6.863 Final Project

Kevin Kelley (kelleyk)

May 18, 2011

Color

The project idea that I proposed centered around examining and classifying the “color” metadata field of products listed for sale on ecommerce platforms. As we’ll see shortly, classifying each item into a “color family” is tricky. There are the usual data quality issues to contend with (typos, run-ons, etc.), but there are also some unique challenges. The most interesting among these are the synonym problem (e.g., “navy” instead of “blue” or “espresso” instead of “black”) and the issue of distinguishing between imperfect but legitimate color information and nonsense. (I use the ⊥ symbol to refer to the class of items that have only nonsense information and should not be otherwise classified.)

My corpus is drawn from Amazon’s own listings and third-party Marketplace listings, from Newegg, and from Best Buy, through a combination of affiliate API use and website scraping. (This is not quite as trivial as it sounds, for a number of reasons, but the details aren’t relevant.) From approximately 11,000 product listings, I found 5,880 products with some color information provided; those items had 498 unique values for the “color” field. The products in question are cameras, mobile phones, GPS units, camcorders, and other consumer electronics products that seemed to be available in a variety of colors.

This information is summarized in colors.dat, a plain-text file that stores (frequency, color) pairs. The colors stored here have been independently preprocessed in a number of ways; for example, they were lowercased, split apart (for multicolored items, e.g. those with / or , in the color property), and had whitespace trimmed. The most common unique “colors” are summarized in Figure 1.
<table>
<thead>
<tr>
<th>Class</th>
<th>Color</th>
<th>Freq.</th>
<th>Class</th>
<th>Color</th>
<th>Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>* black</td>
<td>1864</td>
<td>black nylon black</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* silver</td>
<td>712</td>
<td>pink nylon pink</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* red</td>
<td>439</td>
<td>purple plum</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* blue</td>
<td>383</td>
<td>green olive</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* white</td>
<td>210</td>
<td>grey titanium</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* pink</td>
<td>209</td>
<td>one color</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* grey</td>
<td>127</td>
<td>purple violet</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* green</td>
<td>124</td>
<td>blue arctic blue</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>grey</td>
<td>111</td>
<td>orange bronze</td>
<td>14</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* purple</td>
<td>90</td>
<td>black blk</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* orange</td>
<td>79</td>
<td>off-white khaki</td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* brown</td>
<td>62</td>
<td>blue navy</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* clear</td>
<td>56</td>
<td>grey gun metal</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>* yellow</td>
<td>50</td>
<td>off-white desert tan</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pink</td>
<td>42</td>
<td>off-white sand</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>blue</td>
<td>35</td>
<td>grey dark gray</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>red</td>
<td>30</td>
<td>pink hot pink</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>yellow</td>
<td>28</td>
<td>grey nickel</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>green</td>
<td>26</td>
<td>black black revolt</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>off-white</td>
<td>24</td>
<td>blue eva blue</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1: Frequent colors

Luckily, although perhaps unsurprisingly, the most colors corresponding to the class or “color family” names are far and away the most prevalent. The first thing that I did, using the `analyze_colors.py` tool that I wrote, was to take advantage of this fact.

By unique colors:
287 matched (57.6%); 27 multi-matched (5.4%); 211 not matched (42.4%)

By items:
5255 matched (89.4%); 39 multi-matched (0.7%); 625 not matched (10.6%)

As we can see, a substantial number of colors are not matched by this technique (nearly half!), and they represent a small but significant minority of the items (10.6%). Approximately two-thirds of these values that haven’t been matched do have some meaning that we should be able to extract.
**Future work:** There are also a number of colors that match multiple classes. Most of these are legitimately multi-colored. I haven’t addressed this special case here, although it’s something that deserves consideration in this and subsequent classification attempts.

What sort of values failed to be matched? From examining the list of them (which is also something that `analyze_colors.py` can produce for you), it seems like they fall into three general groups. First, there are some that would have been matched using our simple first-pass effort but for typographical errors. Then, there are some that use synonyms or “Crayola colors,” perhaps in an attempt to make their product seem more interesting. Finally, there are the nonsense entries that I anticipated. Examples of actual values falling into each category are given in Figure 2.

<table>
<thead>
<tr>
<th>Group</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>typos</td>
<td>rd, skyblue, blk, slver, purpl, slim nylon red,</td>
</tr>
<tr>
<td>synonyms</td>
<td>cyan, sand, teal, sand, navy, bone, espresso</td>
</tr>
<tr>
<td>nonsense</td>
<td>daze-e, fun in fields, kupkake, dale earnhardt,</td>
</tr>
<tr>
<td></td>
<td>panda, 512mb built in memory, 2 pack</td>
</tr>
<tr>
<td>misc</td>
<td>alpine plaid, wooden, panda, brass, no color</td>
</tr>
</tbody>
</table>

Figure 2: Examples of unmatched colors

I decided to examine the synonyms group first. Some of these are not likely to occur together with their “real” color (e.g., you wouldn’t see “violet purple” or “grey gray”) but many seem likely to (e.g., “navy blue” or “alpine white” both seem reasonable). Thus, while looking at word pairings won’t solve all of these issues, I was hopeful that it might be able to help with many of them.

I call the list of tokens that occurred in a phrase, other than the one(s) that caused it to be classified, the “secondary vocabulary” associated with that color family. This information, along with per-class counts for each secondary vocabulary token, can also be generated by the `analyze_colors.py` tool, and is presented in Figure 7.

Some of these words are clearly the synonyms that we’re interested in! Brown has “chestnut” and “mocha;” blue has “indigo,” “cyan,” and “navy” (among others); red has “ruby,” “bordeaux,” and “crimson;” and so on.

Naturally, there is the question of how to apply this information. I chose to compute the proportion of the time that each secondary vocabulary word
was associated with each primary vocabulary word. These are in the range [0, 1]. Since I wanted to ensure that having more secondary vocabulary terms increased probability, I chose to rescale them into the range [1, 2] and then multiply the scores per token, per class.

Thus, for example, if “foo bar” is being scored, and foo is always associated with blue, but bar is split between blue and red, we would find a score of $2.0 \cdot 1.5 = 3.0$ for blue and a score of 1.5 for red. Thus, we would classify “foo bar” as blue.

There definitely are some neat results from this effort. For example, “cyan” and “teal” were identified as blue; “wine” was identified as red; “charcoal” and “iron” were identified as gray; and “lime” and “leaf” were identified as green. However, because the counts were so low (all single-digit, and many two or three) there were a good number of mistaken identifications. For example, something about “pentax optio 60 6mp digital camera with 3x optical zoom” produced a very high-probability classification of “silver.” (Somewhat humorously, the item in question actually is silver.)

**Future work:** Some of these words are relevant but aren’t going to help us—take, for example, “light,” “dark,” “glossy,” “matte,” etc. It might be interesting to put together a list of stopwords that includes these, and explore its impact.

**Future work:** We should see whether these problems start to go away as the corpus becomes larger. Even with 11,000 items, since the yield of these secondary words per item is low, there just isn’t enough to learn from.

**Future work:** I also have available a word-count summary of a large corpus of written and spoken English. Words that have very high counts in general might be worth discarding.

By unique colors:
322 matched (64.7%); 27 multi-matched (5.4%); 176 not matched (35.3%)

By items:
5359 matched (91.1%); 39 multi-matched (0.7%); 521 not matched (8.9%)

This effort was able in identify a further 8% or so of the color phrases, or about one-third of the non-nonsense color phrases that were not classified by exact matching.
String similarity

Next, I decided to examine string similarity metrics. Given two strings $s$ and $t$, a similarity function $\text{SimFn}(s, t)$ returns some value in the range $[0, 1]$ where 0 indicates that the strings have nothing in common, and 1 indicates that they are precisely and exactly equal.

Edit distance

I began by examining the Damerau-Levenshtein distance, which is an edit distance metric that permits the insert, deletion, mutation, and transposition operations, all with unit weight. In short, one counts the minimum number of these operations necessary to transform one string into the other. Results for a small group of test pairs are presented in Figure 3.

\begin{table}[h]
\centering
\begin{tabular}{ccc}
\hline
$s$ & $t$ & $\text{LevDist}(s, t)$ \\
\hline
silver & silver & 0 \\
silver & slrv & 4 \\
silver & slvr & 2 \\
slrv & blue & 3 \\
MARTHA & MARHTA & 2 \\
DWAYNE & DUANE & 2 \\
DIXON & DICKSONX & 4 \\
piunk & pink & 1 \\
blk & black & 2 \\
white & with & 2 \\
silver & sliver & 2 \\
red & red & 1 \\
titanum & titanium & 1 \\
\hline
\end{tabular}
\caption{Results using the Damerau-Levenshtein distance}
\end{table}

This is fairly good! However, there are a number of problems. First, this isn’t a similarity score; we’d have to find some way to scale it down (probably depending on the length of the string). Second, while it would be easy to change this, mutations are too cheap, so “slvr” and “blue” are very close (three edits!).

5
Next I decided to try a purely string-based approach. The Dice metric involves breaking $s$ and $t$ up into character bigrams (the sets of which I will call $S$ and $T$, respectively) and then looking at the overlap between them. The similarity score attempts to measure the number of those bigrams which are common to both strings by computing

$$
\text{DICE Sim}(s, t) = \frac{|S \cup T|}{|S||T|}
$$

Results using this metric are given in Figure 4.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$t$</th>
<th>DICE Sim($s, t$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>silver</td>
<td>silver</td>
<td>100.00%</td>
</tr>
<tr>
<td>silver</td>
<td>slrv</td>
<td>0.00%</td>
</tr>
<tr>
<td>silver</td>
<td>slvr</td>
<td>25.00%</td>
</tr>
<tr>
<td>slrv</td>
<td>blue</td>
<td>0.00%</td>
</tr>
<tr>
<td>MARTHA</td>
<td>MARHTA</td>
<td>40.00%</td>
</tr>
<tr>
<td>DWAYNE</td>
<td>DUANE</td>
<td>22.22%</td>
</tr>
<tr>
<td>DIXON</td>
<td>DICKSONX</td>
<td>36.36%</td>
</tr>
<tr>
<td>piunk</td>
<td>pink</td>
<td>57.14%</td>
</tr>
<tr>
<td>blk</td>
<td>black</td>
<td>33.33%</td>
</tr>
<tr>
<td>white</td>
<td>with</td>
<td>28.57%</td>
</tr>
<tr>
<td>silver</td>
<td>sliver</td>
<td>40.00%</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>80.00%</td>
</tr>
<tr>
<td>titanium</td>
<td>titanium</td>
<td>76.92%</td>
</tr>
</tbody>
</table>

Figure 4: Results using the Dice similarity metric

These, unfortunately, are inadequate. Since the strings are so short, even one mistake can create a large difference between the bigram sets $S$ and $T$. The metric is especially weak at identify matches where mutations or transpositions have taken place.

For completeness, I also implemented the similar cosine similarity metric, given by

$$
\text{COSINE Sim}(s, t) = \frac{\vec{S} \cdot \vec{T}}{|S||T|}
$$
The chief difference here is that we treat the bigram sets $S$ and $T$ as sparse vectors whose elements have magnitude (i.e., the bigram count is significant). Additionally, the inner product in the numerator consists of pairwise multiplications, so the importance of frequent bigrams is emphasized. Results using this metric are summarized in Figure ??.

<table>
<thead>
<tr>
<th>$s$</th>
<th>$t$</th>
<th>$\text{COSINE SIM}(s, t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>silver</td>
<td>silver</td>
<td>100.00%</td>
</tr>
<tr>
<td>silver</td>
<td>slrv</td>
<td>0.00%</td>
</tr>
<tr>
<td>silver</td>
<td>slvr</td>
<td>6.67%</td>
</tr>
<tr>
<td>slrv</td>
<td>blue</td>
<td>0.00%</td>
</tr>
<tr>
<td>MARTHA</td>
<td>MARHTA</td>
<td>16.00%</td>
</tr>
<tr>
<td>DWAYNE</td>
<td>DUANE</td>
<td>5.00%</td>
</tr>
<tr>
<td>DIXON</td>
<td>DICKSONX</td>
<td>14.29%</td>
</tr>
<tr>
<td>piunk</td>
<td>pink</td>
<td>33.33%</td>
</tr>
<tr>
<td>blk</td>
<td>black</td>
<td>12.50%</td>
</tr>
<tr>
<td>white</td>
<td>with</td>
<td>8.33%</td>
</tr>
<tr>
<td>silver</td>
<td>sliver</td>
<td>16.00%</td>
</tr>
<tr>
<td>red</td>
<td>red</td>
<td>66.67%</td>
</tr>
<tr>
<td>titanium</td>
<td>titanium</td>
<td>59.52%</td>
</tr>
</tbody>
</table>

Figure 5: Results using the cosine similarity metric

As you can see, they are little better; words as close as “silver” and “sliver” are given a 16% score; “piunk” and “pink” get only 33%.

These clearly would not do. Finally, I turned to the Jaro similarity metric, which was developed for use in the United States Census and thus anticipates both many common human transcription errors and relatively short strings.

The Jaro metric is more complex than either the Dice or cosine metrics. We begin by finding the ordered subset of characters common to both $s$ and $t$, the inputs. Two characters (one in each string) are considered “common” if they have not already been paired with another character, have the same value, and are within half of the length of the shorter of $s$ and $t$. This is intended to account for transpositions, insertions, and deletions, which might otherwise affect matching too much.

The magnitude of this set is $\sigma$. The similarity of two strings depends both on how large $\sigma$ is relative to the lengths $|s|$ and $|t|$ and on how many
transpositions occur in those common characters. The final score is computed as

$$\text{JAROSIM}(s, t) = \frac{\sigma + \sigma_{\text{Transp}(s, t)}}{3}$$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>JAROSIM(s, t)</th>
</tr>
</thead>
<tbody>
<tr>
<td>silver</td>
<td>silver</td>
<td>100.00%</td>
</tr>
<tr>
<td>silver</td>
<td>slrv</td>
<td>75.00%</td>
</tr>
<tr>
<td>silver</td>
<td>slvr</td>
<td>88.89%</td>
</tr>
<tr>
<td>slrv</td>
<td>blue</td>
<td>50.00%</td>
</tr>
<tr>
<td>MARTHA</td>
<td>MARHTA</td>
<td>94.44%</td>
</tr>
<tr>
<td>DWAYNE</td>
<td>DUANE</td>
<td>82.22%</td>
</tr>
<tr>
<td>DIXON</td>
<td>DICKSONX</td>
<td>76.67%</td>
</tr>
<tr>
<td>piunk</td>
<td>pink</td>
<td>93.33%</td>
</tr>
<tr>
<td>blk</td>
<td>black</td>
<td>68.89%</td>
</tr>
<tr>
<td>white</td>
<td>with</td>
<td>85.00%</td>
</tr>
<tr>
<td>silver</td>
<td>sliver</td>
<td>94.44%</td>
</tr>
<tr>
<td>red</td>
<td>red_</td>
<td>91.67%</td>
</tr>
<tr>
<td>titanium</td>
<td>titanium</td>
<td>95.83%</td>
</tr>
</tbody>
</table>

Figure 6: Results using the Jaro similarity metric

Results are presented in Figure 6. Finally! These are quite nice. With few exceptions, a threshold of approximately 75% would allow even relatively badly misspelled pairs to match, and keep dissimilar pairs (such as the previously-discussed “slrv” and “blue”) from matching.

**Future work:** Some pairs, such as “white” and “with,” are legitimately quite close. However, a human looking at the two would understand that “with” is a very common English word and that it is unlikely to be a misspelling of “white.” We could leverage English word frequency information in an attempt to combat this problem.

Let’s see how this performs on the actual data. I added a new classification step that applies the Jaro metric with a threshold of 75%.

By unique colors:
366 matched (73.5%); 29 multi-matched (5.8%); 132 not matched (26.5%)
By items:
5462 matched (92.9%); 42 multi-matched (0.7%); 418 not matched (7.1%)

These look good on the surface (another 9% of items classified!) and we are, indeed, solving some problems; “slvr” becomes “silver,” “red,” becomes “red,” “clack” becomes “black,” and so forth. However, the test has many false positives. The previously-discussed case of “white” and “with” is one (85% similarity). The most amusing example I found was that of “grey” and “turkey” (75% similarity on the nose).

**Future work:** Most of these false identifications seem to be a result of considering the input one token at a time instead of as a unit. Having extraneous words should produce a classification penalty—if we don’t understand those words, we likely don’t understand the context that the word we’re interested in is being used in.

**Hardwired synonyms**

There are a lot of synonyms not caught by the secondary vocabulary system, as I expected. These are things like “violet” for “purple”—unlike “navy blue,” one never sees “violet purple.”

Looking at the corpus, I produced a small list of these and added them to the top of the analysis tool. They can be enabled or disabled with a switch at the top of the driver code section.

By unique colors:
422 matched (84.7%); 65 multi-matched (13.1%); 76 not matched (15.3%)
By items:
5704 matched (97.0%); 105 multi-matched (1.8%); 176 not matched (3.0%)

With this in place, there are very few unmatched words remaining.

**Future work:** Some unmatched values, like “lightblue,” seem to be due to errors that impacted tokenization. Our strict treatment of word separations is probably causing this issue.

**Parsing**
Code listings

Color

analyze_colors.py

#!/usr/bin/env python

import sys
import kflib
import stringmatch

SILENT = False

COLOR_VOCAB_SYN = {
    'black': ['jet', 'espresso', 'midnight', 'carbon'],
    'red': ['burgundy', 'ruby', 'crimson', 'cherry', 'maroon', 'merlot', 'fire'],
    'green': [],
    'white': [],
    'blue': ['sky', 'turquoise'],
    'orange': [],
    'silver': ['titanium', 'chrome'],
    'brown': ['sepia', 'chocolate', 'latte'],
    'gray': ['grey', 'gunmetal', 'titanium', 'graphite'],
    'off-white': ['beige', 'pearl', 'champagne', 'ivory', 'khaki', 'oatmeal', 'sand'],
    'gold': ['brass'],
    'yellow': ['mustard'],
    'purple': ['violet', 'plum', 'lavender', 'lilac', 'orchid'],
    'pink': ['magenta', 'raspberry'],
    'clear': ['translucent'],
}

COLOR_VOCAB = {
    'black': [],
    'red': [],
    'green': [],
    'white': [],
    'blue': [],
}
'orange': [],
'silver': [],
'brown': [],
'gray': ['grey'],
'off-white': [], #['beige', 'pearl', 'champagne']
'gold': [],
'yellow': [],
'purple': [], #['violet'],
'pink': [],
'clear': [],
}

def parse_vocab(vocab):
    primaryvocab = []
synonyms = {}
    for pvword in vocab:
        primaryvocab.append(pvword)
        for synword in vocab[pvword]:
            synonyms[synword] = pvword
    return (primaryvocab, synonyms)

# Make sure we're being called the right way

def usage():
    print "Usage: "+sys.argv[0]+"
    sys.exit(-1)
if len(sys.argv) != 1:
    usage()

# Utility functions

def get_colors():
    colors = {} 
    f = open('colors.dat')
    for line in f:
        if 0 != len(line) and '#' == line[0]: continue
        freq, color = line.split(',',1)
        freq = int(freq)

color = color.strip().lower()
colors[color] = (phrase_to_words(color), freq, [])
return colors

def phrase_to_words(phrase):
    words = [phrase]
    for sep in [None, '&', '/', '(', ')']:
        tmp, words = words, []
        for t in tmp:
            words.extend(t.split(sep))
    return [x for x in words if x != '']

def vocabmatch(colors, primaryvocab, synonyms={}, verbose=False):
    for color, info in colors.items():
        tokens, freq, classes = info
        primarymatches = []
        for tok in tokens:
            if tok in synonyms:
                tok = synonyms[tok]
            classes.extend([pvword for pvword in primaryvocab if tok == pvword])
        return

def svmatch(colors, primaryvocab, secondaryvocab, threshold=1.6, synonyms={}, verbose=False):
    for color, info in colors.items():
        tokens, freq, classes = info
        if 0 != len(classes): continue
        probs = {}
        for tok in tokens:
            if not tok in secondaryvocab: continue
            for pvword in secondaryvocab[tok]:
                if not pvword in probs: probs[pvword] = 1.0
                probs[pvword] *= (1.0 + secondaryvocab[tok][pvword])
        if 0 == len(probs): continue
        maxclass, maxprob = max(probs.items(), key=lambda x: x[1])
        if maxprob >= threshold:
            if verbose:
                print '{0:.5} {1} <- {2}'.format(maxprob, maxclass, color)
            classes.append(maxclass)
        return

12
# Builds a map of each primary vocab word to a list of related words and their counts.

def find_related(colors, primaryvocab, synonyms={}):
    related = {}
    for pvword in primaryvocab:
        related[pvword] = {}
    for color, info in colors.items():
        tokens, freq, classes = info
        for tok in tokens:
            for result in classes:
                if tok in primaryvocab or tok in synonyms:
                    continue
                if not tok in related[result]:
                    related[result][tok] = 0
                related[result][tok] += 1
    return related

# Turns the result of find_related into a map from related words to the associated primary vocab words and a probability-value.

def build_secondary(related):
    secondaryvocab = {}
    for pvword in related:
        for svword, freq in related[pvword].items():
            if not svword in secondaryvocab: secondaryvocab[svword] = {}
            secondaryvocab[svword][pvword] = freq
    for svword in secondaryvocab:
        totalcount = float(sum(secondaryvocab[svword].values()))
    for pvword in secondaryvocab[svword]:
        secondaryvocab[svword][pvword] /= totalcount
    return secondaryvocab

def fuzzymatch(colors, primaryvocab, fn, synonyms={}, threshold=0.7, verbose=False):
    for color, info in colors.items():
        tokens, freq, classes = info
        if 0 != len(classes): continue
        primarymatches = []
        for tok in tokens:
            if 0 != len(classes): continue
            primarymatches = []
            for tok in tokens:
for pvword in primaryvocab:
    score = fn(pvword, tok)
    if score >= threshold:
        if not pvword in classes:
            if verbose: print '{2:.3} {0} <- {1} ({3}, {4})' \
                .format(pvword, color, score, pvword, tok)
            classes.append(pvword)
for synword in synonyms:
    score = fn(synword, tok)
    if synword == 'magenta': print score
    if score >= threshold:
        if not pvword in classes:
            if verbose: print '{2:.3} {0} <- {1} ({3}, {4})' \
                .format(synonyms[synword], color, score, synword, tok)
            classes.append(synonyms[synword])
return

def unigram_counts(phrases, vocab=None):
    if None == vocab:
        vocab = dict()
    for p in phrases:
        for w in phrase_to_words(p.lower()):
            if not w in vocab:
                vocab[w] = 0
            vocab[w] += 1
    return vocab

def print_stats(colors):
    total = len(colors)
    n_match = len([x for x in colors.values() if 0 != len(x[2])])
    n_multimatch = len([x for x in colors.values() if len(x[2]) > 1])
    n_nomatch = total - n_match
    total = float(total)
    print 'By unique colors:'
    print '{0} matched ({3:.1%}); {1} multi-matched ({4:.1%}); {2} not matched
          .format(n_match, n_multimatch, n_nomatch, n_match/total, n_multimatch/total)
    total = sum([x[1] for x in colors.values()])
    return total

print_stats(colors)
n_match = sum([x[1] for x in colors.values() if 0 != len(x[2])])
n_multimatch = sum([x[1] for x in colors.values() if len(x[2]) > 1])
n_nomatch = total - n_match

print 'By items:'
print '{0} matched ({3:.1%}); {1} multi-matched ({4:.1%}); {2} not matched
      .format(n_match, n_multimatch, n_nomatch, n_match/total, n_multimatch/total)

def unmatched_phrases(colors):
  return [key for key, val in colors.items() if 0 == len(val[2])]

# Driver code

if True: # Enable to use human-generated synonym corpus
  primaryvocab, synonyms = parse_vocab(COLOR_VOCAB_SYN)
else:
  primaryvocab, synonyms = parse_vocab(COLOR_VOCAB)

colors = {}

colors = get_colors()

vocabmatch(colors, primaryvocab, synonyms=synonyms)

if True: # Enable if you want to see first-cut results
  print '--[ Exact matching ]--'
  print_stats(colors)
  print 'Unmatched:'
  print ',', ', '.join(unmatched_phrases(colors))

related_words = find_related(colors, primaryvocab, synonyms=synonyms)
secondaryvocab = build_secondary(related_words)

if False: # Enable if you want to see the secondary vocab
  print secondaryvocab

# Enable fuzzy-matching
if True:
    if True: # Enable to allow fuzzy-matching on synonyms
        fuzzymatch(colors, primaryvocab, fn=stringmatch.jaro, threshold=0.75, \
                    verbose=True, synonyms=synonyms)
    else:
        fuzzymatch(colors, primaryvocab, fn=stringmatch.jaro, threshold=0.75, \
                    verbose=True)
    print '---[ Fuzzy matching ]---'
    print_stats(colors)
    print 'Unmatched:'
    print ', '.join(unmatched_phrases(colors))

# Enable if you want to see results including secondary vocab
if True:
    svmatch(colors, primaryvocab, secondaryvocab, threshold=1.6, \
            synonyms=synonyms, verbose=True)
    print '---[ Secondary vocab matching ]---'
    print_stats(colors)
    print 'Unmatched:'
    print ', '.join(unmatched_phrases(colors))

# TODO: penalize a potential fuzzy-match if the word is a real english word (wi
#for phrase, matches in fuzzymatch(cat_nomatch, primaryvocab, stringmatch.jaro, 
#   print phrase, matches

if False: # Enable to see unigram counts, most frequent first
    phrases = get_colors()
    counts = unigram_counts(phrases)
    counts = sorted(counts.items(), key=lambda x: x[1], reverse=True)
    for word, count in counts:
        print word+'	'+str(count)

if not SILENT:
    print "...done!"
stringmatch.py

#!/usr/bin/env python

# Returns the characters in s that are in common with those in t
def commonchars(s, t):
    tt = str(t)
    maxdist = min(len(s), len(t)) / 2
    common = []
    for i in xrange(len(s)):
        pos = tt.find(s[i], max(0, i-maxdist), i+maxdist+1)
        if pos < 0:
            continue
        tt = tt[:pos] + '\0' + tt[pos+1:]
        common.append(s[i])
    return common

# The Jaro similarity metric (0-1 scale)
def jaro(s, t, verbose=False):
    if 0 == len(s) or 0 == len(t): return 0.0
    if s == t: return 1.0
    # Find characters common to the two strings
    scom = commonchars(s, t)
    tcom = commonchars(t, s)
    assert len(scom) == len(tcom)
    sigma = len(scom)
    if 0 == sigma:
        return 0.0
    # Count transpositions among common characters
    transp = 0
    for i in xrange(sigma):
        if scom[i] != tcom[i]:
            transp += 1
    transp /= 2
    if verbose:
        print 'sigma={0}, transp={1}, len_s={2}, len_t={3}'.format(sigma, transp, len(s), len(t))
    # Compute the actual similarity score
    sigma = float(sigma)
    17
return ((\sigma/\text{len}(s))+(\sigma/\text{len}(t))+((\sigma-\text{transp})/\sigma))/3.0

# The Damerau-Levenshtein distance
# (edit distance with unit weight and ins/del/transp ops)
# "Widely used in practice, but not suitable when whole segments of a
# string differ, e.g., when one string is a prefix of the second
# string ("Prof. John Doe" vs "John Doe") or when strings use
# abbreviations ("Peter J Miller" vs "Peter John Miller"). For
# these problems, see Smith-Waterman distance or affine gaps."
# WARNING: Very slow! If this actually is useful, best implement something less

def lev(s, t):
    if not s: return len(t)
    if not t: return len(s)
    return min(lev(s[1:], t[1:]) + (s[0] != t[0]),
               lev(s[1:], t)+1, lev(s, t[1:])+1)

def ngram_counts(s, n):
    counts = {}
    for ngram in [s[i:i+n] for i in range(0, len(s)-n+1)]:
        if not ngram in counts: counts[ngram] = 0
        counts[ngram] += 1
    return counts

# Dice’s similarity metric chops the strings up into n-grams.
def dice_sim(s, t, n=2):
    s_grams, t_grams = ngram_counts(s, n), ngram_counts(t, n)
    common = float(len([x for x in s_grams if x in t_grams]))
    return (2 * common)/(len(s_grams) + len(t_grams))

def cosine_sim(s, t, n=2):
    s_grams, t_grams = ngram_counts(s, n), ngram_counts(t, n)
    pairwise = float(sum([s_grams[x]*t_grams[x] for x in s_grams if x in t_grams])
    magprod = sum(s_grams.values()) * sum(t_grams.values())
    return (pairwise*pairwise) / magprod

if __name__ == '__main__':
    def test(s, t, verbose=False):
        lev_st, lev_ts = lev(s,t), lev(t,s)
jaro_st, jaro_ts = jaro(s,t,verbose=verbose), jaro(t,s,verbose=verbose)
dice_st = dice_sim(s, t)
cosine_st = cosine_sim(s, t)
assert lev_st == lev_ts
assert jaro_st == jaro_ts
# print '{0} & {1} & {2:.2%} & {3:.2%} & {4:.2%} \\'
# .format(s, t, lev_st, jaro_st, dice_st, cosine_st)
print '{0} & {1} & {2:.2%} \\'.format(s,t,jaro_st)
print 'Words	Lev	Jaro	Dice	Cosine'
test('silver','silver')
test('silver','slrv')
test('silver','slvr')
test('slrv','blue')
test('MARTHA','MARHTA')
test('DWAYNE','DUANE')
test('DIXON', 'DICKSONX')
test('piunk','pink')
test('blk', 'black')
test('white', 'with')
test('silver', 'sliver')
test('red', 'red_')
test('titanum', 'titanium')
# print jaro('Prof. John Doe','Dr. John Doe',verbose=True)
# print commonchars('CRATE','TRACE')
# print commoncharsB('CRATE','TRACE')
class TokenSeq(object):
    def __init__(self, string, offsets=None):
        self.string = string
        self.offsets = [] if None == offsets else offsets
        # Adds a new token over characters [i:j] in the underlying string.
        # Returns the ID of the new token.
        def add(self, i, j):
            self.offsets.append((i, j))
            return len(self.offsets)-1
        # Returns the tokens in [i:j] (or just i, if j is not given) as a
        # string, the way that they appeared in the original string (e.g.,
        # preserving any separators).
        def tokenstr(self, i, j=None):
            if None == j: j = i+1
            return self.string[self.offsets[i][0]:self.offsets[j-1][1]]
        # Returns a list of the tokens in [i:j].
        def tokenseq(self, i=None, j=None):
            return [self.string[ii:jj] for ii, jj in self.offsets[i:j]]
        def __repr__(self):
            return '<TokenSeq {0}>'.format(str(self.tokenseq))
        def __len__(self):
            return len(self.offsets)
        def prettyprint(self):
            return '
'.join([self.string[i:j] for i, j in self.offsets])

class NgramDictionary(object):
    # Builds an ngram dictionary supporting up to max_n-grams.
    def __init__(self, max_n):
        self.max_n = max_n
self.ngrams = [{},] * self.max_n
# Associates the production with the given ngram with probability p.
def add(self, ngram, production, p):
    n = len(ngram)
    if 0 == n or n > self.max_n: return
    d = ngrams[n]
    if not ngram in d:
        d[ngram] = []
    d[ngram].append((production, p))

# the "misc" character class (cc) is different in that its characters
# don't "stick together" to form tokens; each character is a different
# token
CC_WSPACE, CC_LETTER, CC_DIGIT, CC_MISC = range(0, 4)
CHARCLASSES = [(CC_WSPACE, string.whitespace),
               (CC_LETTER, string.ascii_letters+"'"),
               (CC_DIGIT, string.digits)]
def classify_char(c):
    for cc, chars in CHARCLASSES:
        if c in chars:
            return cc
    return CC_MISC
def tokenize(s):
    s = ''.join(s.split())  # just to clean up whitespace
    tokens = TokenSeq(s)
    edges = []
    if 0 == len(s): return tokens, edges
    last_char_class = classify_char(s[0])
    last_tok_start = 0
    for i in xrange(1, len(s)):
        cur_char_class = classify_char(s[i])
        # If we've changed character classes or last char was MISC
        # (whose chars are always split into different tokens), stop
        # here and add a token.
        if cur_char_class != last_char_class or last_char_class == CC_MISC:
            if last_char_class != CC_WSPACE:
                21
token_id = tokens.add(last_tok_start, i)
edges.extend([ParseEdge(token_id, token_id+1, tag, features) \
    for tag, features in \
    find_terminals(s[last_tok_start:i], last_char_class)
    last_char_class, last_tok_start = cur_char_class, i
]
token_id = tokens.add(last_tok_start, len(s))
edges.extend([ParseEdge(token_id, token_id+1, tag, features) \
    for tag, features in \
    find_terminals(s[last_tok_start:], last_char_class)]
return tokens, edges

import logging

# The characters s[i:k] are all of class cc, so let’s see what
# terminals we can make out of them!
def find_terminals(token, cc):
    log = logging.getLogger('kf.backend.parse.lexer')
    log.info('Looking for terminals in str: {0}'.format(token))

    if '' == token: return ''
    nts = []

    if '.' == token:
        nts.append(('DOT', {}))
    if re.match('^[0-9]+$', token):
        nts.append(('NUM', {}))

    if 0 == len(nts):
        nts.append(('UNKNOWN', {}))
    return nts

if __name__ == '__main__':
    unittest.main()
import unittest
import logging
import string
import re

class ParseEdge(object):
    def __init__(self, begin, end, tag, features=None, p=1.0, rule=None, dot=0, 
                 best_traversal=None):
        self.begin, self.end, self.tag, self.p = begin, end, tag, p
        self.rule, self.dot = rule, dot
        self.features = {} if None == features else features
        self.best_traversal = best_traversal

    def has_feature(self, key, val=None):
        if None == val:
            return key in self.features
        else:
            # TODO this'll need to get more complicated
            return key in self.features and self.features[key] == val

    def is_terminal(self):
        return None == self.rule

    def is_active(self):
        return None != self.rule and self.dot < len(self.rule.rhs)

    def string(self):
        return tokens.tokenstr(self.start, self.stop)

    # For an active edge, creates a new edge like this one but
    # advancing the dot one symbol via the passive edge (or
    # contextual, passive edge) other. The result edge may be active
    # or passive.
    def advance(self, other):
        assert self.is_active()

        if type(other) == ParseEdge:
            return ParseEdge(self.begin, other.end, self.tag, features={}, 
                              p=self.p*other.p, rule=self.rule, dot=self.dot+1, 
                              best_traversal=(self, other))

        elif type(other) == ContextualEdge:
            return ContextualEdge(self, other)
return ParseEdge(self.begin, self.end, self.tag, features={}, p=self.p*other.p, rule=self.rule, dot=self.dot, best_traversal=(self, other))

else:
    raise Exception('Unknown type. ')
# For an active edge, returns the next symbol on the left hand
# side of the rule (the next symbol after the dot).
def next_symbol(self):
    assert self.is_active()
    return self.rule.rhs[self.dot][0]
def edge_def(self):
    return (self.tag, self.begin, self.end)
def span_eq(self, other):
    return self.edge_def() == other.edge_def()

def __repr__(self):
    if None == self.rule:
        detail = self.tag
    else:
        detail = self.rule.lhs + '->' +
            .join([x[0] for x in self.rule.rhs[:self.dot]]) + ' ' +
            .join([x[0] for x in self.rule.rhs[self.dot:]])
    return '<Edge p={3} {2}[{0}:{1}]>'.format(self.begin, self.end, detail, self.p)

class ContextualEdge(object):
    def __init__(self, tag, features=None, p=1.0):
        self.tag, self.features, self.p = tag, features, p
    def __repr__(self):
        return '<CtxEdge p={0} {1}>'.format(self.p, self.tag)
    def is_terminal(self):
        return True

# Only rules marked 'accepting' will produce parse trees.
class ParseRule(object):
    def __init__(self, lhs, rhs, p=1.0, accepting=False):
        self.lhs, self.rhs = lhs, rhs
        self.p = p
        self.accepting = accepting
class ParserChart(object):
    # Creates an n-by-n parser chart. Typically n is len(tokens)+1.
    def __init__(self, n):
        self.log = logging.getLogger('kf.backend.parse.parserchart')
        self.log.info('Creating chart of size {0}.'.format(n))
        self.n = n
        self.chart = [[[[] for i in range(0, n)] for j in range(0, n)]
    # Adds the edge to the chart if it’s not already in it. Returns true iff it was added, and false if it was already there.
    def add(self, e):
        cell = self.chart[e.begin][e.end]
        if not e in cell:
            self.log.info('Adding edge to chart: ' + str(e))
            cell.append(e)
            return True
        else:
            self.log.info('Edge already in chart: ' + str(e))
            return False
    # Returns all edges covering [i:j].
    def edges_in_cell(self, i, j):
        return list(self.chart[i][j])
    # Returns all edges with endings at j.
    def edges_ending_at(self, j):
        edges = []
        for row in self.chart:
            edges.extend(row[j])
        return edges
    # Returns all edges with beginnings at i.
    def edges_beginning_at(self, i):
        edges = []
        for cell in self.chart[i]:
            edges.extend(cell)
        return edges
    # Returns true iff an edge equal to this one already exists in the chart.
    def is_discovered(self, e):
        pass
return None != self.get_existing_edge(e)
# If an edge equal to this one exists in the chart, it is
# returned; otherwise, None.
def get_existing_edge(self, e):
    for edge in self.edges_in_cell(e.begin, e.end):
        if e == edge:
            return edge
    return None

def __repr__(self):
    return '<ParserChart> -------' + \
    '
    Finishing Agenda\n' + \
    '
    '.join([str(x) for x in self.finishing_agenda]) + \
    '
    Exploration Agenda\n' + \
    '
    '.join([str(x) for x in self.exploration_agenda])

class ParseTree(object):
    def __init__(self, spanning_edge):
        self.root = self.walk_down(spanning_edge)
    def walk_down(self, e):
        if e.is_terminal():
            return (e, None)
        else:
            return (e, self.walk_back(e))
    def walk_back(self, e):
        if None == e: return []
        return self.walk_back(e.best_traversal[0]) + \
                [self.walk_down(e.best_traversal[1])]
    def node_to_str(self, node, n=0):
        edge, children = node
        if None == children:
            return (' '*(n*2))+'({0} p={1})'.format(edge.tag, edge.p)
        return (' '*(n*2))+'({0} p={1}
{2})' \
                .format(edge.tag, edge.p,
                        '
                        '.join([self.node_to_str(x, n+1) for x in children]))
    def __repr__(self):
        return self.node_to_str(self.root)

class ProbParser(object):
```python
def __init__(self, rules, tokens, edges, ctx):
    self.rules, self.tokens, self.edges, self.ctx = rules, tokens, edges, ctx
self.edgetable = {}
def parse(self, shortcircuit=False):
    self.shortcircuit = shortcircuit
    self.log = logging.getLogger('kf.backend.parse.probparser')

    # Contains traversals (active edges)
    self.exploration_agenda = []
    # Contains edges
    self.finishing_agenda = list(self.edges)
    self.chart = ParserChart(len(self.tokens) + 1)

    while 0 != len(self.finishing_agenda):
        while 0 != len(self.exploration_agenda):
            t = self.exploration_agenda.pop()
            self.explore_traversal(t)
            e = self.pop_finishing_item()
            if shortcircuit and self.is_solution(e):
                self.print_edge(e, 'Solution')
                return [ParseTree(e)]
            self.finish_edge(e)

    if shortcircuit:
        return None
    return [ParseTree(e) for e in self.chart.edges_in_cell(0, len(self.tokens)) if self.is_solution(e)]

def find_ctx_actives(self):
    self.ctx_active = {}
    for rule in self.rules:
        pass

def pop_finishing_item(self):
    return self.finishing_agenda.pop(self.finishing_agenda.index(max(self.finishing_agenda, key=lambda x: x.p)))
def is_solution(self, e):
    return self.is_spanning(e) and None != e.rule and e.rule.accepting
def is_spanning(self, e):
    return not e.is_active() and 0 == e.begin and len(self.tokens) == e.end

def print_edge(self, e, note):
    coverage = '-'*e.begin + '='*(e.end-e.begin) + '-'*(self.chart.n-1-e.end)
    self.log.info(coverage + ' ' + note + ' ' + str(e))
```

def explore_traversal(self, t):
    e1, e2, r = t
    self.print_edge(r, 'Exploring')
    if not self.chart.is_discovered(r):
        self.finishing_agenda.append(r)
        self.relax_edge(r, t)
    def relax_edge(self, e, t):
        self.print_edge(e, 'Relaxing')
        if e.edge_def() in self.edgetable:
            e = self.edgetable[e.edge_def()]
            e.best_traversal = t
            e.best_score = t[0].p * t[1].p
            pass
    def finish_edge(self, e):
        self.print_edge(e, 'Finishing')
        self.chart.add(e)
        self.apply_fundamental_rule(e)
        self.apply_bottom_up_rule(e)
    def apply_fundamental_rule(self, e):
        self.log.info(' Applying FR...')
        if e.is_active():
            # Active rule -- look for passive edges beginning at e.end
            for p in [x for x in self.chart.edges_beginning_at(e.end) \n                      if not x.is_active()]:
                if e.next_symbol() == p.tag:
                    r = e.advance(p)
                    self.log.info(' Produced result edge {0}'.format(r))
                    self.exploration_agenda.append((e, p, r))
            # Also look for possible contextual edges, which are all passive.
            for c in self.ctx:
                if e.next_symbol() == c.tag:
                    r = e.advance(c)
                    self.log.info(' Produced (ctx) result edge {0}'.format(r))
                    self.exploration_agenda.append((e, c, r))
            pass
        else:
            # Passive rule -- look for active edges ending at e.begin
            for a in [x for x in self.chart.edges_ending_at(e.begin) \n
if x.is_active():
    if a.next_symbol() == e.tag:
        r = a.advance(e)
        self.log.info(‘ Produced result edge {0}’ .format(r))
        self.exploration_agenda.append((a, e, r))
    return

def apply_bottom_up_rule(self, e):
    # Only passive edges can produce new edges via the bottom-up rule.
    if e.is_active(): return
    self.log.info(‘ Applying BUR...’)  
    for rule in self.rules:
        if rule.rhs[0][0] == e.tag:
            edge = ParseEdge(e.begin, e.end, rule.lhs, features={},
                             p=e.p*rule.p, rule=rule, dot=1,
                             best_traversal=(None, e))
            self.log.info(‘ Bottom-up rule creates ’ + str(edge))
            self.finishing_agenda.append(edge)
    return

# Each element in ctx is a ContextualEdge representing an edge that # can be inserted wherever convenient.
def parse(tokens, edges, rules, ctx=[], shortcircuit=False):
    parser = ProbParser(rules, tokens, edges, ctx)
    return parser.parse(shortcircuit)

if __name__ == '__main__':
    unittest.main()
features.py

import logging
import string
import re

def eval_features(pattern, bindings):
    result = {}
    for key in pattern:
        pval = pattern[key]
        if type(pval) == str and '?' == pval[0]:
            pval = bindings[pval[1:]]
            if type(pval) == dict:
                result[key] = eval_features(pval, bindings)
            else:
                result[key] = pval
    return result

# Tries to match the given features against the given feature pattern
# under the given bindings, which are updated as new variables are
# encountered.
def match_features(features, pattern, bindings):
    result = {}
    for pkey in pattern:
        if not pkey in features:
            return False
        pval = pattern[pkey]
        if type(pval) == str and '?' == pval[0]:
            if pval[1:] in bindings:
                # variable already bound; make sure bound value matches
                pval = bindings[pval[1:]]
            else:
                # variable not bound; bind and consider this matched
                bindings[pval[1:]] = features[pkey]
                continue
        if type(pval) == dict:
            if not type(features[pkey]) == dict: return False
            continue
if not match_features(features[pkey], pval, bindings): return False
continue
if features[pkey] != pval:
    return False
return True

import unittest
class TestFeatureMatch_Features(unittest.TestCase):
    def test_simple_eq(self):
        f={ ' a ' :' x ' ,' b ' :' y ' }
        p={ 'b' : ' y ' }
        self.assertTrue(match_features(f, p, {}))
    def test_simple_ne(self):
        f={ ' a ' :' x ' ,' b ' :' y ' }
        p={ 'b' : ' z ' }
        self.assertFalse(match_features(f, p, {})) # wrong value
        self.assertFalse(match_features({}, p, {})) # no key
    def test_binding(self):
        f={ ' a ' :' x ' ,' b ' :' y ' }
        p={ 'a' : '?a' }
        b={}
        match_features(f, p, b)
        self.assertEquals(b['a'], 'x')
    def test_multibind(self):
        f1 = {'a': 'x', 'b': 'y'}
        f2 = {'a': 'xx', 'b': 'y'}
        p1 = {'a': 'x', 'b': '?v'}
        p2 = {'b': '?v'}
        b={}
        self.assertTrue(match_features(f1, p1, b))
        self.assertTrue(match_features(f2, p2, b))
    def test_multibind_ne(self):
        f1 = {'a': 'x', 'b': 'y'}
        f2 = {'a': 'xx', 'b': 'yy'}
        p1 = {'a': 'x', 'b': '?v'}
        p2 = {'b': '?v'}
        b={}

31
self.assertTrue(match_features(f1, p1, b))
self.assertFalse(match_features(f2, p2, b))

def test_eval_a(self):
    p = {'a': 'x', 'b': 'y'}
    b = {'x': 'xx'}
    r = {'a': 'xx', 'b': 'y'}
    self.assertEqual(eval_features(p, b), r)

if __name__ == '__main__':
    unittest.main()
grammar.py

import re
import ply.lex
import ply.yacc
from parser import ParseRule

tokens = ['IDENT']
literals = ['[', ']', ',', '=', '+', '-']
t_ignore = (r' ')  
t_IDENT = (r'\w+')
def t_error(s):
    raise TypeError('Unknown text: {0}.format(s.value))
ply.lex.lex()

def p_error(p):
    print 'Syntax error at: {0}.format(p.value)

def p_symbollist(p):
    ""
    symbollist : symbol
    symbollist : symbol symbollist
    ""
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = [p[1]] + p[2]

def p_symbol(p):
    ""
    symbol : IDENT
    symbol : IDENT '[' featurelist ']'  
    ""
    if len(p) == 2:
        p[0] = (p[1], {})
    else:
        p[0] = (p[1], p[3])

def p_featurelist(p):
    ""
featurelist : feature
featurelist : feature ',' featurelist

if len(p) == 2:
    p[0] = {p[1][0]: p[1][1]}
else:
    p[0] = dict(p[3])
    p[0][p[1][0]] = p[1][1]

def p_value(p):
    value : IDENT
    value : '\[' featurelist '\']'

if len(p) == 2:
    p[0] = p[1]
else:
    p[0] = p[2]

def p_feature(p):
    feature : '+' IDENT
    feature : '-' IDENT
    feature : IDENT '=' value

if len(p) == 3:
    if '+' == p[1]:
        p[0] = (p[2], True)
    else:
        p[0] = (p[2], False)
else:
    p[0] = (p[1], p[3])

ply.yacc.yacc()

#ply.lex.input('a=b, b=c, c=d')
#for tok in iter(ply.lex.token, None):
#    print repr(tok.type), repr(tok.value)
def load_grammar(filename):
    rules = []
    f = open(filename, 'r')
    for line in f:
        line = line.strip()
        if 0 == len(line) or '#' == line[0]: continue
        if '*' == line[0]:
            accepting = True
            line = line[1:]
        else:
            accepting = False
        lhs_str, rhs_str = line.split('->', 1)
        p, lhs = [x.strip() for x in lhs_str.split(None, 1)]
        p = float(p)
        rhs = ply.yacc.parse(rhs_str.strip())
        rules.append(ParseRule(lhs, rhs, p=p, accepting=accepting))
    f.close()
    return rules
<table>
<thead>
<tr>
<th>Class</th>
<th>Secondary vocabulary</th>
</tr>
</thead>
<tbody>
<tr>
<td>pink</td>
<td>butterfly, slim, dk, vibrant, dusty, eva, foam, stripes, light, metallic, dark, hot, bliss, memory, baby, precious, nylon, skulls, vivid</td>
</tr>
<tr>
<td>brown</td>
<td>chestnut, java, dk, light, matte, mocha, drk, champagne</td>
</tr>
<tr>
<td>gray</td>
<td>charcoal, gunmetal, smoke, light, slate, slim, metal, gun, dark, titanium, warm, camo, iron, eva, olive, graphite, skulls</td>
</tr>
<tr>
<td>purple</td>
<td>eva, slim, royal, candy, metallic, dark, lt, glossy, mystic</td>
</tr>
<tr>
<td>clear</td>
<td>3mp, indigo, vg, lotus, aqua, frost, slim, midnight, deep, cobalt, bright, sea, navy, metallic, sky, usb, royal, lake, ice, imperial, camera, digital, memory, arctic, scientific, noble, ultramarine, cosmic, hawaiian, slate, foam, ds6310-b, oregon, wave, dark, flag, regular, cyan, cool, case, insert, eva, metal, artic, candy, ocean, graphite, tea, light, azure, nylon, scuba</td>
</tr>
<tr>
<td>blue</td>
<td>mini, dmc-df20, classic, tuxedo, slim, midnight, edition, matte, medium, glove, goo-black, heritage, wi-fi, jet, gloss, espresso, shoe, pearl, calm, memory, microfiber, piano, revolt, retro, steel, soft hard, ym, slate, foam, bundle, regular, olive, raspberry, special, case, leather, eva, metal, candy, racy, revolk, graphite, glossy, w, nylon</td>
</tr>
<tr>
<td>yellow</td>
<td>bright</td>
</tr>
<tr>
<td>orange</td>
<td>citrus, trim, bright, nylon, fiery</td>
</tr>
<tr>
<td>green</td>
<td>tea, slim, sage, mint, lime, leaf, pine, forest, memory, pinon, foam, wasabi, regular, moss, olive, plaid, eva, aqua, candy, pistachio, glossy, nylon, eco</td>
</tr>
<tr>
<td>white</td>
<td>whirl, medium, hard, arctic, ceramic, glaze, candy, pearl, glossy, soft, nerd</td>
</tr>
<tr>
<td>silver</td>
<td>and, gold, essence, titanium, zebra, 3x, with, camera, digital, champagne, u-ca3, 3.2mp, digimax, dark, warm, optical, brushed, brilliant, metal, zoom, bag, dmc-fz20, samsung</td>
</tr>
<tr>
<td>red</td>
<td>bundle, mini, rockstar, slim, deep, candy, rally, kodak, scientific, velvet, union, ruby, gloss, shimmer, camera, cardinal, digital, memory, crimson, brick, glossy, dk, foam, ds6310-b, oregon, regular, jack, with, inferno, plaid, flames, eva, metal, aztec, 3mp, blossom, flame, bordeaux, nylon, wine, raspberry, berry</td>
</tr>
</tbody>
</table>

Figure 7: Words associated with each class