Mathematics for Machine Learning Tesina

Vincenzo De Marco, s290373@studenti.polito.it

Abstract

This Tesina is composed of an extensive and accurate analysis of the Dry Bean Dataset, found in the UCI machine learning repository. Valuable pre-processing steps have been included in the pipeline and different Machine Learning methods have been carefully selected and implemented, to correctly classify many kinds of beans. Two types of metrics have been used to test the models' performances, *accuracy* and *F1 score*. Overall, the results achieved are accurate, and the whole pipeline has been demonstrated to be optimal for the chosen task.

1859 Politecnico di Torino

I. Introduction

Machine Learning is a powerful tool than can be adapted to many real-world problems. With the right shrewdness, a well-designed pipeline can hugely help in situations where a prediction is helpful or needed, be it classification or regression.

In our scenario, we are interested in analyzing which property of a dry bean can be used to predict its species, and which Machine Learning model can be implemented to achieve good performances.

II. Dataset Overview

Seven different types of dry beans were used in this research, taking into account the features such as form, shape, type, and structure by the market situation. A computer vision system was developed to distinguish seven different registered varieties of dry beans with similar features in order to obtain uniform seed classification. For the classification model, images of 13,611 grains of 7 different registered dry beans were taken with a high-resolution camera. Bean images obtained by computer vision system were subjected to segmentation and feature extraction stages, and a total of 16 features; 12 dimensions and 4 shape forms, were obtained from the grains.

Attributes informations:

- Area (A): The area of a bean zone and the number of pixels within its boundaries.
- Perimeter (P): Bean circumference is defined as the length of its border.
- Major axis length (L): The distance between the ends of the longest line that can be drawn from a bean.
- Minor axis length (l): The longest line that can be drawn from the bean while standing perpendicular to the main axis.
- Aspect ratio (K): Defines the relationship between L and l.
- Eccentricity (Ec): Eccentricity of the ellipse having the same moments as the region.
- Convex area (C): Number of pixels in the smallest convex polygon that can contain the area of a bean seed.
- Equivalent diameter (Ed): The diameter of a circle having the same area as a bean seed area.
- Extent (Ex): The ratio of the pixels in the bounding box to the bean area.
- Solidity (S): Also known as convexity. The ratio of the pixels in the convex shell to those found in beans.

- Roundness (R): Calculated with the following formula:
$$R = (4\pi A)/(P^2)$$
- Compactness (CO): Measures the roundness of an object: CO = Ed/L
- ShapeFactor1 (SF1)
- ShapeFactor2 (SF2)
- ShapeFactor3 (SF3)
- ShapeFactor4 (SF4)

The Class of the dataset can assume 7 categorical values: Seker, Barbunya, Bombay, Cali, Dermosan, Horoz and Sira. Each of them is a different dry bean species.

III. Data Exploration and Preprocessing

The first and most important step of a proper Machine Learning pipeline is to accurately explore the data and pre-process it accordingly, keeping in mind best practices and the desired outcome.

First of all, since many of the pre-processing steps are non-deterministic, we set a parameter, *seed*, as a default value, in this case 42, to make sure that every (random) result can be replicated.

*A. **Class Encoding***

We start by noticing that the Class' values are categorical, and to feed a ML algorithm is always better to convert categorical values into numerical ones. In this case, since the Class is the only categorical one, the best thing would simply be to encode it using integer values, from 0 to 6. The resulting Class values would then be: 0 (for Seker), 1 (for Barbunya), 2 (for Bombay), 3 (for Cali), 4 (for Dermosan), 5 (for Horoz) and 6 (for Sira).

Anyway, if there happens to be any other categorical value, a proper encoding method, like 1-Hot Encoding, should be implemented.

*B. **Duplicates and Missing Values***

Duplicates are redundant data that might lead to over-fitting, so, if the dataset is extensive enough, it is always better to drop them. We achieve this by using *Pandas' DataFrame.drop_duplicates()* function, which will remove any row with the same attributes' values of any other row, leaving us only unique data. This brought the number of instances in our dataset from 13.611 to 13.543.

After dropping duplicates, we are interested in checking if our dataset contains any NaN value, i.e. a missing value, because they can alter a correct learning process. Running *Pandas' DataFrame.info()* function, we see that for each attribute we have exactly 13.543 non-null values, meaning that there are no missing values in our dataset.

*C. **Attributes' Correlation and Collinearity***

The next data exploration step consists in the analysis of the attributes' correlation, which will result in a deeper insight among features relationship. The heatmap on the left, in figure 1, highlights the correlation values: as expected, there are features that are related to each other. In fact, as an example, features such as *Area* and *Perimeter* can be mathematically derived from *Major axis length* and *Minor axis length*. It is clear that we have an high *collinearity* in the dataset, and this can lead to problems in the classification setting, since it can be difficult to separate the individual effects of collinear variables on the dataset's classes.

For this reason, after analyzing the attributes' correlation one by one, we decide to drop the following attributes: Area, Perimeter, Convex Area, Equiv Diameter, Compactness and Aspect Ration. Each of these attributes either doesn't have enough correlation with the *class* to bring meaningful information in the training process, or that information is already conveyed by one or more other attributes. The attributes' correlation of the resulting dataset can be observed in the heapmap on the right of figure 1.

The last issue remaining is that *multicollinearity*, i.e. correlation between three or more attributes that can not be identified considering them in pairs, can not be detected using an heatmap, but since the remaining attributes are few, we don't investigate further, because, considering the remaining features, the level of collinearity should be acceptable.

If one wanted to investigate anyway, computing the *variance inflation factor* (VIF) will better assess the level of collinearity in the dataset.

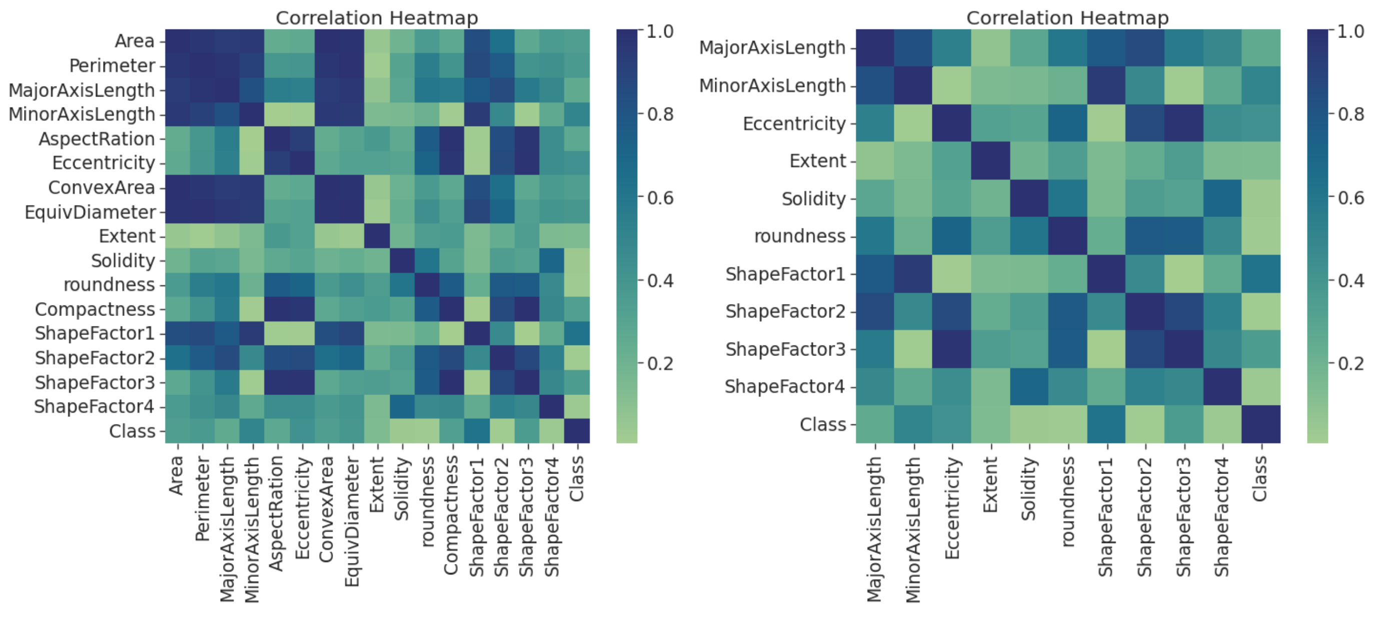

Fig. 1. Heatmaps showing attributes' correlation, before (left) and after (right) dropping attributes with high collinearity

D. *Dataset Splits*

After dropping correlated attributes, we are interested in splitting the dataset in two splits: *Train* and *Test*. This is fundamental because we have to be sure that the next pre-processing steps will not be applied on, and so influence, the test data, since the test split needs to reflect measurements taken from a real-world scenario.

We achieve this using *sklearn train_test_split()*, setting the *seed* as random state and 0.15 as test size, meaning that a random 15% of the data will be kept for testing. This will leave us with 11511 entries in the training dataset and 2032 entries dedicated to model evaluation, which will be enough to extensively assess model performances.

E. *Outliers*

Now that we have our training dataset, we can analyze its values and remove *outliers*, i.e. instances with "strange" attributes' values among its class. To recognize and iteratively remove outliers, we define a function that calculate, for each attribute, the values distribution's 0.01 and 0.99 *quantiles*, i.e. the values before (0.01) and after (0.99) which lies the 1% of the data in the distribution, and use them to find and delete outliers; this means that if a record has a value for that attribute that is lower than the 0.01 quantile or higher than the 0.99 quantile, it can be labaled as outlier and removed.

We repeat this process once per class, since the quantiles should be calculated only using data from the same class, otherwise their values will be altered and that can lead to remove data that is not actually an outlier for its class.

This process will reduce the amount of training data from 11511 instances to only 9343. It is highly probable that, in those 2168 records, we removed also data that was not really an outlier, but since we still have more that enough data to train the models with, it will not be a problem.

F. *Attributes Scaling*

Usually, when training complex methods with non-scaled data, the algorithm will iterate more, wasting time and resources and risking to achieve lower performances. For this reason, we decide to scale the data, but, before doing so, it is necessary to check the density distributions of the values we aim to scale, in order to choose the right normalization procedure.

Figure 2 shows said distributions for two attributes, *Major Axis Length* and *Solidity*, taken as examples. We can clearly see how each class values' distribution can be approximated by a normal distribution. This property holds also for every other attribute, even if we don't report here each plot. Since this property holds, we can safely apply normalization using *sklearn.preprocessing.StandardScaler()*, which will standardize features by removing the mean and scaling to unit variance:

$$z = (x - \mu)/\sigma$$

where z is the normalized value, x is the original value, and μ and σ are the mean and standard deviation of the values' distributions, respectively.

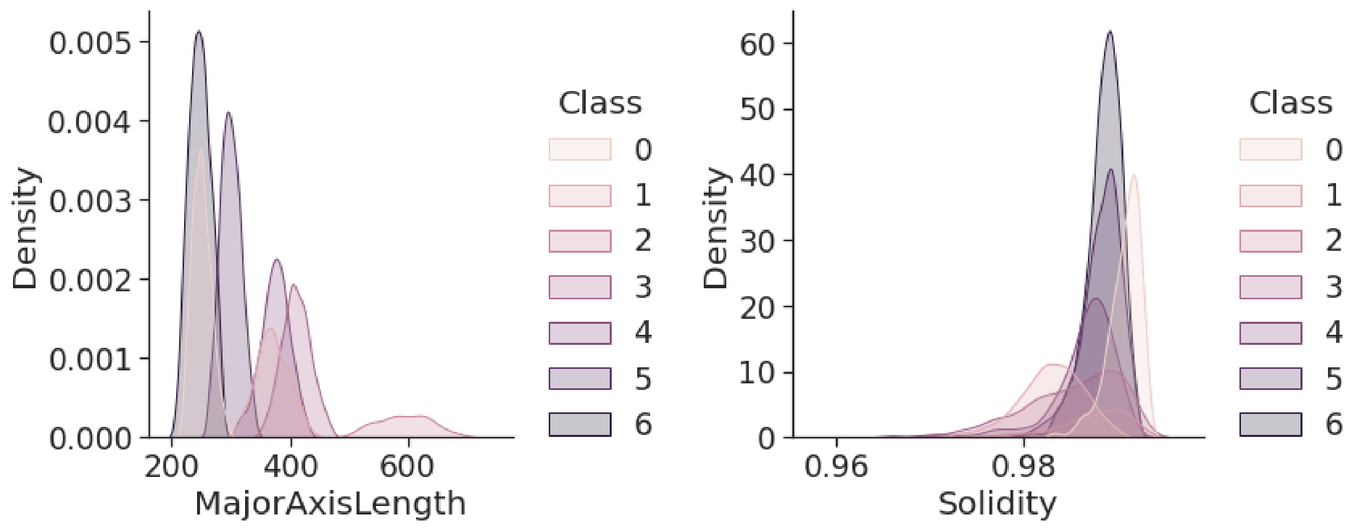

Fig. 2. Major Axis Length and Solidity values' density distributions per class, before the scaling process

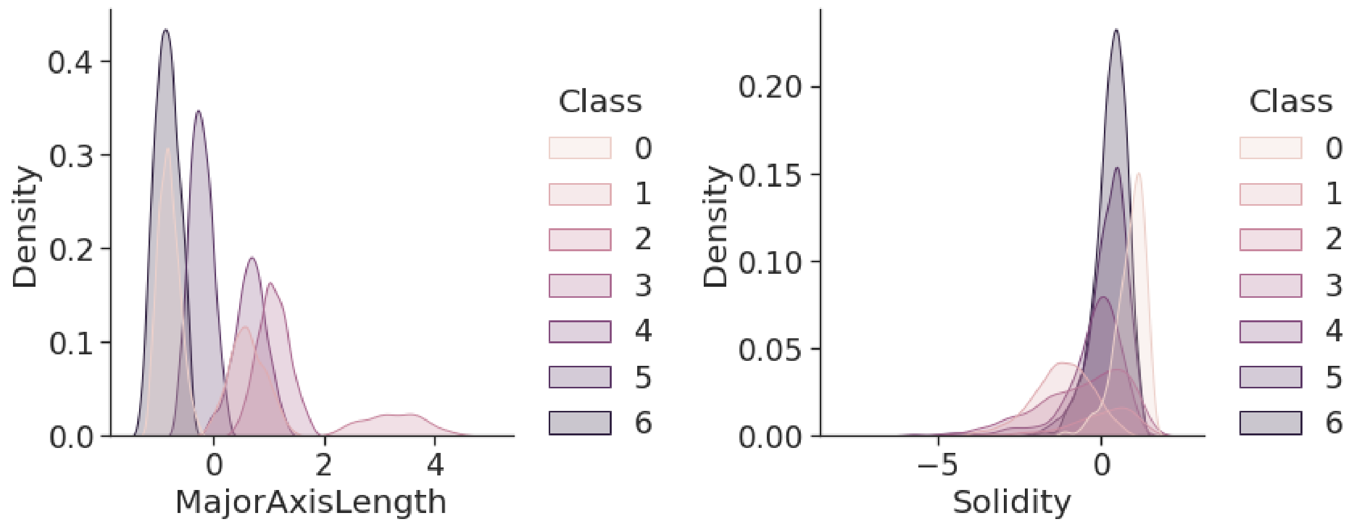

Fig. 3. Major Axis Length and Solidity values' density distributions per class, after the scaling process

Figure 3 shows the same plot as before but after the scaling process. It's easy to notice how the distributions remain the same but the density values are scaled, w.r.t. before.

Anyway, if we train the models with a scaled dataset, also the data used to evaluate them must be scaled, otherwise the predictions will not be meaningful nor accurate. For this reason we save the mean vector (μ, one per attribute) and the standard deviation vector (σ, one per attribute) of the distributions of the training data and use them to scale the test dataset. Notice that we use the training data scaling values instead of calculating them again for the test data: this is because in a real-world scenario it would be impossible to calculate such statistics for the data on which we want to make predictions.

G. Classes Balance

The last pre-processing step regards classes' balance. Training with an imbalanced dataset, i.e. a dataset with one or more classes that have a notably bigger sample size w.r.t. the other ones, will lead to potential over-fitting on the classes with more instances, because the algorithm will focus more on the patterns in the attributes of said class or classes, and potential under-fitting on the classes with less data available, because, having seen less examples, it will not generalize well on those patterns.

In Figure 4, the plot on the right shows that our training dataset is pretty imbalanced, so we will *re-sample* the data to balance it before feeding it to the models. We will consider the number of instances of class 4 as the average to apply *oversampling*,

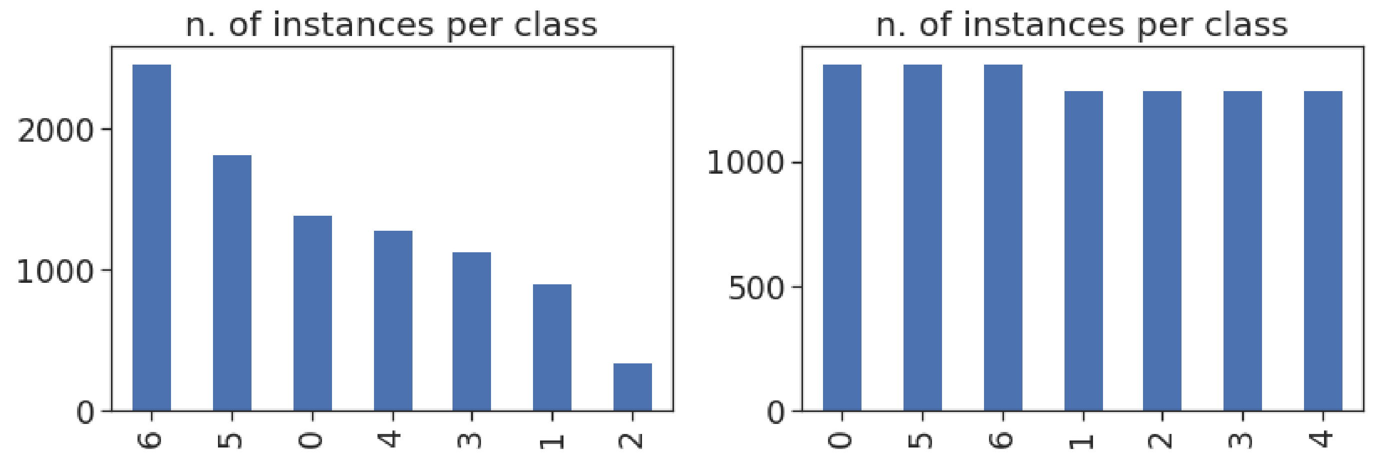

Fig. 4. Number of instances per class, before (left) and after (right) the re-sampling process

while we use the number of instanced of class 0 to apply *undersampling*. This is because, even if in the end the number of instances will not be exactly the same, we will avoid creating too many synthetic (duplicated) samples in the under-populated classes and deleting too many original samples in the over-populated ones.

To balance the classes with a lower number of instances than the average, we use the *imblearn.over_sampling.RandomOverSampler*, creating and giving it as parameter a dictionary, with the goal to apply oversampling only on classes 1, 2 and 3, until each one has 1284 samples, the same number of records that class 4 has. The *Random Over Sampler* will then, for each class, randomly select a record with replacement from the available ones, until the total number of samples reaches the desired amount. This process is similar to what happens in *bootstrapping* a dataset.

Regarding classes 5 and 6, since they have too many samples, we will use the *imblearn.under_sampling.RandomUnderSampler* to randomly delete records in each class, until only 1392 instances remain, the same amount that class 0 has.

This process will leave us with a training dataset composed of 9312 instances, where the classes are almost completely balanced, as can be seen in the plot on the right of Figure 4.

IV. Training and Evaluation settings

Now that we have properly analyzed and pre-processed the data, we will go through the training settings and the metrics that we will use to correctly train and evaluate any chosen method.

A. *Model Selection*

First of all, every method has its own *hyperparameters*, i.e. parameters with which the model is generated that describe one of its possible behaviors. Such hyperparameters need to be *tuned*, meaning that we would like to choose the best hyperparameters for our scenario, so that the resulting model fits our data as well as possible. Anyway, training and testing many different models will be time-consuming and will waste a lot of resources.

For these reasons, we will use the *sklearn.model_selection.GridSearchCV*, a *grid search* that uses brute force and *k-fold cross-validation* to find the best hyperparameters set.

1) Grid Search: It consists of a grid search scenario where, given a dictionary with the desired subset of hyperparameters and their possible values, every possible configuration of the model will be created and tested using *k-fold cross-validation*. Then, the best model (the one with the highest validation score), its validation score and its hyperparameters will be saved and used for evaluation.

2) K-Fold Cross-Validation: It is based on splitting the training dataset in K sub-datasets, and then use one of them as test set and the other *(K-1)* as training sets. This training and evaluation process is repeated K times with the same model, using each of the K different splits as test set once. Then, the K scores obtained are averaged to gain the final model's validation score. In our case, 5 is chosen as the value for K, and Figure 5 shows the 5-fold cross-validation process.

The metric chosen as validation score is the *accuracy*, which we will describe later.

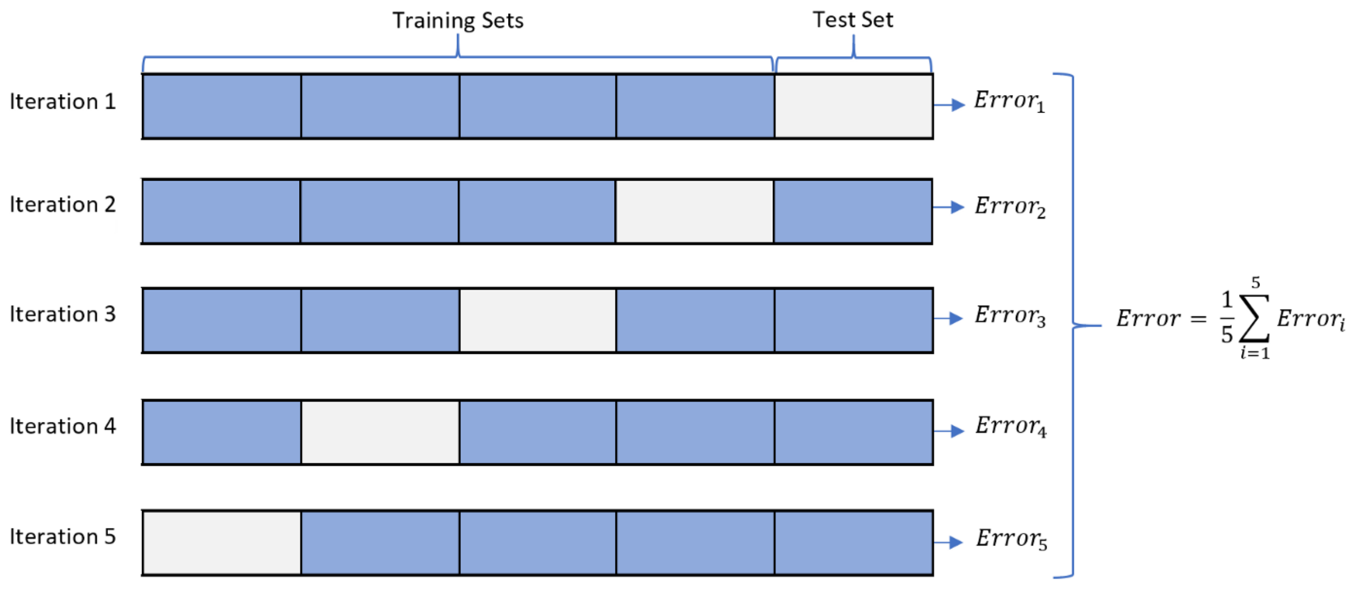

Fig. 5. 5-Folds Cross-Validation process

B. Metrics

When Validating and Evaluating a model, the choice of the right metric is essential to make meaningful comparisons and conclusions. In our classification pipeline, we will use the *accuracy* for both validation and evaluation score, and include the *f-1 score* as an additional evaluation metric.

1) Accuracy: It is the most know and used metric to evaluate classification models. It reaches its best score at 1 and worst at 0, and it is calculated as:

$$accuracy = 1 - \frac{\sum_{i=1}^{n} I(y_i \neq Y_i)}{n}$$

where *n* is the number of predictions, *y* is the predicted class and *Y* is the true class, and the indicator variable *I()* returns 1 if *y*=*Y* and 0 otherwise. The part of the equation after the - sign is known as *classification error*.

2) F1 Score: The F1 score can be interpreted as a harmonic mean of the *precision* and *recall*, where an F1 score reaches its best score at 1 and worst at 0. The relative contribution of precision and recall to the F1 score are equal. The precision is the fraction of true positive examples among the examples that the model classified as positive, while the recall, also known as sensitivity, is the fraction of examples classified as positive, among the total number of positive examples.

The formula to calculate the f1 score is:

$$f_1 = 2\frac{precision \cdot recall}{precision + recall} = \frac{tp}{tp + ((fp + fn)/2)}$$

where *tp* is the number of *True Positives* and *fp* and *fn* are the number of *False Positives* and *False Negatives*, respectively.

The main benefit of f1 score over accuracy is that it is much lower if the model is not able to classify correctly a certain class due to dataset imbalance, like in our case; for this reason, a good f1 score means that the re-sampling strategy works as intended.

Regardless, it is easy to understand that the f1 score can be calculated only in a binary classification scenario: in fact, in our multi-class setting, it is calculated as the average of the F1 score of each class with weighting depending on the number of instances for that class. This will take further into account classes' imbalance.

V. Selected Models

The methods we choose for our classification problem are *Random Forest Regressor* and *Support Vector Classifier*. Both are complex methods that perform well in many circumstances, and in our multi-class scenario, they are the perfect choice, since they can maintain a good bias-variance tradeoff also when the decision boundaries that separate classes are non-linear.

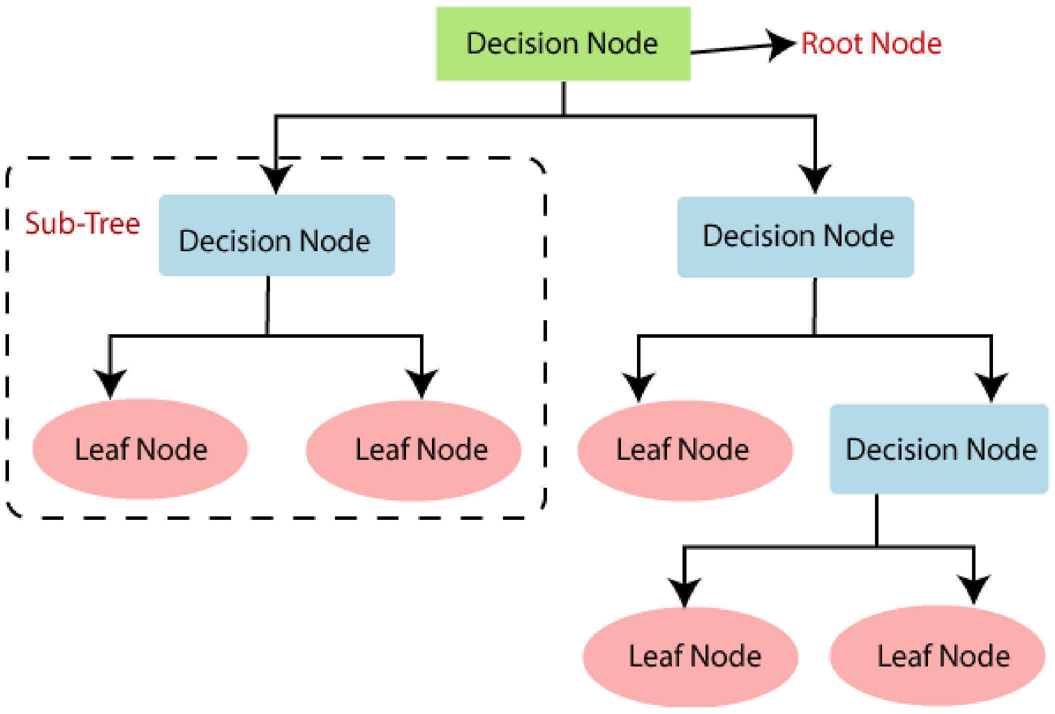

Fig. 6. Example of a simple decision tree classifier

A. *Random Forest Classifier*

A random forest is a meta classifier that fits a number of *decision tree classifiers* on various sub-samples of the dataset and uses majority voting to improve the predictive accuracy and control over-fitting.

1) Decision Tree Classifiers: Tree-based methods involve stratifying or segmenting the predictor space into a number of simple non overlapping regions, with the goal being to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. It is called decision tree classifier because it is based on taking a decision at each step, based on the attributes' values, and when built resembles an *upside-down tree*, Figure 6 pictures an example.

Each *decision node* consists in splitting the feature space of a chosen attribute in two parts by applying a condition (i.e. $a1 > 3$), using a top-down, greedy approach called *recursive binary splitting*, which consists in choosing the best possible split at each step; the criterion to decide which attribute and value will result in the best possible split can be the *gini index* or the *cross-entropy*, both will be discussed in the *hyperparameters* section.

After enough splits have been made, or reaching any stopping condition (i.e. maximum number of splits or maximum depth), a *leaf node* will be created, and, when making a prediction on a test sample, the majority class in the leaf node to which the sample arrives will be its assigned class.

The main drawback of decision trees is that generally they do not have a good level of predictive accuracy, and if the tree over-grows it will lead to over-fit the data. These problems are solved by aggregating many decision trees in what are called *random forest classifiers*

2) Random Forest Classifiers: A random forest classifier is composed by aggregating many decision trees in a single structure. When making a prediction, each tree will output its predicted class, and the final decision will be made by majority voting. This, supposed that each tree is a bit different from the others, and so may give a different prediction, will ensure a good generalization of the ensemble and good performances.

To ensure this diversity in the trees' population, each tree is trained on a different subset of the dataset, usually obtained by *bootsrtapping*. This process, called *bagging*, reduces the *varinace*, that is usually high in a single decision tree. Instead of bagging, there is another approach, called *boosting*, that can be used to reduce variance, and it is based on training trees sequentially, iteratively using the information from previously grown trees to grow the next one. We do not discuss it further because we will use bagging.

Beside bagging, another important property, used in random forests to reduce variance, is forcing the algorithm to consider only a subset of attributes to choose the best split for each node. This will ensure that, if there is a very strong attribute among the predictors, also other splits will be considered, because that strong attribute will not be in some of the allowed subsets.

3) Hyperparameters: The hyperparameters used in the cross-validation and their set of possible values are shown in the left table of Table I. These are:

- n_estimators: The number of decision trees in the forest.
- max_features: The number of features to consider (the dimension of the attributes' subset) when looking for the best split.
- max_depth: The maximum depth of the tree. We don't want it to grow indefinitely because that will cause over-fitting to the training data.
- min_samples_split: The minimum number of samples required to split an internal node. Splitting nodes with too few samples will lead to over-fitting.
- bootstrap: Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.
- criterion: The function to measure the quality of a split. We included both the gini index, and the cross-entropy.
  The gini index is calculated as:
  $$G = \sum_{k=1}^{K} p_{mk}(1 - p_{mk})$$
  where *K* is the number of classes and p_{mk} is the proportion of training observations in the *m*th region that are from the *k*th class.
  The cross-entropy is calculated as:
  $$D = -\sum_{k=1}^{K} p_{mk} \cdot log(p_{mk})$$
  Both are very similar between eachother, and are considered as measures of node *purity*, taking small values, indicating that a node is pure, if all of the p_{mk} are close either to 0 or 1.

B. Support Vector Classifier

Support Vector Classifier is a supervised ML model whose goal is to find an hyperplane that maximize the distance between classes in the feature space. An optimal separation is achieved by the hyper-plane that has the largest distance to the nearest training data points.

1) Maximal Margin Classifier: The maximal margin classifier tries to find *optimal separating hyperplanes* in the feature space to define decision boundaries to make predictions. Given *p* predictors, the feature space will be p-dimensional, and in these settings an *hyperplane* is a (p-1)-dimensional flat subspace, defined by the mathematical equation:

$$\beta_0 + \beta_1 X_1 + ... + \beta_p X_p = 0$$

for parameters β_0, β_1, ... , β_p. Any hyperplane divides the feature space in 2 parts, and so it can be interpreted as a *decision boundary*, i.e. if a data point lies on one side of the hyperplane it will be of class $y_i = 1$, otherwise if it lies on the other side it will be of class $y_i = -1$. The concept of this separating hyperplane can be expressed with the following mathematical property:

$$y_i(\beta_0 + \beta_1 X_1 + ... + \beta_p X_p) > 0$$

for all *i = 1, ..., n*. If such a hyperplane exists, a test observation can be assigned a class simply depending on its position w.r.t. the hyperplane.

Defining a *margin* as the minimum distance between the hyperplane and the closest data point(s), the maximal margin classifier tries to find the separating hyperplane with the maximum margin. Figure 7 shows an example with binary classification. From the mathematical point of view, the maximal margin hyperplane is the solution to the optimization problem:

$$\text{find } \beta_0, \beta_1, ..., \beta_p \text{ to maximize } M$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1$$

$$y_i(\beta_0 + \beta_1 X_{i1} + ... + \beta_p X_{ip}) \geq M$$

for any *i = 1, ..., n*.

An interesting property of the maximal margin classifier is that the coefficients β_i will be equal to 0 for every data point but the ones with the minimum distance from the optimal separating hyperplane. These observations, as can be seen in Figure 7, are called *support vectors*, because the maximal margin hyperplane depends directly only on them, and not on any other observation.

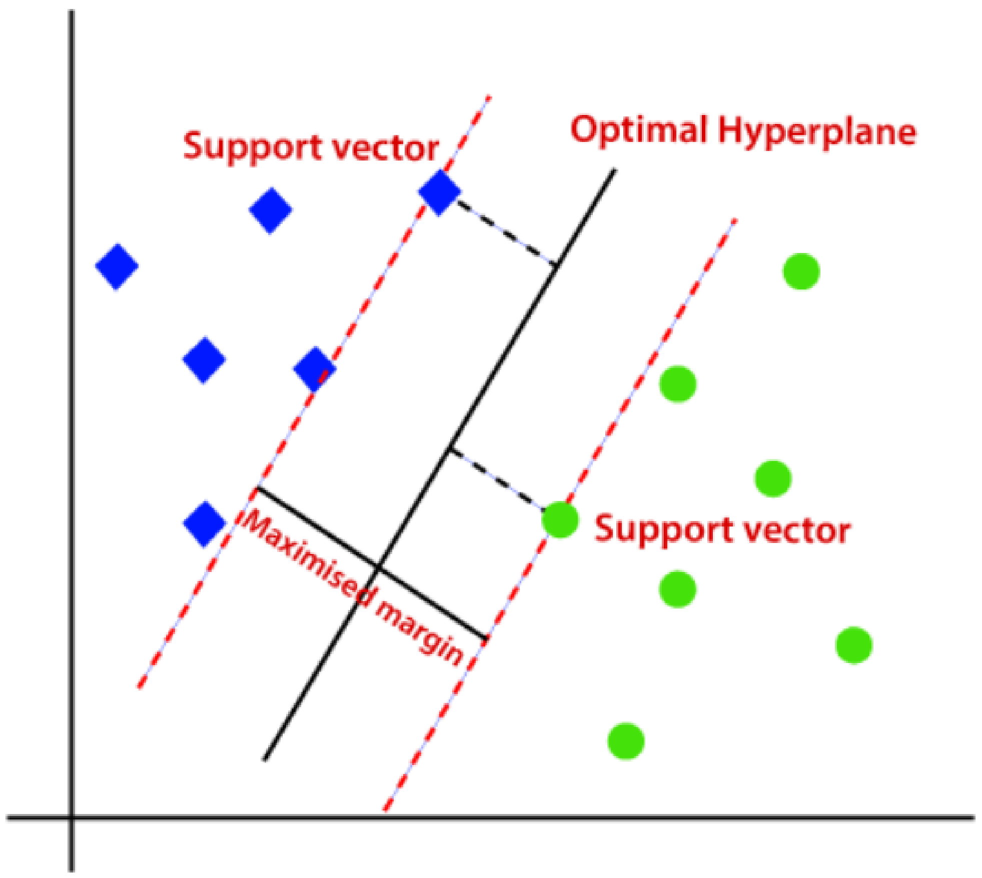

Fig. 7. Example of a maximal margin classifier

2) Support Vector Machines: The support vector classifier has two main differences from the maximal margin classifier: it uses a *soft margin*, that allows some observation to be in the wrong side of the margin or even the hyperplane, and it uses *kernels* to enlarge the feature space, that are functions that quantify the similarity between two observations.

Starting from the soft margin, its major benefit is that it allow greater robustness to single observations, while granting better classification of most of the training observations. The soft margin alter the optimization problem above by modifying the last equation and adding a constraint:

$$y_i(\beta_0 + \beta_1 X_{i1} + ... + \beta_p X_{ip}) \geq M(1 - \epsilon_i)$$

$$\epsilon_i \geq 0, \ \sum_{i=1}^{n} \epsilon_i \leq C$$

where C is a non-negative tuning parameter, which determines the number and severity of the violations allowed, and ϵ_1, ... , ϵ_n are *slack variables*, that allow individual observations to be on the wrong side of the margin or hyperplane.

Furthermore, one can decide to consider enlarging the feature space to address classification problems that are not linear. This can be done using functions of the predictors, such as quadratic or cubic terms, but then finding a solution to the optimization problem will require a lot more computing power. A smarter and more efficient way to enlarge the feature space is to use kernels, generalizations of the inner product between two observations of the form $K(x_i, x_{i'})$. For example, the linear kernel is defined as:

$$K(x_i, x_{i'}) = \sum_{j=1}^{p} x_{ij} \cdot x_{i'j}$$

and the linear support vector classifier can be represented as:

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i K(x, x_i)$$

where there are *n* parameters α_i, one per training observation. Note that only the α_i of the support vectors will be different from 0, for the property discussed above.

3) Multi-Class SVC: As we discussed, the support vector classifier works by separating the (enlarged) feature space in 2 parts, so it can be applies only on binary classification problems. To extend it for multi-class scenario, there are two approaches:

- *One-Versus-One*: this approach constructs and trains one SVC for each pair of classes in the dataset, comparing them. The test classification is performed by assigning to the sample the class to which it was most frequently assigned by every pair-wise SVC.
- *One-Versus-All*: In this approach *K* SVCs, one per class, are created and fitted. Each SVC compares one of the classes to the remaining *(K-1)*. The test classification is achieved by assigning to the sample the class *k*, where *k* comes from the related SVC (which compares class *k* to the rest) that had the biggest confidence in assigning to the sample its class compared to the remaining ones.

4) Hyperparameters: The hyperparameters used in the cross-validation and their set of possible values are shown in the table on the right of Table I. These are:

- C: Regularization parameter for the soft margin. It must be strictly positive and the strength of the regularization is inversely proportional to C.
- gamma: Kernel coefficient for ‘rbf’, ‘poly’ and ‘sigmoid’ kernels.
- kernel: Specifies the kernel type to be used in the algorithm.
- decision_function_shape: Whether to return a one-vs-rest decision function or the original one-vs-one decision function. However, internally, the one-vs-one multi-class strategy is always used in training, so an one-vs-rest matrix is only constructed from the one-vs-one matrix.

VI. Results and Conclusions

After training and validating with each parameters’ configuration using cross-validation, the best performing model is saved and used to predict classes on the test set.

The models show a good adaptation to the patterns in the dataset, and the confusion matrices and the f1 scores confirm that the pre-processing steps applied to the training data helped the generalization and accuracy of the trained models. In the following sections we analyze the results more in depth.

*A. **Random Forest Classifier***

The accuracy and f1 score are reported in the left table of Table II, and the best hyperparameters are highlighted in Table I. Overall, the random forest classifier seems to adapt well to the pre-processed dataset, and the performances are satisfying. A larger amount of decision trees, with a high depth is preferred, and the test scores confirm that they don’t lead to over-fitting. As expected, bootstrap helps in obtaining better performances, while the criterion chosen between gini and cross-entropy doesn’t alter the performances that much.

Figure 8 shows, on the left, the confusion matrix obtained from the model’s predictions. The only classes that seem to be very close to each other are 5 and 6, and in fact around 40 instances each are miss-classified. There are also some sistematic errors considering classes pairs 0-6 and 3-1. Besides this, that may be due to the similarities between bean species, the correctly classified instances are the absolute majority, regardless of the class considered.

Figure 9 shows instead the importance of the features when considering the best split in a decision tree. As expected, Major Axis Lenght and Minor Axis Lenght are the two strongest features. It is interesting to notice, though, that the shape factors, which are some values extracted from the characteristic shape of each bean type, come immediately after the two features mentioned before. Also. with the exception of the last 3 attributes, each one seems pretty important in describing the bean.

*B. **Support Vector Classifier***

The accuracy and f1 score are reported in the table on the right of Table II, and the best hyperparameters are highlighted in Table I. The support vector classifier shows slightly better performances compared to the random forest classifier. The high value of *C* shows that a strong margin doesn’t generalize well in our 7-classes scenario, and the one-vs-rest strategy is preferred, even if ’artificial’.

Furthermore, the Radial Basis Function kernel works better than the standard linear one. The ’rbf’ kernel is defined as:

$$K(x_i, x_{i'}) = exp(-\gamma||x_i - x_{i'}||^2)$$

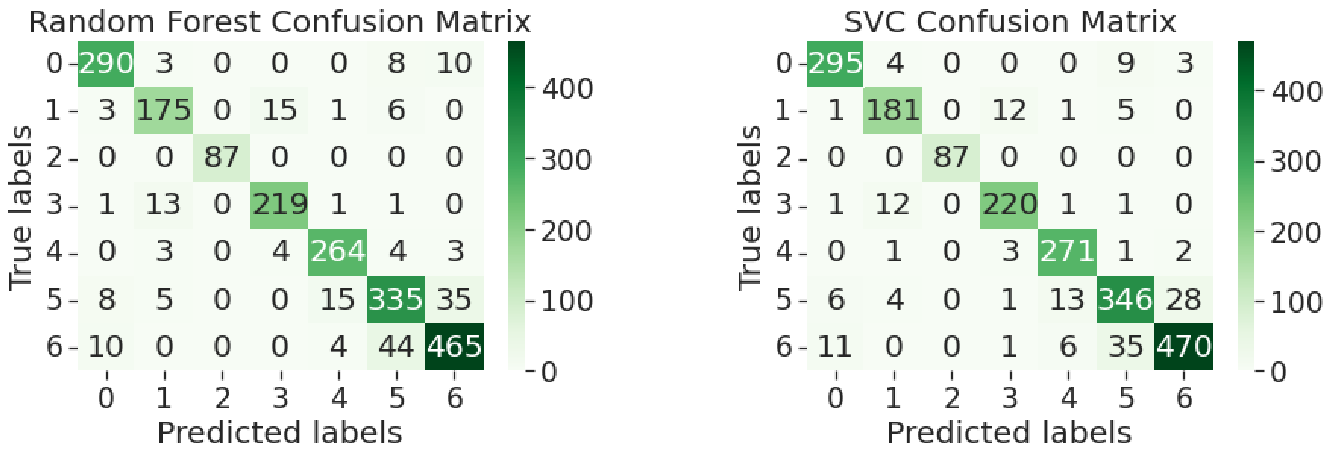

Fig. 8. Random forest classifier (left) and SVC (right) evaluation Confusion Matrix

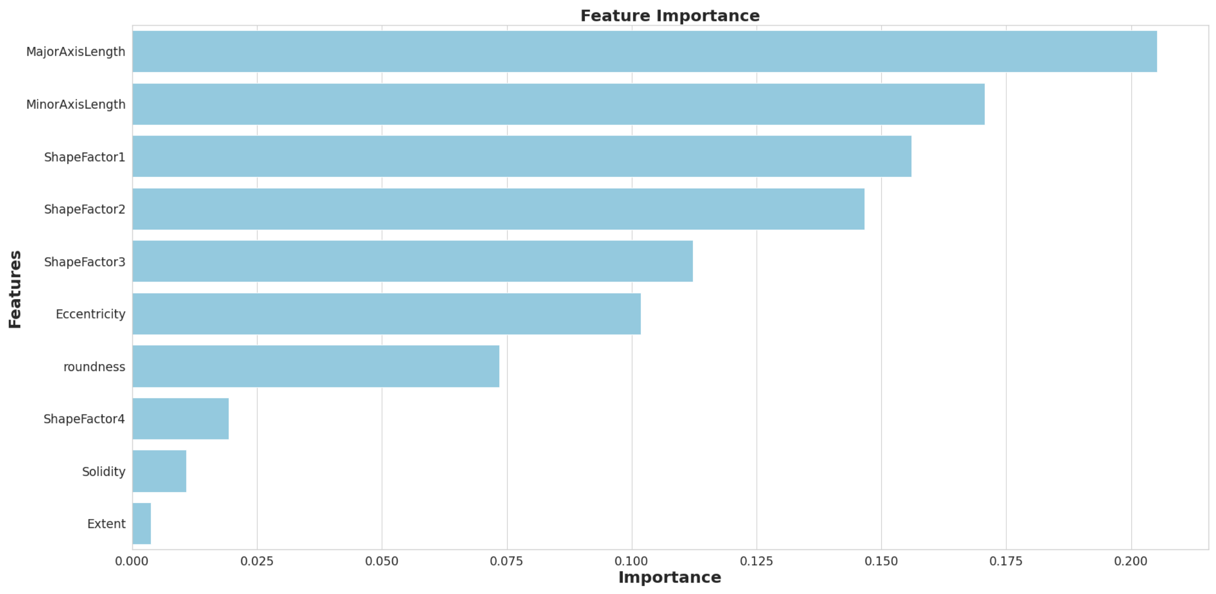

Fig. 9. Feature importance when considering the best split in a decision tree

where $||x_i - x_{i'}||^2$ is the squared Euclidean distance between the two feature vectors, and γ is a free parameter, in our case equal to 0.1. A visualization of the radial basis function kernel applied on a binary classification can be seen in Figure 10.

Figure 8 shows, on the right, the confusion matrix obtained from the model's predictions. The number of correct classifications is a bit higher, but the same consideration made for the random forest confusion matrix still apply here.

$$X_i < q_{0.01}$$

$$X_i > q_{0.99}$$

Parameters	Values				
n_estimators	100	200	**300**		
max_features	**'log2'**	'sqrt'	0.33	0.6	0.8
max_depth	5	10	15	**20**	
min_samples_split	2	**4**	6	8	
bootstrap	**'True'**	'False'			
criterion	'gini'	**'entropy'**			

Parameters	Values				
C	0.1	0.5	1	10	**100**
gamma	1	0.75	0.25	**0.1**	0.01
kernel	'linear'	**'rbf'**	'poly'	'sigmoid'	
decision_function_shape	'ovo'	**'ovr'**			

TABLE I
RANDOM FOREST (LEFT) AND SVC (RIGHT) HYPERPARAMETERS GRID

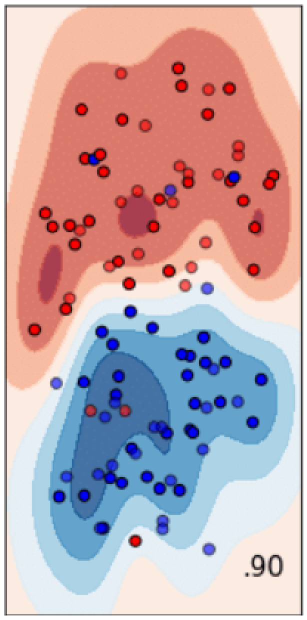

Fig. 10. Example of radial basis function kernel

Model	Val accuracy	Test accuracy	Test F1 score
Random Forest	0.9794	0.9040	0.9038
SVM	0.9775	0.9203	0.9201

TABLE II
ACCURACY AND F1 SCORES

Random Forest	
Val accuracy	0.9794
Test accuracy	0.9040
Test F1 score	0.9038

TABLE III
ACCURACY AND F1 SCORES RANDOM FOREST

SVC	
Val accuracy	0.9775
Test accuracy	0.9203
Test F1 score	0.9201

TABLE IV
ACCURACY AND F1 SCORES SVC