Entity Matching for Online Marketplaces

Applying Machine Learning to Product Matching

Kyle Gilde

5/14/2019

Abstract: This paper explores the intersection of entity matching, machine learning and online marketplaces. Entity matching, or finding references that point to the same real-world object, has existed as a field for more than half a century. The field of machine learning classification began about twenty-five years ago and has seen breakthroughs recently with processing natural language. In the last ten years, online multi-vendor marketplaces have proliferated. To maximize the benefits of sellers competing for customers, the marketplace firm must be able to consolidate the product offers in its catalog that belong to the same real-world product. This paper addresses the following questions: 1. To what degree can machine learning algorithms solve the entity matching problem for online marketplaces? 2. Which types of feature representations of text produce the best classification results when the offer catalog is rife with missing values? The results demonstrate that machine learning classifiers can detect offer *matches* and *non-matches*, but exhibit the tradeoff between precision and recall. Additionally, despite the sparsity in the dataset, an attribute comparison approach to feature representation proves superior to the single-document representation.

Key Words: Entity Matching, Product Matching, Machine Learning, Natural Language Processing

1 Introduction

In the last ten years, the proportion of e-commerce sales within the U.S. retail industry has more than doubled from less than 4% to nearly 10%[1] and is projected to continue in the years to come.[2] Within the e-commerce trend, a growing number of companies have adopted a multi-seller virtual marketplace model. This model allows retailers to vastly expand the

[1] U.S. Census Bureau News (2018, November 19) QUARTERLY RETAIL E-COMMERCE SALES *Census.gov*. Retrieved 10 February 2019, from https://www.census.gov/retail/mrts/www/data/pdf/ec_current.pdf

[2] Keyes, D. (2017, August 11) *E-Commerce will make up 17% of all US retail sales by 2022 – and one company is the main reason*. *Business Insider*. Retrieved 10 February 2019, from https://www.businessinsider.com/e-commerce-retail-sales-2022-amazon-2017-8

assortment of their virtual inventory without the costs associated with holding physical inventory.[3] Additionally, online retailers can leverage marketplace competition to create a better customer experience and increase sales conversion. Instead of having offers for the same product appear as separate product listings, the virtual marketplace can decrease customers' cognitive burden by identifying these offers as the same product and consolidating them on to a single web page. Juxtaposing these multiple offers stimulates vendor competition, and customers benefit by getting lower prices and better shipping service. The business hosting the online marketplace gains increased sales conversion, customer loyalty, and a wealth of information about both consumer and seller behaviors. In 2016, half of global online retail sales occurred on multi-seller virtual marketplaces, and this proportion is forecasted to rise to two-thirds by 2021.[4]

Before maximizing the benefits of marketplace competition, online retailers must devise a method for determining whether offers from different sellers refer to the same real-world product. Since the advent of electronic databases, a number of entity matching (EM) techniques have been developed to accomplish this information retrieval task that Mudgal et al. (2018) describe as "[finding] data instances that refer to the same real-world entity." Initially, EM techniques were a means for researchers to create new datasets by combining different sources of data. In the age of Internet, EM methodology falls within the domains of natural language processing and machine learning (ML). While reviewing the preceding practices, this paper will focus on which ML techniques can most effectively match instances of text.

2 Literature Review

Throughout its practice, EM, which has also been known as entity resolution, record linkage and duplicate detection, aims to solve the problem of what Elmagarmid et al. (2007) describe as *lexical heterogeneity*, i.e. textual variations among references for the same referent. The sources of these variations are twofold. First, instances of data vary in quality and format, which is caused by the absence of standardized conventions, incomplete information and transcription errors (Elmagarmid et al., 2007). Secondly, human language allows for the same information to be communicated in a multitude of ways. Vastly differing written symbols can be used to express the same idea. The process of EM can be represented as three tasks: blocking, feature representation and classification.

---

[3] Rigg, O. (2018, December 13) *The Pros And Cons Of The Marketplace Model For E-Commerce*. *Forbes.com*. Retrieved 10 February 2019, from https://www.forbes.com/sites/forbesnycouncil/2018/12/13/the-pros-and-cons-of-the-marketplace-model-for-e-commerce/#feb71485935d

[4] Howland, D. (2017, September 14) *Forrester: Half of online sales occur on marketplaces*. (2017). *Retail Dive*. Retrieved 10 February 2019, from https://www.retaildive.com/news/forrester-half-of-online-sales-occur-on-marketplaces/504913/

2.1 Blocking

Blocking is a technique to reduce search space to a computationally-manageable size (Kopcke and Rahm, 2009). In blocking, the values in one or more attributes are employed to to partition the data instances into smaller groups, i.e. blocks, before generating all of the possible document pairs. Blocking is a useful first step for any corpus of moderate-to-large size because the number of reference pairs increases exponentially. For a corpus of *n* documents, the number of pairwise combinations is represented by $n(n-1)/2$. Therefore, a corpus of ten thousand references generates nearly fifty million pairs, and and a corpus of one hundred thousand yields nearly five billion pairs. While facilitating greater efficiencies in the steps to follow, the downside of blocking is that segregating the documents by attribute values that are too particular may preclude legitimate matches from consideration.

2.2 Feature Representation

Feature representation generally involves three parts: text normalization, encoding and comparison. Text normalization, the process of transforming the text into a standard format, can reduce lexical heterogeneity. It effectively eliminates variations caused by formatting differences, but only marginally mitigates variation from different symbolic representations of the same meaning. The Russell soundex system was one of the first methods of normalizing text and was used by Newcombe et al. (1959), who were among the earliest of EM practitioners. Using punch-card records and computers, they attempted to match the parental names on birth certificates to the names on marriage records by encoding the names as phonetic representations.

Stemming is a text normalization technique that truncates words to their roots by removing their suffixes, which is another way to minimize symbolic differences in the text. Porter (1980) used the example of stemming the words *connected*, *connecting*, *connection* and *connections* to the root word *connect*. This method reduces some of the symbolic disparities so that string-based and token-based similarity approaches can recognize the shared meaning.

Encoding transforms text into numeric representations. A widely adopted encoding method for paragraph-length text is the token-based bag-of-words model. It encodes a corpus as a sparse matrix of term frequencies with the rows representing the documents and the columns representing the terms. Once encoded, a distance function like cosine similarity or Euclidean can be used to compare the texts. However, as its name implies, the drawbacks of the bag-of-words model include that it does not account for the order of words or for the shared meaning of the different symbols (Mikolov and Le, 2014).

An alternative to the BOW, which skips the encoding step, is to use an edit distance function to measure the character similarity between short- to sentence-length strings. For instance, the Levenshtein function measures the number of character deletions, insertions and substitutions required to make one string match another (Elmagarmid 2007). The limitation of these string-based approaches is that they may report large textual dissimilarity between a pair of data instances that human being would interpret as being synonymous. For example, a human being understands that the terms HP and Hewlett Packard represent the same company. However, a scaled Levenshtein similar function finds that these strings have a match rate of only 13%.

2.3 Classification

The final step of EM is classification, which is used to determine whether two texts are matches or not. In the twentieth century, EM practitioners like Newcombe et al. (1959) and Fellegi and Sunter (1969) accomplished classification by combining string-similarity approaches with probabilistic Naive Bayes models and decision-rule frameworks. More recently, ML approaches to EM view the task as a applying classification algorithms to pairs of references that are either *matches* or *not-matches*.

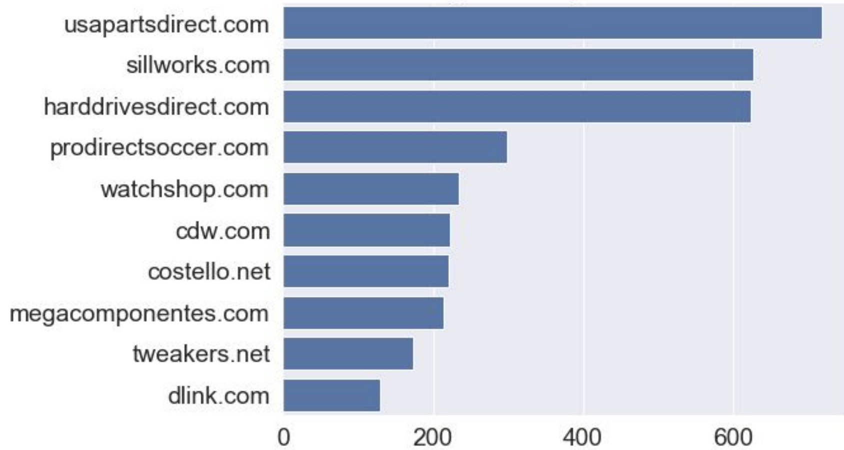
Chollet (2017) posits the origin of the current era of ML development originated in the 1990s with the development of the kernel-based support vector machine (SVM) algorithm, which employs a computationally efficient method to find good decision boundaries in classification problems. In the 2000s, Breiman (2001) formulated the Random Forest (RF) algorithm, which surpassed SVM performance. In recent years, the stochastic gradient boosting machines (SGB) algorithm, proposed by Friedman (2001), has surpassed the performance and popularity of RF. Both RF and SGB rely on the ML concepts of ensembling and bagging. Bagging is short for bootstrap aggregation, and it refers to the process of selecting a random subset of the training observations for each tree model. Ensembling is a technique that creates several weak models and combines them to yield better predictions. Both techniques produce models with reduced variances and less overfitting to the training data. The primary difference between the two algorithms is that in RF the tree models are built independently of each other while they are built successively in SGB. In SGB, after the first tree model is fit, each successive model is fit to the gradient, i.e. residuals, of the previous model, which produces iteratively smaller gradients.

3 Data Exploration

The dataset for this paper is the *WDC Training Dataset and Gold Standard for Large-Scale Product Matching* (LSPM), which was created from the Common Crawl web-data corpus and published in December 2018 by a group of researchers at the University of Mannheim.[5] Their goal was to create an entity-matching benchmark dataset with a high degree of heterogeneity and a large enough sample size to train word embeddings and deep-learning matchers. In total, it consists of sixteen million English-language product offers from forty-three thousand websites.

[5] http://webdatacommons.org/largescaleproductcorpus/index.html

Figure 1: Top Domains



The subset of offers in the training and test datasets originate from 471 domains, and the ten most frequent are shown in Figure 1. This sort of large, amalgamated offer catalog is an apt approximation of the multi-vendor offers a business would collect to start an online marketplace.

3.1 Offer Features

After parsing the millions of JSON of arrays, the dataset yields nine attributes that could be considered as offer features. However, as Figure 2 shows, six of them are missing the majority of their values. The fill rates for the primary sources of text, the offer *name* and *description*, are the two highest at near 100% and 75%, respectively. This challenging level of sparsity resembles the inconsistent quantity and quality of information submitted to online marketplaces.

Figure 2. % Missing Values

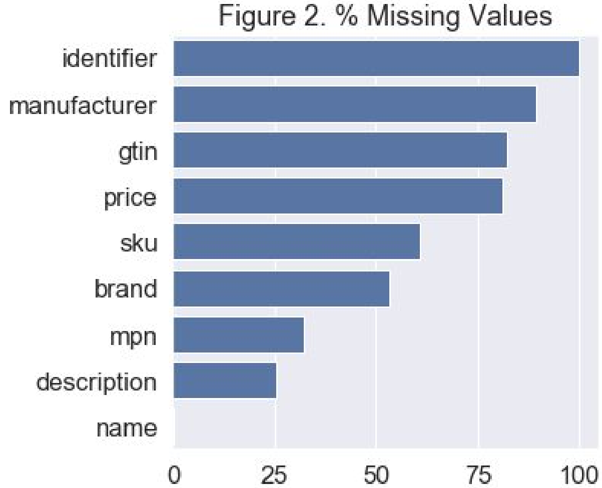
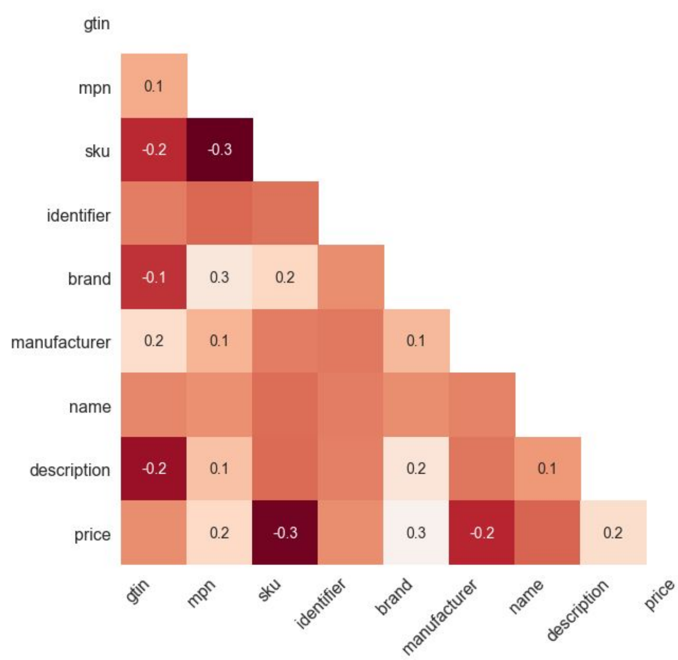


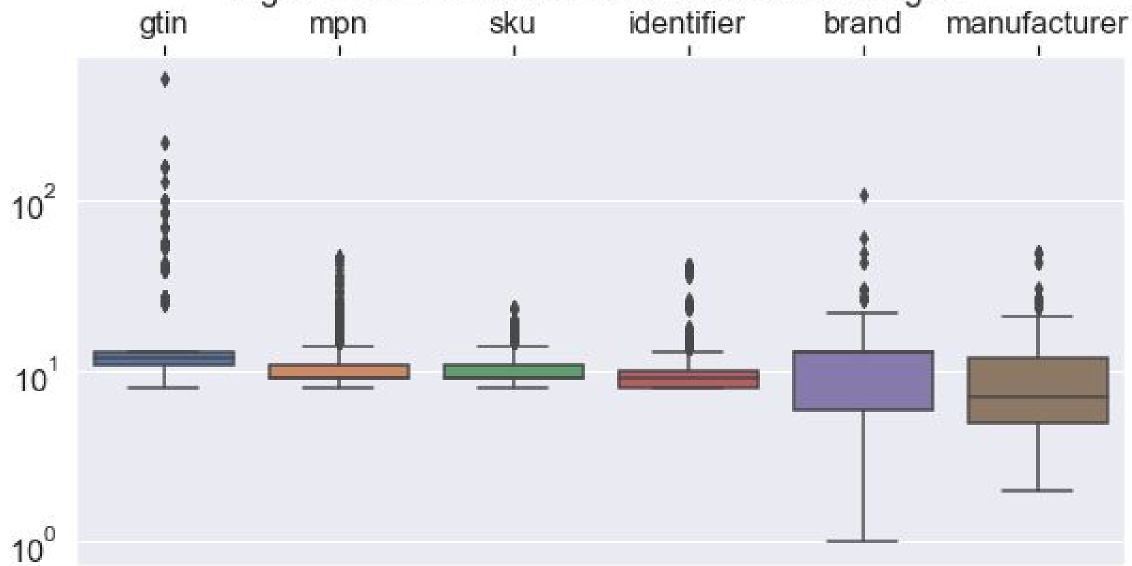
Figure 3 shows that there is little correlation between the missing values of the variables. This indicates that the data points are likely missing completely at random and not a source of bias. The cells in Figure 3 without numbers have correlations near zero and comprise more than half of the variable combinations. The highest correlations are only positive and negative 0.3.

Figure 3. Missing Value Correlation



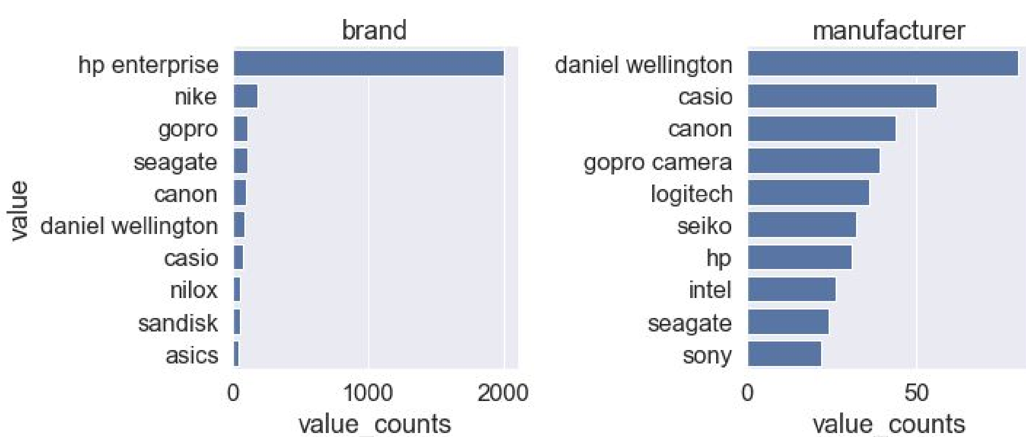
Apart from the price, the dataset contains eight textual attributes, and there are three broad feature types. First, there are six atomic text features for which edit distance functions can be used to measure the similarity. Four of these are alphanumeric product identifiers with a median length of around ten characters (Fig. 4). If used consistently, the *gtin* (global trade item number), *mpn* (manufacturer part number), *sku* (stock-keeping unit) and a generic *identifier* can definitively identify products. However, because the majority of these values are missing, they are likely to not be as determinative.

Figure 4. Atomic Text Feature Character Lengths



The other two atomic features are the *brand* and *manufacturer*. While having similar median character lengths as the gtin, mpn, sku, and identifier attributes, the brand and manufacturer attributes have wider interquartile ranges with greater character length variance. Figure 5. shows the ten most frequent values and how these variables share much of the same information. In fact, when both attributes have a value, these values match in 75% of the instances.

Figure 5. Most Frequent Values



The second and third types of textual features are the offer *name* and offer *description* features (Fig 6.). With a median of eight tokens, the *name* feature is approximately the length of a sentence, and this length is still short enough for edit distance functions. With a median of twenty-three words and a much greater variability in word counts, the paragraph-length *description* is an ideal candidate for the bag-of-words approach to measuring similarity. These attributes are the primary sources of textual features.

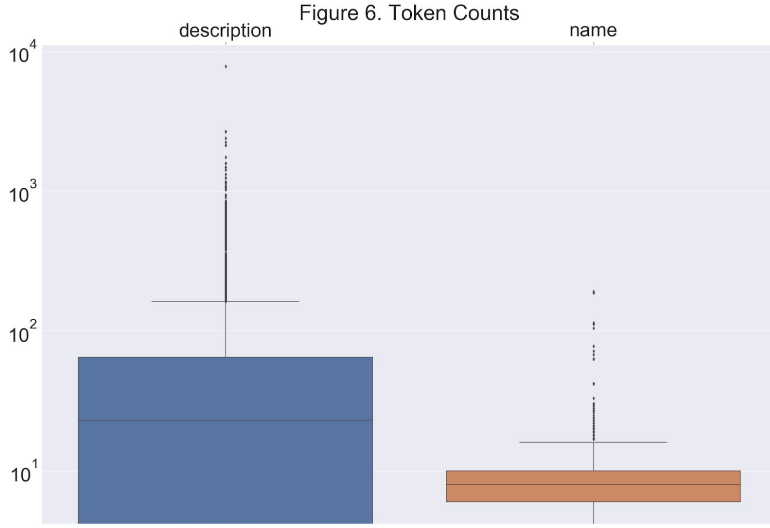


Figure 6. Token Counts

3.1 Training & Test Dataset

With 82,000 offer pairs for training and 2,100 for testing, the size of Mannheim group's training data is considerably larger than many other publicly available hand-labeled data (Fig. 7). While the training set is composed of less than 8,000 unique offers, the testing set contains a significantly larger proportion of unique offers with 2,000 (Fig. 8).

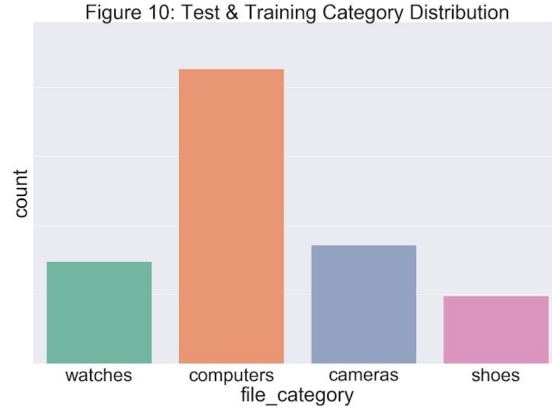
Figure 7: Number of Training & Test Samples

Figure 8: Number of Unique Offers

The *match* (1) and *non-match* (0) classes are slightly imbalanced with the *match* labels accounting for 45% of the test and training samples (Fig. 9). Additionally, the offer pairs originate from four categories of products (Fig. 10). *Computers and accessories* has more than twice the samples as the next largest category *cameras*, followed by watches and then shoes.

Figure 9: Test & Training Class Distribution

Figure 10: Test & Training Category Distribution



However, the composition of the test dataset is different. Figure 11 shows that the class imbalance is larger with the *match* labels comprising only 30% of the samples, and in Figure 12, the offer-pair observations are uniformly distributed between the four categories. Ideally, the test and training sets would have more similar compositions, but the differences are not prohibitive.

Characteristics of the Testing Dataset

Figure 11: Class Distribution

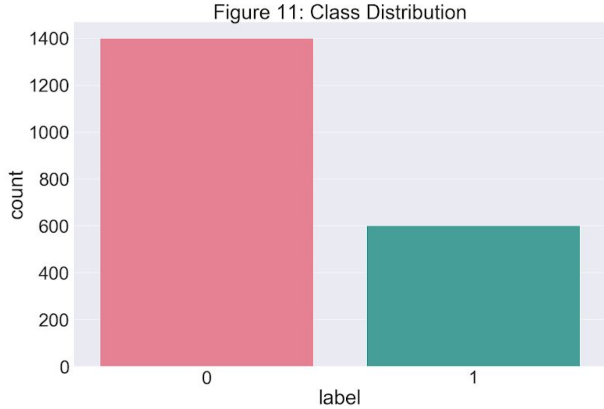
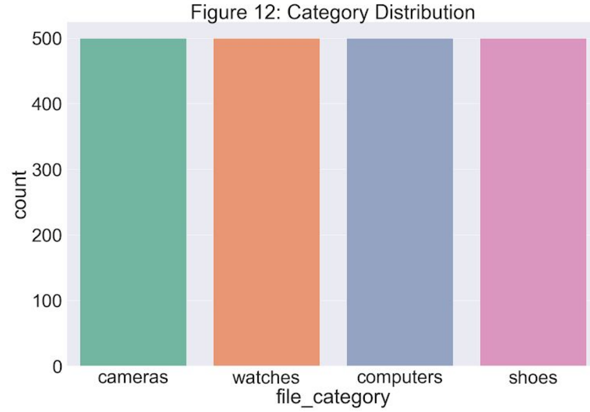


Figure 12: Category Distribution



3.2 Offer Pairs

Table 1 provides an example of a *matched* offer pair from each category. They demonstrate the type of lexical heterogeneity that the feature engineering and modeling must overcome. Specifically, we can see instances where string-based and token-based feature representations may not be effective. For example, the edit distance and BOW approaches would perform well in identifying the similarity between the computer offers; the two sequences of tokens are the same but for a handful of characters. However, the symbolic approach will not perform as well in matching the camera offers in the last row. While the beginnings of the names match exactly, edit distance functions and BOW vectors would view these instances as being of low similarity because the second name contains a much longer description of the item.

Table 1: Pairs of Offer Names

file_category	name_1	name_2
shoes	nike sportswear air force 1 07 white	sneakers buty nike air force 1 07 low white 315122 111
computers	323146 b21 bl20p g2 1p xeon 3 06ghz	323146 b21 bl20p xeon 3 06ghz
watches	daniel wellington dapper sheffield rose gold	daniel wellington men s dapper 38mm sheffield rose watch
cameras	canon eos rebel t5i	canon eos rebel t5i 18 135mm is stm digital slr camera kit black

4 Methodology

I applied the EM process of feature representation and classification to the LSPM product offers; blocking was unnecessary because the offers in the dataset are already blocked by category. To normalize the textual attributes, I applied the following framework:

- Converted the letters to lowercase
- Removed the HTML tags
- Removed the characters that were not either alphanumeric or a single space
- Removed the stopwords, the most common words in a language. (They comprise a disproportionate amount of the text, but add relatively little to its meaning. Removing them is a feature selection method so the models can prioritize the tokens that disproportionately contribute to a text's meaning.)
- Stemmed the *name* and *description* with the Snowball algorithm (Porter, 1980).

This normalized dataset served as the input to the other features sets, and I applied two feature representation strategies, which I am describing as *attribute comparisons* and *single document comparisons* (Table 2).

Table 2: Feature Sets

Type	Number of Features	% Variance Explained
Attribute Comparison	7	-
Attribute Comparison	9	-
Single-Doc Comparison	9	6.2%
Single-Doc Comparison	100	25.1%

4.1 Attribute-Comparison Features

The similarity between the corresponding attribute values of the offer pairs was measured to create the attribute comparison feature set. In the resulting dataset, the rows represent the similarity measures for the attribute pairs, whereas the columns contain the output of a similarity function applied to the corresponding attribute values. Three similarity functions were employed to create the similarity values. The *price* similarity was calculated by the absolute difference scaled by the larger value. Next, a scaled Levenshtein distance function created the similarity values for the short-to-medium length text attributes, including the offer *name*, *brand*, *manufacturer*, *gtin*, *mpn*, *sku* and *identifier*.

Lastly, since using the Levenshtein function would be computationally expensive, the similarity for the large *description* attribute was calculated with BOW, truncated singular value decomposition (SVD) and cosine similarity. BOW was employed to embed the text as numeric vectors. Instead of using the term frequencies, I used a term-frequency inverse-document-frequency (TF-IDF) model. Multiplying the term frequency by the natural logarithm of the inverse document frequency allows for each term-document combination to represent how important the term is to both the document and the corpus as a whole. If a term appears in nearly every document, then its discriminatory power within a corpus is small, and the IDF will be near zero. However, if the word is less common within the corpus, then its discriminatory power is greater and IDF multiplier is larger (Ramos 2003). Additionally, to compensate for the unordered nature of BOW, I parsed the tokens into unigrams, bigrams and trigrams. The resulting matrix contained tens of thousands columns representing each of the unique one-to-three token permutations. Truncated SVD was used to reduce the dimensionality; selecting the top 3,000 components explained 99% of the variance. Next, a cosine similarity function was used to calculate the similarity value. The outcome of this feature engineering was a nine-column set of features containing the similarity values between zero and one for each of attributes shared between each set of offer pairs. If either of the attributes values were missing, the similarity value was set to zero.

4.2 Single-Document Comparison Features

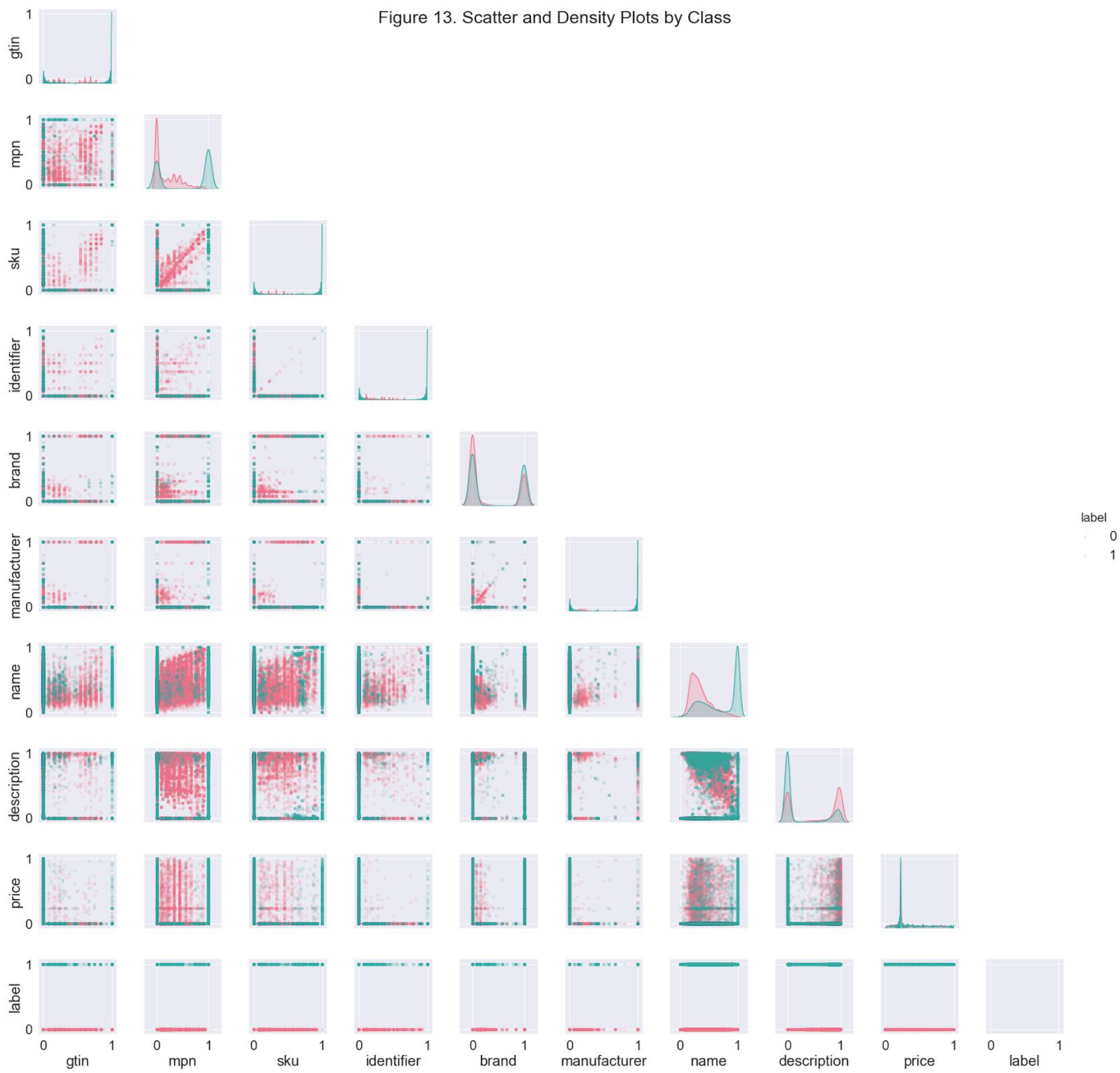
Based upon the approaches presented in Ebraheem et al. (2018) and Shah et. al (2018), the second feature strategy treats each set of offer attributes as a single document by concatenating the values. I theorized that this approach might minimize the effect of the missingness and create a dense set of non-collinear features. To the newly-created string, I then applied the same BOW model using TF-IDF and unigrams, bigrams and trigrams, followed by truncated SVD. I used truncated SVD instead of principal components analysis because it performs better on the sparse matrices created by BOW models.[6] Then, to compare the offer pairs, I calculated the absolute difference between the offer vectors. I created two different versions of the feature set. The first version contained only the nine most informative features from the truncated SVD, which explained only 6.2% of the feature variance. The model results will reveal whether the nine features performed better or worse than the nine features in the attribute-comparison feature set. The second version contained the 100 most informative features and explained 25% of the variance.

[6] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html

5 Feature Analysis

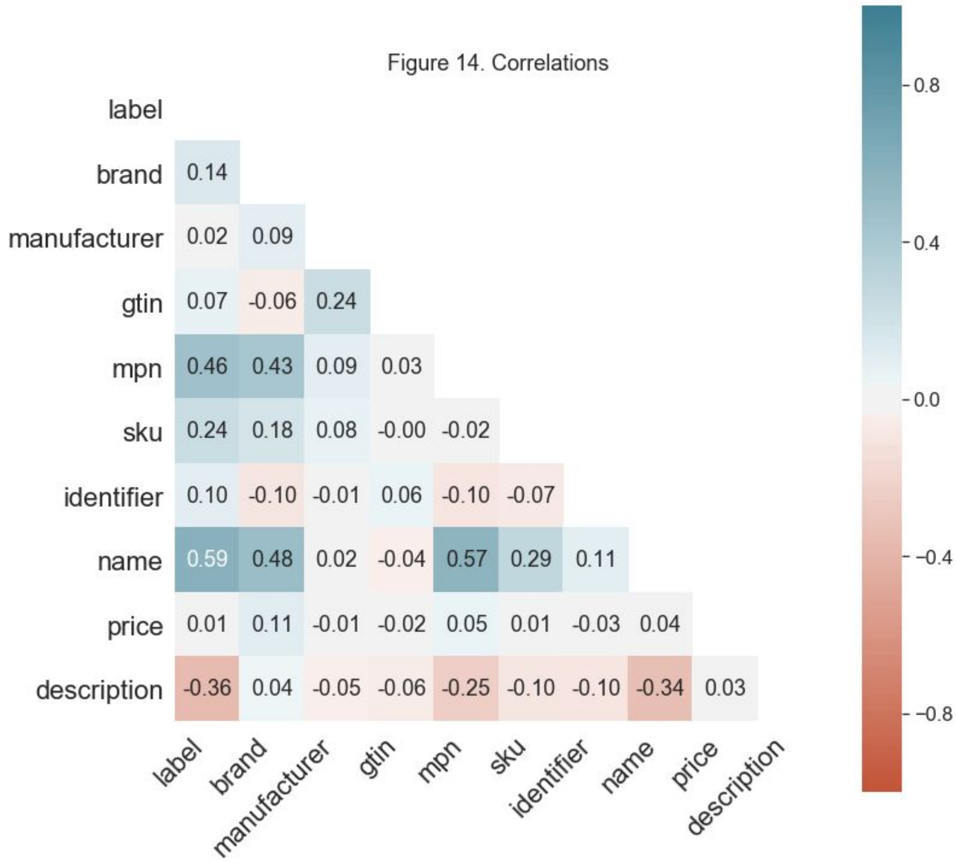
The next series of plots examines the relationships among the predictor variables and between the predictors and the response variable *label* in the attribute-comparison feature set. Figure 13 shows the scatter and density plots with *match* and *not-match* labels represented in green and red. While there appears to be an unexpected inverse relationship between the *name* and *description*, the offer *name* has slight correlations with about five of the other features. In the density plots for the *name, mpn, sku, identifier* and *manufacturer*, the *match* labels are more concentrated around the higher similarity values. These may be important to training the model. However, the density plots for the *description* and *price* show that the *matches* are clustered around the lower similarity values, which indicates that these will likely not be important.

Figure 13. Scatter and Density Plots by Class



5.1 Correlation

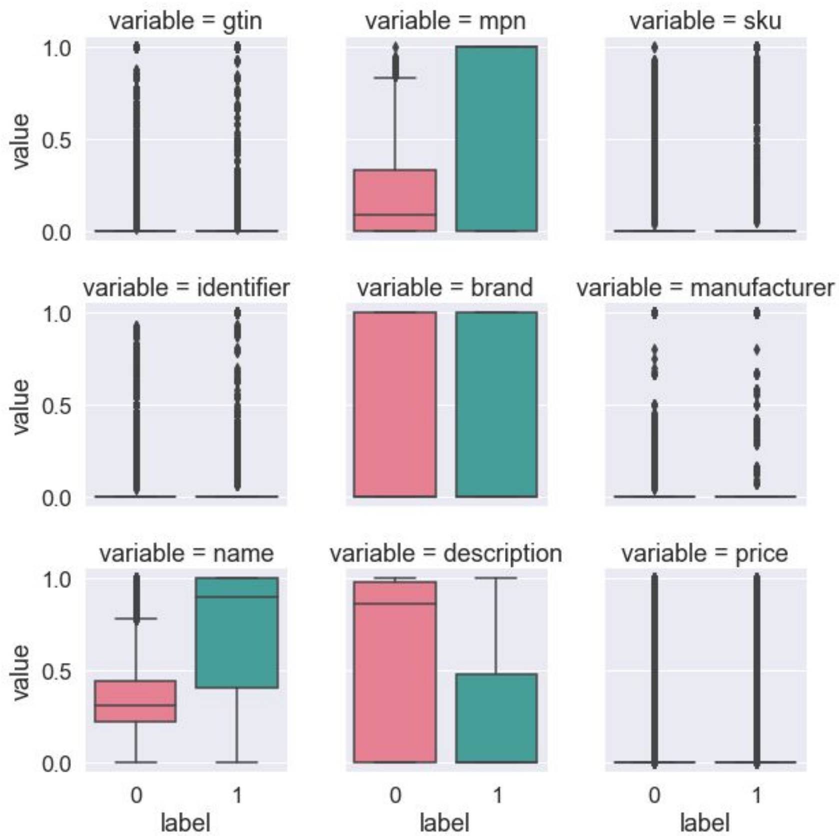
Figure 14 shows that there is only a moderate amount of correlation between some of the similarity variables. The highest correlation is between the *name* and *label* features, which indicates that the name is likely to be important. The *mpn* has the second highest correlation with the response variable and is also moderately correlated with the *name*. This plot confirms that the similarity values for the *description* have a small negative correlation with the *name*. Additionally, the *description* has a small negative correlation with the response variable.



5.2 Boxplots by Class

Figure 15 shows how the similarity values are distributed across the classes. Most of the plots are uninformative because the variables with many missing values are zero-inflated, and their interquartile ranges are centered at zero. However, for the *name*, *description* and *mpn* features, the plots do offer some insights. The nearly-separated interquartile ranges for the *name* indicate that this will be an important feature. The *match* median similarity value is greater than .8 and the *not-match* median is less than .4. Additionally, for the *mpn*, the interquartile range for the *not-matches* is associated with smaller similarity values. For the *description*, the plot shows the same inverse relationship with the *matches* and similarity.

Figure 15. Side-by-Side Boxplots



5.3 Logistic Regression

To gain a better understanding of the independent effect of each attribute, an elastic-net logistic regression from the *Caret* package was fit to the 9-feature, attribute-comparison set. The four variables with the most missingness, the *gtin*, *identifier*, *manufacturer* and *price*, had too little variance for the regression assumptions and were removed. Since there was no high correlation among the variables, none of them were removed to meet the non-collinear assumption.

Of the five remaining variables, the *name* was the most important in that its coefficient had the largest absolute value (Fig. 16).[7] The *brand* and *mpn* had a little more than half the importance of the *name*, followed by the *sku* and and ending with the *description*. The importance of the *name*, *brand* and *mpn* have a roughly inverse relationship with the number of missing values. However, while *description* attribute had the second highest fill-rate with fewer than 25% of its values missing, the model found that it had relatively little importance. In fact,

[7] https://rdrr.io/cran/caret/man/varImp.html

Figure 16 shows that the *description* and *brand* attributes actually had a negative coefficients; which means that the higher the similarity values, the less likely the offer pair were a match. Given the findings from Figure 14, a negative coefficient for the *description* is not surprising. Based upon this insight, a 7-feature version of the attribute-comparison features was created to explore whether it performed better than all nine features (Table 2).

Figure 16. Variable Importance

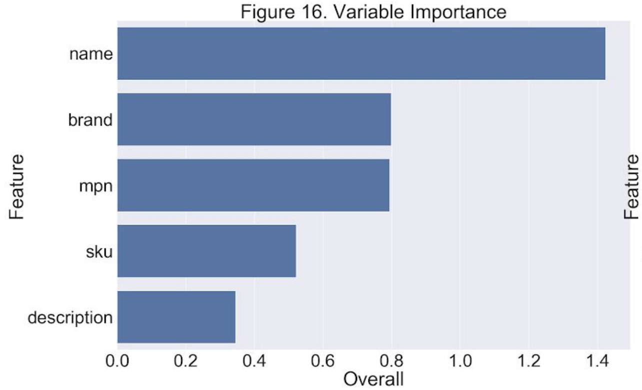
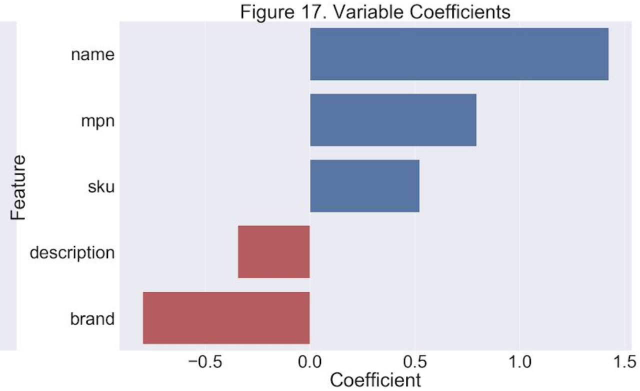


Figure 17. Variable Coefficients



5.4 Description Discrepancy

One possible explanation for the inverse relationship between the *match* labels and the *description* similarity values are vastly different lengths of text between the offer pairs. Table 3 shows two examples where *description_2* contains all of the text in *description_1*, but also contains much more information. Because *description_2* is a two to three times longer and because the similarity values were scaled by the longest text, this will report a lower similarity score.

Table 3: Examples of Different-Length Descriptions

description_1	description_2
long lasting comfort defines the new nike pegasus 33 which has a new finely tuned outsole that delivers a smooth snappy ride a cushlon midsole provides soft and responsive cushioning while its engineered mesh upper provides lightweight and comfortable support and ventilation as well as impact absorbing cuts to the crash rail that also enhance grip waffle pistons and a radiused heel all combine to give the shoe its renowned ride	long lasting comfort defines the new nike pegasus 33 which has a new finely tuned outsole that delivers a smooth snappy ride a cushlon midsole provides soft and responsive cushioning while its engineered mesh upper provides lightweight and comfortable support and ventilation as well as impact absorbing cuts to the crash rail that also enhance grip waffle pistons and a radiused heel all combine to give the shoe its renowned ride features include horizontal and vertical cuts in the crash rail enhance grip cushlon midsole provides soft yet responsive cushioning flywire cables deliver the ultimate in lockdown engineered mesh upper provides ventilation and support zoom air units in the forefoot and heel provide low profile responsive cushioning weight 8 6oz 244g women s size 8 offset 10mm last mr 10
these pants feature nike tech fleece fabric that helps keep you warm with a relaxed fit that tapers at the cuff to show off your sneakers	these pants feature nike tech fleece fabric that helps keep you warm with a relaxed fit that tapers at the cuff to show off your sneakers features include nike tech fleece fabric gives you warmth without weight max tapered fit is relaxed on top then tapers near the cuff adjustable straps on the sides of the waistband create a custom fit zippered fly with a button closure offers easy on and off

6 Models

Four types of models were trained and tested to predict whether the offer pairs were matches. To represent the pre-ML probabilistic EM frameworks, a Naive Bayes model was fit as a baseline. The second model was SVM. After testing the linear, radial basis function (RBF) and polynomial kernels, the RBF kernel was found to produce the best F1 score. The third and fourth models were Random Forest (RF) and Stochastic Gradient Boosting (SGB). The latter three models will automate the feature selection. To tune the hyperparameters, I executed cross-validated grid searches. Since SVM requires it, all features sets were centered and scaled. Additionally, for the SVM and RF models, the sci-kit learn package's class-weighting functionality was used to account for the class imbalance.

7 Results

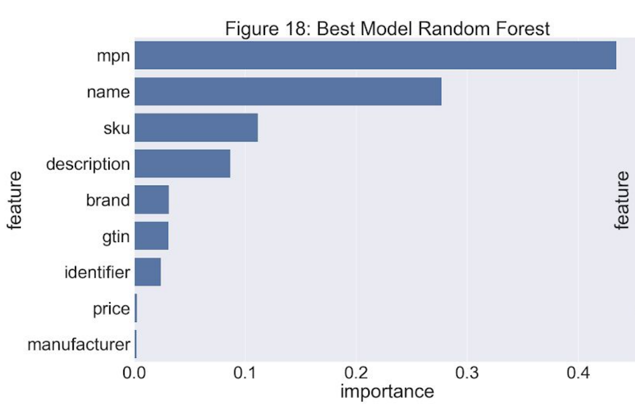
Table 4 contains the test results for the sixteen combinations of models and feature sets. The higher F1 scores are shaded with a darker shade of green. Overall, at 0.798, the RF model and attribute-comparison-9 feature set had the highest F1 score, followed closely by the same feature set with the SGB model. The worst-performing combination was the SVM model with the 9-feature, single-document set.

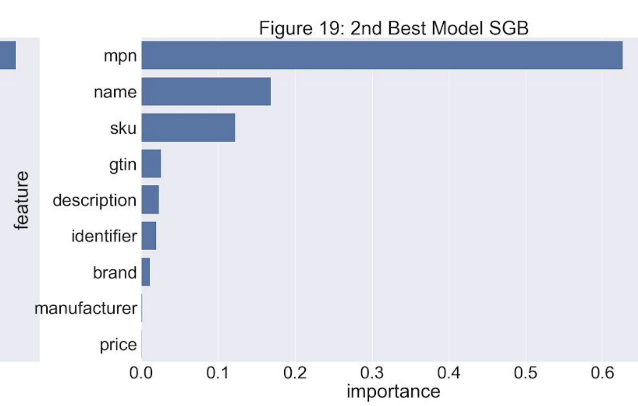
Table 4: Test Results

	Naive Bayes			SVM			Random Forest			Gradient Boosting		
Feature Set	P	R	F1	P	R	F1	P	R	F1	P	R	F1
Attribute Comparison-7	.460	.600	.521	.731	.759	.745	.758	.808	.782	.780	.793	.786
Attribute Comparison-9	.433	.507	.467	.743	.801	.771	.755	.846	.798	.761	.831	.795
Single Doc-9	.274	.911	.422	.276	.858	.418	.707	.552	.620	.698	.545	.612
Single Doc-100	.287	.883	.433	.359	.564	.439	.649	.600	.624	.661	.622	.641

7.1 Feature Importance

Figures 18 and 19 demonstrate an unexpected and significantly different order of variable importance than the initial logistic regression model. The *mpn* attribute was actually the most important feature in the top two models. The online retailers must have been using the *mpn* with consistency even though a value was missing in 30% of the offers. The offer *name* was the second most important feature in both models. In the RF model, it was more than half the importance of the *mpn*. In both models, the *sku* was the third most important features, despite having more null values than the *brand* or *description*.

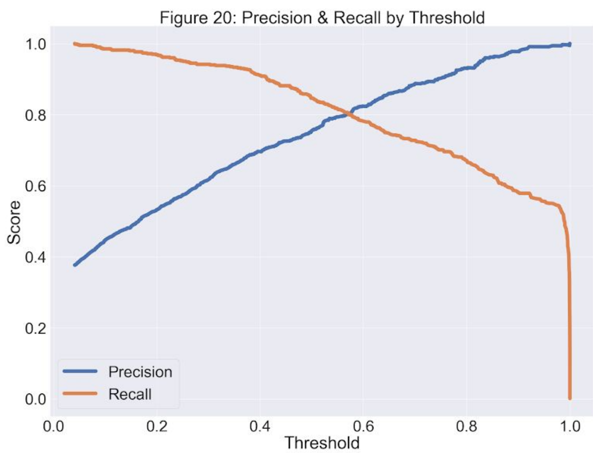


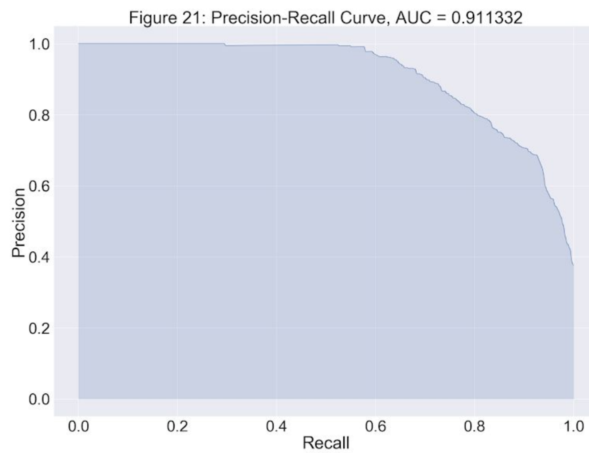


7.2 Precision & Recall

Overall, at the default class threshold of 0.5, the recall score exceeded the precision in twelve of the sixteen model-feature combinations; the rate of false positives was greater than the rate of false negatives (Table 4). These models were better at returning matches than correctly labeling matches.

The same was true for the top RF model, and Figures 20 and 21 reflect the superior recall for the model. In the first plot, the red line more rapidly approaches the recall maximum, and in the second one, the curve on the right side has a steeper slope. Overall, the area under the precision-recall curve is relatively high at 0.91.



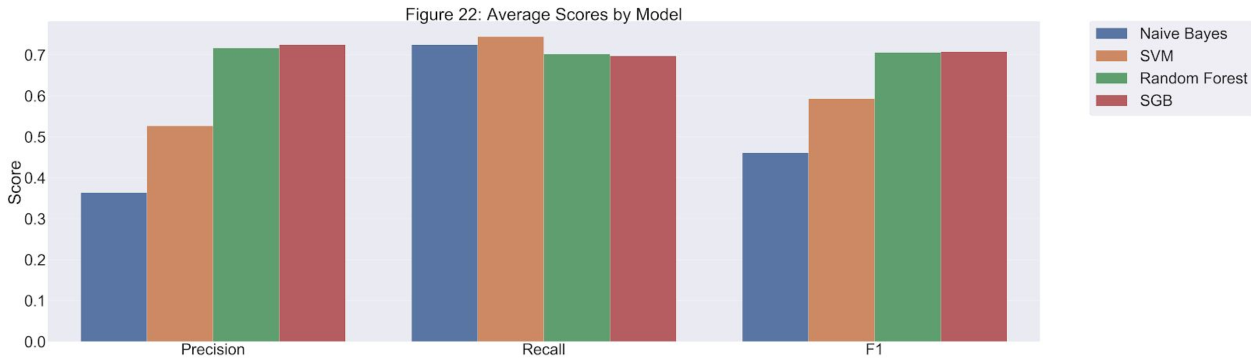


From a customer and business perspective, a precision score of 0.755 at the default threshold is too low. The retailer would be selling the customer an incorrect product 24% of the time. If the class threshold is increased to 0.571, Figure 13 shows that the precision and recall intersect at a score of approximately 0.8. To have no false positives with the test dataset labels, the class threshold would need to be set to 0.999, which would decrease recall to 0.296. On the other hand, to minimize false negatives, the threshold could be set to 0.041, which would yield a precision score of 0.376. The higher precision corresponding with perfect-recall scenario than the recall in the perfect-precision scenario indicates that the model's recall more rapidly approaches a perfect score than its precision.

7.2 Performance by Model

Figure 22 displays the average scores for the models across the four feature sets. The average F1 scores are approximately what I would expect based upon the literature review. The mean F1 score for SGB model marginally exceeded the score for the RF model, which indicates that SGB performed better than RF on the non-optimal feature sets. Notably, the SGB model had the best mean precision score and the smallest average recall. This model may benefit from further development if the priority is precision. These models are followed by the performance of

SVM with a mean F1 of 0.6. As expected, the Naive Bayes performed the least well with a score of 0.46.

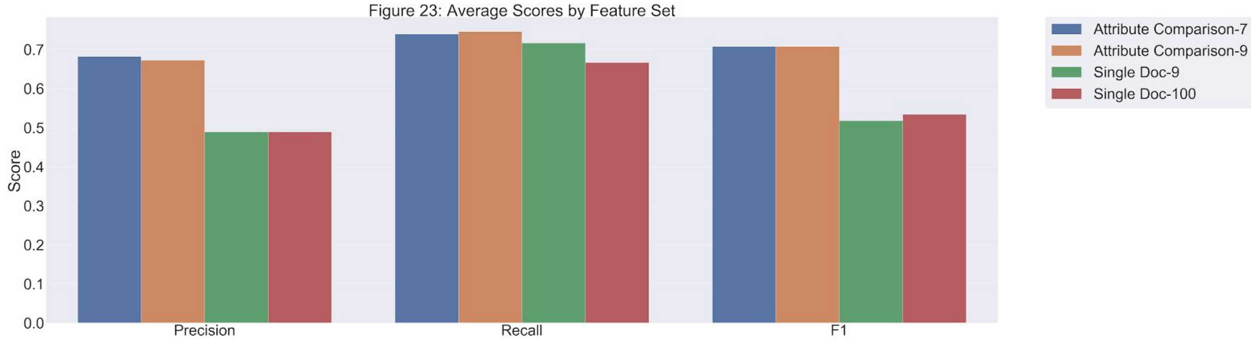


From a training and cross-validation perspective, I had the best experience with tuning the parameters of the RF model. This is primarily because all the weak tree models can be fit independently and in parallel. The SGB was the next easiest model to train and tune, but took considerably longer than RF since the tree models are created in succession. The radial-basis-version of SVM was the most difficult to train and tune and was not a scalable solution.

7.3 Performance by Feature Set

Figure 23 exhibits the mean scores for each feature set across the four models. The attribute-comparison models had the highest F1 at approximately 0.708. The single-document features performed significantly worst, mainly due to poor precision scores. Notably, attribute-comparison-7 features, where I removed the *brand* and *description* attributes, had the highest average precision while retaining a high recall. This feature set may be suitable for further high-precision development. Additionally, Naive Bayes, the model without any automated feature selection, confirmed the insight gained from the logistic regression model. Creating a more parsimonious feature set without the *brand* and *description* attributes yielded an increase in the Naive Bayes F1 score from 0.467 to 0.521 for the attribute-comparison-9 and attribute-comparison-7 features (Table 4).

Figure 23: Average Scores by Feature Set

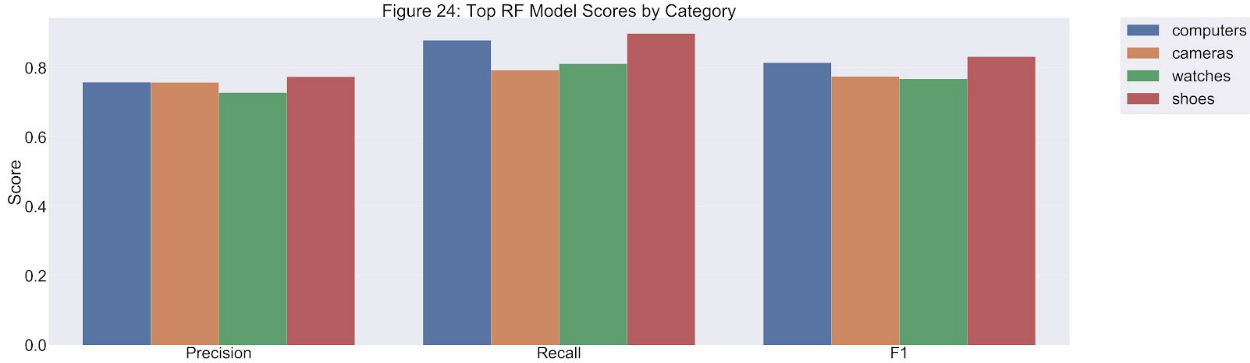


Among the single-document feature sets, while increasing the variables from nine to 100 did marginally improve upon a poor F1 score, the amount of variance explained was low at 25%. Additionally, the higher dimensionality dramatically increased the amount of time needed to train and tune the models. Hence, this feature representation approach is likely not scalable.

7.4 Performance by Category

Figure 24 shows how the best model performed by category. Despite having more than twice as many training samples as *cameras*, the *computers and accessories* category did not have the highest F1 score. *Shoes*, the category with the smallest number of samples, performed the best. After that, the F1 scores then correspond with how many training samples the category had.

Figure 24: Top RF Model Scores by Category



None of the quantitative characteristics of the categories in test dataset explain why the small *shoes* category performed the best. The distributions of the number of missing values per offer pair were no different than the *watches* or *cameras* categories, and the *computers* category had even fewer missing values (Fig. 25). Additionally, the red bars in Figure 26 show the proportion of *mpn* values, the most important feature to the RF model, that were missing in the offer pairs. The *shoes* category actually had the highest missing rate for the *mpn*. The length of the *name* attribute did not explain the performance differential either (not pictured). Other factors like the quality of the *name* attribute likely explain the differences in category performance.

Figure 25: Number of Missing Values

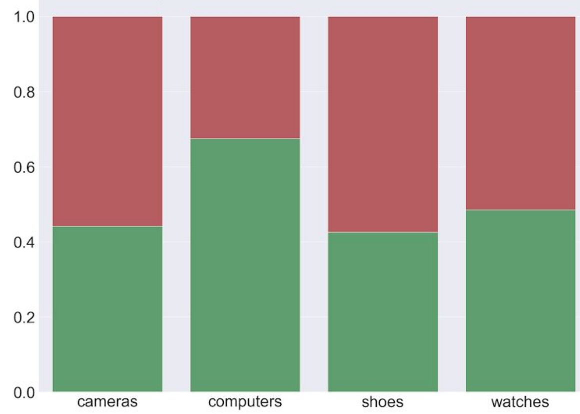
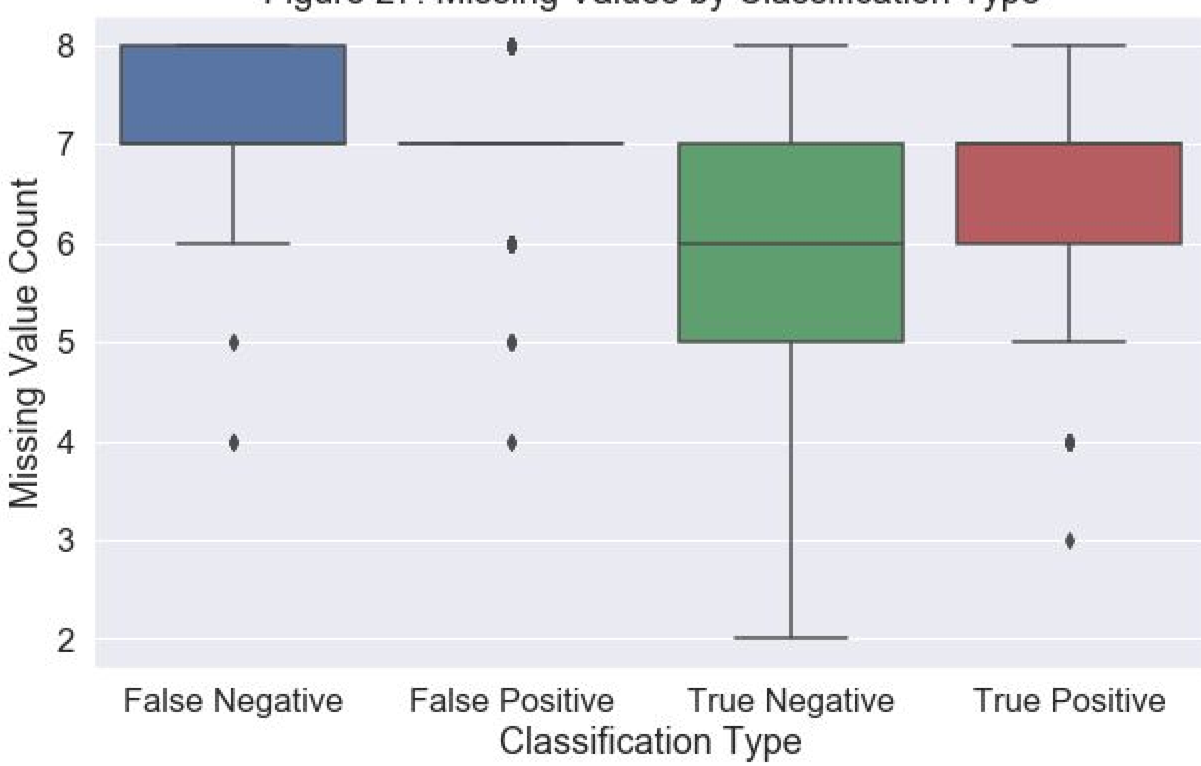


Figure 26: Proportion of MPN Values Present

7.5 Effect of the Missing Values

Figure 27 displays the distributions of the missing values in the test set by the RF model's classification types. The missingness in the data appears to have had little influence on whether the label was correctly predicted. The median number of missing values for all but the true negatives was seven. The interquartile ranges for the true positives and negatives were marginally lower than those for the incorrect predictions.

Figure 27: Missing Values by Classification Type



8 Conclusions & Future Research

8.1 Models

The results of this analysis demonstrate that machine learning algorithms profoundly improve upon the Naive Bayes approach and can make progress toward resolving entity matching for online marketplaces. The tree-based bagged, ensemble models were easiest to train and yielded the best predictions. Random Forests had the top F1 score overall and marginally outperformed Stochastic Gradient Boosting model. Still, a rather steep trade off persistently exists between recall and precision. Given a perfect-precision requirement, the classifiers and feature representations in this project produced an unacceptably low level of recall that would exclude a significant number of offer matches from being in a consolidated marketplace.

Recent developments in deep learning classification would likely narrow the gap between precision and recall. Two types of neural networks are particularly well suited for the EM task. First, the architecture of Siamese networks natively ingests pairs of inputs. It consists of two identically-structured, parallel branches or subnetworks that are connected by only the output layer. Secondly, the bidirectional long short-term memory version of recurrent neural networks are adept at modeling sequential data like time series and natural language. Their forward and backward internal loops can model the contextual nature of text information (Mueller & Thyagarajan, 2016).

8.2 Features

For the feature representations, the attribute-to-attribute comparison approach proved superior to representing sets of attributes as unitary text documents. The idea motivating the single-document approach was to engineer new features in order to compensate for the large volume of missing values in the corpus. However, the higher dimensionality of this feature set is a temporal impediment to training the classifier algorithms. With the attribute-comparison features, the correct and incorrect predictions did not appear to be affected by the distribution of the missing values. However, the missingness in general likely affected the overall precision, recall and F1 scores.

In future iterations, the following adaptations may be able to improve upon the attribute comparison approach: Instead of scaling the similarity value between zero and one, an unscaled similarity measure might account for the different lengths of text and provide more information to the model. For example, an edit distance score where 75% of the characters match is likely to be more significant if the pair of texts are 100 characters long rather than only ten-characters long. Additionally, extracting missing attribute values from the available text may increase the importance of some of the atomic attributes. For example, the existing values for the *brand* and *manufacturer* attributes can likely be used to extract some of the missing instances from the other text attributes.

The study of neural networks has also yielded new ways of engineering textual features that may improve upon bag-of-words and edit distances. Mikolov et al. (2013) and Le and Mikolov (2014) developed efficient methods for learning distributed representations of words and documents through neural language models. Text embeddings use the context to probabilistically encode words and documents into a coherent vector space; the relationships between vectors closely approximate a human being's understanding of them. The approaches employed in Ebraheem et al. (2018) and Mudgal et al. (2018) provide the foundation for further research into how distributed representations can resolve the EM problem for online marketplaces.

9 References

Breiman, L. (2001). Random Forests. Machine Learning, 45, 5-32.

Chollet, F. (2017). Deep Learning with Python., 8-19.

Ebraheem, M., Thirumuruganathan, S., Joty, S.R., Ouzzani, M., & Tang, N. (2017). DeepER - Deep Entity Resolution. CoRR, abs/1710.00597.

Elmagarmid, A.K., Ipeirotis, P.G., & Verykios, V.S. (2007). Duplicate Record Detection: A Survey. IEEE Transactions on Knowledge and Data Engineering, 19, 1-16.

Friedman, J.H. (2001). Greedy function approximation: A gradient boosting machine. Ann. Statist. 29, no. 5, 1189-1232.

Köpcke, H., & Rahm, E. (2010). Frameworks for entity matching: A comparison. Data Knowl. Eng., 69, 197-210.

Le, Q.V., & Mikolov, T. (2014). Distributed Representations of Sentences and Documents. ICML.

Mikolov, T., Chen, K., Corrado, G.S., & Dean, J. (2013). Efficient Estimation of Word Representations in Vector Space. CoRR, abs/1301.3781.

Mudgal, S., Li, H., Rekatsinas, T.I., Doan, A., Park, Y., Krishnan, G., Deep, R., Arcaute, E., & Raghavendra, V. (2018). Deep Learning for Entity Matching: A Design Space Exploration. SIGMOD Conference.

Mueller, J., & Thyagarajan, A. (2016). Siamese Recurrent Architectures for Learning Sentence Similarity. AAAI.

Newcombe, H.B., Kennedy, J.M., Axford, S.J., & James, A.P. (1959). Automatic linkage of vital records. Science, 130 3381, 954-9 .

Porter, M.F. (1980). An algorithm for suffix stripping. Program, 14, 130-137.

Ramos, J.E. (2003). Using TF-IDF to Determine Word Relevance in Document Queries.

Shah, K., Köprü, S., & Ruvini, J. (2018). Neural Network based Extreme Classification and Similarity Models for Product Matching. NAACL-HLT.

10 Code Appendix

Repository: https://github.com/kylegilde/Entity-Matching-in-Online-Retail

List of Scripts

01-parse-json-to-dfs.py
02-normalize-features.py
03-create-attribute-comparison-features.py
03-create-single-doc-similarity-features.py
04-train-cv-models.py
04-train-non-cv-models.py
05-tune-individual-models.py
json_parsing_functions.py
utility_functions.py
LogReg Variable Importance.Rmd (see repo)
Presentation.ipynb (see repo)

Scripts

01-parse-json-to-dfs.py

```
# !/usr/bin/env/python3
# -*- coding: utf-8 -*-
"""
Created on Feb 10, 2019
@author: Kyle Gilde

This script is used to process the  the WDC Training Dataset and
Gold Standard for Large-Scale Product Matching (LSPM),
which was created from the Common Crawl web-data corpus
and published in December 2018 by a group of researchers at the
University of Mannheim.
```

```
Data can be downloaded here:
http://webdatacommons.org/largescaleproductcorpus/index.html

"""
import os
from datetime import datetime

import numpy as np
import pandas as pd
from pandas.io.json import json_normalize

import matplotlib.pyplot as plt
from json_parsing_functions import *
from utility_functions import *

# Initialize some constants
TRAIN_TEST_CATEGORIES = ['Computers_and_Accessories',
'Camera_and_Photo', 'Shoes', 'Jewelry']
PRICE_COLUMN_NAMES = ['price','parent_price']
DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
os.chdir(DATA_DIRECTORY)

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 0)


################################################################
#
################### Load Train and Test Data
################################################################
#

# Load Train & Test Data
train_df, test_df = read_train_test_files(DATA_DIRECTORY +
'/training-data/'),\
```

```
                   read_train_test_files(DATA_DIRECTORY +
'/test-data/')

print(train_df.info(memory_usage='deep'))
print(test_df.info(memory_usage='deep'))

plt.gcf().clear()
train_df.filename.value_counts().plot.bar()

train_test_df = pd.concat([train_df, test_df],
axis=0).reset_index()
print(train_test_df.info())
train_test_df = create_file_categories(train_test_df)

# remove some duplicates
train_test_df =
train_test_df[~train_test_df.duplicated(subset=['offer_id_1',
'offer_id_2', 'filename', 'dataset', 'label'])]

# remove a bad index
bad_index = train_test_df.index[(train_test_df.offer_id_1 ==
'_:nodede8ccd8d6cc033e333fc23a88f31fad
http://www.prodirectselect.com/products/asics-womens-gelnoosa-tr
i-11-white-noise-womens-shoes-white-white-black-120149.aspx') &\
             (train_test_df.offer_id_2 ==
'_:nodefa2169b488f32926e188bbf2f8567bf
https://footstop.com/producto/asics-gel-noosa-tri-11-t676q-0101/
') &
             (train_test_df.label == 0)]
train_test_df.drop(bad_index, inplace=True)

print(train_test_df.info())

train_test_df.to_csv('train_test_df.csv', index=False)

train_test_offer_ids = rbind_train_test_offers(train_df,
test_df)
print(train_test_offer_ids.info())
```

```
unique_train_test_offer_ids = pd.DataFrame({'offer_id' :
train_test_offer_ids.index.unique()}).set_index('offer_id')
print(len(unique_train_test_offer_ids))


################################################################
#
################ Clean & Tidy the Offers data
################################################################
#

# if file exists read it, otherwise create it
if 'train_test_offer_features.csv' in os.listdir():
   train_test_offer_features =
reduce_mem_usage(pd.read_csv('train_test_offer_features.csv',

index_col='offer_id'))
   print(train_test_offer_features.info(memory_usage='deep'))
else:
   start = datetime.now()
   # Read or Create the Cluster ID-Category Mappings
   # category_cluster_ids = get_cluster_ids()

   # get offers data for train-test categories
   offers_reader = pd.read_json('offers_consWgs_english.json',
                                orient='records',
                                chunksize=1e6,
                                lines=True)

   columns_with_json = ['identifiers', 'schema_org_properties',
'parent_schema_org_properties']
   columns_to_drop = ['parent_NodeID'] + columns_with_json # ,
'relationToParent'
   more_cols_to_drop = ['availability']


   offers_df_list = []
   for i, chunk in enumerate(offers_reader):
       print(i)
```

```
        # chunk = next(offers_reader)


        # create the unique offer_id
        chunk['offer_id'] = chunk.nodeID.str.cat(chunk.url, sep='
')
        # inner join on the train-test offer ids
        new_chunk = reduce_mem_usage(chunk)\
                        .set_index('offer_id')\
                        .join(unique_train_test_offer_ids,
how='inner')

        # if any rows remain after inner join, parse the data
        if len(new_chunk):
            # clean column names
            new_chunk.columns =
new_chunk.columns.str.replace('.', '_')
            # parse the website domain into column
            new_chunk['domain'] =
new_chunk.url.apply(parse_domain)
            # set offer_id to index and drop its components
            new_chunk = new_chunk.reset_index()\
                .set_index('offer_id')\
                .drop(['nodeID', 'url'], axis=1)

            parsed_df_list = []
            json_columns_df = new_chunk[columns_with_json]
            for column in columns_with_json:
                # column = columns_with_json[2]
                print(column)
                df =
parse_json_column(json_columns_df[column].apply(json_normalize))
\
                       .apply(lambda x: x.str.strip() if x.dtype ==
"object" else x)\
                       .replace('null', np.nan)
                print(df.columns)

                df.index = new_chunk.index
```

```
                # coalesce the gtins
                if column == 'identifiers':
                    df = coalesce_gtin_columns(df)
                elif column == 'parent_schema_org_properties':
                    df.columns = 'parent_' + df.columns
                # parse the price columns
                if np.sum(df.columns.isin(PRICE_COLUMN_NAMES)):
                    df = parse_price_columns(df,
PRICE_COLUMN_NAMES)

                parsed_df_list.append(df)

            print(datetime.now() - start)
            # Drop the 3 parsed columns
            new_chunk.drop(columns_to_drop, axis=1, inplace=True)
            # Concatenate the chunk to the 3 parsed columns & add
it to the df list
            parsed_df_list.append(new_chunk)
            # combine the parent child columns
            new_chunk =
coalesce_parent_child_columns(pd.concat(parsed_df_list, axis=1,
sort=False))

            # Remove the terms null and description
            new_chunk['name'] =
new_chunk.name.combine_first(df.parent_title)\
                .str.replace('^null\s*?,|,\s*?null$', '')
            new_chunk['description'] =
new_chunk.description.str.replace('^description', '')
            offers_df_list.append(new_chunk)


    # Combine all the offers & save the output
    print('Saving as CSV...')
    train_test_offer_features =
reduce_mem_usage(pd.concat(offers_df_list, axis=0)\

.drop(more_cols_to_drop, axis=1)\
```

```
                                                  .dropna(axis=1,
how='all'))


train_test_offer_features.to_csv('train_test_offer_features.csv'
, index_label='offer_id')

   print(train_test_offer_features.describe(include='all'))
   print(train_test_offer_features.info(memory_usage='deep'))
   calculate_percent_nulls(train_test_offer_features)

   ################################
   # Add the Category Attribute
   ################################
   print('Adding the category from clusters file...')
   english_clusters_reader =
pd.read_json('clusters_english.json',
                                                    lines=True,

orient='records',

chunksize=1e6)
   english_cluster_list = []
   for i, chunk in enumerate(english_clusters_reader):
       print(i)

       another_chunk = chunk[['id', 'category', 'id_values']]\
           .rename(columns={'id':'cluster_id'})\
           .set_index('cluster_id')

       english_cluster_list.append(another_chunk)

   train_test_offer_features = train_test_offer_features\
       .reset_index()\
       .set_index('cluster_id', drop=False)\
       .join(pd.concat(english_cluster_list, axis=0))\
       .set_index('offer_id')

   #
```

```
    train_test_offer_features =
create_file_categories[train_test_offer_features]

    # Save df
    print('Saving as CSV...')

train_test_offer_features.to_csv('train_test_offer_features.csv'
, index_label='offer_id')
    # missingness plot
    # sns.heatmap(train_test_offer_features[['brand',
'manufacturer']].isnull(), cbar=False)

    calculate_percent_nulls(train_test_offer_features)
    train_test_offer_features.domain.value_counts()
    train_test_offer_features.brand.value_counts()



#################################################################
#
#### Create the df for only the offers in train/test set
#################################################################
#

# if file exists read it, otherwise create it
if 'train_test_feature_pairs.csv' in os.listdir():
    train_test_feature_pairs =
reduce_mem_usage(pd.read_csv('train_test_feature_pairs.csv'))

else:
    # join the offer details to the pair of offer ids
    # and add suffixes to them
    # move label column to 1st position
    # drop completely null columns
    train_test_feature_pairs =
train_test_df.set_index('offer_id_1', drop=False)\
        .join(train_test_offer_features.add_suffix('_1'),
how='left')\
        .set_index('offer_id_2', drop=False)\
```

```
        .join(train_test_offer_features.add_suffix('_2'),
how='left')\
        .sort_index(axis=1)\
        .set_index('label')\
        .reset_index()


train_test_feature_pairs.to_csv('train_test_feature_pairs.csv',
index=False)

    calculate_percent_nulls(train_test_feature_pairs)
```

02-normalize-features.py

```
# !/usr/bin/env/python365
"""

Created on Apr 23, 2019
@author: Kyle Gilde

This script ingests the train_test_offer_features.csv created by
01-parse-json-to-dfs.py
and does the following:

   - Removes non-alphanumeric/space characters from all text
columns
       and creates the train_test_normalized_features.csv to be
used to
       create the semantic features
   - Stems the name description and other text features

Creates output file called train_test_stemmed_features.csv

"""
import os

import numpy as np
import pandas as pd
```

```python
from utility_functions import *

import nltk
from nltk.corpus import stopwords


def clean_text(series):
    """

    1. remove html tags
    2. converts to lowercase
    3. replace 2 consecutive non-alphnum-space characters with a
space
    4. remove remaining non-alphnum-space characters
    5. remove some English-language indicators

    :param series:
    :return:
    """
    return series.str.replace(r'<.*?>', '')\
        .str.lower()\
        .str.replace(r'[^a-z0-9 ]{2,}', ' ')\
        .str.replace(r'[^a-z0-9 ]', '')\
        .str.replace(r'\W(en|enus)\W', '')


def stem_and_remove_stopwords(doc, stemmer):
    """

    :param doc:
    :param stemmer:
    :return:
    """
    if pd.isnull(doc):
        return doc
    else:
        stemmed_tokens = [stemmer.stem(token) for token in
nltk.word_tokenize(doc)\
```

```
                                if token not in
stopwords.words('english')]

        return ' '.join(stemmed_tokens)


# initialize constants
DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
os.chdir(DATA_DIRECTORY)
MAX_SVD_COMPONENTS = 3000
N_ROWS_PER_ITERATION = 2000

TEXT_FEATURES = ['name', 'description']
SHORT_TEXT_FEATURES = ['brand', 'manufacturer']
IDENTIFIER_FEATURES = ['gtin', 'mpn', 'sku', 'identifier']
ALL_TEXT_FEATURES = TEXT_FEATURES + SHORT_TEXT_FEATURES +
IDENTIFIER_FEATURES

STRONGLY_TYPED_FEATURES = ['category']
NUMERIC_FEATURE = ['price']

ALL_FEATURES = ALL_TEXT_FEATURES + STRONGLY_TYPED_FEATURES +
NUMERIC_FEATURE

sb_stemmer = nltk.stem.SnowballStemmer('english')

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 0)

# load parsed features
if 'train_test_feature_pairs.csv' in os.listdir():
    train_test_offer_features =
reduce_mem_usage(pd.read_csv('train_test_offer_features.csv'))\
        .set_index('offer_id')
    # the identifier and productID columns are mutually
exclusively null
```

```
    train_test_offer_features['identifier'] =
train_test_offer_features.identifier\
        .mask(pd.isnull, train_test_offer_features.productID)


train_test_normalized_features =
train_test_offer_features[ALL_FEATURES]

# loop through text features and clean them
for column in train_test_normalized_features.columns:
    if column in ALL_TEXT_FEATURES:
        train_test_normalized_features[column] =
clean_text(train_test_normalized_features[column])

# save to file
train_test_normalized_features.reset_index().to_csv('train_test_
normalized_features.csv', index=False)

train_test_stemmed_features =
train_test_normalized_features.copy()

for column in train_test_stemmed_features.columns:
    if column in TEXT_FEATURES:
        train_test_stemmed_features[column] =
train_test_normalized_features[column]\
            .apply(lambda x: stem_and_remove_stopwords(x,
stemmer=sb_stemmer))

# save to file
train_test_stemmed_features.reset_index().to_csv('train_test_ste
mmed_features.csv', index=False)
```

03-create-attribute-comparison-features.py

```
# !/usr/bin/env/python365
"""
Created on Apr 27, 2019
@author: Kyle Gilde
```

```
This script takes 2 input files:

   1. 'train_test_stemmed_features.csv' created from
02-normalized-features.py
   2. 'train_test_df.csv' created from 01-parse-json-to-dfs.py

For each of the 4 dataset categories, this script compares each
of the corresponding attribute values for each pair of offers.

   1. For the 7 short-to-medium length text features, it
calculates the scaled Levenshtein similarity score.
   2. For the long description feature, it encodes the text as
bag-of-words TF-IDF approach for unigrams, bigrams & trigrams.
       Truncated SVD is used to select only the components that
explain 99.9% of the variances, and then the cosine similarity
       is calculated for the 2 vectors.
   3. For the price, the absolute percent diffence is calculated
   4. For the strongly-typed offer category, a 0 or 1 indicates
whether they are an exact match.

It outputs symbolic_similarity_features.csv,
which is a dataframe that contains the similarity vectors of the
offer pairs
 for both the test and training sets.

"""

import os
import gc

from datetime import datetime
import nltk
import numpy as np
import pandas as pd

from utility_functions import *

from sklearn.feature_extraction.text import TfidfVectorizer
```

```
from sklearn.decomposition import TruncatedSVD
from scipy.spatial.distance import cosine


def levenshtein_similarity(df_row):
    """

    Calculates the scaled Levenshtein similarity for 2 strings.

    :param df_row: a list-like object containing 2 strings
    :return: a float between 0 and 1
    """

    str1, str2 = df_row[0], df_row[1]

    if pd.isnull(str1) or pd.isnull(str2):
        return 0
    else:
        return 1 - nltk.edit_distance(str1, str2) /
max(len(str1), len(str2))


def elementwise_cosine_similarity(df_row, n_features):
    """
    Calculates the elementwise cosine similarity with the apply
method

    :param df_row: a DF row where the first n_features are the
left side features
        and the last n_features are the right side features
    :param n_features: this is the number of features each side
of the DTM has
    :return: float between 0 and 1
    """

    s1, s2 = df_row[:n_features], df_row[n_features:]

    if np.sum(s1) == 0 or np.sum(s2) == 0:
        return 0
```

```
    else:
        return cosine(s1, s2)


start_time = datetime.now()

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 250)

# initialize constants
DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
DATA_DIRECTORY = '//files/share/goods/OI Team'
os.chdir(DATA_DIRECTORY)
MAX_SVD_COMPONENTS = 3000
VARIANCE_EXPLAINED = 0.999
N_ROWS_PER_ITERATION = 2000

# initialize the variable types
SHORT_TEXT_FEATURES = ['brand', 'manufacturer']
IDENTIFIER_FEATURES = ['gtin', 'mpn', 'sku', 'identifier']
ALL_SHORT_TEXT_FEATURES = SHORT_TEXT_FEATURES +
IDENTIFIER_FEATURES + ['name']
LONG_TEXT_FEATURES = ['description']
STRONGLY_TYPED_FEATURES = ['category']
NUMERIC_FEATURE = ['price']

# the description column must be last in the list
ALL_FEATURES = ALL_SHORT_TEXT_FEATURES + STRONGLY_TYPED_FEATURES
+ NUMERIC_FEATURE + LONG_TEXT_FEATURES

OFFER_PAIR_COLUMNS = ['offer_id_1', 'offer_id_2', 'filename',
'dataset']

# set display options
pd.set_option('display.max_rows', 3000)
```

```
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 0)

# load files
assert 'train_test_stemmed_features.csv' in os.listdir() and
'train_test_df.csv' in os.listdir(), 'An input file is missing'

train_test_stemmed_features =
reduce_mem_usage(pd.read_csv('train_test_stemmed_features.csv'))
\
   .set_index('offer_id')

train_test_df =
reduce_mem_usage(pd.read_csv('train_test_df.csv'))


file_categories = train_test_df.file_category.unique()
category_df_list = []
start_time = datetime.now()

for the_category in file_categories:

   print('the_category:', the_category)
   # the_category = 'shoes'

   cat_symbolic_similarity_features =
train_test_df[train_test_df.file_category ==
the_category].copy()
   unique_offer_ids =
pd.concat([cat_symbolic_similarity_features.offer_id_1.astype('o
bject'),

cat_symbolic_similarity_features.offer_id_2.astype('object')])\
       .unique()


cat_symbolic_similarity_features.set_index(OFFER_PAIR_COLUMNS,
inplace=True)
```

```
    for column in ALL_FEATURES:
        print('column:', column)
        get_duration_hours(start_time)

        # put the left and right side feature into a df
        both_features =\

cat_symbolic_similarity_features.reset_index()[OFFER_PAIR_COLUMN
S]\
            .set_index(OFFER_PAIR_COLUMNS[0], drop=False)\

.join(train_test_stemmed_features[[column]].add_suffix('_1'),
how='inner')\
            .set_index(OFFER_PAIR_COLUMNS[1], drop=False)\

.join(train_test_stemmed_features[[column]].add_suffix('_2'),
how='inner')\
            .set_index(OFFER_PAIR_COLUMNS)

        if column in ALL_SHORT_TEXT_FEATURES:

            # scaled Levenshtein similarity score
            cat_symbolic_similarity_features[column] =
both_features.apply(levenshtein_similarity, axis=1)
            cat_symbolic_similarity_features =
reduce_mem_usage(cat_symbolic_similarity_features)

        elif column in STRONGLY_TYPED_FEATURES:

            # exact match
            cat_symbolic_similarity_features[column] =
pd.Series(both_features.iloc[:, 0]\
                                                                      ==
both_features.iloc[:, 1]).astype('int8')
            cat_symbolic_similarity_features =
reduce_mem_usage(cat_symbolic_similarity_features)

        elif column in NUMERIC_FEATURE:
```

```
            # absolute percent difference
            cat_symbolic_similarity_features[column] = \
                np.nan_to_num(np.absolute(both_features.iloc[:,
0] - both_features.iloc[:, 1]) / \
                              np.maximum(both_features.iloc[:,
0], both_features.iloc[:, 1]))
            cat_symbolic_similarity_features =
reduce_mem_usage(cat_symbolic_similarity_features)

        elif column in LONG_TEXT_FEATURES:
            # column = 'description'

            # bag-of-words TF-IDF with Truncated SVD and cosine
similarity
            vectorizer = TfidfVectorizer(ngram_range=(1, 3))

            # create a document-term matrix from the unique
column values
            unique_column_values =
train_test_stemmed_features.loc[unique_offer_ids][[column]].fill
na('')

            dtm =
vectorizer.fit_transform(unique_column_values[column])
            print('dtm dimensions:', dtm.shape)

            get_duration_hours(start_time)
            print('Use Truncated SVD to select a smaller number
of important features')
            svd_model =
TruncatedSVD(n_components=MAX_SVD_COMPONENTS).fit(dtm)
            print('n_components:',
len(svd_model.explained_variance_ratio_))
            print(svd_model.explained_variance_ratio_.sum(),
'variance explained')
```

```
            n_features =
sum(svd_model.explained_variance_ratio_.cumsum() <=
VARIANCE_EXPLAINED)
            print(n_features, "features explain this much of the
variance:", VARIANCE_EXPLAINED)

            print('fit the svd model and convert to df')
            dtm_svd =
reduce_mem_usage(pd.DataFrame(svd_model.transform(dtm)[:,
:n_features],

index=unique_column_values.index))

            print('post-SVD DTM dimensions:', dtm_svd.shape)
            print(dtm_svd.info(memory_usage='deep'))

            del dtm, svd_model, unique_column_values
            gc.collect()

            get_duration_hours(start_time)
            print("Let's create a df to hold both sides of the
DTM")

            both_sides_dtm_svd =\
                reduce_mem_usage(
                    both_features
                        .reset_index()
                        .drop(['description_1', 'description_2'],
axis=1)
                        .set_index('offer_id_1', drop=False)
                        .join(dtm_svd.add_suffix('_1'),
how='inner')
                        .set_index('offer_id_2', drop=False)
                        .join(dtm_svd.add_suffix('_2'),
how='inner')
                        .set_index(OFFER_PAIR_COLUMNS)
                )

            print(both_sides_dtm_svd.info())
```

```
            del both_features
            gc.collect()

            get_duration_hours(start_time)
            print("Let's calculate the cosine similarity.")
            assert both_sides_dtm_svd.shape[1] == 2 * n_features,
"Something is wrong. Your df row length is not 2 x n_features"
            cat_symbolic_similarity_features[column] =
both_sides_dtm_svd.apply(elementwise_cosine_similarity,

n_features=n_features,

axis=1)
            del both_sides_dtm_svd
            gc.collect()

            # append category DF

category_df_list.append(cat_symbolic_similarity_features)

   print('Combine all the categories')
   symbolic_similarity_features =\
       reduce_mem_usage(pd.concat(category_df_list, axis=0))\
       .reset_index()

   print('Summary stats')
   print(symbolic_similarity_features.columns)
   print(symbolic_similarity_features.shape)
   print(symbolic_similarity_features.describe())
   print(symbolic_similarity_features.info())

   print("symbolic_similarity_features saved")

symbolic_similarity_features.to_csv('symbolic_similarity_feature
s.csv', index=False)

   get_duration_hours(start_time)
```

03-create-single-doc-similarity-features.py

```
# !/usr/bin/env/ python3
# -*- coding: utf-8 -*-

"""
Created on Apr 27, 2019
@author: Kyle Gilde

This script takes the outputs of create-symbolic-features.py
and parse-json-to-dfs.py.

It outputs a df that contains the similarity vectors of the
offer pairs
in the test and training sets. The df is saved as
symbolic_single_doc_similarity_features.csv

"""

import os
import gc

from datetime import datetime
# import nltk
import numpy as np
import pandas as pd

from utility_functions import *

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.decomposition import TruncatedSVD

start_time = datetime.now()

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
```

```
pd.set_option('display.max_colwidth', 250)

# global variables
DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
DATA_DIRECTORY = '//files/share/goods/OI Team'
os.chdir(DATA_DIRECTORY)

VARIANCE_EXPLAINED_MAX = 0.999
# feature list
ALL_FEATURES = ['name', 'description', 'brand', 'manufacturer',
'gtin', 'mpn', 'sku', 'identifier', 'price']
# offer pair index columns
OFFER_PAIR_COLUMNS = ['offer_id_1', 'offer_id_2', 'filename',
'dataset', 'label', 'file_category']

# user inputs
svd_components_to_retain = int(input('Enter the number of SVD
components to retains'))
output_file_name = 'symbolic_single_doc_similarity_features-' +
str(svd_components_to_retain) + '.csv'

# load files
assert 'train_test_stemmed_features.csv' in os.listdir() and
'train_test_df.csv' in os.listdir(), 'An input file is missing'

print('load the stemmed features')
train_test_stemmed_features =
pd.read_csv('train_test_stemmed_features.csv')\
   .set_index('offer_id')
print(train_test_stemmed_features.info())

print('load the offer pairs')
train_test_df =
reduce_mem_usage(pd.read_csv('train_test_df.csv')\
                               .set_index(OFFER_PAIR_COLUMNS))
print(train_test_df.info())
assert len(train_test_df.columns) == 0, 'The DF contains extra
column(s)'
```

```
start_time = datetime.now()

print("Let's concatenate all features into single documents")

train_test_stemmed_features['price'] =
train_test_stemmed_features.price.astype('str')
unique_column_values =\
   train_test_stemmed_features[ALL_FEATURES]\
       .apply(lambda x: ' '.join(x.dropna()), axis=1)\
       .str.replace('\Wnan$', '')

unique_column_value_indices = unique_column_values.index

print("Let's take a look at the concatenation")
print(unique_column_values.reset_index().head())

# reclaim some memory
del train_test_stemmed_features
gc.collect()

print("Let's create a DTM using TF-IDF")
vectorizer = TfidfVectorizer(ngram_range=(1, 3))

dtm = vectorizer.fit_transform(unique_column_values)
print('dtm dimensions:', dtm.shape)

# reclaim some memory
del unique_column_values
gc.collect()

get_duration_hours(start_time)
print('Use Truncated SVD to select a smaller number of important
features')
svd_model =
TruncatedSVD(n_components=svd_components_to_retain).fit(dtm)

print('n_components:', len(svd_model.explained_variance_ratio_))
variance_explained = svd_model.explained_variance_ratio_.sum()
print(variance_explained, 'variance explained')
```

```
n_features = sum(svd_model.explained_variance_ratio_.cumsum() <=
VARIANCE_EXPLAINED_MAX)
print(n_features, "features explain this much of the variance:",
variance_explained)

print('fit the svd model and convert to df')
dtm_svd =
reduce_mem_usage(pd.DataFrame(svd_model.transform(dtm)[:,
:n_features],

index=unique_column_value_indices))

print('post-SVD DTM dimensions:', dtm_svd.shape)
print(dtm_svd.info(memory_usage='deep'))

# reclaim some memory
del dtm, svd_model
gc.collect()

get_duration_hours(start_time)
train_test_df.info()
print("Let's create a df to hold both sides of the DTM")
symbolic_single_doc_similarity_features =\
   reduce_mem_usage(
       train_test_df\
           .reset_index()
           .set_index('offer_id_1', drop=False)
           .join(dtm_svd.add_suffix('_1'), how='inner')
           .set_index('offer_id_2', drop=False)
           .join(dtm_svd.add_suffix('_2'), how='inner')
           .set_index(OFFER_PAIR_COLUMNS)
   )
print(symbolic_single_doc_similarity_features.info())

assert symbolic_single_doc_similarity_features.shape[1] == 2 *
n_features, "Something is wrong. Your df row length is not 2 x
n_features"
```

```
print('Calculate the absolute difference between each offer
pair')
symbolic_single_doc_similarity_features_df =\
   (symbolic_single_doc_similarity_features.iloc[:,
:n_features]\
   .sub(symbolic_single_doc_similarity_features.iloc[:,
n_features:].values))\
   .abs()

del symbolic_single_doc_similarity_features
gc.collect()

print('Summary stats')
print(symbolic_single_doc_similarity_features_df.shape)
print(symbolic_single_doc_similarity_features_df.info())
print(symbolic_single_doc_similarity_features_df.columns)
print(symbolic_single_doc_similarity_features_df.describe())

get_duration_hours(start_time)
print("symbolic_similarity_features saved:", output_file_name)
symbolic_single_doc_similarity_features_df.to_csv(output_file_na
me)

print(variance_explained, 'variance explained')
get_duration_hours(start_time)
```

04-train-cv-models.py

```
# !/usr/bin/env/ python3
# -*- coding: utf-8 -*-

"""
Created on Apr 27, 2019
@author: Kyle Gilde

This script is used to tune and train CV versions of the models.
```

```python
This script takes the feature set outputs of
03-create-attribute-comparison-features.py
and 03-create-single-doc-similarity-features.py
"""

import os
import gc

import numpy as np
import pandas as pd
import pickle
from utility_functions import *

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split,
GridSearchCV, StratifiedKFold, RandomizedSearchCV,
cross_val_score
from sklearn.metrics import classification_report,
confusion_matrix, precision_score, recall_score, f1_score,
make_scorer

from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier


# global variables
DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
# DATA_DIRECTORY = '//files/share/goods/OI Team'
os.chdir(DATA_DIRECTORY)

RANDOM_STATE = 5
FOLDS = 2
DEV_TEST_SIZE = .5
METRIC_NAMES = ['Precision', 'Recall', 'F1_score']
# ALL_FEATURES = ['brand', 'manufacturer', 'gtin', 'mpn', 'sku',
'identifier', 'name', 'price', 'description'] # 'category'
```

```
OFFER_PAIR_COLUMNS = ['offer_id_1', 'offer_id_2', 'filename',
'dataset', 'label', 'file_category']

# list of models to fit
# MODEL_NAMES = ['Naive Bayes', 'SVM', 'Random Forest',
'Gradient Boosting']
MODELS = [GaussianNB(),
         SVC(random_state=RANDOM_STATE, class_weight='balanced',
verbose=2), # probability=True,
         RandomForestClassifier(random_state=RANDOM_STATE,
class_weight='balanced', verbose=2),
         GradientBoostingClassifier(random_state=RANDOM_STATE,
n_iter_no_change=30, verbose=2)]

model_names = [model.__class__.__name__ for model in MODELS]
model_dict = dict(zip(model_names, MODELS))

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 250)

# provide input file
input_file_name = 'attribute_comparison_features-7.csv' #
input('Input the features file')
assert input_file_name in os.listdir(), 'An input file is
missing'

# read input file
symbolic_similarity_features =
reduce_mem_usage(pd.read_csv(input_file_name))
print(symbolic_similarity_features.columns.tolist())

# get the train & test indices
train_indices, test_indices =
symbolic_similarity_features.dataset.astype('object').apply(lamb
da x: x == 'train').values,\
```

```
symbolic_similarity_features.dataset.astype('object').apply(lamb
da x: x == 'test').values

# get the labels
all_labels = symbolic_similarity_features.label
train_labels, test_labels = all_labels[train_indices],
all_labels[test_indices]
class_labels = np.sort(all_labels.unique())

# create features df
symbolic_similarity_features.set_index(OFFER_PAIR_COLUMNS,
inplace=True)

print(symbolic_similarity_features.columns.tolist())
print(symbolic_similarity_features.info())
print(symbolic_similarity_features.shape)
print(symbolic_similarity_features.describe())

# center and scale for SVM
scaler = StandardScaler()
symbolic_similarity_features =
scaler.fit_transform(symbolic_similarity_features)

# train and test features
train_features, test_features =
symbolic_similarity_features[train_indices, :],\

symbolic_similarity_features[test_indices, :]

dev_train_features, dev_test_features, dev_train_labels,
dev_test_labels =\
   train_test_split(train_features, train_labels,
test_size=DEV_TEST_SIZE, random_state=RANDOM_STATE)

print('Dev Train Feature Shape')
print(dev_train_features.shape)

nb_grid_params = None
```

```
# svc_grid_params = {'kernel': ['linear', 'poly', 'rbf']}

svc_grid_params = {'C': np.logspace(-1, 2, 4),
                   'gamma': np.logspace(0, 1, 2)}

rf_grid_params = {'max_depth': [None, 5],
                  'max_features': ['auto', None],
                  'max_leaf_nodes': [None],
                  'min_impurity_decrease': [0.0],
                  'min_impurity_split': [None],
                  'min_samples_leaf': [1],
                  'min_samples_split': [2, 4],
                  'min_weight_fraction_leaf': [0.0],
                  'n_estimators': [500, 1000]}

count_cv_models(rf_grid_params, FOLDS, .18)

gbm_grid_params = {'learning_rate': [.025, .05],
                   'max_depth': [None, 5],
                   'max_features': ['auto', None],
                   'max_leaf_nodes': [None],
                   'min_impurity_decrease': [0.0],
                   'min_samples_leaf': [1],
                   'min_samples_split': [2],
                   'min_weight_fraction_leaf': [0.0],
                   'n_estimators': [150, 300],
                   'subsample': [.33, .67]}

count_cv_models(gbm_grid_params, FOLDS, .07)

grid_param_list = [nb_grid_params, svc_grid_params,
rf_grid_params, gbm_grid_params]
grid_param_dict = dict(zip(model_names, grid_param_list))

# create stratified folds
skf = StratifiedKFold(n_splits=FOLDS, random_state=RANDOM_STATE)

# output DF
```

```
test_metrics = []
dev_test_metrics = []
model_durations = []
best_params_list = []

# save diagnostics
test_predictions = []
class_probabilities_list = []
fit_models = []
classification_reports = []
confusion_matrices = []

for model_name, model in model_dict.items():

   print(model_name, model)

   # i = 1
   # model = MODELS[1]
   start_time = datetime.now()
   model_params = grid_param_dict[model_name]
   full_fit_model = model

   if model_params is None:

       # no CV grid search for NB
       full_fit_model.fit(train_features, train_labels)

       fit_models.append(full_fit_model)
       best_params_list.append(None)

   else:

       cv_model = GridSearchCV(model, model_params, cv=skf,
n_jobs=-1, verbose=2)
       cv_model.fit(dev_train_features, dev_train_labels)

       # get the best CV parameters
       best_params = cv_model.best_params_
       best_params_list.append(best_params)
```

```
        # make dev test predictions & calculate scores
        dev_test_pred = cv_model.predict(dev_test_features)
        dev_test_scores = calculate_scores(dev_test_labels,
dev_test_pred)
        dev_test_metrics.append(dev_test_scores)
        print(dev_test_scores)

        print('fit model to full training set')
        full_fit_model.set_params(**best_params)
        full_fit_model.fit(train_features, train_labels)
        get_duration_hours(start_time)

    fit_models.append(full_fit_model)

    # make test predictions
    test_pred = full_fit_model.predict(test_features)
    # test_class_probabilities =
full_fit_model.predict_proba(test_features)

    test_predictions.append(test_pred)
    # class_probabilities_list.append(test_class_probabilities)

    # get the classification report
    class_report = classification_report(test_labels, test_pred)
    classification_reports.append(class_report)
    print(class_report)

    # get confusion matrix
    confusion_df = pd.DataFrame(confusion_matrix(test_labels,
test_pred),
                                columns=["Predicted Class " +
str(class_name) for class_name in class_labels],
                                index=["Class " + str(class_name)
for class_name in class_labels])
    confusion_matrices.append(confusion_df)
    print(confusion_df)

    # get scores
```

```
    model_metrics = calculate_scores(test_labels, test_pred)

    print(METRIC_NAMES)
    print(model_metrics)
    test_metrics.append(model_metrics)

    # get training duration
    hours = get_duration_hours(start_time)
    model_durations.append(hours)

print('Create Metrics DF')

sklearn_models_df = pd.DataFrame(test_metrics,
columns=METRIC_NAMES, index=model_names)
sklearn_models_df['training_time'],
sklearn_models_df['best_params'] = model_durations,
best_params_list
print(sklearn_models_df.iloc[:, :4])

print('Save the results')

# create output directory if it doesn't exist
output_directory = input_file_name[:
input_file_name.find('.csv')]
if not os.path.exists(output_directory):
    os.makedirs(output_directory)

# create output filename
output_file_name = output_directory + '-results.csv'

os.chdir(output_directory)
with open('sklearn_models.pkl', 'wb') as f:
    pickle.dump(fit_models, f)

with open('sklearn_test_predictions.pkl', 'wb') as f:
        pickle.dump(test_predictions, f)

with open('sklearn_class_probabilities.pkl', 'wb') as f:
    pickle.dump(class_probabilities_list, f)
```

```
with open('sklearn_confusion_matrices.pkl', 'wb') as f:
   pickle.dump(confusion_matrices, f)

# save the model metrics
sklearn_models_df.reset_index().to_csv(output_file_name,
index=False)
```

04-train-non-cv-models.py

```
# !/usr/bin/env/ python3
# -*- coding: utf-8 -*-

"""
Created on Apr 27, 2019
@author: Kyle Gilde

This script is used to train non-CV versions of the models.

This script takes the feature set outputs of
03-create-attribute-comparison-features.py
and 03-create-single-doc-similarity-features.py

"""

import os

import numpy as np
import pandas as pd
import pickle
from utility_functions import *

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score,
f1_score
```

```
from sklearn.naive_bayes import GaussianNB #alpha smoothing?
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 250)

DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
# DATA_DIRECTORY = '//files/share/goods/OI Team'
os.chdir(DATA_DIRECTORY)

# global variables
RANDOM_STATE = 5
FOLDS = 2
DEV_TEST_SIZE = .5
OFFER_PAIR_COLUMNS = ['offer_id_1', 'offer_id_2', 'filename',
'dataset', 'label', 'file_category']
METRIC_NAMES = ['Precision', 'Recall', 'F1_score']

# list of models to fit
MODELS = [GaussianNB(),
         SVC(random_state=RANDOM_STATE, class_weight='balanced',
verbose=2),
         RandomForestClassifier(random_state=RANDOM_STATE,
n_estimators=1000, class_weight='balanced', verbose=2),
         GradientBoostingClassifier(random_state=RANDOM_STATE,
n_estimators=300, n_iter_no_change=30, verbose=2)]

MODELS = [RandomForestClassifier(random_state=RANDOM_STATE,
n_estimators=1000, class_weight='balanced', verbose=2),
         GradientBoostingClassifier(random_state=RANDOM_STATE,
n_estimators=300, n_iter_no_change=30, verbose=2)]

model_names = [model.__class__.__name__ for model in MODELS]
model_dict = dict(zip(model_names, MODELS))
```

```
# provide input file
input_file_name = 'single_doc_similarity_features-100.csv'
#'attribute_comparison_features-9.csv' #
'symbolic_single_doc_similarity_features-9.csv' #input('Input
the features file')
assert input_file_name in os.listdir(), 'An input file is
missing'

# read input file
symbolic_similarity_features =
reduce_mem_usage(pd.read_csv(input_file_name))
print(symbolic_similarity_features.columns.tolist())

# get the train & test indices
train_indices, test_indices =
symbolic_similarity_features.dataset.astype('object').apply(lamb
da x: x == 'train').values,\

symbolic_similarity_features.dataset.astype('object').apply(lamb
da x: x == 'test').values

# get the labels
all_labels = symbolic_similarity_features.label
train_labels, test_labels = all_labels[train_indices],
all_labels[test_indices]
class_labels = np.sort(all_labels.unique())

# create features df
symbolic_similarity_features.set_index(OFFER_PAIR_COLUMNS,
inplace=True)

print(symbolic_similarity_features.columns.tolist())
print(symbolic_similarity_features.info())
print(symbolic_similarity_features.shape)
print(symbolic_similarity_features.describe())

# center and scale for SVM
scaler = StandardScaler()
```

```
symbolic_similarity_features =
scaler.fit_transform(symbolic_similarity_features)

# train and test features
train_features, test_features =
symbolic_similarity_features[train_indices, :],\

symbolic_similarity_features[test_indices, :]

# create dev test and train sets
dev_train_features, dev_test_features, dev_train_labels,
dev_test_labels =\
   train_test_split(train_features, train_labels,
test_size=DEV_TEST_SIZE, random_state=RANDOM_STATE)

print('Dev Train Feature Shape')
print(dev_train_features.shape)

# output DF
test_metrics = []
model_durations = []
best_params_list = []

# save diagnostics
test_predictions = []
class_probabilities_list = []
fit_models = []
classification_reports = []
confusion_matrices = []

for model_name, model in model_dict.items():

   print(model_name, model)

   start_time = datetime.now()

   model.fit(train_features, train_labels)
   test_pred = model.predict(test_features)
   test_predictions.append(test_pred)
```

```
    # get scores
    model_metrics = [precision_score(test_labels, test_pred),
                     recall_score(test_labels, test_pred),
                     f1_score(test_labels, test_pred)]

    print(METRIC_NAMES)
    print(model_metrics)
    test_metrics.append(model_metrics)

    fit_models.append(model)

    # get training duration
    hours = get_duration_hours(start_time)
    model_durations.append(hours)

print(model_durations)

sklearn_models_df = pd.DataFrame(test_metrics,
columns=METRIC_NAMES, index=model_names)
sklearn_models_df['training_time'] = model_durations
print(sklearn_models_df.iloc[:, :4])
```

05-tune-individual-models.py

```
# !/usr/bin/env/ python3
# -*- coding: utf-8 -*-

"""
Created on Apr 27, 2019
@author: Kyle Gilde

This script is used to more extensively tune and train CV
versions of the models.

This script takes the feature set outputs of
03-create-attribute-comparison-features.py
and 03-create-single-doc-similarity-features.py
```

```
"""

import os
import gc

import numpy as np
import pandas as pd
import pickle
from utility_functions import *

from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import train_test_split,
GridSearchCV, StratifiedKFold, RandomizedSearchCV,
cross_val_score
from sklearn.metrics import classification_report,
confusion_matrix, precision_score, recall_score, f1_score,
make_scorer

from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier,
GradientBoostingClassifier


# global variables
DATA_DIRECTORY = 'D:/Documents/Large-Scale Product Matching/'
# DATA_DIRECTORY = '//files/share/goods/OI Team'
os.chdir(DATA_DIRECTORY)

RANDOM_STATE = 5
FOLDS = 3
DEV_TEST_SIZE = .7
METRIC_NAMES = ['Precision', 'Recall', 'F1_score']
OFFER_PAIR_COLUMNS = ['offer_id_1', 'offer_id_2', 'filename',
'dataset', 'label', 'file_category']

# list of models to fit
```

```
MODELS = [RandomForestClassifier(random_state=RANDOM_STATE,
class_weight='balanced', verbose=2),
         GradientBoostingClassifier(random_state=RANDOM_STATE,
n_iter_no_change=30, verbose=2)]

model_names = [model.__class__.__name__ for model in MODELS]
model_dict = dict(zip(model_names, MODELS))

# set display options
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 500)
pd.set_option('display.max_colwidth', 250)

# provide input file
input_file_name = 'attribute_comparison_features-7.csv' #
input('Input the features file')
assert input_file_name in os.listdir(), 'An input file is
missing'

# read input file
symbolic_similarity_features =
reduce_mem_usage(pd.read_csv(input_file_name))
print(symbolic_similarity_features.columns.tolist())

# get the train & test indices
train_indices, test_indices =
symbolic_similarity_features.dataset.astype('object').apply(lamb
da x: x == 'train').values,\

symbolic_similarity_features.dataset.astype('object').apply(lamb
da x: x == 'test').values

# get the labels
all_labels = symbolic_similarity_features.label
train_labels, test_labels = all_labels[train_indices],
all_labels[test_indices]
class_labels = np.sort(all_labels.unique())
```

```
# create features df
symbolic_similarity_features.set_index(OFFER_PAIR_COLUMNS,
inplace=True)

print(symbolic_similarity_features.columns.tolist())
print(symbolic_similarity_features.info())
print(symbolic_similarity_features.shape)
print(symbolic_similarity_features.describe())

# center and scale for SVM
scaler = StandardScaler()
symbolic_similarity_features =
scaler.fit_transform(symbolic_similarity_features)

# n_features = symbolic_similarity_features.shape[1]

# train and test features
train_features, test_features =
symbolic_similarity_features[train_indices, :],\

symbolic_similarity_features[test_indices, :]

dev_train_features, dev_test_features, dev_train_labels,
dev_test_labels =\
   train_test_split(train_features, train_labels,
test_size=DEV_TEST_SIZE, random_state=RANDOM_STATE)

print('Dev Train Feature Shape')
print(dev_train_features.shape)

rf_grid_params = {'max_depth': [None, 10],
                  'max_features': ['auto', None],
                  'max_leaf_nodes': [None],
                  'min_impurity_decrease': [0.0, .1],
                  'min_impurity_split': [None],
                  'min_samples_leaf': [1, 5],
                  'min_samples_split': [2, 5],
                  'min_weight_fraction_leaf': [0.0],
                  'n_estimators': [1000, 1500]}
```

```
count_cv_models(rf_grid_params, FOLDS, .21)

gbm_grid_params = {'learning_rate': [.01, .025],
                   'max_depth': [None, 10],
                   'max_features': ['auto', None],
                   'max_leaf_nodes': [None, 10],
                   'min_impurity_decrease': [0.0, .1],
                   'min_samples_leaf': [1, 5],
                   'min_samples_split': [2, 5],
                   'min_weight_fraction_leaf': [0.0],
                   'n_estimators': [150, 300, 450],
                   'subsample': [.15, .33],
                   'warm_start': [True, False]}

count_cv_models(gbm_grid_params, FOLDS, .07)

grid_param_list = [rf_grid_params, gbm_grid_params]
grid_param_dict = dict(zip(model_names, grid_param_list))

# create stratified folds
skf = StratifiedKFold(n_splits=FOLDS, random_state=RANDOM_STATE)

# output DF
test_metrics = []
dev_test_metrics = []
model_durations = []
best_params_list = []

# save diagnostics
test_predictions = []
class_probabilities_list = []
fit_models = []
classification_reports = []
confusion_matrices = []

for model_name, model in model_dict.items():

   print(model_name, model)
```

```
    start_time = datetime.now()
    model_params = grid_param_dict[model_name]
    full_fit_model = model

    if model_params is None:

        # no CV grid search for NB
        full_fit_model.fit(train_features, train_labels)

        fit_models.append(full_fit_model)
        best_params_list.append(None)

    else:

        cv_model = GridSearchCV(model, model_params, cv=skf,
n_jobs=-1, verbose=2)
        cv_model.fit(dev_train_features, dev_train_labels)

        # get the best CV parameters
        best_params = cv_model.best_params_
        best_params_list.append(best_params)

        # make dev test predictions & calculate scores
        dev_test_pred = cv_model.predict(dev_test_features)
        dev_test_scores = calculate_scores(dev_test_labels,
dev_test_pred)
        dev_test_metrics.append(dev_test_scores)
        print(dev_test_scores)

        print('fit model to full training set')
        full_fit_model.set_params(**best_params)
        full_fit_model.fit(train_features, train_labels)
        get_duration_hours(start_time)

    fit_models.append(full_fit_model)

    # make test predictions
    test_pred = full_fit_model.predict(test_features)
```

```
    # test_class_probabilities =
full_fit_model.predict_proba(test_features)

    test_predictions.append(test_pred)
    # class_probabilities_list.append(test_class_probabilities)

    # get the classification report
    class_report = classification_report(test_labels, test_pred)
    classification_reports.append(class_report)
    print(class_report)

    # get confusion matrix
    confusion_df = pd.DataFrame(confusion_matrix(test_labels,
test_pred),
                               columns=["Predicted Class " +
str(class_name) for class_name in class_labels],
                               index=["Class " + str(class_name)
for class_name in class_labels])
    confusion_matrices.append(confusion_df)
    print(confusion_df)

    # get scores
    model_metrics = calculate_scores(test_labels, test_pred)

    print(METRIC_NAMES)
    print(model_metrics)
    test_metrics.append(model_metrics)

    # get training duration
    hours = get_duration_hours(start_time)
    model_durations.append(hours)

print('Create Metrics DF')

sklearn_models_df = pd.DataFrame(test_metrics,
columns=METRIC_NAMES, index=model_names)
sklearn_models_df['training_time'],
sklearn_models_df['best_params'] = model_durations,
best_params_list
```

```python
print(sklearn_models_df.iloc[:, :4])

print('Save the results')

# create output directory if it doesn't exist
output_directory = input_file_name[:
input_file_name.find('.csv')]
if not os.path.exists(output_directory):
    os.makedirs(output_directory)

# create output filename
output_file_name = output_directory + '-tuned-results.csv'

os.chdir(output_directory)
with open('sklearn_models.pkl', 'wb') as f:
    pickle.dump(fit_models, f)

with open('sklearn_test_predictions.pkl', 'wb') as f:
        pickle.dump(test_predictions, f)

with open('sklearn_class_probabilities.pkl', 'wb') as f:
    pickle.dump(class_probabilities_list, f)

with open('sklearn_confusion_matrices.pkl', 'wb') as f:
    pickle.dump(confusion_matrices, f)

# save the model metrics
sklearn_models_df.reset_index().to_csv(output_file_name,
index=False)
```

json_parsing_functions.py

```python
# !/usr/bin/env/ python3
# -*- coding: utf-8 -*-

"""
Created on Feb 10, 2019
@author: Kyle Gilde
```

```
These are the functions that are used in the
01-parse-json-to-dfs.py script

"""


import os
from urllib.parse import urlparse
import numpy as np
import pandas as pd

# functions for reading the test and train offer pairs

def read_train_test_files(file_dir, sep='#####',
col_names=['offer_id_1', 'offer_id_2', 'label']):
   """
   Read all files from the director & use the file name for the
category column

   :param file_dir: a director
   :param delimitor: default is #####
   :return:
   """
   original_wd = os.getcwd()
   os.chdir(file_dir)
   files = os.listdir()

   df_list = []
   for file in files:
       df = pd.read_csv(file, names=col_names, sep=sep,
engine='python')
       df['filename'] = file
       if 'train' in file:
           df['dataset'] = 'train'
       elif 'gs_' in file:
           df['dataset'] = 'test'
       df_list.append(df)
   os.chdir(original_wd)
```

```
    return reduce_mem_usage(pd.concat(df_list, axis = 0,
ignore_index=True))


def rbind_train_test_offers(train_df, test_df):
    """
    Combines the test and train dfs

    :param train_df:
    :param test_df:
    :return:
    """
    # row bind all the offer ids
    df1, df2, df3, df4 = test_df[['offer_id_1', 'filename',
'dataset']].rename(columns={'offer_id_1': 'offer_id'}),\
        test_df[['offer_id_2', 'filename',
'dataset']].rename(columns={'offer_id_2': 'offer_id'}),\
        train_df[['offer_id_1', 'filename',
'dataset']].rename(columns={'offer_id_1': 'offer_id'}),\
        train_df[['offer_id_2', 'filename',
'dataset']].rename(columns={'offer_id_2': 'offer_id'})

    # Aggregate the IDs
    train_test_offer_ids = pd.concat([df1, df2, df3, df4],
axis=0, ignore_index=True, sort=False)\
        .drop_duplicates().set_index('offer_id')

    return train_test_offer_ids

# functions for parsing the massive json file

def get_cluster_ids(json_file='D:/Documents/Large-Scale Product
Matching/clusters_english.json'):
    """
    Get cluster IDs for the 4 categories in the train-test sets
    :param json_file:
    :return:
    """
    if 'category_cluster_ids.csv' in os.listdir():
```

```
        category_cluster_ids =
reduce_mem_usage(pd.read_csv('category_cluster_ids.csv',

index_col='cluster_id'))
    else:
        product_categories_df =
reduce_mem_usage(pd.read_json(json_file, lines=True))


        category_cluster_ids = product_categories_df\
            .rename(columns={'id':'cluster_id'})\

.set_index('cluster_id').loc[product_categories_df.category.isin
(TRAIN_TEST_CATEGORIES).values, ['category']]

        print(category_cluster_ids.info(memory_usage='deep'))

        # get only the cluster ids needed for the train-test
categories
        category_cluster_ids.to_csv('category_cluster_ids.csv')

        return category_cluster_ids


def merge_nan_rows(df):
    """
    Merges rows with NaNs into one
    It's passed the parse_json_column() function
    :param df: a DataFrame
    :return: one row without NaNs
    """
    try:
        s = df.apply(lambda x: x.dropna().max())
        return pd.DataFrame(s).transpose()
    except Exception as e:
        print('merge_nan_rows', e)


def parse_json_column(a_series):
```

```
    """
    Parses the remaining JSON into a DF
    :param a_series: a pandas Series
    :return: a DF
    """

    # Concatenate DFs and remove the beginning & ending brackets
    df = pd.concat([merge_nan_rows(x) for x in a_series],
sort=True)\
        .apply(lambda x: x.str.replace('^\[|\]$', ''))
    # clean column names
    df.columns = df.columns.str.strip('/')

    return df


def coalesce_gtin_columns(df):
    """
    Since a product can have only one gtin,
    this function coalesces these columns to one column
    :param df:
    :return:
    """
    # select the gtin columns
    gtin_df = df.filter(regex='gtin')
    gtin = gtin_df.iloc[:, 0]
    if len(gtin_df.columns) > 1:
        # start the loop on the 2nd column
        for col in gtin_df.columns[1:]:
            gtin = gtin.mask(pd.isnull, gtin_df[col])
    df['gtin'] = gtin
    df.drop(gtin_df.columns, axis=1, inplace=True)
    return df


def parse_domain(url):
    """

    :param url:
```

```
    :return:
    """
    return urlparse(url).netloc


def parse_price_columns(df, price_column_names):
    """

    :param price_series:
    :return:
    """
    price_columns =
df.columns[df.columns.isin(price_column_names)]

    for price_column in price_columns:
        df[price_column] = df[price_column]\
            .str.replace(r'[a-zA-Z",]+', '')\
            .str.strip()\
            .str.replace(r' (\d\d)$', r'.\1')\
            .str.replace(r'\s', '')\
            .apply(lambda x: pd.to_numeric(x, errors='coerce',
downcast='integer'))

    return df


def coalesce_parent_child_columns(df):
    """

    :param df:
    :return:
    """
    parent_columns =
df.columns[df.columns.astype(str).str.startswith('parent_')]
    nonparent_columns =
df.columns[~df.columns.isin(parent_columns)]

    child_parent_pairs = []
    for parent_column in parent_columns:
```

```
        for nonparent_column in nonparent_columns:
            if parent_column.endswith('_' + nonparent_column):
                child_parent_pairs.append((parent_column,
nonparent_column))

    for child_parent_pair in child_parent_pairs:
        parent, child = child_parent_pair
        df[child] = df[child].combine_first(df[parent])
        df.drop(parent, axis=1, inplace=True)

    return df


def create_file_categories(df):
    """
    Creates a df column for the test and train category

    :param df: the df containing a column called filename
    :return: the df with a column called file_category
    """
    if 'filename' in df.columns:

        file_categories = ['computers', 'cameras', 'watches',
'shoes']

        for file_category in file_categories:
            df.loc[df['filename'].str.contains(file_category),
'file_category'] = file_category

        return df
    else:
        print('The df does not have a column called filename')


def pairwise_cosine_dist_between_matrices(a, b):
    """

    :param a:
    :param b:
```

```
    :return:
    """
    cosine_matrix = np.dot(a, b.T) / \
                    np.dot(np.sqrt(np.dot(a, a.T).diagonal()),
                           np.sqrt(np.dot(b, b.T).diagonal()).T)
    # the truncated SVD creates some slightly negative values in
the calculation
    # change these to zero
    return pd.Series(np.maximum(cosine_matrix.diagonal(), 0))
```

utility_functions.py

```
# !/usr/bin/env/ python3
# -*- coding: utf-8 -*-

"""
Created on Feb 10, 2019
@author: Kyle Gilde

These are some functions that are used in several of the
scripts.

"""
from datetime import datetime
import numpy as np
import pandas as pd
from pandas.api.types import is_numeric_dtype, is_string_dtype

from sklearn.metrics import precision_score, recall_score,
f1_score


def reduce_mem_usage(df, n_unique_object_threshold=0.30):
    """

    Converts the data type when possible in order to reduce
memory usage
```

```
    source:
https://www.kaggle.com/arjanso/reducing-dataframe-memory-size-by
-65
    :param df: a DataFrame
    :return: returns a smaller df if possible
    """
    print("___________________________________")
    start_mem_usg = df.memory_usage().sum() / 1024**2
    print("Starting memory usage is %s MB" %
"{0:}".format(start_mem_usg))
    # record the dtype changes
    dtype_df = pd.DataFrame(df.dtypes.astype('str'),
columns=['original'])

    for col in df.columns:
        if is_numeric_dtype(df[col]):
            # make variables for max, min
            mx, mn = df[col].max(), df[col].min()
            # If no NaNs, proceed to reduce the int
            if np.isfinite(df[col]).all():
                # test if column can be converted to an integer
                as_int = df[col].astype(np.int64)
                delta = (df[col] - as_int).sum()

                # Make Integer/unsigned Integer datatypes
                if delta == 0:
                    if mn >= 0:
                        if mx < 255:
                            df[col] = df[col].astype(np.uint8)
                        elif mx < 65535:
                            df[col] = df[col].astype(np.uint16)
                        elif mx < 4294967295:
                            df[col] = df[col].astype(np.uint32)
                        else:
                            df[col] = df[col].astype(np.uint64)
                    else:
                        if mn > np.iinfo(np.int8).min and mx <
np.iinfo(np.int8).max:
                            df[col] = df[col].astype(np.int8)
```

```
                        elif mn > np.iinfo(np.int16).min and mx <
np.iinfo(np.int16).max:
                            df[col] = df[col].astype(np.int16)
                        elif mn > np.iinfo(np.int32).min and mx <
np.iinfo(np.int32).max:
                            df[col] = df[col].astype(np.int32)
                        elif mn > np.iinfo(np.int64).min and mx <
np.iinfo(np.int64).max:
                            df[col] = df[col].astype(np.int64)

                # Make float datatypes 32 bit
                else:
                    df[col] = df[col].astype(np.float32)

                # Print new column type
                # print("dtype after: ", df[col].dtype)
        elif is_string_dtype(df[col]):
            if df[col].astype(str).nunique() / len(df) <
n_unique_object_threshold:
                df[col] = df[col].astype('category')

    # Print final result
    dtype_df['new'] = df.dtypes.astype('str')
    dtype_changes = dtype_df.original != dtype_df.new

    if dtype_changes.sum():
        print(dtype_df.loc[dtype_changes])
        new_mem_usg = df.memory_usage().sum() / 1024**2
        print("Ending memory usage is %s MB" %
"{0:}".format(new_mem_usg))
        print("Reduced by", int(100 * (1 - new_mem_usg /
start_mem_usg)), "%")

    else:
        print('No reductions possible')

    print("------------------------------------")
    return df
```

```
def calculate_percent_nulls(df, print_series=True,
return_series=False):
    """

    Counts the NaNs by column

    :param df: a Pandas dataframe
    :param print_series: print statement
    :param return_series: print statement
    :return: a series
    """
    percentages = df.isnull().sum() / len(df) * 100
    percentages_sorted = percentages.sort_values(ascending=False)

    if print_series:
        print(percentages_sorted)

    if return_series:
        return(percentages_sorted)


def get_duration_hours(start_time):
    """
    Prints and returns the time difference in hours

    :param start_time: datetime object
    :return: time difference in hours
    """
    time_diff = datetime.now() - start_time
    time_diff_hours = time_diff.seconds / 3600
    print('hours:', round(time_diff_hours, 2))
    return time_diff_hours


def count_words(s):
    """

    Counts the words in Series of text
```

```
    :param s: a Pandas object Series
    :return: a Series containing the respective word counts
    """
    return s \
        .str.split(expand=True) \
        .apply(lambda x: np.sum(pd.notnull(x)), 1) \
        .sort_values(ascending=False)


def count_cv_models(param_dict, folds,
est_hours_per_model=None):
    """

    Measures and prints how many models will be fit given the
search parameters and CV folds

    :param param_dict: a dictionary containing lists of
parameters to search
    :param folds: the number of CV folds
    :param est_hours_per_model: optional, if provided, it will
print a time estimate
    :return: None
    """

    if param_dict is not None:
        n_models = np.prod([len(v) for v in param_dict.values()])
* folds
        print("models: ", n_models)
    else:
        print("No parameters")

    if est_hours_per_model is not None:
        print("est. hours: ", n_models * est_hours_per_model)


def calculate_scores(test_labels, test_pred):
    """
```

```
    Calculates the precision, recall and F1 for the actuals and
predictions

    :param test_labels: the actual labels
    :param test_pred: the predictions
    :return: a list containing the precision, recall and F1
    """
    return [precision_score(test_labels, test_pred),
            recall_score(test_labels, test_pred),
            f1_score(test_labels, test_pred)]
```