**Report on Part 1A - Product Comparison Algorithm:** Split into 3 components.

1. **Parsing the csv files:** The csv files are converted to Pandas dataframe and iterated by rows to extract the data required for preprocessing. A product's 'title' ('name' in *google.csv*), 'description' and 'manufacturer' are explicitly casted to string type (to avoid type errors, as the preprocessing function treats its parameter as string type) and passed into preprocessing(string) for preprocessing. The result is concatenated into a single, cleaned-up list of tokens.
2. **Preprocessing raw data:** preprocessing(string) treats its parameter as string type. The function uses several string methods and data cleaning techniques to return a list of cleaned-up tokens. The list of tokens satisfy the following criteria:
   - Does not contain numbers / single letters / punctuation / whitespaces. (achieved via. translate() and split() string methods)
   - Does not contain uppercase letters. (achieved via. casefold() string method)
   - Is tokenised. (achieved via. tokenizer.tokenize())
   - Does not contain stopwords. (achieved via. stopwords.words('english') from nltk)
   - Is stemmed. (achieved via. PorterStemmer.stem() from nltk library)
3. **Scoring similarity between two products:** The metrics used to score similarity are two token-based algorithms (Sorensen-Dice Coefficient[1], Cosine Similarity[2]), and a sequence-based algorithm (Ratcliff-Obershelp Similarity[3]). An average of the three is taken. Token-based algorithms are chosen as main comparison method between keywords is prioritised in this situation, since products normally sprinkle keywords throughout their data to increase their appearance frequency in recommender systems. Additionally, one sequence-based algorithm is chosen to consider the common grammar between titles. Edit-based algorithms are forfeited as they are inefficient for long strings. For each Amazon record, a most-similar Google record is found. A threshold of 0.3 is implemented after trial-n-error. If the similarity score between an Amazon record and its most-similar Google record exceeds threshold, it is added to the result.

**Overall Performance:** My algorithm successfully classified 119 matches, out of 130 matches that my algorithm **should have** classified as True Positive, which evaluates to a Recall of ~92%. In terms of Precision, 119 matches are relevant (True Positive) out of 144 matches that my algorithm classified as positive, which evaluates to a Precision of ~83%.

**Improvement Opportunities:** Implementing weighted score between the string similarity metrics instead of taking average, especially since token-based algorithms are conceptually similar. Measuring similarities between n-grams or bag of words for each product may also improve Recall and Precision, but is not implemented due to its computational cost since every Amazon record is compared to every Google record in Part 1A.

**Report on Part 1B - Blocking Implementation:** Similar to Part 1A, the blocking implementation is also split into 3 components.

1. **Parsing the csv files:** Same as above. The resulting list of tokens will be used to allocate each row into appropriate blocks later.
2. **Preprocessing raw data:** The preprocessing(string) function is identical to Part 1A.
3. **Allocating product to appropriate block:**

**Firstly**, the list of cleaned-up tokens that represents each product is passed into nltk.bigrams() to find its combination of bigrams.

**Secondly**, This list of bigrams is FreqDist from nltk library is used to find the frequency of each bigram. Bigrams are chosen as unigrams do not take relative order of tokens into consideration, and higher order n-grams cause the data to be overly sparse. This is an indicator of how the total number of bigrams are distributed across the vocabulary items that describe this product. **Thirdly**, a THRESHOLD is set for finding a number of most frequently-occurring bigrams. This threshold is set to 10 after trial-n-error and resulted in an ideal level of tradeoff between speed (execution time ~0.003s) and accuracy of blocking (PC and RR both 90%+). **Finally**, each product is allocated into THRESHOLD number of blocks, with each bigram as a block key.

**Overall Performance:** Since all blocks with one of the top 10 most common bigram in each record as block key would contain that specific record, this method is highly effective in allocating records with similar bigram frequency to the same block. This feature of my blocking method contributes to my 90%+ Pair Completeness as the number of True Positive pairs is very high in comparison to False Negative pairs (making denominator only marginally larger than numerator in PC formula). Although the distribution of records are uneven amongst the blocks in my blocking method (many records can possibly have similar top 10 bigrams), even the largest block (with block key = *'encor softwar'*) merely produce 3140 comparisons, which is still an optimistic reduction from the worst-case scenario of comparing every single Amazon record to every single Google record (3226 * 1363 = 4,397,038 comparisons). The very-high True Negative count (4356705) plays a major role in offsetting my Reduction Ratio to reach an ideal 99%+.

[1] **Sorensen-Dice Coefficient:** 2*Number of common tokens, divided by number of total tokens. Dice always overestimates similarity since its denominator does not remove duplicate tokens like Jaccard Index formula.
[2] **Cosine Similarity:** Measures the cosine of the angle betweem two tokens in their vector forms. Essentially how close they are on a vector plane.
[3] **Ratcliff-Obershelp Similarity:** 2*Number of common characters / Total number of characters in both strings. The common characters are defined as the longest common substring + recursively the number of common characters in the non-matching regions on both sides of the longest common substring.

**Improvement Opportunities:** Potential opportunities for improvement include having a second check after the initial allocation. This second check can utilise other string similarity metrics, such as Jaccard Index and Cosine Similarity, to evaluate the similarity between strings in the same block. If this similarity is lower than a certain threshold, the blocking may need to be reconsidered. However, this idea may not align with the linear-time requirement of Part 1B, hence was not implemented.