Home Depot 2015

February 14, 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("<h3>First three rows of %s</h3>"%name))
    preview_df = pd.read_csv(file)
    print display(preview_df.head(3))

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 ...

     search_term  relevance
0  angle bracket        3.0
1      l bracket        2.5
2      deck over        3.0


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
1   4       100001  Simpson Strong-Tie 12-Gauge Angle   metal l brackets
2   5       100001  Simpson Strong-Tie 12-Gauge Angle   simpson sku able


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...


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...
```

```
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)))
```

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 [6]: # 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"],
        ["w", "watt", "watts"],
        ["v", "volt", "volts"]
    ]

    # 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%s

    # 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)))
    # 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
# This will get called with a word in the QUERY
def extension_words(word):

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

    # Start building a list of the words we will be returning
    # Add abbreviation group if applicable and add the word itself
    ret_words = abbrev(w)
    ret_words.append(w)

    # If the word is small (<4 chars) or contains a number, only add s and abbreviations
    if any(str(i) in w for i in xrange(10)) or len(list(word)) < 4:
        ret_words.append("%ss"%w)
        return ret_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']

    # 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 ret_words
```

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 [7]:
```
# 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
# Since multiple words may match,
#   return 1 if anything matches or 0 if nothing matches
#--------------------------------------------------------
def match_word(word, compare_words):

    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_words else 0, strings ) ))


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

## FRACTIONAL matching
#---------------------------------------------------------------------

# 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), 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(" "))

## 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
```

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

        # 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):
            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):
            b = str(a).split(" ")
            c = np.hstack(np.array(map(lambda x: x.split("-"), b)))
            d = np.hstack(np.array(map(lambda x: x.split("*"), c)))
```

```
    return len(d)
    #return l if l > 1 else 1

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

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]) )


## 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'] = 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) )


    # Title columns
    title_zip = np.dstack( (np.array(df['search_term']), np.array(df['product_title']) ))[0]
    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) )


    # 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'])

# Now build the columns normally
attr_zip = np.dstack( (np.array(df['search_term']), np.array(df['attr']) ))[0]
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) )


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

# Word lengths
df['desc_word_l'] = pd.Series( POOL.map(lambda_word_len, df['product_description']) )#.apply
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)
df['attr_word_l'] = pd.Series( POOL.map(lambda_word_len, df['attr']) )#.apply(ln)


## 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 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()-s
```

```
        return clean_df, ids
```

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

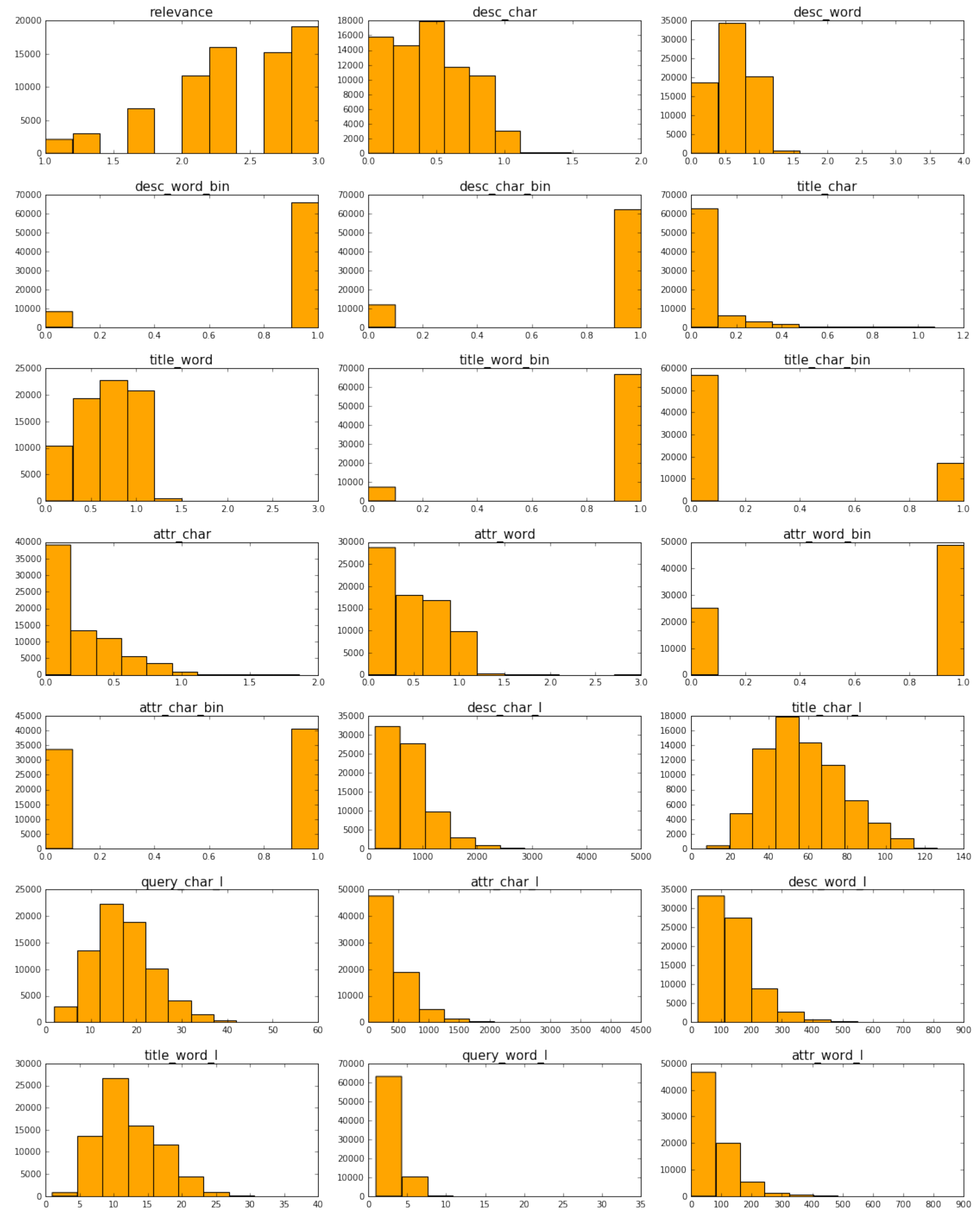
```
df size: (143828, 22)
clean_df size: (74067, 21)

<IPython.core.display.HTML object>


None
```

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 [10]: # 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 [11]: # 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 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 [12]:
```
# 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 [13]:
```
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
```

```python
#    @ 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 [14]: from sklearn.linear_model import Ridge, Lasso
         from sklearn.ensemble import GradientBoostingRegressor as GBR
```

```
In [15]: """args = {
             'alpha': [i/10. for i in range(0, 10)],
         }
         r_args = {'args': args, 'static': {}}

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

```
Out[15]: "args = {\n    'alpha': [i/10. for i in range(0, 10)],\n}\nr_args = {'args': args, 'static': {}
```

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 [17]: """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)"""
```

```
Out[17]: "abr_dynamic_args = {\n    #'loss': ['linear', 'exponential', 'square'],\n    #'n_estimators':
```

```
In [18]: #eval_model(train_X, train_y, ABR(**abr_static_args))
```

7.1.2 Gradient Boosting Regressor

```
In [19]: 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': 500,
    'min_samples_leaf': 3,
    #'min_samples_split': 5,
    'loss': 'ls',
    #'min_weight_fraction_leaf': 0.05,
    'max_depth': 8
}

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

```
In [20]: #eval_model(train_X, train_y, GBR(**gbr_static_args))
```

7.1.3 Random Forest

```
In [44]: rf_dynamic_args = {
             #'n_estimators': [5, 10, 15, 20, 25, 30],
             'max_depth': [4, 5, 6, 7, 8],
             'min_samples_split': [2, 3, 4],
             #'min_samples_leaf': [1, 2, 3, 4],
             #'min_weight_fraction_leaf': [0.1, 0.2, 0.3, 0.4, 0.5],
         }

         rf_static_args = {
             'n_estimators': 500,
             #'max_depth': 10,
             'min_samples_split': 3,
             '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 [45]: #M = ABR(**abr_static_args)
         #M = GBR(**gbr_static_args)
         M = RF(**rf_static_args)
         M.fit(train_X,train_y)

Out[45]: RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=None,
                    max_features='auto', max_leaf_nodes=None, min_samples_leaf=1,
                    min_samples_split=3, min_weight_fraction_leaf=0.0,
                    n_estimators=500, n_jobs=-1, oob_score=False, random_state=None,
                    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.

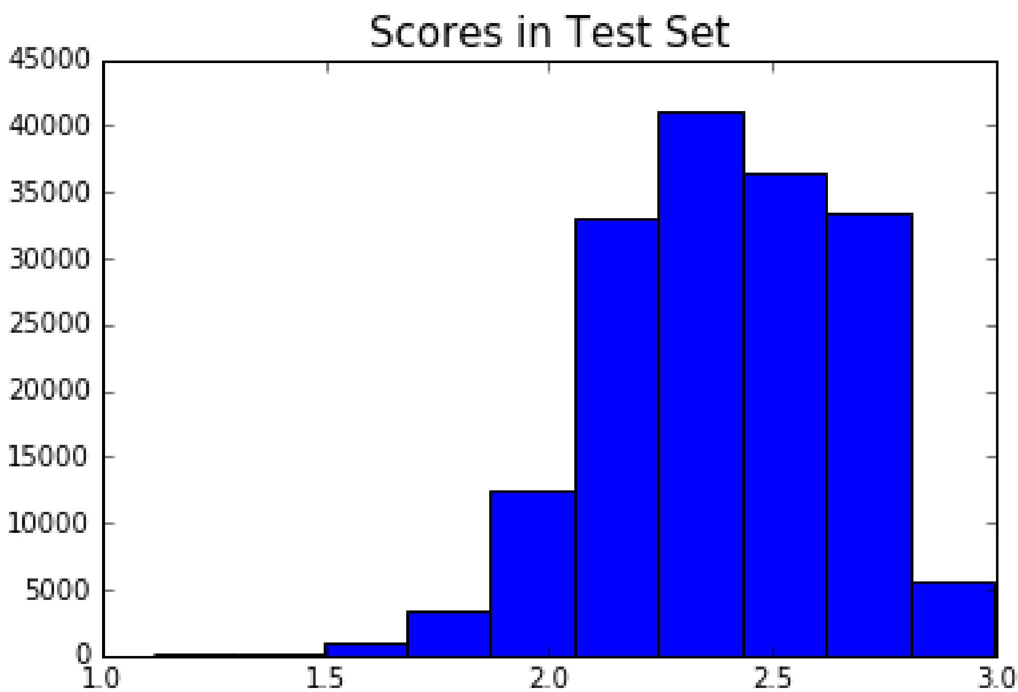
```
In [23]: df_test, df_test_ids = pipeline(f_test)
```

```
df size: (193661, 21)
clean_df size: (166693, 20)

<IPython.core.display.HTML object>


None
```

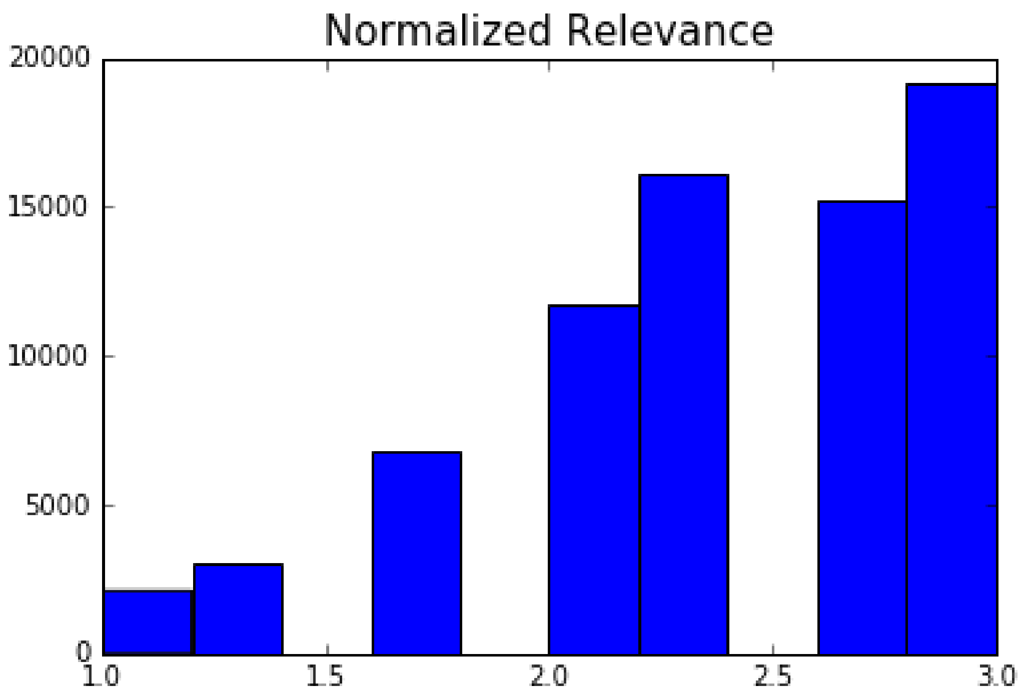
```
In [46]: # Define a new X and predict y using the model we fit earlier
         test_X = np.array(df_test.copy())
         test_y = M.predict(test_X)

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




```
In [26]: 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')
```




```
In [48]: 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: 2.3914822714, std: 0.264701760661
Training mean: 2.38163379103, std: 0.533980343669
```

10 Submission

Now I am finally ready to write the submission file!

```
In [49]: ## 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 [50]: submit(df_test_ids, final_y, 'submission2')
```