COMP SCI 784: Stage-3 & Stage-4

Amrita Roy Chowdhury	Ekta Sardana	Roney Michael
roychowdhur2@wisc.edu	ekta@wisc.edu	rmichael2@wisc.edu

This report details our development process for the Stage-3 & Stage-4 of the COMP SCI 784 project for the Spring 2016 semester. Overall we were able to achieve **Precision**, **Recall** and **F1-Scores** of 96.05%, 90.19% & 93.03%, and **96.21%**, **91.18%** & **93.63%**, for the Stage-3 and Stage-4 parts of the project respectively.

We have also included a **write-up on the py_stringmatching package** in Section V of this report.

I. Overall Performance

Overall we were able to obtain the following values for the performance measures of interest:

- Precision: 96.25%
- Recall: 91.10%
- F-1 score: 93.60%
- Accuracy: 93.79%

We will cover the details of how the matcher was built and the different ideas we tried in section II, where we will first explain our attempts at feature extraction and the application and tuning of various Machine Learning approaches in stage-3, followed by some detail on what did did in stage-4 to improve upon the performance. This will be followed by an analysis of why we believe we were not able to further improve upon the performance, and discussions of what we think might be interesting ideas to explore to make the matcher better, in section III. Section IV consists of an appendix comprising of the code for our feature extractor and the best performing matcher, followed by an appendix on the discussion of the py_stringmatching package[1] and our key observations about the same.

[1] http://anhaidgroup.github.io/py_stringmatching/

II. The Matcher

The purpose of the matcher here is in the simplest of terms, to predict a MATCH or a MISMATCH when provided with input pertaining to two product descriptions. There are many ways to achieve this such as by writing rules, using machine learning, leveraging crowdsourcing, etc., but for the purposes of this project, the base matcher built in stage-3 leverages classification approaches from Machine Learning. Stage-4 then took this a bit further by handling sets of corner cases by white- and black-list rules. We go into more detail below.

Stage-3

Building the Matcher

For the machine learning approach we adopted, the building of the matcher consisted of essentially two distinct parts:

1. The first part consisted of converting an input product pair of non-precise structure into a distinct feature vector representation that may be easily analysed and understood by common ML approaches.
2. The second involved applying various ML approaches for classification to the problem, and tuning the first part accordingly.

Feature Extraction

When considering extracting features from the JSON representation of product pairs we were provided, we were faced with some tough questions:

- What attributes of the products are useful and what are noise?

  Modern machine learning approaches are generally considered to be adept at filtering out useless attributes from consideration while building the model. How this is achieved is different for different algorithms and may incorporate ideas such as entropy and mutual information of the attributes.

  Keeping with this general theme we decided to go ahead with preparing a massive feature set and trimming it down later if necessary. The idea was to generate as many features as may be available from every single instance in the dataset, and let the ML algorithms decide what to pick-up and what to discard.

  We did a bit of cleaning to the data before we started comparing them field-by-field. Specifically, we removed all html tags from the attribute values and also replaced all sequences of non-alphanumeric characters with a white-space,

since these did not provide any reasonable degree of distinguishing information. Additionally, we also made use of the stop_words python package and removed from our fields all such “noise” words.

- How would two products be compared?

  We came up with four distinct similarity measures to compare product pairs attribute-wise. Each similarity measure was applied to every possible available attribute pair.

  We used the py_stringmatching package as our basic starting point for comparing to products in a product pair. We started off using the monge_elkan measure with whitespace based tokenisation. This was chosen due to it’s apparent resiliency to minor noise in the data set for similar products, being a derivative form of the Levenshtein distance.

  This worked reasonably well, but after analysing some preliminary results we found that certain matching products differed a great deal in their product long descriptions though they retained the same keywords with minor variations. This difference occurred in terms of the overall length as well as the position at which the keywords occurred. Due to these factors, we decided to add a second similarity measure to compare each attribute, which was the overlap_coefficient similarity with 3-gram tokenization. The 3-gram tokenization allowed us to be resilient to misspellings and spelling variations, while the overlap_coefficient allowed comparison of attributes independent of the difference in their lengths, being that the denominator of the similarity function was based on the 3-gram set of smaller size.

  On analysing the results after adding the second similarity measure, we noticed there there were two distinct more or less mutually exclusive cases which were often misinterpreted.

  a. Cases where two MISMATCH products matched completely except for the mention of different model numbers like character sequences in the product name or description fields.

  b. Cases where two MATCH products had their names and descriptions worded quite differently except for certain model number like character sequences in the product name or description fields.

  Note that even though we mention only the name and description fields here, the same problem occurred occasionally with other attributes as well.

  To handle these cases, we came up with a methodology by which, we would also compare two products in a pair after preprocessing the pair by:

a. Removing all numeric characters from the attribute value and only comparing the resultant string.

b. Removing all non-numeric characters from the attribute value and only comparing the resultant string.

This helped us account for subtle differences in model numbers which were most often sequences of numbers, at least in part. The final similarity measure for these last two features/attribute, also utilised the overlap_coefficient function with 3-gram tokenization.

- What can we do with missing values?

  As we had found in stage-1, the dataset had a considerable problem with missing values. That is, different products had JSON representations consisting of very minimal subsets of the entire attribute set. The issue was so problematic that a large chunk of products had only about 5 attributes while the entire attribute set contained more than 500.

  A brand extractor had been developed for the project as part of stage-2 and we used that to fill in the missing values for the 'Brand' attribute wherever we were able to extract it from the 'Product Name' field. Our experiments in stage-2 had show the precision of this exercise to be greater than 99%, so we expect that this will only improve our net results if at all affected.

  For all four feature values for each attribute, we treated missing values in a pessimistic fashion by setting the corresponding similarities to zero. This essentially treats a missing value as a complete attribute mismatch. The same idea was used even in cases where both products in the pair were missing the attribute. We adopted this approach since the primary objective here was to get the precision of the resultant matcher above 96%. By adopting a pessimistic approach, we conceptually enforce the condition that a missing value will in no way contribute positively to a product pair being declared a match.

- Heuristics

  We also observed an interesting class of cases in which one product would have a 'Manufacturer Part Number' attribute while the other would have the same value, but in some other attribute, usually in the name or description. To handle this we made our feature extraction so that when one product had this attribute and the other had the corresponding value in its body, we would set the similarity measures for the 'Manufacturer Part Number' to the maximum value of one, whereas it would otherwise have been zero due to it being a missing attribute.. This helped the ML algorithm catch and correct many false negatives, improving

both the precision and the recall. Based on this success, we also applied the same idea to the 'Actual Color' attribute.

After doing all the above for feature extraction, our matcher's precision was still marginally below the targeted 96% with the default confidence threshold of 0.5. To bring this value safely above the threshold, we set the minimum confidence threshold for predicting the positive class (i.e., a MATCH) to 0.55, which helped us achieve our target.

The final code for our feature extractor is provided in Appendix-A of this report.

The Matcher

Building the matcher was a fairly straightforward task.

We made use of the scikit-learn python package[2] which had many state-of-the-art machine learning approaches built into it. Various approaches were tried, including, but not limited to Decision Trees, Random Forest, Naive Bayes, Support Vector Machine, Logistic Regression, Neural Network, Adaptive Boosting, Bagging, Extra Trees and so on, as well as ensemble approaches which attempted to combine the prediction of these approaches through various weighted voting schemes.

Five fold cross-validation was used on the set-X instances to identify and correct issues with our feature extraction and final metrics were reported on the set-Y instances.

We found that the Extra Trees with Adaptive Boosting and the Random Forest Classifier performed the best and fairly equivalently. The code for our matcher based on the Random Forest Classifier is provided in Appendix-B on this report. We are omitting the code for our other approaches, since there is nothing really interesting there; the only differences required to change the classifier is the initialization step for the classifier variable. Ensembles may similarly be trivially constructed by doing the same activity twice and taking a majority vote of the results (if equally weighted).

A few points of note from applying these machine learning approaches were:

- Naive Bayes gave terrible performance of less than 50% for the precision for this problem. This was quite surprising because since it was a 2 class problem, this would mean that it more accurately gets it wrong than right.
- Increasing the complexity of the neural network did not make a strictly quantifiable improvement to the performance. We observed almost equivalent performances for this problem, for a single perceptron as for larger multi-layer networks.

[2] http://scikit-learn.org/

- Adaptive boosting and bagging both almost always improved the performance of the base classifiers, but this improvement was not observed in the case of rondom forests, most likely due the the fact that the classifier is already an ensemble of decision trees in a sense.
- Other than the two tree based approaches, logistic regression was the only classifier which gave us an f1 score above or near 90%.

Performance

Overall we achieved average Precision, Recall and F1-Scores of **96.05%**, **90.19%** & **93.03%** on set Y with our Stage-3 matcher.

We were informed that the performance of our matcher on the blind data set was such that the Precision, Recall and F1-Scores (rounded off to two decimal places) were **96.45%**, **90.72%** and **93.50%** respectively.

Stage-4

Modifications Made

The objective here was to improve the recall of the matcher as much as possible while still maintaining a precision above 96% (i.e., essentially improving the f1 score). Do do so we decided to follow the idea of adding overriding blacklist and whitelist rules to filter out some identifiable classes of false positives and false negatives.

We tried a host of different approaches where many failed due to being so broad that it not only corrected existing mistakes but also muddled up some previously correct predictions. In the end we had to settle for a set of four rules which we found to enhance our performance marginally, boosting the net f1 score by about 0.50%.

The rules applied were as follows:

- **Whitelist for UPC and GTIN:** The UPC and GTIN could be considered to be “keys” where product matches were concerned. However these were seldom of use since the fields almost exclusively only occurred in one of the two products in the pair. These values did however occur rarely in the body of the other product’s JSON, and we therefore adopted a method similar to that for the ‘Manufacturer Part Number’ described below, but instead of modifying the feature values, directly set the result to be a match. In doing so, we realized that there was a second class of GTIN & UPC related false negatives, which were those in which the UPC of one product would provide a partial key (generally of length greater than five) in the body of the other, which we would also consider as a match.

- **Whitelist for Manufacturer Part Number:** Due to extraneous noise (symbols, etc), there were false negative cases which were not being covered by the 'Manufacturer Part Number' feature rule, but which were seemingly matches. We therefore added a whitelist rule such that if an Manufacturer Part Number match was found after deleting all non-alphanumeric characters, then it would be considered a MATCH overall. This rule would only be effected when the confidence of the positive class output by the classifier was at least 0.2.
- **Blacklist for Condition:** We found a non-trivial number of false positive cases where the products were adjudged a MATCH and they indeed were almost perfectly similar, except that one would be a new product while the other was refurbished. We created a rule to handle that, but this also seemed to affect some of our true positives negatively. Nevertheless we kept this on as it seemed to do more good than harm. This rule would only be effected when the confidence of the negative class output by the classifier was at least 0.2.
- **Whitelist for Product Long Description:** The long description being fairly long was a good indicator of mathes. However, due to differences in punctuation, such similarities were often lost. A rule was therefore added to whitelist cases where the Product Long Description of the two products matched exactly, barring differences in punctuation, separators and spacing. This rule would only be effected when the confidence of the positive class output by the classifier was at least 0.45.

There were also a number of ideas which we tried but found to do more harm than good and were therefore discarded. A couple of note were:

- If two products had a staggering difference in the number of attributes, generate a sim score with only those common attributes (minus the long description, since that was a very open ended field) and predict the pair as a MATCH when the sim score was a perfect one.
- Use product names to lookup the UPC number on an online UPC knowledge base[3] by crafting a query URL and querying for a product each half-second. The first UPC result (if any) would be considered as the UPC for the product and if this matched for both the products, declare the pair as a MATCH.

Performance

After incorporating all the changes made in stage-4, our Precision, Recall and F1-Score on the set-Y, all improved marginally to **96.21%**, **91.18%** & **93.63%** respectively.

[3] http://www.yoopsie.com/

III. Analysis & Future Work

Performance Limitations

The major reason why we think the performance cannot be improved further is because of the inherent flaw in the labeling of the dataset. We found a recurring error pattern in the labelled work data provided as is delineated below.

False Positives

While inspecting the product pairs falling in set of false positives returned by our matcher set we found that which means our classifier does a better job than the provided label. Product pairs, which on human inspection is quite evidently found to be a definitive match, is wrongly labeled as a mismatch. The pairs are of the following patterns:

1. The most important attributes like ‘Product Name’, ’Long Description’, ’Brand’ , ’Product Segment’ and Product Type, etc of both the products match, in most cases, verbatim.The only difference between the two products is that the number of attributes for one is more than that of the other. But the extra attributes appearing in one of the products are like ‘Warranty Information’ , ‘Country of origin’, ’E-Waste Recycling Compliance Required’, ’ etc. These attributes have very less frequency over the entire dataset and carry little information, hence they should contribute meagerly to the matcher. Hence our matcher correctly labelled them as a true match in contrast to the label of the golden data. This anomaly was the most prominent one in the set of false positives (around 90%).

   Example:

   Product 1:


```
"Product Name":"PARKER 66PLP-5-2, Female Connector, NP Brass, 5/16 In, PK
10"
"Product Type":"Pipe Fittings & Couplers"
"Manufacturer Part Number":"66PLP-5-2"
"Brand":"Parker”
"Category":"Water Valves & Fittings"
"Product Long Description":"Female Adapter, Basic Material Metal, Body
Material Nickel Plated Brass, Connection Type Tube x FNPT, Tube Size 5/16
In., Pipe Size 1/8 In., Max. Pressure 300 psi, Temp. Range 0 Degrees to
200 Degrees F, Package Quantity 10<br><b>Features</b><ul><li>Item :
Female Adapter</li><li>Length : 1.25</li><li>Material of Construction :
Nickel Plated Brass</li><li>Outside Dia. : .025</li><li>Package Quantity
: 10</li><li>Pressure (PSI) : 300</li><li>Temp. Range : 0 Degrees to 200
Degrees F</li><li>Thread Size : 1/8</li><li>Body Material : Nickel Plated
```

```
Brass</li><li>Connection : Tube x FNPT</li><li>Pipe Size :
1/8</li><li>Tube Size : 5/16</li><li>Max. Pressure : 300
psi</li><li>Connection Type : Tube x FNPT</li><li>Fittings Sub-Category :
Adapters</li><li>Basic Material : Metal</li></ul>"
"GTIN":"II58OLTEma6815"
"Product Short Description":"Female Connector, NP Brass, 5/16 In, PK 10"
"Warranty Information":"Y”
"UPC":"58OLTEma6815"
"Product Segment":"Electronics"
```


Product 2:


```
"Product Type":"Cable Connectors"
"Product Name":"PARKER 66PLP-5/32-2 Female Connector NP Brass 5/32 In PK
10"
"Brand":"Parker"
"Product Long Description":"Female Adapter Basic Material Metal Body
Material Nickel Plated Brass Connection Type Tube x FNPT Tube Size 5/32
In. Pipe Size 1/8 In. Max. Pressure 300 psi Temp. Range 0 Degrees to 200
Degrees F Package Quantity 10<br><b>Features</b><ul><li>Item : Female
Adapter</li><li>Length : 1.17</li><li>Material of Construction : Nickel
Plated Brass</li><li>Outside Dia. : .125</li><li>Package Quantity :
10</li><li>Pressure (PSI) : 300</li><li>Temp. Range : 0 Degrees to 200
Degrees F</li><li>Thread Size : 1/8</li><li>Body Material : Nickel Plated
Brass</li><li>Connection : Tube x FNPT</li><li>Pipe Size :
1/8</li><li>Tube Size : 5/32</li><li>Max. Pressure : 300
psi</li><li>Connection Type : Tube x FNPT</li><li>Fittings Sub-Category :
Adapters</li><li>Basic Material : Metal</li></ul>”
"Product Segment":"Electronics”
```


2. The two products had a very subtle difference between them like in the colour/watt/ size etc and was a verbatim match otherwise.

Example:

Product 1:


```
"Product Name":"HP - Power supply ( plug-in module ) - 700 Watt - United
States - for HP 5406R, 5406R zl2, 5412R, 5412R zl2"
```


Product 2:


```
"Product Name":"HP - Power supply ( plug-in module ) - 1100 Watt - for HP
5406R 5406R zl2 5412R 5412R zl2"
```


Our reasoning is that since the overall matching for both the products is very high, it results in a reasonably high similarity score. The small difference is usually a

single word difference, hence our matcher fails. For features like color which have explicit attributes 'Color' our matcher takes care of it as described under heuristics. But for the rest of the features, as seen in the examples, the universal set of possibilities can be huge. Moreover it largely depends on the type of the product at hand, i.e., a laptop will differ in features like OS used, USB ports 3.0 or 2.0, etc while a headphone can differ in whether the fit is circumaural or in-ear. Hence it was not feasible to come up with a reasonable set of all such features in the given time frame to improve our matcher.

False Negatives

In order to improve the recall we need to reduce the number of false negatives.Hence we started debugging by looking into the product pairs in the set of false negatives. We saw identical pattern of product pairs as seen in the case of False Positives.

The product pairs were of the following form:

1. Product names, types, brand and description matches but the number of attributes in one of the products is much higher than of the other. While one has typically 4 -5 attributes (product name, product type, long description, brand segment), the other can have as high as 40 different attributes, i.e, the product with the larger number of attributes is a more specific version of the other. This is similar to the case 1 seen in false positives.
   The occurrence of the similar data pattern in both false positives and false negatives indicate that there is an inconsistency in the labeling of the golden data provided, i.e., while some pairs of this form are labeled as a match , the rest is not. This flaw, we believe, confuse our trainer and result in this anomalous outcome. We tried out a heuristic of matching only the common attributes in the event the number of a large disparity in the number of attributes in the product pair, while it improved false negatives, it introduced new pairs in the false positive. This can again be attributed to the golden data labelling anomaly.

   Example:

   Product 1:


```
"Product Name":"1000FT Cat 6 Bulk Bare Copper Ethernet Network Cable UTP,
Stranded, In-Wall Rated, 550MHz, 24AWG - Gray"
"Product Type":"Network Cables"
"Brand":"Monoprice"
"Category":"Networking Cables & Connectors"
"Product Long Description":"Avoid having excess Ethernet cable lying
around by building your own cables to the exact length needed using this
Bulk Ethernet Copper Cable from Monoprice!<br><br>This 1000 foot roll of
unshielded (UTP) Cat6 cable uses stranded conductors and features 550MHz
bandwidth capacity. Stranded conductors are more flexible and less prone
```

to breakage than solid conductors, making stranded cables well suited for less permanent runs and for connecting client computers. This cable carries a CM fire safety rating, which means that it is safe for use within the walls of commercial buildings.

Monoprice Cat5e and Cat6 cables are made of 100% bare copper wire, as opposed to copper clad aluminum (CCA) wire, and are therefore fully compliant with UL Code 444 and National Electrical Code TIA-568-C.2 fire and safety standards, which require pure bare copper wire in Cat5e and Cat6 communications cables.

*** 30-day easy returns. No restocking fee. Free lifetime technical support. Limited lifetime warranty. ***"
"Country of Origin: Components":"Imported"
"GTIN":"00844660022671"
"Product Short Description":"Avoid having excess Ethernet cable lying around by building your own cables to the exact length needed using this Bulk Ethernet Copper Cable from Monoprice!

This 1000 foot roll of unshielded (UTP) Cat6 cable uses stranded conductors and features 550MHz bandwidth capacity. Stranded conductors are more flexible and less prone to breakage than solid conductors, making stranded cables well suited for less permanent runs and for connecting client computers. This cable carries a CM fire safety rating, which means that it is safe for use within the walls of commercial buildings.

Monoprice Cat5e and Cat6 cables are made of 100% bare copper wire, as opposed to copper clad aluminum (CCA) wire, and are therefore fully compliant with UL Code 444 and National Electrical Code TIA-568-C.2 fire and safety standards, which require pure bare copper wire in Cat5e and Cat6 communications cables.

*** 30-day easy returns. No restocking fee. Free lifetime technical support. Limited lifetime warranty. ***"
"Actual Color":"Multicolor"
"Color":"Multicolor”
"UPC":"844660022671"
"Product Segment":"Electronics"

Product 2:

"Product Type":"Computer Cables”
"Product Name":"1000FT Cat 6 Bulk Bare Copper Ethernet Network Cable UTP Stranded In-Wall Rated 550MHz 24AWG - Gray"
"Brand":"Monoprice"
"Product Long Description":"Avoid having excess Ethernet cable lying around by building your own cables to the exact length needed using this Bulk Ethernet Copper Cable from Monoprice!

This 1000 foot roll of unshielded (UTP) Cat6 cable uses stranded conductors and features 550MHz bandwidth capacity. Stranded conductors are more flexible and less prone to breakage than solid conductors making stranded cables well suited for less permanent runs and for connecting client computers. This cable carries a CM fire safety rating which means that it is safe for use within the walls of commercial buildings.

Monoprice Cat5e and Cat6 cables are made of 100% bare copper wire as opposed to copper clad

aluminum (CCA) wire and are therefore fully compliant with UL Code 444 and National Electrical Code TIA-568-C.2 fire and safety standards which require pure bare copper wire in Cat5e and Cat6 communications cables.

*** 30-day easy returns. No restocking fee. Free lifetime technical support. Limited lifetime warranty. ***"
"Product Segment":"Electronics"

2. ‘Product Name’, ‘Brand’, ’Product Type’ matches , ‘Product Long Description’ is semantically equivalent for both the products but string matching methodologies return poor result. Additionally one of the products have a substantially longer description than the other, mostly greater than twice the size of the other. In some cases the long descriptions mostly consist of non overlapping information.Hence the string matching similarity scores would be <=0.5.
   Our belief is that this low score is causing the matcher to predict it incorrectly. We tried out a heuristic, if the product name, brand and type match exactly and then for product long description similarity scores above threshold T, where T was varied from 0.4 to 0.5, we bypass the classification of the matcher and go ahead and label the product pair as a match. However though this decreased the false negatives, it introduced quite a few false positives which decreased the precision.We tried out varying the parameters like changing exact match of the names to a high threshold match but none seemed to improve our performance. The reason why the false positives increased is again the aforementioned inconsistency in the golden dataset labeling.

   Example:

   Product 1:

"Product Name":"Tripplite Srcableduct1u Smartrack 1u Horiz Cbl Manager"
"Product Type":"Cable Organizers"
"Brand":"Tripp Lite"
"Category":"Cable Management Kit"
"Product Long Description":" Tripp Lites SmartRack SRCABLEDUCT1U helps eliminate cable stress for your rack enclosure cabinet. The 1U 19-inch horizontal cable manager (finger duct with cover) organizes cables within the SmartRack enclosure or open rack. Required mounting hardware is included. <U>Product Information</U>

Cable Manager Type: Cable Cover

<U>Physical Characteristics</U>

Color: Black
Material: Steel
Rack Height: 1U
Panel Width: 19

<U>Miscellaneous</U>

Package Contents: Horizontal Cable Manager
1U 19 inch Horizontal Cable Manager -finger duct with cover, mounting hardware
Additional Information: Helps

eliminate cable stress
Organizes cables within SmartRack enclosure or open rack
Compatibility: SmartRack "
"GTIN":"II58OKCXfr1194"
"Product Short Description":" Tripp Lites SmartRack SRCABLEDUCT1U helps eliminate cable stress for your rack enclosure cabinet. The 1U 19-inch horizontal cable manager (finger duct with cover) organizes cables within the SmartRack enclosure or open rack. Required mounting hardware is included. "
"UPC":"58OKCXfr1194"
"Product Segment":"Electronics"

Product 2:

"Product Type":"Computer Rack Hardware & Accessories"
"Product Name":"Tripplite SRCABLEDUCT1U SmartRack 1U Horiz Cbl Manager"
"Brand":"Tripplite"
"Product Long Description":"1U 19 Horizontal Cable Manager -finger duct with cover- Helps eliminate cable stress- Organizes cables within SmartRack enclosure or open rack- Required mounting hardware included- Cold rolled steel with Black finish- SKU: DHSRCABLEDUCT1U"
"Product Segment":"Electronics"

Potential Improvements

We list out some improvements which we believe could potentially improve the performance.

1. One obvious approach can be trying out more evolved and sophisticated techniques like deep learning algorithms Deep Boltzmann Machine (DBM), Deep Belief Networks (DBN), Convolutional Neural Network (CNN) , Stacked Auto-Encoders etc.
2. Trying out a dictionary based semantic matching algorithm for addressing case in false negatives.
3. Generating a set of features for the different types of products, then designing a suitable extractor to extract the features from the product name or description to address case 2 under false positives.

IV. Appendix

Appendix-A: Code for the Feature Extractor

```
import re
import os
import sys
import json
from operator import itemgetter
from stop_words import get_stop_words
from py_stringmatching import simfunctions, tokenizers

def DEBUG(debug):
    if IS_DEBUG:
        print debug
    return

def sublistExists(list1, list2):
    return ''.join(map(str, list1)) in ''.join(map(str, list2))

def buildBrandExtractor():
    if os.path.isfile('elec_brand_dic.txt'):
        with open('elec_brand_dic.txt') as f:
            for line in f:
                line = unicode(line, errors='ignore') #For character which
are not utf-8
                data = line.strip().split('\t') #.lower()
                brand = data[0]
                brands.update({data[0]:(brands.get(data[0], 0) +
int(data[1]))})
            f.close()

    global brands_list, symbol_re, number_re, nonnumber_re, nonAlphaNum_re,
brands_re, preposition_re, limitIndex, refurbished_re, CaSe_re, CASE_re

    brands_list = sorted(brands.items(), key=lambda x: len(x[0]))

    symbol_re = re.compile('[^\\w]+', re.IGNORECASE)
    number_re = re.compile('[0-9A-Z]+')
    nonnumber_re = re.compile('[^0-9A-Z][^0-9A-Z]+')
    nonAlphaNum_re = re.compile('[^\\dA-Z]+', re.IGNORECASE)
    brands_re = list()
    for brand, count in brands_list:
        brands_re.append(re.compile('\\b' + symbol_re.sub('[^\\w]{0, 3}',
brand) + '\\b', re.IGNORECASE))
    #Heuristics
    preposition_re = re.compile('\\bfor|with\\b', re.IGNORECASE)
    limitIndex = 50
    refurbished_re = re.compile('^Off\s*Lease\s*REFURBISHED\s*',
```

```
re.IGNORECASE)
    CaSe_re = re.compile('^[A-Z]+[a-z]+-?[A-Z]+\\w+\\b')
    CASE_re = re.compile('^[A-Z]{5, }\\b') #CAPS, but not A, AN, THE, etc.

    return

def extractBrand(productName):
    #if not BRAND_EXTRACTOR:
    #   buildBrandExtractor()
    try:
        matched = CaSe_re.search(productName).group(0)
    except:
        matched = None
    try:
        for idx, brand in enumerate(brands_re):
            match = brand.search(productName)
            if match is not None:
                if matched == None:
                    matched = match.group(0)
                else:
                    matchIndex = refurbished_re.sub('',
productName).index(match.group(0))
                    matchedIndex = refurbished_re.sub('',
productName).index(matched)
                    if matchIndex <= matchedIndex:
                        matched = match.group(0)
        if matched is None:
            try:
                matched = CASE_re.search(productName).group(0)
            except:
                matched = None
        if matched is not None:
            matchedIndex = refurbished_re.sub('',
productName).index(matched)
            if matchedIndex > limitIndex:
                matched = None
        if matched is not None:
            matchedIndex = productName.index(matched)
            prepositionMatch = preposition_re.search(productName)
            if prepositionMatch is not None:
                prepositionIndex =
productName.index(prepositionMatch.group(0))
                if prepositionIndex < matchedIndex:
                    matched = None
    except:
        print >> sys.stderr, productName
        matched = None
    return matched

def initMain():
    with open(sys.argv[1]) as f:
```

```
        inputLines = f.readlines()
        f.close()
    global BRAND_EXTRACTOR, products, attributes, pairs, brands, stop_words
    BRAND_EXTRACTOR = False
    products = dict()
    attribute_counts = dict()
    pairs = list()
    brands = dict()
    stop_words = set()
    stop_words.update(get_stop_words('en'))

    html_re = re.compile('((<[^<]*?>)|( ))+', re.IGNORECASE)
    junk_re = re.compile('[;]+', re.IGNORECASE)

    for line in inputLines:
        line = unicode(line, errors='ignore') #For character which are not
utf-8
        line = junk_re.sub(' ', html_re.sub('', line))
        data = line.strip().split('?')

        pairId = data[0]
        prod1_id = data[1] + '_1'
        if (data[2]):
            prod1_json = json.loads(data[2])
        else:
            prod1_json = dict()
        prod2_id = data[3] + '_2'
        if (data[4]):
            prod2_json = json.loads(data[4])
        else:
            prod2_json = dict()
        products.update({prod1_id:prod1_json})
        products.update({prod2_id:prod2_json})
        pairs.append((prod1_id, prod2_id))
        for k in prod1_json.keys():
            attribute_counts.update({k:(attribute_counts.get(k, 0) + 1)})
        for k in prod2_json.keys():
            attribute_counts.update({k:(attribute_counts.get(k, 0) + 1)})
        if prod1_json.get('Brand', None) is not None:

brands.update({prod1_json['Brand'][0]:(brands.get(prod1_json['Brand'][0], 0)
+ 1)})
        if prod2_json.get('Brand', None) is not None:

brands.update({prod2_json['Brand'][0]:(brands.get(prod2_json['Brand'][0], 0)
+ 1)})

    attributes = sorted(attribute_counts.items(), key=itemgetter(1),
reverse=True)
    attributes = [x[0] for x in attributes]
    return
```

```
def updateMPN(product):
    if products[product].get('Manufacturer Part Number', None) is not None:
        mpn = products[product].get('Manufacturer Part Number',
None)[0].strip().split()
        if ' ' in mpn:
            for idx in range(0, len(mpn)):
                if mpn[idx] in brands_list:
                    mpn[idx] = ''
            mpn = ' '.join(' '.join(mpn).split())
            products[product].update({'Manufacturer Part Number':[mpn]})
    return

def updateBrand(product):
    if products[product].get('Brand', None) is None:
        brand = extractBrand(products[product].get('Product Name', [''])[0])
        if brand is not None:
            products[product].update({'Brand':[brand]})
    return

def updateAllBrands():
    for p in products:
        updateBrand(p)
    return

def delNumber(product, feature):
    featureValue = ''
    if products[product].get(feature, None) is not None:
        featureValue = number_re.sub('', products[product].get(feature)[0])
    return featureValue

def delNonNumber(product, feature):
    featureValue = ''
    if products[product].get(feature, None) is not None:
        featureValue = nonnumber_re.sub(' ',
products[product].get(feature)[0]).strip()
    return featureValue

def monge_elkan(pair, feature):
    if feature in products[pair[0]] and feature in products[pair[1]]:
        return
simfunctions.monge_elkan(tokenizers.whitespace(products[pair[0]].get(feature
, [''])[0].lower()), tokenizers.whitespace(products[pair[1]].get(feature,
[''])[0].lower()))
    else:
        return noneValue

def overlap_coefficient(pair, feature):
    if feature in products[pair[0]] and feature in products[pair[1]]:
        val = [products[pair[0]].get(feature, [''])[0].lower(),
products[pair[1]].get(feature, [''])[0].lower()]
```

```
        val[0] = ' '.join([word for word in val[0].split() if word not in
stop_words])
        val[1] = ' '.join([word for word in val[1].split() if word not in
stop_words])
        return
simfunctions.overlap_coefficient(tokenizers.whitespace(val[0]),
tokenizers.whitespace(val[1]))
    else:
        return noneValue

def isDifferentModel_1(pair, feature):
    if feature in products[pair[0]] and feature in products[pair[1]]:
        return
simfunctions.overlap_coefficient(tokenizers.qgram(delNumber(pair[0],
feature), 3), tokenizers.qgram(delNumber(pair[1], feature), 3))
    else:
        return noneValue

def isDifferentModel_2(pair, feature):
    if feature in products[pair[0]] and feature in products[pair[1]]:
        return
simfunctions.overlap_coefficient(tokenizers.qgram(delNonNumber(pair[0],
feature), 3), tokenizers.qgram(delNonNumber(pair[1], feature), 3))
    else:
        return noneValue

def getFeatureVector(pair):
    featureVector = list()
    for feature in features:
        featureVector.append('{0:.3f}'.format(monge_elkan(pair, feature)))
        featureVector.append('{0:.3f}'.format(overlap_coefficient(pair,
feature)))
        featureVector.append('{0:.3f}'.format(isDifferentModel_1(pair,
feature)))
        featureVector.append('{0:.3f}'.format(isDifferentModel_2(pair,
feature)))
    return featureVector

def main():
    initMain()
    buildBrandExtractor()

    global features, noneValue
    num_attributes = len(attributes)
    if len(sys.argv) > 2:
        num_attributes = int(sys.argv[2])
    features = attributes[0:num_attributes]
    idx_MPN = features.index('Manufacturer Part Number')
    idx_AC = features.index('Actual Color')
    noneValue = 0
    num_feature_variants = 4 #Edit accordingly
```

```
    featureNames = list()
    for feature in features:
        f = symbol_re.sub('', feature)
        for i in range(0, num_feature_variants):
            featureNames.append(f + '_' + str(i))
    print ' '.join(featureNames)
    for pair in pairs:
        updateBrand(pair[0])
        updateBrand(pair[1])
        featureVector = getFeatureVector(pair)

        updateMPN(pair[0])
        updateMPN(pair[1])
        if 'Manufacturer Part Number' in products[pair[0]]:
            if nonAlphaNum_re.sub('', products[pair[0]]['Manufacturer Part
Number'][0]).lower() in nonAlphaNum_re.sub('',
str(products[pair[1]].values())).lower():
                featureVector[4*idx_MPN + 0] = '1.000'
                featureVector[4*idx_MPN + 1] = '1.000'
                featureVector[4*idx_MPN + 2] = '1.000'
                featureVector[4*idx_MPN + 3] = '1.000'
        elif 'Manufacturer Part Number' in products[pair[1]]:
            if nonAlphaNum_re.sub('', products[pair[1]]['Manufacturer Part
Number'][0]).lower() in nonAlphaNum_re.sub('',
str(products[pair[0]].values())).lower():
                featureVector[4*idx_MPN + 0] = '1.000'
                featureVector[4*idx_MPN + 1] = '1.000'
                featureVector[4*idx_MPN + 2] = '1.000'
                featureVector[4*idx_MPN + 3] = '1.000'

        if 'Actual Color' in products[pair[0]]:
            if nonAlphaNum_re.sub('', products[pair[0]]['Actual
Color'][0]).lower() in nonAlphaNum_re.sub('',
str(products[pair[1]].values())).lower():
                featureVector[4*idx_AC + 0] = '1.000'
                featureVector[4*idx_AC + 1] = '1.000'
                featureVector[4*idx_AC + 2] = '1.000'
                featureVector[4*idx_AC + 3] = '1.000'
        elif 'Actual Color' in products[pair[1]]:
            if nonAlphaNum_re.sub('', products[pair[1]]['Actual
Color'][0]).lower() in nonAlphaNum_re.sub('',
str(products[pair[0]].values())).lower():
                featureVector[4*idx_AC + 0] = '1.000'
                featureVector[4*idx_AC + 1] = '1.000'
                featureVector[4*idx_AC + 2] = '1.000'
                featureVector[4*idx_AC + 3] = '1.000'

        print ' '.join(featureVector)
    return

if __name__ == '__main__':
```

```
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print >> sys.stderr, 'Usage: python ' + sys.argv[0] + ' <input>
[<num_attributes>]'
        exit()
    global IS_DEBUG
    IS_DEBUG = False
    main()
```

Appendix-B: Code for the Matcher

```
import os
import sys
import re
import json
import numpy as np
from sklearn.cross_validation import train_test_split
from sklearn.ensemble import RandomForestClassifier
from py_stringmatching import simfunctions, tokenizers

#Global flag variables
IS_VALIDATION = False
POS_THRESH = 0.55
MIN_KEY_LEN = 5
PARTIAL_KEY = False

if IS_VALIDATION:
    if len(sys.argv) != 3:
        print 'Invalid number of arguments.'
        print 'Usage: python ' + sys.argv[0] + ' <train_file>
<json_train_file>'
        exit()
else:
    if len(sys.argv) != 4:
        print 'Invalid number of arguments.'
        print 'Usage: python ' + sys.argv[0] + ' <train_file> <test_file>
<json_test_file>'
        exit()

with open(sys.argv[1], 'r') as f:
    length = len(f.readline().strip().split(' '))

#Output filename
filename = sys.argv[0][:sys.argv[0].rfind('.')].replace('_', '') + '_' +
sys.argv[1][:sys.argv[1].rfind('.')]
for i in range(2, len(sys.argv)):
    filename = filename + '_' + sys.argv[i]
filename = filename + '.txt'

if IS_VALIDATION:
    #Read
```

```
    _data = np.genfromtxt(sys.argv[1], dtype=float, delimiter=' ',
skip_header=1, usecols=range(1, length))
    #Seperate features from labels
    X_data = _data[:, range(0, _data.shape[1]-1)]
    y_data = _data[:, _data.shape[1]-1]

    #Set random_state for testing
    X_train, X_test, y_train, y_test, i_train, i_test =
train_test_split(X_data, y_data, range(0, len(y_data)), test_size=0.20,
random_state=42)
else:
    #Read
    _data = np.genfromtxt(sys.argv[1], dtype=float, delimiter=' ',
skip_header=1, usecols=range(1, length))
    #Seperate features from labels
    X_train = _data[:, range(0, _data.shape[1]-1)]
    y_train = _data[:, _data.shape[1]-1]
    i_train = range(0, len(y_train))

    #Read
    X_test = np.genfromtxt(sys.argv[2], dtype=float, delimiter=' ',
skip_header=1, usecols=range(1, length-1))
    i_test = range(0, X_test.shape[0])
#ENDIF

classifier = RandomForestClassifier(n_estimators=100, n_jobs=-1,
criterion='entropy')

classifier.fit(X_train, y_train)

old_stdout = sys.stdout
sys.stdout = open(os.devnull, 'w')
results = classifier.predict(X_test) #May produce junk output
results_proba = classifier.predict_proba(X_test) #May produce junk output
sys.stdout.close()
sys.stdout = old_stdout

for idx in range(0, len(results)):
    if results_proba[idx][1] < POS_THRESH:
        results[idx] = 0

global products, pairs, pair_ids, actual
products = dict()
pairs = list()
pair_ids = list()
actual = dict()
with open(sys.argv[-1], 'r') as fr:
    for line in fr:
        line = unicode(line, errors='ignore') #For character which are not
utf-8
        data = line.strip().split('?')
```

```
        pairId = data[0]
        prod1_id = data[1] + '_1'
        if (data[2]):
            prod1_json = json.loads(data[2])
        else:
            prod1_json = dict()
        prod2_id = data[3] + '_2'
        if (data[4]):
            prod2_json = json.loads(data[4])
        else:
            prod2_json = dict()
        if len(data) == 6:
            label = data[5]
        else:
            label = 'UNKNOWN'

        products.update({prod1_id:prod1_json})
        products.update({prod2_id:prod2_json})
        pairs.append((prod1_id, prod2_id))
        pair_ids.append(pairId)
        actual.update({pairId:label})
    fr.close()

symbol_re = re.compile('[^A-Z\d\s]+', re.IGNORECASE)
nonNum_re = re.compile('\D+', re.IGNORECASE)
withoutNum_re = re.compile('\b\D+\b', re.IGNORECASE)
nonWord_re = re.compile('\W+', re.IGNORECASE)
html_re = re.compile('((<[^<]*?>)|( ))+', re.IGNORECASE)

labels = dict()
labels.update({1:'MATCH'})
labels.update({0:'MISMATCH'})

def setMatchNumKeyField(pair, fields):
    data1 = nonNum_re.sub(' ', str(products[pair[1]])).strip().split()
    if data1:
        for d1 in data1:
            if len(d1) >= MIN_KEY_LEN:
                if PARTIAL_KEY:
                    data2 = tokenizers.qgram(d1, MIN_KEY_LEN)
                else:
                    data2 = tokenizers.qgram(d1, len(d1))
                for d2 in data2:
                    for field in fields:
                        try:
                            if (d2 in products[pair[0]][field][0]):
                                return 1
                        except:
                            continue
    return 0
```

```
def MatchMPN2PN(pair, probab):
    if 'Manufacturer Part Number' in products[pair[0]]:
        MPN = symbol_re.sub('', products[pair[0]]['Manufacturer Part
Number'][0])
        PN = products[pair[1]]['Product Name'][0].strip().split()
        for pn in PN:
            if ((MPN in pn) or (pn in MPN)) and
(simfunctions.levenshtein(MPN, pn) <= 1):
                #print '>', '-'.join(pair), actual['-'.join(pair)], probab
                return 1
    if 'Manufacturer Part Number' in products[pair[1]]:
        MPN = symbol_re.sub('', products[pair[1]]['Manufacturer Part
Number'][0])
        PN = products[pair[0]]['Product Name'][0].strip().split()
        for pn in PN:
            if ((MPN in pn) or (pn in MPN)) and
(simfunctions.levenshtein(MPN, pn) <= 1):
                #print '>', '-'.join(pair), actual['-'.join(pair)], probab
                return 1
    return 0

def NewRefurbished(pair, probab):
    if 'refurbished' not in products[pair[0]]['Product Name'][0].lower():
        if 'refurbished' in products[pair[1]]['Product Name'][0].lower():
            #print '<', '-'.join(pair), actual['-'.join(pair)], probab
            return 0
    if 'refurbished' not in products[pair[1]]['Product Name'][0].lower():
        if 'refurbished' in products[pair[0]]['Product Name'][0].lower():
            #print '<', '-'.join(pair), actual['-'.join(pair)], probab
            return 0
    return 1

def matchWord(pair, field):
    if field in products[pair[0]] and field in products[pair[1]]:
        if nonWord_re.sub('', products[pair[0]][field][0]) ==
nonWord_re.sub('', products[pair[1]][field][0]):
            return True
        return False
    return True

def lessAttributes(pair):
    if (len(products[pair[1]]) < len(products[pair[0]])) and
(len(products[pair[1]]) <= 5):
        set1 = set()
        set2 = set()
        for f in products[pair[1]]:
            list1 = symbol_re.sub(' ', html_re.sub(' ',
products[pair[0]][f][0])).strip().split()
            list2 = symbol_re.sub(' ', html_re.sub(' ',
products[pair[1]][f][0])).strip().split()
```

```
            if list1 and list2:
                set1.update(list1)
                set2.update(list2)
        sim_score = simfunctions.overlap_coefficient(set1, set2)
        if sim_score == 1:
            return 1
    return 0

f = open('predict1_' + sys.argv[-2], 'w')
for idx, i in enumerate(i_test):
    f.write(pair_ids[i] + ', ' + labels[results[idx]] + '\n')
f.close()

f = open('predict2_' + sys.argv[-2], 'w')
for idx, i in enumerate(i_test):
    if results[idx] == 0:
        results[idx] = setMatchNumKeyField(pairs[i], ['GTIN', 'UPC'])
    if results[idx] == 0:
        results[idx] = setMatchNumKeyField(pairs[i][::-1], ['GTIN', 'UPC'])
    if results[idx] == 0 and results_proba[idx][1] >= 0.2:
        results[idx] = MatchMPN2PN(pairs[i], results_proba[idx])
    if results[idx] == 1 and results_proba[idx][0] >= 0.2:
        results[idx] = NewRefurbished(pairs[i], results_proba[idx])
    if results[idx] == 0 and results_proba[idx][1] >= 0.45:
        if matchWord(pairs[i], 'Product Long Description'):
            results[idx] = 1
    f.write(pair_ids[i] + ', ' + labels[results[idx]] + '\n')
f.close()


f = open('probab_' + sys.argv[-2], 'w')
for idx, i in enumerate(i_test):
    f.write(' '.join([str(t) for t in results_proba[idx]]) + '\n')
f.close()
```

V. Bonus: Write-up on the py_stringmatching Package

Pros

We used the py_stringmatching package in our project and found it to have various sensible features:

- Similarity functions are easy to use. We didn't have to write much boilerplate code for this which increased our efficiency and reduced the development time to some extent.
- The examples are easy to understand and gave us a very good quick start. It also helped us understand new similarity measures which we were not aware of like, Monge-elkan.
- We also checked out the code on GitHub and it helped us understand the intricate details of how it is actually implemented.
- Tokenizers, which did the preprocessing of the tuples by converting the attributes in n-grams really saved our time in implementing it from scratch.

Cons

- Documentation
    - Specific cases for similarity measures describing which one should be used where, can help the user in choosing the functions more judiciously.
    - Though the documentation is easy and does the job quite well, we still see some scope of adding more details about the sim measures. E.g.: A brief explanation of the scoring measure or citing a relevant reference.
    - It would be easy to search if there is a proper heading for each similarity measure in the documentation, so that they be directly linked to in the side-bar.
- Code
    - Have each of the sim measures overloaded, so as to perform operation on lists as well and return the element wise sim measure in a list.
        - Current Implementation : <float> some_sim(string, string)
        - Our Suggestion : <List<float>> some_sim(List<string>, List<string>)

- We also had to do a lot of pre-processing specifically regarding case-sensitiveness and special characters. The library can be extended to handle this gracefully by giving the user an optional argument of whether he wants it to be case-sensitive versus insensitive and to remove special characters within.
  - Current Implementation : <float> some_sim(string, string)
  - Our Suggestion : <float> some_sim(string, string, ignorecase=False, removeSpecialChar=False)

Other

Additional similarity measures may be added. Eg: Tversky Index, Sorensen Dice coefficient, etc.