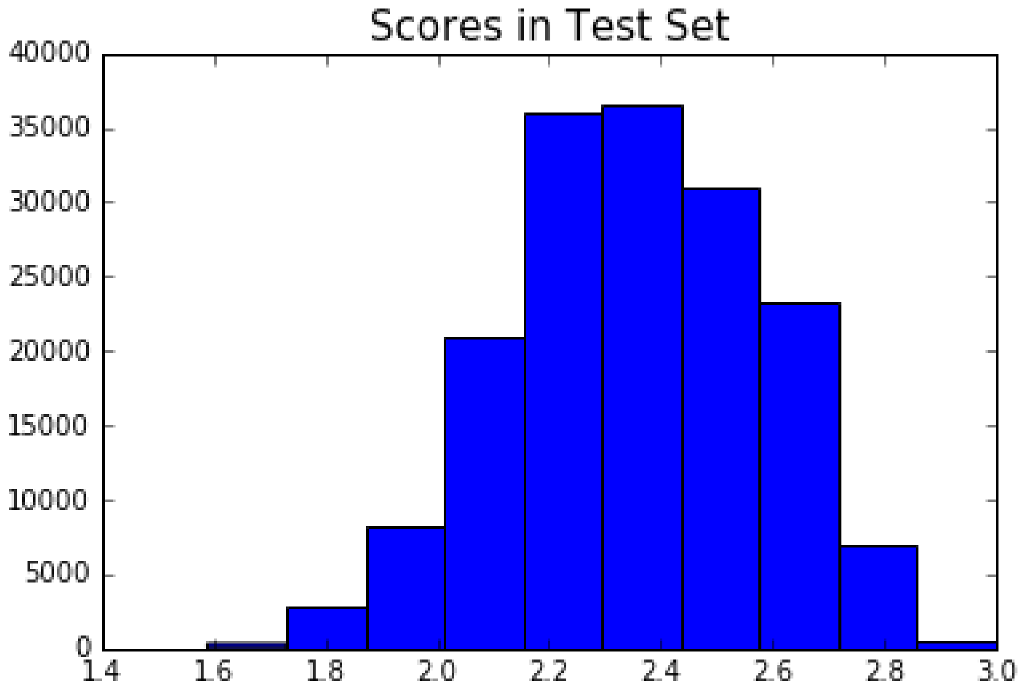
```
In [20]: df_test, df_test_ids = pipeline(f_test)
```

```
df size: (193661, 25)
clean_df size: (166693, 24)

<IPython.core.display.HTML object>


None
```

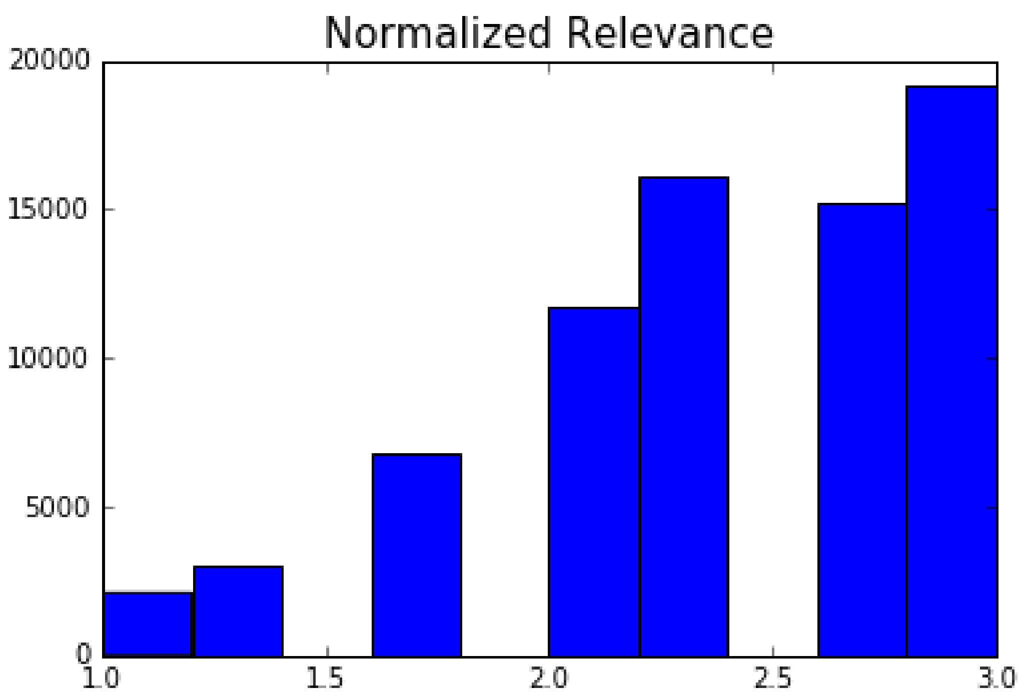
```
In [66]: # Define a new X and predict y using the model we fit earlier
         test_X = np.array(df_test.copy())
         test_y = M.predict(test_X)

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

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

```
In [67]: plot_hist(final_y, 'Scores in Test Set')
```




```
In [38]: new_train_y = map(lambda x: 1 if x<(1./3) else 3 if x>(1.) else x*3., train_y)
         plot_hist(new_train_y, 'Normalized Relevance')
```



Those distributions do not look very similar. First of all, the standard deviation in the test scores is way too small. Let's get the mean and standard deviation and then try to increase the deivation.

```
In [68]: print "Test mean: %s, std: %s"%(np.mean(final_y), np.std(final_y))
         print "Training mean: %s, std: %s"%(np.mean(new_train_y), np.std(new_train_y))
```

```
Test mean: 2.35189951593, std: 0.22618452121
Training mean: 2.38163379103, std: 0.533980343669
```

```
In [ ]: # Clear test sets out of memory if needed
        %reset_selective test_X
        %reset_selective test_y
        %reset_selective final_y
        %reset
```

10 Submission

Now I am finally ready to write the submission file!

```
In [69]: ## 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 [70]: submit(df_test_ids, final_y, 'submission1')
```

```
In [71]: x = pd.read_csv("%ssubmission1.csv"%DATADIR)
         x.head()
```

Out[71]:

	id	relevance
0	1	2.080428
1	4	2.171023
2	5	2.474501
3	6	2.677501
4	7	2.433761

```
In [ ]:
```