Task1a

For product comparison, I first load two csv file into data frames and replace all na into “” for later matching (pre-processing). I take a nested loop where for each product in amazon table, compute similarity for all products in google table and find assume the one with max similarity as matched product. The similarity function I use is the Jaccard Similarity. It is a token-based function, where more the number of common tokens, more is the similarity between the sets. In mine case, the string of name/title can be transformed into sets of words by splitting using a delimiter (default=space) Note, here tokens of different length have equal importance. This is the first reason why I choose Jaccard instead of other edit-based methods such as hamming. Since I need to do fuzzy match for string, a combination of words, it is better to value a word as a unit instead of an index character, where Jaccard’s rationale exactly matches my demand.

Secondly, by finding intersection between sets of words, it overcomes the shortage in approaches which depend on text distance and string size. For instance, if two names respectively from amazon and google table are of different size, result similarity from hamming distance may greatly decreased due to more places the string vary. Also, sequence-based methods would not be suitable since it finds the longest common substring from two strings, where order of words in a string matter.

Final scoring function consists of 3*similarity on “name/title” + 1*similarity on “price”. Instead of taking all features into account, strings of “description” and “manufacturer” are both of large sizes and so have overall low similarity since denominators are large. Moreover, there are many missing values in “manufacturer” and recorded ones coincide frequently, which makes this feature less reliable. Unselect this feature avoids overfitting. The similarity on “name” is weight more than “price”, as the string of name is more unique and homogeneous for a product compared to price.

The matched product is acknowledged only if its similarity is greater than the threshold. In this case, I set threshold = 1.2 as the possible maximum score is 4 in total, product records with field values that are less than 30% similar are unlikely to be assigned to the same cluster. It serves for limiting a minimal similarity among all matched products, as there may be unmatched products with low similarity being incorrectly classified as matched product.

The overall performance is pretty good with Recall of 0.9 and Precision of 0.9, which shows 90% matched records are found and 90% of the records my algorithm identified as matched actually are matched. It could be improved by lowering the value of threshold score, allow more potential records being recalled. Yet more records mean greater probability of match records which are not accurate, hence lowering precision.

Task1b

Since all features except price are of type string and approximately matched by similarity function, the future “price” of type float as block key is more direct and convenient. In addition, there are fewer missing values (price 0) in “price” compared to “manufacturer”. Those missing values may lead to lots of false matches so that the feature “price” has a lower error probability compared to others.

The process starts with pre-processing the data, unifying prices in both tables to be AUD (price in google table is in GBP). Then I discretised the data by cutting the continuous variable price into bins. Since prices in higher level of ranges have a greater difference, equal width method will result in a bad distribution. In order to control the number of records within a block, I manually let price falls in larger-value interval be cut by a larger value per bin. For example, prices that are ultra-high (fall in range from 10000AUD to 60000AUD) are cut 5000AUD per bin, where prices that are low (0 to 500AUD) are only cut 20 AUD per bin. After all bins have been but, I allocate prices into responding blocks and the blocking is done.

The result I receive for this is a Pair Completeness (PC) of 0.63692308 and a Reduction Ratio (RR) of 0.8539803385825. These two percentages are both relatively high, which shows this blocking method performs well. Yet the percentage of PC is relatively lower. This is because of small blocks (49 blocks compared to about 3000 records). Standard blocking trades off pairs completeness with reduction ratio performance as the number of blocks b increases. More smaller blocks results in less comparisons (RR increases) but more true match pairs are missed. (PC decreases) Since blocking emphasis more on efficiency (RR), it is reasonable to have a relatively low PC. To improve, the first and foremost solution is to select blocking keys and intervals more carefully so that records could lie more evenly. Instead of cutting bins by different intervals for approximately the same number of records, add additional algorithms to cut bins of equal frequency. There could also be additional matching algorithms worded together with blocking methods to get the right decisions, such as EM algorithm, which find maximum-likelihood estimates for model parameters when your data is incomplete and could be implemented when price = 0.