Project 2 Report

Hoang Dang 1080344

Task 1a. The algorithm takes the data into *DataFrame*'s and cleans the data by removing all characters that are not numerical or alphabetical and enforces all characters to lowercase. It was manually determined that *abt'*s name field always starts with the brand and ends with the serial number of the product. The *buy* data is not always nicely formatted, however, most of this data is in a similar format to *abt*. Therefore, the program assumes that the first and last words in a product name are brands and serial numbers respectively and checks all the data for cases that perfectly match both in brand and serial number. In the case of *buy*, the manufacturer is also checked if brand is not a match. After that, all matched products are removed from the tables, and a second check is carried out to find as many edge cases as possible. These checks include: shortening the length of the current serial number by 1 character to allow for small errors in serial numbers; finding all other substrings inside the data that may potentially be serial numbers and cross checking; and using *fuzzywuzzy*'s *token_set_ratio()* to check the names and descriptions for at least 90% matching in wording. The condition for edge cases is that the brands must first match, and then any of the above conditions satisfied, as the probability of such intersection being a false positive is low.

Evaluation. Overall, the algorithm performs well with > 0.9 Precision and Recall, but still struggles to cover all the edge cases where data is not well-formatted. To improve performance, future implementations can sacrifice storage for better language processing. In particular, if two products are matched, the brand and serial number may be kept in dictionaries to help with brand-identification in edge cases and help remove duplication of products with the same serial number. The algorithm can also implement blocking where products that don't match on brand and serial number can be put into blocks, then compared on the basis of noun and/or adjective matching using the *nltk* library. If the products match at least 90% for noun and/or adjective matching, as well as some matching of unrecognised substrings (ie. 1080p, 48GB) – as these usually help distinguish products – there is a good probability of this matching being a true positive.

Task 1b. This script cleans data similarly to part 1a, leaving only alphabetical and numerical characters and enforcing all data into lowercase. The blocking method is primarily based on the insight that brand information is usually the first substring in the product name. Hence, the program assumes that the first words in the name field of all products are brands and puts all products of the same brand into one block. Because the *abt* data generally has better formatting, the algorithm starts by taking all the brands from *abt* and assigning blocks; then it iterates through *buy* and checks to see if the brand of the *buy* product is already part of the dictionary of blocks. In the case of *buy*, if the brand doesn't match any existing blocks, the manufacturer is also checked for matches. If no existing blocks can be found, new blocks are also assigned for *buy*.

Evaluation. This algorithm works well for its purpose, achieving almost perfect Pair Completeness and excellent Reduction Ratio. However, it still generates a substantial number of False Positive counts given the simplistic nature of the design. To further reduce this False Positive count and reduce potential build up in computational time in later processing, a more precise model should be implemented. To achieve better precision, current blocks can be further broken down into smaller categories by noun or adjective. Again, the use of the *ntlk* library may come in handy here. For example, a product with the brand 'Apple' may have blocks 'ApplePhone', 'AppleWatch', 'AppleCase', etc… The main issue with this approach is picking which substring to add to the brand for blocking, since picking the wrong substring would actually harm both Precision and Pair-Completeness. One simple strategy is to sort all the nouns of the product in ascending order and pick the first one in the list.

Note: In task 2a and task 2b, *DataFrame*'s are used wherever applicable. All random states are 200 wherever applicable (*train_test_split*, *DecisionTreeClassifier*, *LogisticRegression*, *KMeans*, *mutual_info_classif*, *PCA*)

Task 2a.

Pre-processing. The two data files are read into two pandas *DataFrame*'s and then merged in an inner join on 'Country Code'. It is then sorted in ascending order also on 'Country Code' and columns that are not features are dropped. The class feature is chosen to be 'Life expectancy at birth (years)', the rest of the features are used for analysis. The data is split using *sklearn*'s *train_test_split* function, giving 70% training and 30% test data. This produces a training set (X_train), a test set (X_test), a training class vector (y_train) and a test class vector (y_test). A *SimpleImputer* from *sklearn* is fitted with X_train for median imputation, then used to transform both X_train and X_test. The data is standardized using *sklearn*'s *StandardScaler* method. This method is fitted with X_train, then used to transform both X_train and X_test. The statistics from *SimpleImputer* and *StandardScaler* are used to produce the 'task2a.csv' file containing median, mean and variance for each feature.

Building and testing the model. The k-nn model is built using the *KNeighborsClassifier* class with *n_neighbors=7* for 7-nn and *n_neighbors=3* for 3-nn. In each iteration, the model is fitted with X_train, and used to predict X_test. This produces a y_predicted vector, which is compared to the y_test vector using *sklearn*'s *accuracy_score* function. The 3-nn model achieved a score of 0.673, while the 7-nn model achieved a score of 0.727. The decision tree model is built using *sklearn*'s *DecisionTreeClassifier* class. The model is constructed with *max_depth=3*. It is fitted with X_train and y_train and used to predict X_test; similarly, producing a y_predicted vector using its own prediction. This is compared to the y_test vector using *accuracy_score*. The decision tree achieved a score of 0.709.

Evaluation. The 7-nn model performed better than the decision tree, which in turn is better than the 3-nn model. It is not surprising that the 3-nn model performed the worst, since the initial high variances (as seen in task2a.csv) indicate that data points can be reasonably far apart, and the small k value makes the algorithm susceptible to a lot of noise. The decision tree may have performed better because decision trees make use of splitting up data to maximal impurity, thus variation does not affect the algorithm as much. Lastly, 7-nn is also less susceptible to random noise and variation. However, overall, the three models performed relatively similar to each other and the accuracy difference may be due to randomness.

Task2b.

Pre-processing. The data is taken in and pre-processed exactly the same as part 2a, with the exception of splitting data into 66% training and 34% test instead of 70/30, to increase the accuracy of model evaluation and avoid overfitting. This produces X_train, X_test, y_train, y_test. In the same procedure as task 2a, the data is median-imputed and standardized.

Feature Engineering.

K-Means: The k-Means model is tested for k in the range from 1 to 20 using *sklearn*'s *KMeans* class. In each iteration, the *KMeans* model is fitted with X_train and the sum of squared distances within clusters are calculated (*KMeans' inertia_* attribute). This information is plotted on a graph of *inertia_* against k values (see task2bgraph1). Using the elbow method for picking k, the point furthest away from the line connecting k=1 and k=20 is automatically chosen through algebraic calculations. With the optimal k chosen, *KMeans* is fitted again with X_train and used to produce a 'fclusterlabel' vector for X_train. Then 'fclusterlabel' is generated for X_test by iterating through each X_test data point and assigning it to the closest cluster centroid (using *KMeans' cluster_centers* attribute). In execution, the optimal k chosen by the program was 4.

Interaction term pairs: Interaction term pairs are calculated by multiplying each of the 20 features of the dataset with every other feature, this produces 190 (20C2) new features.

➔ At this stage the dataset has 211 features (191 generated).

Feature Selection: The first step of feature selection is removal of highly correlated features to reduce multicollinearity between the independent vectors. A correlation matrix is generated using *DataFrame*'s *corr()* function and the correlation threshold is chosen to be 0.9. Then a loop is used to find the feature that has the highest number of inter-correlations (above 0.9) and remove it, before re-calculating the correlation matrix and finding the next such feature to remove. This process continues until there are no longer any inter-correlations above 0.9. This procedure removed 129 highly correlated features during execution. The second step of feature selection is removal of unimportant features, in order to prune out features that do not contribute to the prediction model. This is calculated using *sklearn's LogisticRegression* (see task2bgraph2). Logistic regression is deemed suitable because the problem is a classification task where data has been standardized, which allows the regression coefficients to be considered as importance scores. The *LogisticRegression* model is given as a parameter to *sklearn*'s *SelectFromModel* function, this combination is then fitted with X_train and y_train. The importance threshold is left at default to be the mean of importance scores. This procedure further removed 42 unimportant features in execution.

➔ The dataset now has 40 features remaining out of the 211 features.

Three common classification procedures are tested to find the best 4 features of this dataset: ANOVA F, Chi-Square and Mutual Information; performed using *f_classif*, *chi2* and *mutual_info_classif* from *sklearn*. For the Chi2 analysis, the dataset is also scaled to the range 0-1 (using *MinMaxScaler* from *sklearn* fitted with X_train) to eliminate negative values in order to work with the *chi2* function. *f_classif* and *chi2 are* given as parameters to *sklearn*'s *SelectKBest* and then this combination is fitted with X_train to find the best 4 features. *mutual_info_classif* is done manually (to enforce random state) by looking at the resulting vector produced when fitted with X_train and picking the top 4 scores. It is worth noting that from testing, mutual information analysis with *n_neighbors=7* or higher produced the best results; hence 7 is chosen as a constant. In each of the above procedures, the resulting selected features are used to transform both X_train and X_test into the best 4 features as determined by each algorithm. Each set of best 4 features from these selections is then tested with 3-nn classification by fitting the *KNearestNeighbor* function with the 4 selected features from X_train and using that to predict those selected features from X_test. The predicted vectors are individually compared to y_test. In execution, ANOVA F produced an accuracy of 0.619, Chi2 produced an accuracy of 0.651 and Mutual Information produced an accuracy of 0.762.

Naïve first 4 features. The first 4 features of the original dataset (after median-imputing and standardization) are picked and 3-nn classification is done on this set of features by fitting X_train into *KNearestNeighbors* and predicting X_test; then comparing the results with y_test. In execution, this produced an accuracy of 0.667.

Principle Component Analysis. The original set of 20 features (after median-imputing and standardization) are used for PCA analysis using the *PCA* class from *sklearn.* The *PCA* function is fitted with X_train and y_train; and given *n_components=4* as the number of components to keep. This is then used to transform the X_train and X_test sets, which produces 4 principle components (PC) for each of the sets. To evaluate the PCA, a bar chart is plotted to visualise the percentage of explained variance from each of these components (see task2bgraph3), and a scatter plot is also plotted of PC2 against PC1 (see task2bgraph4). The principle components from X_train is then fitted into *KNearestNeighbors* for 3-nn classification, and this is used to predict X_test. The resulting vector is compared to y_test. In execution, this produced an accuracy of 0.762. The bar plot produced in task2bgraph3 shows a good coverage of variance in the principle components (PC1 near 70% explained variance).

Evaluation. The feature engineering procedure and PCA method both produced the best accuracy of 0.762, while the naïve first 4 method only produced an accuracy of 0.667. This is reasonable as picking the first 4 features has no statistical significance compared to doing analysis on the data, hence the difference in performance is justifiable. It is worth noting that the results produced from feature selection in task 2b are relatively similar to results produced without feature selection in part 2a, which indicates that the feature selection procedures were not highly effective. Interestingly, the scatter plot (task2bgraph4 produced during PCA) of PC2 against PC1 revealed a possible quadratic relationship in the data, which may explain why ANOVA F performed poorly and Mutual Information performed well. Overall, the results for feature engineering and PCA are quite good. However, these results cannot be deemed reliable as the sample size of 183 data points is quite small and differences in results can be largely influenced by the randomness in splitting data, even with consistent randomness enforced. In fact, different global *random_state* values were manually tested on the model, which provided varying results each time.

To improve the feature selection model, future implementations can try removing features with low variance first (the *VarianceThreshold* library in *sklearn* can do this), before removing highly correlated and unimportant features. The algorithm may also benefit from further dependence analysis to find inter-dependent features using more than just correlation; and removing such relationships, as independent features should ideally be independent of each other (not of class). To further improve feature selection, algorithms like k-fold cross validation and recursive feature elimination (*RFE*) from *sklearn* can be used instead of *SelectFromModel* to cross validate the importance of features and find the best combination before removing any features that are deemed 'unimportant'. The programmer may also try a different importance metric provided by *sklearn*'s *DecisionTreeClassifier* or *RandomForestClassifier* to go with these algorithms, which would produce different importance scores to *LogisticRegression*.