Dry Beans Classification Report

EE 559 Course Project

Data Set Dry Beans

Panthil Patel, pkpatel@usc.edu

4/27/2024

1. Abstract

The goal of this project is to design and compare several machine learning algorithms and compare them to see what works best on the dry beans classification dataset provided by Kaggle[3]. Several preprocessing and feature engineering techniques will be tested on the following machine learning techniques: perceptron, support vector machines, k nearest neighbor, and logistic regression. The baseline model for comparison will be a nearest means classifier. The best classification results ultimately come from logistic regression after SMOTE, standardization, normalization, shuffling, 3rd order polynomial transformation, LDA down to 6 components, and regularization term of C=1.

2. Introduction

2.1. Problem Statement and Goals

The chosen dataset for this project is the Dry Bean Classification dataset. This dataset contains 13,611 grains from 7 different classes of dry beans. The objective of this project is test out a variety of preprocessing and feature engineering techniques on various Machine Learning methods to see what techniques and model create the best result in the best performance measures. The primary performance measures are classification accuracy, macro averaged F1 score, micro averaged F1 score, and confusion matrix.

3. Approach and Implementation

3.1. Dataset Usage

The first operation performed on the datasets was using pandas to read the .csv file into python. This was done for both the training and testing datasets. The next step was to split up the features from the labels. This was done using the dot operator(.values) in python. Once the labels were separated, each name was assigned an integer value to make processing

easier going forward. Each of the 7 classes were assigned an integer from 1-7. This was done for the training and class labels.

Next, I will go over how the dataset was used in terms of preprocessing for the final combination of preprocessing techniques chosen. Several preprocessing techniques and most classification methods were implemented using sklearn[1]. Direct comparison of all preprocessing techniques tested will be discussed in the next section. The following operations were performed only on the training dataset. To fix the imbalance within the training dataset, SMOTE was utilized to artificially boost the number of data points and provide extra data points for classes that had very few in the original training dataset. Next, a standardization model with 0 mean and 1 standard deviation was fitted to the resampled training dataset features using standard_scaler.fit(). The training and testing datasets were transformed based on this fitted model. The next preprocessing step that was taken across all ML methods was shuffling the whole resampled and standardized training dataset using np.random.shuffle(). Lastly, the training and testing datasets was normalized for logistic regression and SVM ML techniques. For the perceptron, kNN and logistic regression classifiers, a 3^{rd} order non-linear transformation was performed to increase the number of features. LDA was then used to reduce the d.o.f from 970 to 7. For the SVM classifier, a 3^{rd} order non-linear transformation was performed to increase the number of features. PCA was then used to reduce the d.o.f from 970 to 100. After trial and error, 100 features was found to significantly reduce run-time while maintaining classification accuracy when compared to applying a non-linear transformation without PCA.

Cross validation was used to refine hyperparameters for SVM, logistic regression, and kNN. Cross validation was implemented in a sequential for loop format while manually changing parameters each run and recording the results up to 5 runs. The specific hyperparameters and the values tested will be explained in more detail in section 3.5. For all ML methods, 5 fold cross validation was used and all table values report the best result across 5 runs. After SMOTE, each training set consisted of 15,764 data points and each validation set contained 3,941 data points. For each fold, validation set accuracy and F1 scores are recorded. After 5 folds, the values are averaged and reported with their standard deviation. The classifier of the final cross validation fold is used to predict labels on the validation set to obtain the reported confusion matrix.

Finally, once cross validation was done, the classifier was fitted to the full training dataset. Using clf.score(), the testing accuracy on testing dataset was obtained and using clf.predict(), the predicted labels of the test set were obtained to calculate F1 score metrics. The predicted labels were also used in conjunction with the true labels to produce the test set confusion matrix. Overall the test set is used twice 3 times total, once to undergo a standard_scaler transformation, a second time to be normalized for certain ML techniques, and a third time in each ML method to record required performance metrics.

3.2. Preprocessing

As explained earlier, the final choices for preprocessing across all ML methods are SMOTE, standardization, and shuffling. For logistic regression and SVM, I found that normalizing the dataset slightly improved classification results and model fitting runtime.

The problem statement informs us that the dataset is imbalanced which is why SMOTE is used to ensure that each class has an even number of data points. Standardization was used to make all features measurable on the same scale. Simply going over the feature names of the dry bean dataset, one can see that these metrics are not all measured on the same scale. These were the things that jumped out to me first, so SMOTE and standardization were the first pre-processing techniques I tried. SMOTE was implemented using the imblearn library [2]. Their immediate effect can be seen in tables 1-4.

***Note: All ML methods are using results from best hyperparameters but without any feature engineering. These values and the method for obtaining them will be explained in section 3.5**

***Note: The reported values in tables throughout the entire report are the best values after 5 runs. I chose to do best value instead of average values since several values are extremely close to each other and reporting the best results seemed like the best way of comparing the potential of each respective classifier or technique. It is worth noting that even across 5 runs there was very little variance so either way the results would be the same. The cross validation values have standard deviation reported as they are averaged across 5 folds.**

ML Technique: Perceptron		
	Cross Validation Acc	Test Set Acc
SMOTE	0.2541±0.1552	0.353
Standardization	0.897±0.01176	0.881
SMOTE + Standardization	0.919±0.0332	0.8868
No Preprocessing	0.2755±0.0484	0.176

Table 1: Effect of SMOTE and Standardization - Perceptron

ML Technique: SVM		
	Cross Validation Acc	Test Set Acc
SMOTE	0.04542 ±0.0209	0.2755
Standardization	0.91955±0.00609	0.9265
SMOTE + Standardization	0.9459±0.0197	0.9228
No Preprocessing	0.2657±0.01011	0.2755

Table 2: Effect of SMOTE and Standardization - SVM

ML Technique: kNN		
	Cross Validation Acc	Test Set Acc
SMOTE	0.7095±0.0344	0.7175
Standardization	0.9235±0.00396	0.92616
SMOTE + Standardization	0.93925±0.0205	0.92248
No Preprocessing	0.7002±0.004414	0.7226

Table 3: Effect of SMOTE and Standardization - kNN

ML Technique: Logistic Regression		
	Cross Validation Acc	Test Set Acc
SMOTE	0.92723±0.0189	0.909258

Standardization	0.92414±0.00323	0.92616
SMOTE + Standardization	0.9397±0.01951	0.92322
No Preprocessing	0.9134±0.003511	0.9122

Table 4: Effect of SMOTE and Standardization – Logistic Regression

Across all ML techniques, the results between using both SMOTE and standardization vs only standardization are very similar. In general, the average validation accuracy appears to be higher when combining SMOTE and standardization and the variation in test set accuracies is negligible. The higher cross validation classification accuracy indicates that the model is likely the most robust under these conditions and will perform better under a variety of test sets when compared to using only standardization.

Building upon my knowledge that SMOTE and standardization produces good results, I tried 3 other preprocessing techniques to see if they would have any significant effect. The results can be seen in tables 5-8.

SMOTE + Standardization – ML Technique: Perceptron							
Preprocessing	Validation Acc	Validation Micro F1	Validation Macro F1	Test Set Acc	Test Set Micro F1	Test Set Macro F1	Runtime (ms)
Normalize	0.91667±0.02878	0.91667±0.02878	0.7495±0.1942	0.8931	0.8931	0.9026	66.66
Remove Outliers	0.9115±0.046	0.9115±0.046	0.746±0.188	0.900	0.900	0.9122	69.94
Shuffling	0.929±0.00197	0.929±0.00197	0.9287±0.00194	0.9019	0.9019	0.9189	66.64
No Addition	0.9087±0.0112	0.9087±0.0112	0.7485±0.2011	0.891	0.891	0.9045	88.15

Table 5: Effect of Normalization, Removing Outliers, and Shuffling (Perceptron)

SMOTE + Standardization – ML Technique: SVM							
Preprocessing	Validation Acc	Validation Micro F1	Validation Macro F1	Test Set Acc	Test Set	Test Set Macro F1	Runtime (s)

					Micro F1		
Normalize	0.9420±0.0198	0.942±0.0198	0.756±0.225	0.9254	0.9254	0.93877	7.527
Remove Outliers	0.9472±0.0212	0.9472±0.0212	0.761±0.225	0.9254	0.9254	0.9366	15.28
Shuffling	0.9505±0.0019	0.9505±0.0019	0.9505±0.0019	0.9254	0.9254	0.9389	13.87
No Addition	0.9466±0.0206	0.9466±0.0206	0.7599±0.224	0.924	0.924	0.937	14.388

Table 6: Effect of Normalization, Removing Outliers, and Shuffling (SVM)

SMOTE + Standardization – ML Technique: kNN							
Preprocessing	Validation Acc	Validation Micro F1	Validation Macro F1	Test Set Acc	Test Set Micro F1	Test Set Macro F1	Runtime
Normalize	0.923±0.0249	0.9232±0.0249	0.7422±0.2211	0.9085	0.9085	0.9220	1.296
Remove Outliers	0.9389±0.0199	0.9389±0.0199	0.756±0.223	0.925	0.925	0.940	1.28
Shuffling	0.9474±0.0020	0.9474±0.0020	0.9475±0.0017	0.9188	0.9188	0.934	1.247
No Addition	0.938±0.021	0.938±0.021	0.754±0.223	0.921	0.921	0.9362	1.28

Table 7: Effect of Normalization, Removing Outliers, and Shuffling (kNN)

SMOTE + Standardization – ML Technique: Logistic Regression							
Preprocessing	Validation Acc	Validation Micro F1	Validation Macro F1	Test Set Acc	Test Set Micro F1	Test Set Macro F1	Runtime
Normalize	0.9342±0.020	0.9342±0.020	0.7642±0.201	0.9199	0.9199	0.9328	0.4886
Remove Outliers	0.9373±0.0194	0.9373±0.0194	0.754±0.222	0.9228	0.9228	0.9368	0.722

Shuffling	0.9451±0.0013	0.9451±0.0013	0.9451±0.0015	0.9229	0.9229	0.9366	0.8527
No Addition	0.9358±0.0166	0.9358±0.0166	0.753±0.224	0.9221	0.9221	0.9361	0.676

Table 8: Effect of Normalization, Removing Outliers, and Shuffling (LogReg)

The most substantial effect of the tested preprocessing techniques is the effect of shuffling the training dataset on the macro F1 score. Across all ML techniques, shuffling the training dataset substantially boosted the macro averaged F1 score on the average of the cross validation set. This is likely due to the unshuffled training dataset after the effect of SMOTE having a very uneven spread of classes, resulting in certain folds of cross validation being very imbalanced despite the full training dataset being balanced. Beyond macro F1 score, shuffling the dataset also seemed to slightly increase avg cross validation accuracy. This is most likely due to the same reason mentioned earlier where prior to shuffling come cross validation folds were imbalanced.

For both SVM and logistic regression, normalizing the dataset had a fairly substantial effect on reducing model fit time without seeing any substantial effect on classification accuracies (cross validation & testing).

The rule I used to classify outliers is if a specific feature value was more than 3 standard deviations away from the mean feature value across all data points. Removing outliers did not appear to have a significant impact on classification results on any ML method. This is likely due to the sheer number of data points. After SMOTE there are 19,705 data points so keeping a few outliers will likely either have no effect on model performance or serve to simply make the model more robust.

3.3. Feature engineering + Feature Dimensionality Adjustment

A 3rd order polynomial transformation was used to increase the number of features from 16 to 970. Performing classification on such a large number of features usually results in very long runtimes and a potentially overfit model as some features are weighted incorrectly according to the importance. Initially I tried to rectify this using recursive feature elimination(RFE). However, every time I tried to use this no, matter how many features I tried reducing by, my IDE would freeze or crash. I never got any error messages and I was able to use it on other datasets so I am left with the assumption that RFE is simply not suitable with this dataset. When

I have spare time I would like to come back to this and see if there is a combination of preprocessing techniques that can be used to get around this issue.

For now I decided to test other feature reduction techniques, principal component analysis(PCA) and linear discriminant analysis(LDA). The number of components to reduce to for PCA was chosen by heuristic analysis. It is worth mentioning that there is a quite a large range of values that can be chosen here that will yield very similar results. For LDA, the maximum feature value was chosen, min(# of features, 1 – (number of classes)). Reducing features further either negatively impacted classification results or did not have a substantial impact on model fitting time.

SMOTE + Standardization + Shuffle - Perceptron			
Feature Engineering:	Avg Validation Acc	Test Set Acc	Runtime (ms)
3rd order transform	0.9394±0.0031	0.8905	1776.73
3rd order transform + PCA(n=100)	0.9380±0.0082	0.92395	46.23
3rd order transform + LDA(n=6)	0.9415±0.00435	0.92469	52.82

Table 9: Effect of Non-linear transformation & PCA or LDA (Perceptron)

SMOTE + Standardization + Shuffling + Normalization - SVM			
Feature Engineering:	Avg Validation Acc	Test Set Acc	Runtime (s)
3rd order transform	0.94966±0.0032	0.9291	83.924
3rd order transform + PCA(n=100)	0.9502±0.00282	0.9269	11.191

3rd order transform + LDA(n=6)	0.9482±0.0019	0.9199	10.872

Table 10: Effect of Non-linear transformation & PCA or LDA (SVM)

SMOTE + Standardization + Shuffling - kNN			
Feature Engineering:	Avg Validation Acc	Test Set Acc	Runtime (s)
3rd order transform	0.9447±0.0022	0.9166	4.202
3rd order transform + PCA(n=100)	0.9429±0.0015	0.91844	1.487
3rd order transform + LDA(n=6)	0.9557±0.0026	0.9247	1.2998

Table 11: Effect of Non-linear transformation & PCA or LDA (kNN)

SMOTE + Standardization + Shuffling + Normalization- LogReg			
Feature Engineering:	Avg Validation Acc	Test Set Acc	Runtime (s)
3rd order transform	0.94712±0.0051	0.9221	3.268
3rd order transform + PCA(n=100)	0.9465±0.0053	0.92175	0.4616
3rd order transform + LDA(n=6)	0.9585±0.0021	0.9302	3.3676

Table 12: Effect of Non-linear transformation & PCA or LDA (LogReg)

Ultimately, the resulting classification accuracies seemed to be slightly better when using LDA. This was the case for all ML techniques except SVM.

There is not a dramatic difference between the PCA reduction classification results and LDA reduction results. This indicates that the variance of data does not skew in a way that causes varying class labels to overlap after feature reduction. This indicates there are likely a few dominant features that do a good job of separating the data so that both PCA and LDA are effective.

Regarding runtime, both PCA and LDA seemed to have a similar effect on how much runtime was reduced by for perceptron, SVM, and kNN classifiers. For logistic regression, the runtime was significantly lower after PCA reduction compared to LDA reduction. In a real world case, it might be better to go with PCA here since the classification results are very similar and model fitting time can be a very important metric. For the case of this project, I will choose LDA as the better result since it did have slightly higher classification accuracies and the runtime is not so long that it will affect data collection. Furthermore, the required report metrics do not include runtime so I will focus on what the problem statement requests.

3.4. Training, classification and/or regression, and model selection

Perceptron:

After preprocessing, the training data undergoes a 3rd order polynomial transformation using the PolynomialFeatures() function. The testing dataset undergoes the same transformation using the model that was fit to the training dataset. Next, feature engineering occurs using the LinearDiscriminantAnalysis() function to reduce the number of features to 6. The resulting d.o.f. is much less than the number of data points, which should prevent the classifier from becoming overfit. Once again, the model is fit to the training dataset and then both the training & testing datasets are transformed with said model. Next a perceptron classifier is created using sklearn.linear_model.Perceptron() with the maximum iterations set to 1,000,000 to ensure convergence. This value was chosen by heuristics, just picking something that would ensure that the model doesn't get cut off before converging.

The perceptron model is a convergence based technique that uses stochastic gradient descent to refine weight values until reaching a point where there are either no misclassified points or the maximum number of iterations is reached.

After all the set up, 5 fold cross validation begins. The process is done using KFold() and then a for loop to iterate over each fold, splitting the full training dataset into a training and validation set using the indices provided by KFold(), and storing the classification accuracy, micro F1 score, & macro F1 score. After 5 folds, the performance measures are averaged and the standard deviation is recorded. The perceptron classifier does not have

hyperparameters that need to be refined so in this instance cross validation was simply used to obtain an idea of the robustness of the model using various validation and training sets.

Finally, the classifier is fit to the full transformed training dataset and performance measures on the transformed test dataset are recorded. The length of time that the model takes to fit to the full training dataset is also recorded using time.time().

Support Vector Machine:

After preprocessing, the training data undergoes a 3rd order polynomial transformation using the PolynomialFeatures() function. The testing dataset undergoes the same transformation using the model that was fit to the training dataset. Next, feature engineering occurs using the PCA() function to reduce the number of features to 100. The resulting d.o.f. is much less than the number of data points, which should prevent the classifier from becoming overfit. Once again, the model is fit to the training dataset and then both the training & testing datasets are transformed with said model. Next a SVM classifier is created using sklearn.svm.SVC() with the kernel set to 'rbf'(radial basis function). The rbf kernel was chosen by comparing the only 2 versions we have used in EE559. The rbf kernel produced better results and is a bit more interesting, as it is slightly more dynamic compared to the linear kernel. The initial hyperparameters were C=0.01 and Gamma=1.

Support vector machines work by creating a margin around the boundary that is established to increase the robustness of model. The goal is to create as large a margin as possible with the training data so that the resulting decision boundary is less overfit and more applicable to a variety of test sets. The C hyperparameter directly controls the number of support vectors and therefore the size of the margin. A higher C values means less support vectors and smaller margin. The goal is to keep C as small as possible without reducing classification accuracy. In the rbf kernel case there is another hyperparameter, Gamma. A larger gamma value causes the decision boundary to fit itself more closely to the shape and spread of the training data. If the gamma value is too high, the decision boundary will be overfit.

After all the set up, 5 fold cross validation begins. The process is done using KFold() to obtain cross validation indices and then a for loop to iterate over each fold, splitting the full training dataset into a training and validation set using the indices provided by KFold(), and storing the classification accuracy, micro F1 score, & macro F1 score. After 5 folds, the performance measures are averaged and the standard deviation is recorded. When choosing hyperparameters, the range of tested C values were 0.01, 0.1, 1, 10, and 100. The range of tested Gamma values were 1, 10, 100, and 1000. Each pair underwent 5 fold cross validation and the best results are C=1 and Gamma=1.

Finally, the classifier is fit to the full transformed training dataset and performance measures on the transformed test dataset are recorded. The length of time that the model takes to fit to the full training dataset is also recorded using time.time().

K Nearest Neighbor:

After preprocessing, the training data undergoes a 3rd order polynomial transformation using the PolynomialFeatures() function. The testing dataset undergoes the same transformation using the model that was fit to the training dataset. Next, feature engineering occurs using the LinearDiscriminantAnalysis() function to reduce the number of features to 6. The resulting d.o.f. is much less than the number of data points, which should prevent the classifier from becoming overfit. Once again, the model is fit to the training dataset and then both the training & testing datasets are transformed with said model. Next a kNN classifier is created using sklearn.neighbors.KNeighborsClassifier(). The initial hyperparameter was k=3.

K nearest neighbors works by assigning points a class based on the majority of training data points in proximity to the test point. For any specific point the class is assigned by the highest ratio of (points in i class)/k based on the closest k points in terms of Euclidean distance.

After all the set up, 5 fold cross validation begins. The process is done using KFold() to obtain cross validation indices and then a for loop to iterate over each fold, splitting the full training dataset into a training and validation set using the indices provided by KFold(), and storing the classification accuracy, micro F1 score, & macro F1 score. After 5 folds, the performance measures are averaged and the standard deviation is recorded. When choosing hyperparameters, the range of tested K values were 3, 5, 10, 15, 20, and 100. Each k value underwent 5 fold cross validation and the best result was k=10.

Finally, the classifier is fit to the full transformed training dataset and performance measures on the transformed test dataset are recorded. The length of time that the model takes to fit to the full training dataset is also recorded using time.time().

Logistic Regression:

After preprocessing, the training data undergoes a 3rd order polynomial transformation using the PolynomialFeatures() function. The testing dataset undergoes the same transformation using the model that was fit to the training dataset. Next, feature engineering occurs using the LinearDiscriminantAnalysis() function to reduce the number of features to 6. The resulting d.o.f. is much less than the number of data points, which should prevent the classifier from becoming overfit. Once again, the model is fit to the training dataset and then both the training & testing datasets are transformed with said model. Next a logistic regression classifier is created using sklearn.linear_model.LogisticRegression(). The initial hyperparameter was C=0.01.

Logistic Regression is very similar to linear regression which was covered in lecture. However, instead of using a linear function to make approximations, a sigmoid function is used. Similar to SVM, the C parameter is a regularization strength parameter. Smaller C values result in strong regularization.

After all the set up, 5 fold cross validation begins. The process is done using KFold() to obtain cross validation indices and then a for loop to iterate over each fold, splitting the full training dataset into a training and validation set using the indices provided by KFold(), and storing the classification accuracy, micro F1 score, & macro F1 score. After 5 folds, the performance measures are averaged and the standard deviation is recorded. When choosing hyperparameters, the range of tested C values were 0.01, 0.1, 1, 10, and 100. Each C value underwent 5 fold cross validation and the best results came from C=1.

Finally, the classifier is fit to the full transformed training dataset and performance measures on the transformed test dataset are recorded. The length of time that the model takes to fit to the full training dataset is also recorded using time.time().

4. Results and Analysis: Comparison and Interpretation

Best Preprocessing + Feature Engineering + Hyperparameter Combination – ML Methods							
ML Method:	Avg Cross Validation Acc	Validation Micro F1	Validation Macro F1	Test Set Acc	Test Set Micro F1	Test Set Macro F1	Runtime (s)
Perceptron	0.9444±0.0086	0.9444±0.0086	0.944±0.0088	0.9104	0.9104	0.9295	0.04302
SVM	0.9503±0.00424	0.9503±0.00424	0.9503±0.0044	0.9287	0.9287	0.9414	12.3211
kNN	0.9534±0.00268	0.9534±0.00268	0.9535±0.00238	0.9225	0.9225	0.9367	1.320
Logistic Regression	0.9589±0.0035	0.9589±0.0035	0.9588±0.0036	0.9306	0.9306	0.9446	4.165

Table 13: All ML Techniques Comparison

Best ML vs Trivial vs Baseline Required Performance Measures							
ML Method:	Avg Cross Validation Acc	Validation Micro F1	Validation Macro F1	Test Set Acc	Test Set Micro F1	Test Set Macro F1	Runtime (s)

Trivial	0.1751±0.0052	0.1751±0.0052	0.1463±0.003	0.1639	0.1639	0.1308	0
Baseline	0.6169±0.0076	0.6169±0.0076	0.6352±0.00627	0.61866	0.61866	0.6348	0.0127
Baseline + Preprocess + Feature Eng	0.9580±0.00075	0.9580±0.00075	0.9581±0.00092	0.9247	0.9247	0.9392	0.1001
Logistic Regression	0.9565±0.00386	0.9565±0.00386	0.9565±0.00362	0.9291	0.9291	0.9433	2.967

Table 14: Best ML Method vs Baseline vs Trivial

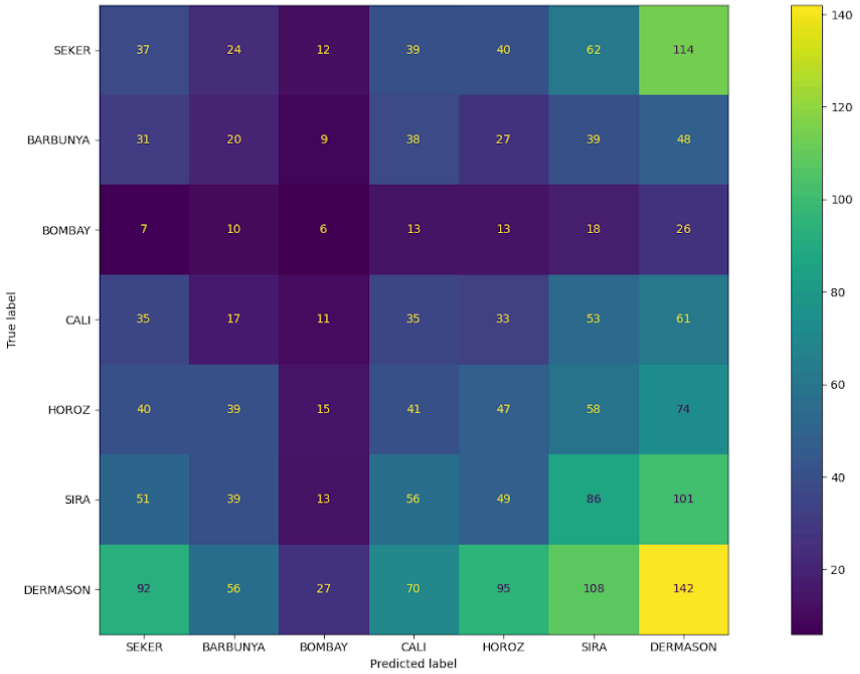

Figure 1: Trivial Solution Confusion Matrix – Final Cross Validation Fold Classifier

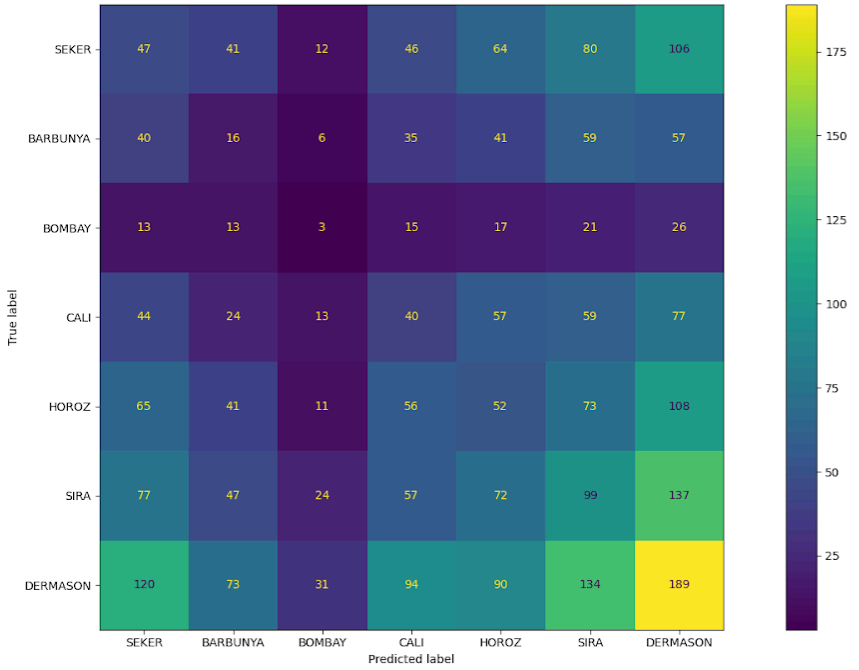

Figure 2: Trivial Solution Confusion Matrix – Test Set

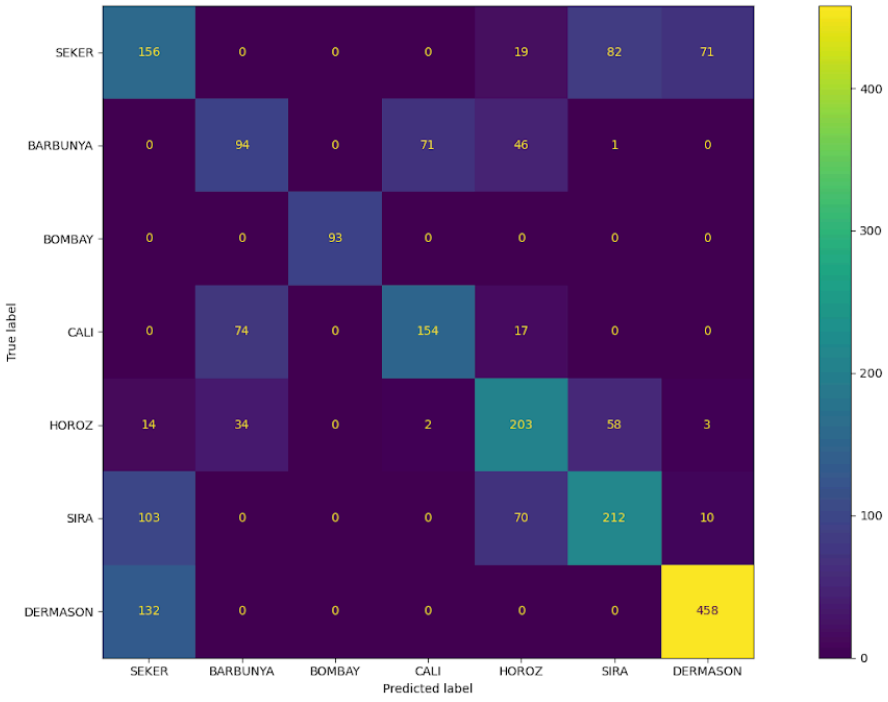

Figure 3: Baseline Solution Confusion Matrix - Final Cross Validation Fold Classifier

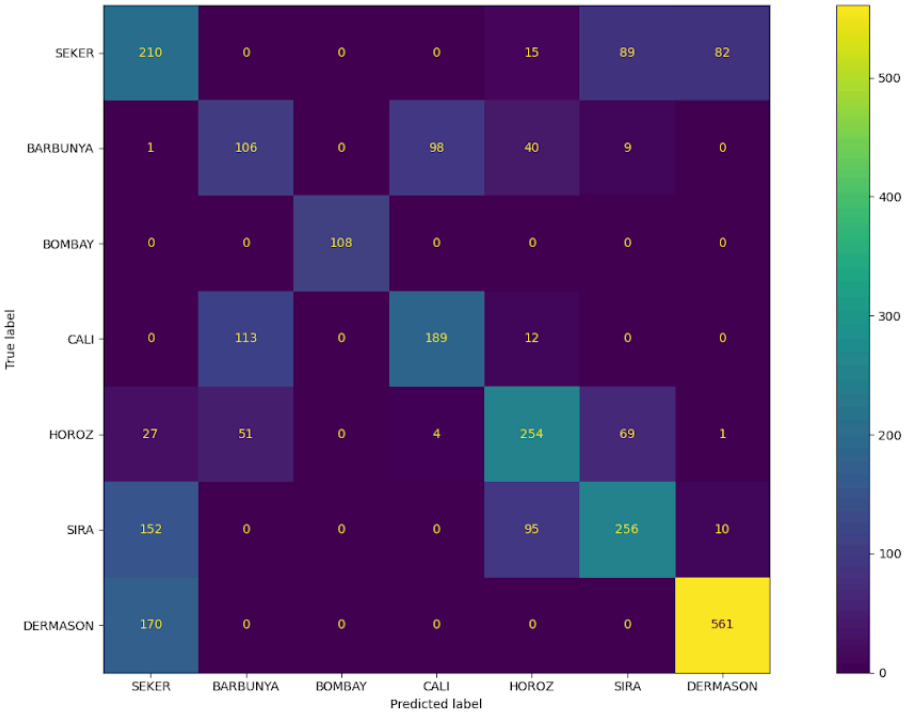

Figure 4: Baseline Solution Confusion Matrix – Test Set

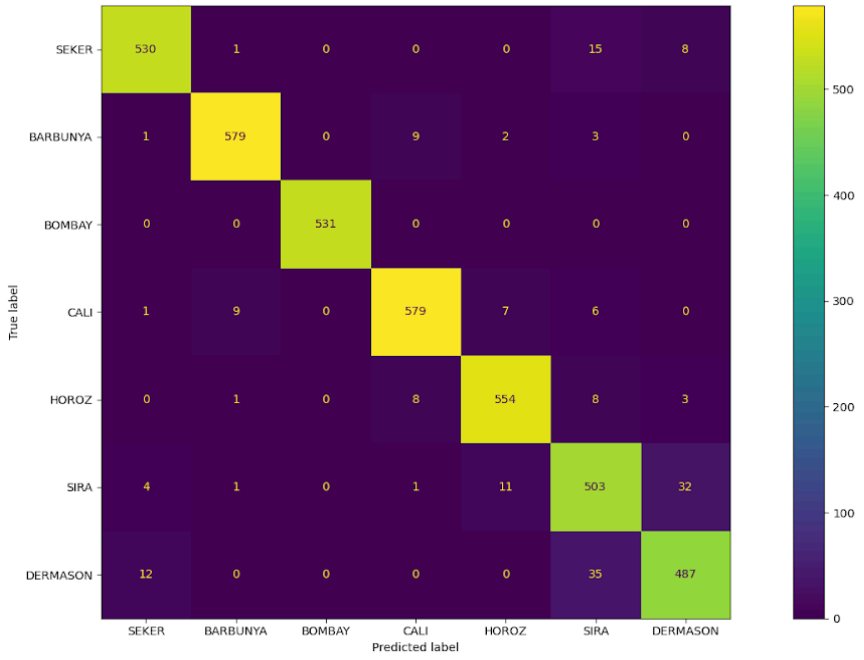

Figure 5: Baseline + Preprocessing Confusion Matrix - Final Cross Validation Fold Classifier

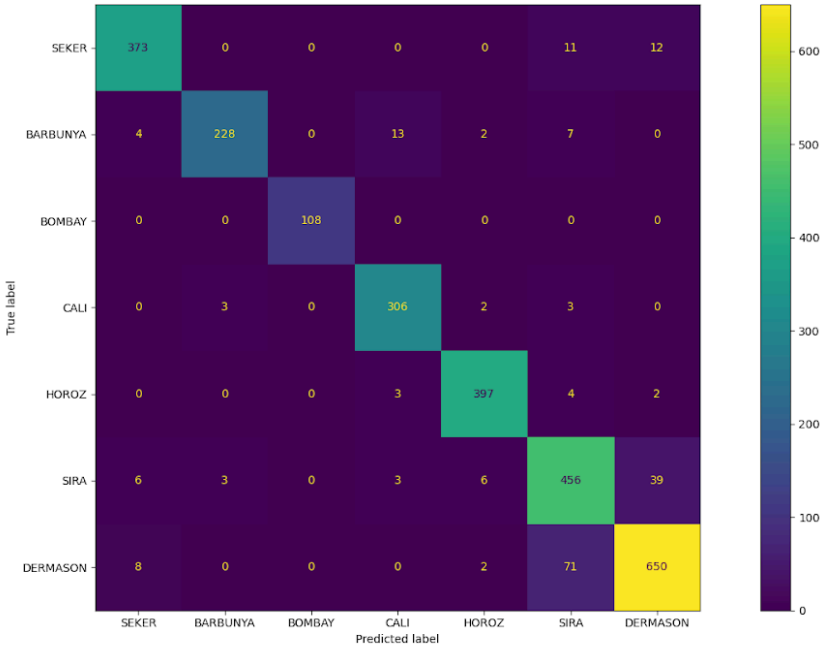

Figure 6: Baseline + Preprocessing Confusion Matrix – Test Set

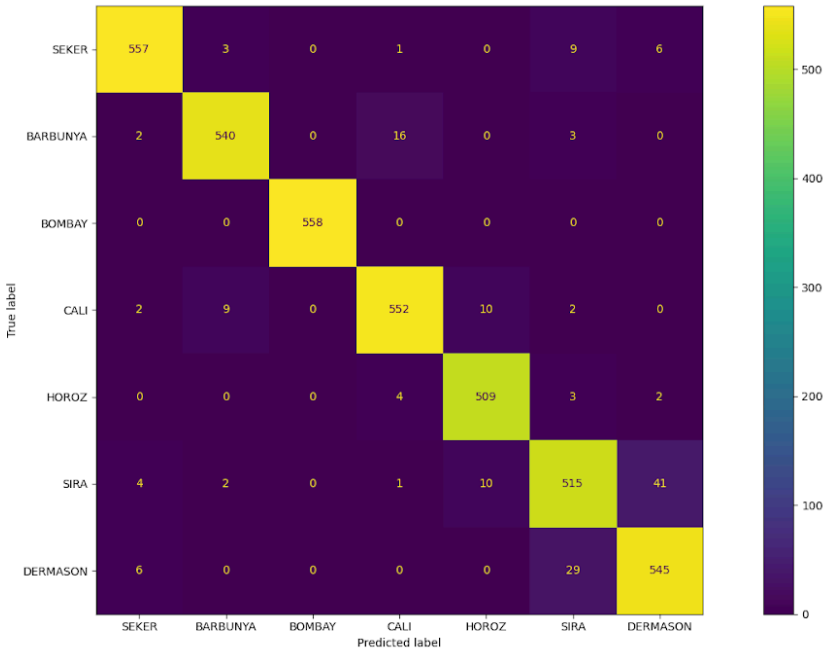

Figure 7: Best ML Method Confusion Matrix - Final Cross Validation Fold Classifier

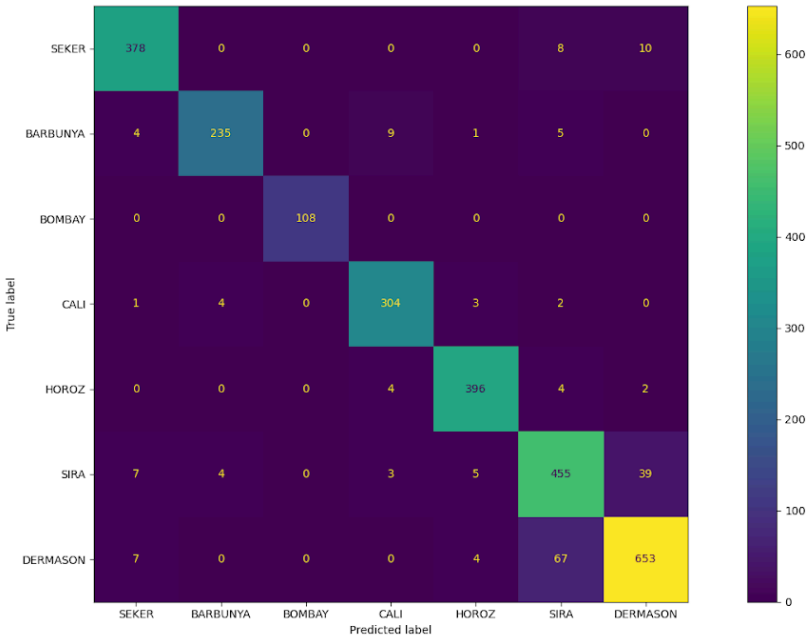

Figure 8: Best ML Method Confusion Matrix – Test Set

When comparing the confusion matrices in figures 3 and 4 with figures 5-8 it is interesting to see how the preprocessed classifiers all have the most struggle when differentiating between the Sira and Dermason beans, however there are very few misclassifications between those classes in figures 3 and 4. This is likely due to the data being unstandardized in figures 3 and 4 causing more nonsensical misclassifications. By analyzing figures 5-8, it seems plausible that the Sira and Dermason beans share similar values in prominent features. The confusion matrix information from figures 1 and 2 is not very helpful besides telling us that the training and test data is most saturated with Dermason beans compared to any other type.

From the results in table 13, it is clear to see that the best performing ML model in regards to classification accuracy and f1 score is logistic regression. However, it is worth noting that the difference between all ML models after optimal pre-processing is not dramatically different. All 4 of the ML techniques tested here would likely end up producing similar results in a real world setting. When considering runtime, perceptron performed by far the best with SVM being the slowest. However, across multiple runs perceptron generally resulted slightly lower classification accuracies compared to the other techniques. In a real life setting, one would need to balance the speed pay off of perceptron vs the 1-2% decrease in classification accuracy. For

the purposes of this assignment I will prioritize the required performance measures. In that regard, my best performing system is logistic regression utilizing SMOTE, standardization, normalization, and training data shuffling for preprocessing; 3rd order polynomial transformation and LDA down to 6 components in terms of feature engineering; and hyperparameters of C=1 and maximum iteration limit of 1,000,000. My best EE559 ML technique is kNN with all of the same preprocessing and feature engineering except no normalization and a hyperparameter of k=10.

From the results in table 14 we can see that when using all the preprocessing techniques honed for the ML models on the baseline model, the resulting classification accuracies are extremely good. The baseline model classification accuracies and f1 scores are extremely close to the logistic regression performance measures. When looking at the best results over 5 runs, the baseline model had a slightly higher cross validation accuracy compared to the logistic regression model. The similarity of performance measures not only between the logistic regression model and baseline + preprocessing model, but all tested ML techniques, leads me to believe that the most important part of getting the best results with this particular dataset lies in the preprocessing. In particular, standardizing the dataset had by far the biggest effect. This makes sense since most of the features are measured on different scales so uniformizing that makes a huge difference. I found it interesting that the logistic regression model had above 90% classification accuracy even without any preprocessing. I am assuming this is due to the nature of using a sigmoid function being much more adaptable compared to other ML techniques.

When analyzing the differences in the feature engineering step, the biggest changes seemed to be in runtime with only very slight changes to cross validation and test set performance measures. This makes sense when considering that after 3rd order polynomial transformation d.o.f = 970 and after SMOTE N_{c_tr} = 19,705 & N_{c_val} = 15,764. Using the rule of thumb: N > (3-10)*d.o.f; we can see that even without feature reduction the model would not be overfit. However, it is still good practice to perform feature reduction to reduce computation time and complexity of the model.

5. Libraries used and what you coded yourself

Libraries Used:

1. sklearn
2. imblearn
3. numpy
4. pandas
5. time

For cross validation, sklearn.model_selection.KFold() was used to obtain indices, however actually splitting the dataset, training & testing the classifier, and recording performance measures was done manually through a for loop.

When testing various preprocessing techniques, removing outliers was coded manually by iterating through the dataset and removing data points that contained any features that had values greater than 3 standard deviations away from the mean of that feature across all data points.

Pandas was used to read the datasets from the .csv files.

Numpy was used for all array creation and manipulation.

Time was used to record how long certain models took to fit.

Imblearn was used in order to apply SMOTE to fix the imbalance within the training dataset.

Sklearn was the most prominent library used as it was used for all classification techniques except the trivial solution, which was done manually. Sklearn was also used for feature engineering: PolynomialFeatures(), PCA(), LinearDiscriminantAnalysis(). Sklearn was also used for certain preprocessing techniques such as standardization and normalization.

6. Contributions of each team member

Solo Project – N/A

7. Summary and conclusions

Four distinct machine learning methods were tested in this project. Ultimately, logistic regression was determined to produce the best results based on the required performance measures. However, after all the preprocessing and feature engineering, the classification accuracies across all ML methods as well as the baseline solution ended up being very close. Ultimately, one of the key things I learned working on this project and this dataset specifically is that preprocessing is much more impactful on final results than the actual ML method used. In theory this makes sense since if the data is a mess, then regardless of the technique the results are not going to be great. It just was not something I really thought about until spending so much time working with this dataset. As I mentioned earlier, I am interested to see if there is some preprocessing technique I can use to get recursive feature estimation to work as that might produce better results than LDA or PCA.

References

[1] "About Us," 2024. [Online]. Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011

[2] "Over-Sampling",2024.[Online]. https://imbalanced-learn.org/stable/over_sampling.html#from-random-over-sampling-to-smote-and-adasyn

[3] Dry Bean Dataset Classification, Access:2024, https://www.kaggle.com/datasets/nimapourmoradi/dry-bean-dataset-classification/data