Edible and Poisonous Mushrooms Classification by Machine Learning Algorithms

1st Vo Minh Thanh
Computer Science, UIT
HCMC, Viet Nam
21522607@gm.uit.edu.vn

2nd Tran Thi Ngoc Nhan
Computer Science, UIT
HCMC, Viet Nam
21522410@gm.uit.edu.vn

3rd Nguyen Nguyen Khoi
Computer Science, UIT
HCMC, Viet Nam
21521009@gm.uit.edu.vn

4th Nguyen Do Quynh Nhu
Computer Science, UIT
HCMC, Viet Nam
21521243@gm.uit.edu.vn

***Abstract*—There are millions of mushroom species growing worldwide, consisting of both edible and poisonous types. Distinguishing between edible and poisonous mushrooms is a challenging task that requires expertise. Therefore, the classification of edible and poisonous mushrooms holds significant importance. Machine learning algorithms provide an alternative approach to classify these mushrooms by utilizing their morphological or physical features. The dataset used in this study is the Mushroom dataset available in the UC Irvine Machine Learning Repository. Based on 15 features in the Mushroom dataset and four different machine learning algorithms, models have been created for the classification of edible and poisonous fungi. Remarkably, these models achieved a 100% success rate in classification using Support Vector Machine, K Nearest Neighbors, Decision Tree, and Random Forest algorithms.**

***Index Terms*—Edible mushrooms, Poisonous mushrooms, Machine learning algorithms, Mushroom dataset, Classification.**

I. INTRODUCTION

Mushrooms are diverse organisms found in various ecosystems globally, encompassing a vast number of species. However, differentiating between edible and poisonous mushrooms can be a complex task that demands expertise and knowledge. The consequences of misidentification can be severe, leading to food poisoning or even fatalities. Therefore, accurate classification of mushrooms into edible and poisonous categories is of paramount importance.

Traditionally, mushroom classification relied on the expertise and extensive knowledge of mycologists. However, this approach is time-consuming, subjective, and prone to errors. In recent years, machine learning algorithms have emerged as powerful tools to automate the classification process by leveraging the morphological and physical characteristics of mushrooms.

This paper aims to explore the application of machine learning algorithms in the classification of edible and poisonous mushrooms. The study utilizes the Mushroom dataset, a well-known and widely used dataset in this domain, providing comprehensive information on 22 mushroom features.

To achieve accurate classification, four popular machine learning algorithms are employed: Support Vector Machine (SVM), K Nearest Neighbors (KNN), Decision Tree, and Random Forest. These algorithms have demonstrated their effectiveness in various classification tasks, and their application to mushroom classification holds significant potential.

The primary objective of this research is to develop robust machine learning models that can accurately classify mushrooms into edible and poisonous categories. The performance of the models will be evaluated based on metrics such as accuracy, precision, recall, and F1-score. The findings of this study can contribute to the development of automated systems for mushroom classification, enhancing food safety and public health.

II. MATERIAL AND METRICS

A. Data

1. Title: Secondary mushroom data
2. Sources:
   (a) Mushroom species drawn from source book: Patrick Hardin.Mushrooms & Toadstools.Zondervan, 1999
   (b) Inspired by this mushroom data: Jeff Schlimmer.Mushroom Data Set. Apr. 1987. url:https://archive.ics.uci.edu/ml/datasets/Mushroom.
   (c) Repository containing the related Python scripts and all the data sets: https://mushroom.mathematik.uni-marburg.de/files/
   (d) Author: Dennis Wagner
   (e) Date: 05 September 2020
3. Relevant information: This dataset includes 61069 hypothetical mushrooms with caps based on 173 species (353 mushrooms per species). Each mushroom is identified as definitely edible, definitely poisonous, or of unknown dibility and not recommended (the latter class was combined with the poisonous class). Of the 20 variables, 17 are nominal and 3 are metrical.
4. Data simulation: The related Python project (Sources (c)) contains a Python module `secondary_data_generation.py` used to generate this data based on `primary_data_edited.csv` also found in the repository. Both nominal and metrical variables are a result of randomization. The simulated and ordered by species version is found in `secondary_data_generated.csv`. The randomly shuffled version is found in `secondary_data_shuffled.csv`.

5. Class information: class poisonous=p, edibile=e (binary)
6. Variable Information:
   (n: nominal, m: metrical; nominal values as sets of values)
   1. cap-diameter (m): The diameter of the mushroom's cap, measured in centimeters.
   2. cap-shape (n): The shape of the mushroom's cap, represented by letters such as bell=b, conical=c, convex=x, flat=f, sunken=s, spherical=p, others=o.
   3. cap-surface (n): The surface texture of the mushroom's cap, indicated by letters such as fibrous=i, grooves=g, scaly=y, smooth=s, dry=d, shiny=h, leathery=l, silky=k, sticky=t, wrinkled=w, fleshy=e.
   4. cap-color (n): The color of the mushroom's cap, represented by letters such as brown=n, buff=b, gray=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y, blue=l, orange=o, black=k.
   5. does-bruise-bleed (n): Indicates whether the mushroom's cap bruises or bleeds, with letters bruises-or-bleeding=t, no=f.
   6. gill-attachment (n): The attachment of the gills to the mushroom's stem, represented by letters such as adnate=a, adnexed=x, decurrent=d, free=e, sinuate=s, pores=p, none=f.
   7. gill-spacing (n): The spacing between the gills on a single gill surface, indicated by letters such as close=c, distant=d, none=f.
   8. gill-color (n): The color of the gills, following the same color codes as the cap-color column, or none (f).
   9. stem-height (m): The height of the mushroom's stem, measured in centimeters.
   10. stem-width (m): The width of the mushroom's stem, measured in millimeters.
   11. stem-color (n): The color of the stem, following the same color codes as the cap-color column, or none (f).
   12. has-ring (n): Indicates whether the mushroom has a ring on the stem, with letters ring (t) or none (f).
   13. ring-type (n): The type of ring on the stem, represented by letters such as cobwebby (c), evanescent (e), flaring (r), grooved (g), large (l), pendant (p), sheathing (s), zone (z), scaly (y), movable (m), none (f).
   14. habitat (n): The habitat where the mushroom is found, represented by letters such as grasses (g), leaves (l), meadows (m), paths (p), heaths (h), urban (u), waste (w), or woods (d).
   15. season (n): The season in which the mushroom is typically found, represented by letters such as spring (s), summer (u), autumn (a), or winter (w).

B. Performance Metrics :
Accuracy, precision, recall and F1_Score metrics are used to evaluate a classifier model.

Mushroom Dataset		Predicted	
		Edible	**Poisonous**
Actual	**Edible**	**(TP)** The number of correctly predicted edible mushroom	**(FN)** Predicted number of edible mushrooms as poisonous mushrooms
	Poisonous	**(FP)** Predicted number of poisonous mushrooms as edible mushrooms	**(TN)** The number of correctly predicted poisonous mushrooms

Fig. 1. CONFUSION MATRIX.

1. Accuracy :

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total Samples}}$$

Accuracy measures the overall correctness of the model's predictions. It represents the ratio of correctly classified samples to the total number of samples.

2. Precision :

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Precision quantifies the model's ability to correctly identify positive samples. It measures the ratio of correctly predicted positive samples to the total number of samples predicted as positive.

3. Recall

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

Recall, also known as sensitivity or true positive rate, measures the model's ability to identify all positive samples. It calculates the ratio of correctly predicted positive samples to the total number of actual positive samples.

4. F1_score

$$\text{F1-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

The F1-score is the harmonic mean of precision and recall. It provides a balanced measure that takes both precision and recall into account.

C. Hyperparameter Tuning: GridSearchCV is a technique used for evaluating and optimizing models by searching over a predefined grid of hyperparameters. It is commonly used to find the best combination of hyperparameters that yields the highest performance for a given model.
The process of GridSearchCV involves the following steps:

1. Define the Model: First, you need to choose a model or an estimator that you want to evaluate and optimize. This could be any machine learning algorithm, such as decision trees, support vector machines, or random forest.

2. Define the Hyperparameter Grid: Next, you need to define a grid of hyperparameters that you want to search over. Hyperparameters are the configuration settings of the model that are not learned from the data, such as learning rate, number of hidden layers, or regularization strength. The grid should include different values or ranges for each hyperparameter.
3. Cross-Validation: GridSearchCV uses cross-validation to evaluate the performance of each combination of hyperparameters. Typically, k-fold cross-validation is used, where the data is divided into k subsets or folds. The model is trained on k-1 folds and evaluated on the remaining fold. This process is repeated k times, with each fold serving as the test set once. The performance metrics, such as accuracy or F1_score, are averaged over the k iterations.
4. Evaluation: For each combination of hyperparameters, the model is trained and evaluated using cross-validation. The average performance metric is calculated, and the combination with the highest performance is selected as the best hyperparameter setting.

GridSearchCV is an iterative and computationally expensive process as it exhaustively searches over all possible combinations of hyperparameters. However, it provides a systematic approach to find the optimal hyperparameter configuration for a model.

D. Data analysis & visualization

a) Counting values in each column:

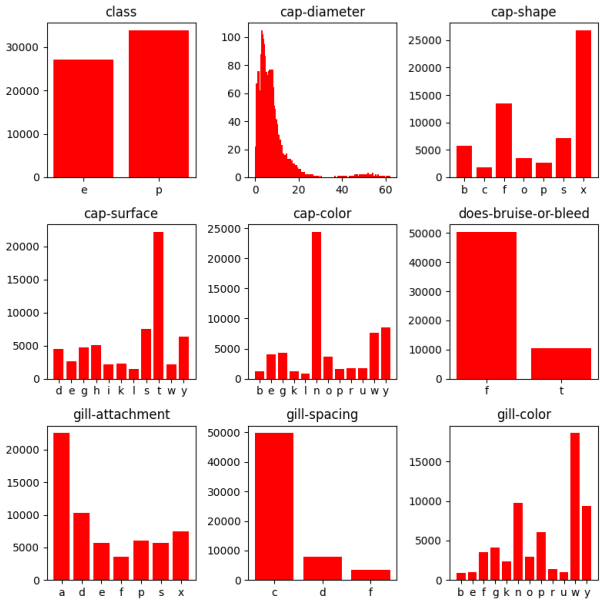

Fig. 2. Counting values in each column

Visualizing the ”class” column, it is clear that the data is fairly balanced. In details, the number of poisonous mushrooms and the number of edible ones differ by 6707. Most mushrooms have yellow, white or brown cap and diameter under 20cm long. Bruises or bleeding could be seen in about $\frac{1}{6}$ of the mushroom population. In most mushrooms, there exists some spaces between the gills but they are usually close.

b) Distribution of edible and poisonous mushrooms over various features:
As can be seen from the chart below, most mushrooms have no special ring type. However, zone-ring mushrooms are mostly poisonous and movable ones are mostly edible.

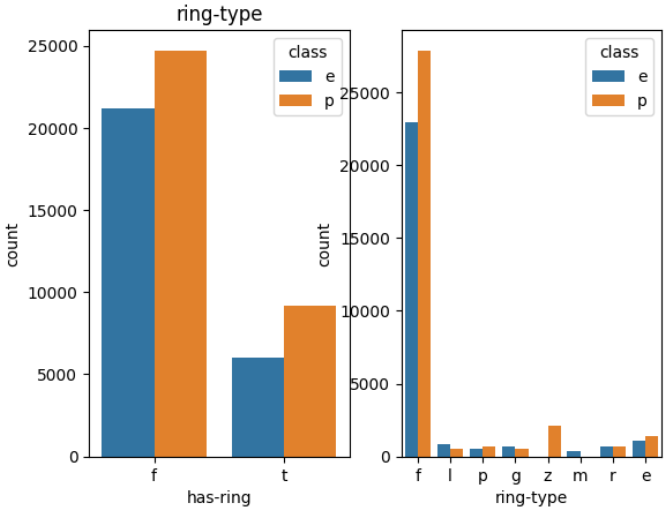

Fig. 3. Distribution of poisonous and edible mushrooms over different ring types

As can be seen from the chart below, most pink-, green- and non-color-stem mushrooms are poisonous. In contrast, most buff-stem mushrooms are edible.

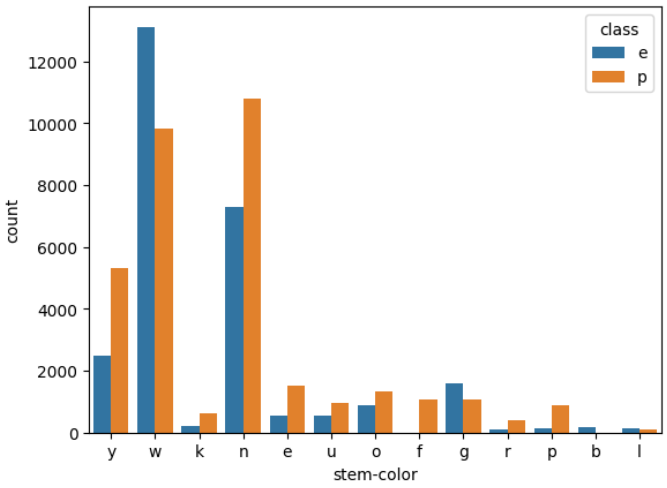

Fig. 4. Distribution of poisonous and edible mushrooms over stem color

As can be seen from the chart below, edible mushrooms have a wider range of stem width than poisonous mushrooms. While the stem of an edible mushrooms may be slightly larger than 0 to more than 100, a

poisonous one has stem that is most likely to be under 60cm wide.

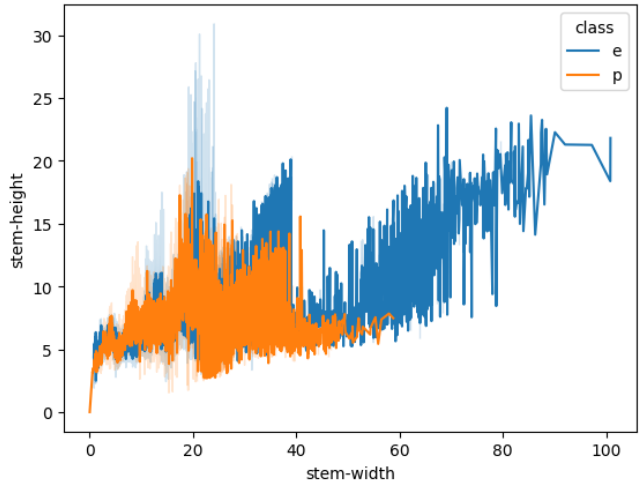

Fig. 5. Distribution of poisonous and edible mushrooms over stem color

Also, edible mushrooms have a wider range of stem height and cap diameter than poisonous ones

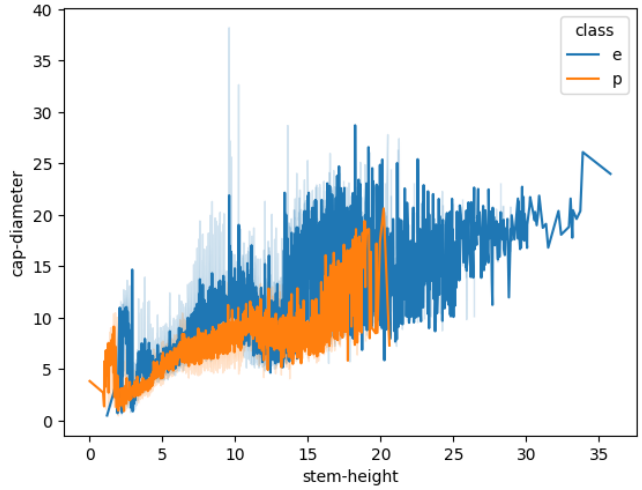

Fig. 6. Distribution of poisonous and edible mushrooms over stem color

III. Methods

A. SVM

1) Definition: SVM (Support Vectors Machine) is a supervised machine learning algorithm used for both classification and regression tasks. Its primary objective is to find the optimal hyperplane that best separates different classes in a dataset. [1] In the context of binary classification, SVM works by finding the hyperplane that maximizes the margin between the two classes. The margin is the distance between the hyperplane and the nearest data points of each class. These data points, known as support vectors, are critical in defining the hyperplane and the margin. [1]

2) Linear SVM: In this report, we use linear SVM because of this 4 reasons:
- Simplification: Linear SVM is a simpler model than SVM with kernel function, which can help avoid overfitting in cases where the data set is not too complex.
- Computational complexity: Linear SVM has lower computational complexity than SVM with kernel functions, especially when the number of data points is large.
- Speed: Linear SVM usually has faster training speed than SVM with kernel functions.
- Generalization: In some cases, when the data set does not contain many important and complexly correlated features, Linear SVM can generalize well and perform highly.

3) Mathematical foundation of linear SVM (Hard-Margin SVM): The SVM technique is a classifier that finds a hyperplane or a function $g(x) = w^T x + b$ that correctly separates two classes with a maximum margin. Figure 1 shows a separating hyperplane corresponding to a hard-margin SVM. [2] Mathematically speaking, given a set of points xi that

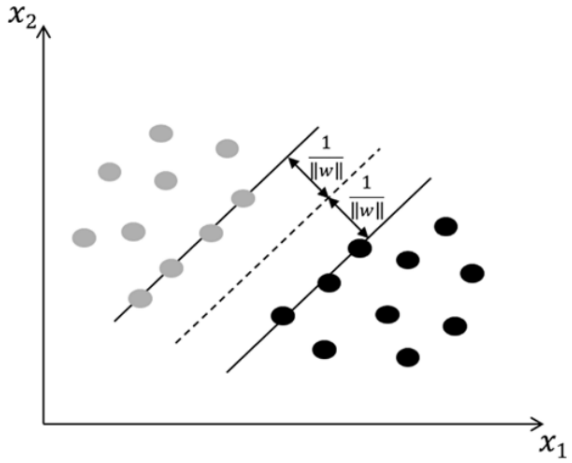

Fig. 7. Hard-maximum-margin separating hyperplane.

belong to two linearly separable classes w_x1, w_2 the distance of any instance from the hyperplane is equal to $\frac{|g(x)|}{||w||}$. SVM aims to find w, b such that the value of g(x) equals 1 for the nearest data points belonging to class w_1 and –1 for the nearest ones of w_2. This can be viewed as having a margin of:

$$\frac{1}{||w||} + \frac{1}{||w||} = \frac{2}{||w||}$$

whereas $w^T x + b = 1$ for $x \in w_1$, and $w^T x + b = -1$ for $x \in w_2$.
This leads to an optimization problem that minimizes the objective function:

$$J(w) = \frac{1}{2}||w||^2$$

subject to the constraint:

$$y_i(w^T x + b) \geq 1,\ i = 1, 2, ...N$$

When an optimization problem—whether minimization or maximization—has constraints in the variables being optimized, the cost or error function is augmented by adding to it the constraints, multiplied by the La-

grange multipliers. In other words, the Lagrangian function for SVM is formed by augmenting the objective function with a weighted sum of the constraints:

$$L(w, b, \lambda) = \frac{1}{2} w^T w - \sum_{i=1}^{N} \lambda_i [y_i (w^T x_i + b) - 1]$$

where w and b are called primal variables, and λ_i is the Lagrange multipliers. The dual problem of SVM optimization is to find:

$$max(\sum_{i=1}^{N} \lambda_i - \frac{1}{2} \sum_{i,j} \lambda_i \lambda_j y_i y_j x_i x_j)$$

subject to

$$\sum_{i=1}^{N} \lambda_i y_i = 0,\ \lambda_i \geq 0 \forall i[2]$$

4) Hyperparameters: SVM has several important hyperparameters that need to be defined before training the model. These hyperparameters impact the model's performance and accuracy. The key hyperparameters are:

- C: float, default=1.0: The regularization parameter C determines the tolerance for misclassification (allowing support points to be on the wrong side of the hyperplane) during optimization. A larger C value makes the model attempt to classify as many training data points correctly as possible, but it may lead to overfitting. On the other hand, a smaller C value makes the model more flexible but may reduce accuracy.
- kernel: string, default='rbf': Select the kernel function to transform the data into a new space. Common choices include 'linear', 'poly', 'rbf' (Radial Basis Function), and 'sigmoid'. The 'linear' kernel is suitable for linear data, while 'poly' and 'rbf' kernels often work well with non-linear data.
- degree: int, default=3: This parameter applies only when the kernel is 'poly'. It determines the degree of the polynomial kernel function.
- gamma: 'scale', 'auto' or float, default='scale': This parameter applies only when the kernel is 'rbf', 'poly', or 'sigmoid'. Gamma determines the influence of a training data point on the classification decision. If gamma is too large, the model may overfit.
- shrinking: bool, default=True: Determine whether the SVM algorithm uses the shrinking technique to remove unnecessary support vectors. Typically, using shrinking can speed up the training process.
- probability: bool, default=False: Determine whether the SVM algorithm computes probability estimates for the output classes. If you want to use probability-based classification, it is necessary to evaluate the model after building it.

5) Hyperparameters tunning: Tuning hyperparameters for an SVM model involves finding the best values for parameters that maximize performance. GridSearch CV is chosen for its exhaustive search, simplicity, optimal performance estimation through cross-validation, and comprehensive evaluation capabilities. It efficiently automates the process, ensuring the best SVM model for the given dataset.

We use some hyperparameter as the following below:

- C: [0.1, 1, 10]
- kernel: linear
- ganma: scale

B. KNN

1) Definition: K Nearest Neighbors (KNN) classifies a given based on the groups of its surrounding 'k' number of neighbors. It uses feature similarity to predict the cluster that the new point will fall into.

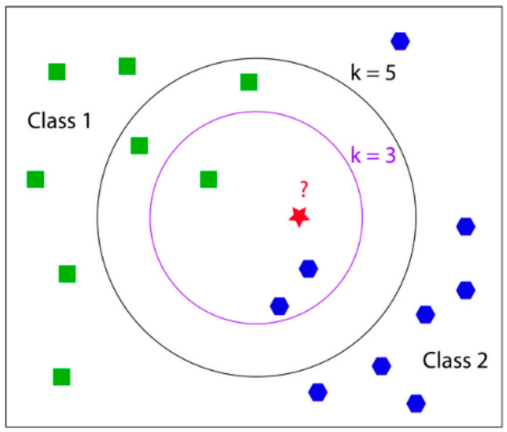

Fig. 8. An illustration of the KNN algorithm.

2) Basic Principle:

- Nearest Neighbor Search: Given a new, unlabeled data point, the KNN algorithm searches for its K nearest neighbors in the training dataset based on a distance metric. The distance can be computed using various measures, such as Euclidean distance, Manhattan distance, or Minkowski distance.
- Majority Voting: Once the K nearest neighbors are identified, the algorithm assigns a class label to the new data point based on the majority class among its neighbors. In other words, the class label of the new data point is determined by the most common class label among its K nearest neighbors.
- Non-Parametric and Instance-Based: KNN is considered a non-parametric algorithm because it does not assume any specific form for the decision boundary or the underlying data distribution. Instead, it directly uses the training instances to make predictions. It is also instance-based because it stores the entire training dataset and uses it directly during the prediction phase. Each instance in the training dataset serves as a data point in the feature space.
- Feature Scaling: It is important to scale the features of the dataset before applying KNN. Since KNN relies on distance calculations, features with larger scales can dominate the distance computations. Therefore, it is common practice to normalize or standardize the feature values to ensure all features contribute equally.

3) Distance Measures:

a) Euclidean Distance: Euclidean Distance is defined as crow flies. It calculates the distance between two real-valued vectors. Euclidean distance is calculated as the square root of the sum of the squared differences between the two vectors.

$$
\begin{aligned}
d(p,q) &= d(q,p) \\
&= \sqrt{(q_1-p_1)^2+(q_2-p_2)^2+...+(q_n-p_n)^2} \\
&= \sqrt{\sum_{i=1}^{n}(q_i-p_i)^2}
\end{aligned}
\tag{1}
$$

In (1), q and p are two different points which have n dimensions

b) Manhattan: Manhattan Distance also called the Taxicab distance or the City Block distance, calculates the distance between two real-valued vectors. It is perhaps more useful to vectors that describe objects on a uniform grid, like a chessboard or city blocks. The taxicab name for the measure refers to the shortest path that a taxicab would take between city blocks (coordinates on the grid). Manhattan distance is calculated as the sum of the absolute differences between the two vectors.

$$
d(p,q) = d(q,p) = \sqrt{\sum_{i=1}^{n}|q_i-p_i|} \tag{2}
$$

In (2), q and p are two different points which have n dimensions

c) Minkowski Distance: The Minkowski distance is a generalized distance measure that includes both the Euclidean and Manhattan distances as special cases. It is defined as:

$$
\left(\sum_{i=1}^{n}|p_i-q_i|^h\right)^{\frac{1}{h}} \tag{3}
$$

In (3):

- q and p are two different points which have n dimensions
- h represents the order of the Minkowski distance. When h is set to 1, the calculation is the same as the Manhattan distance. When h is set to 2, it is the same as the Euclidean distance.

4) Process: In the training phase, the model will only store the data points.

In the testing phase, all the distance from the query point to the other points from the training phase is calculated to classify each point in the test dataset.

5) Pros and cons:

a) Pros:

- The algorithm is simple and easy to comprehend and implement.
- The training phase of K-nearest neighbor classification is much faster compared to other classification algorithms, as there is no need to train a model for generalization. This is why K-NN is known as the simple and instance-based learning algorithm.
- K-NN does provide a relatively high accuracy compared to other algorithms.
- K-NN can be useful in case of nonlinear data. It can be used with the regression problem. Output value for the object is computed by the average of k closest neighbors value.

b) Cons:

- 'Time complexity' and 'space complexity' is enormous, which is a major disadvantage of K-NN.
  Time complexity refers to the time the model takes to evaluate the class of the query point, while space complexity refers to the total memory used by the algorithm. If we have n data points in training and each point is of m dimension $->$ Then time complexity is of order O(nm), which will be huge if we have higher dimension data. Therefore, K-NN is not suitable for high dimensional data.
- Another disadvantage is called 'The curse of dimensionality'.
  It is important to mention that K-NN is very susceptible to overfitting due to the curse of dimensionality. The curse of dimensionality describes the phenomenon where the feature space becomes increasingly sparse for an increasing number of dimensions of a fixed-size training dataset. Intuitively, we can think of even the closest neighbors being too far away in a high-dimensional space to give a good estimate.

6) Hyperparameters: K-Nearest Neighbors only has one hyperparameter 'k', which stands for the number of neighbors required for computation purpose. One thing to notice here, if the value of K is even, it might create problems when taking a majority vote because the data has an even number of classes. Therefore, choose K as an odd number when the data has an even number of classes and an even number when the data has an odd number of classes.

7) Tuning hyperparameters: For a better result, we employ Grid Search to obtain the optimal hyperparameter value. Grid search is a tuning technique that attempts to compute the optimum values of hyperparameters. It is an exhaustive search that is performed on the specific parameter values of a model. In the process, it will be searching all possible hyperparameter values, evaluating, memorizing, and then produce the best value for the model.

In K-Nearest Neighbors, the only hyperparameter is the 'k' number of neighbors, so we give Grid Search a range of 'k' values from 1 to 30 for input. In addition, Scikit-learn offers us another attribute for computing the metric distance as well as weights function. About the metric to use for distance computation, we use 'Euclidean' metric, 'Manhattan' metric and 'Minkowski' metric to figure out the distance. We also use 'uniform' and 'distance' weights function for prediction.

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.

After processing, with default value 'weights':'uniform' and 'metric':'minkowski', the optimal value of n_neighbors we have is 1.

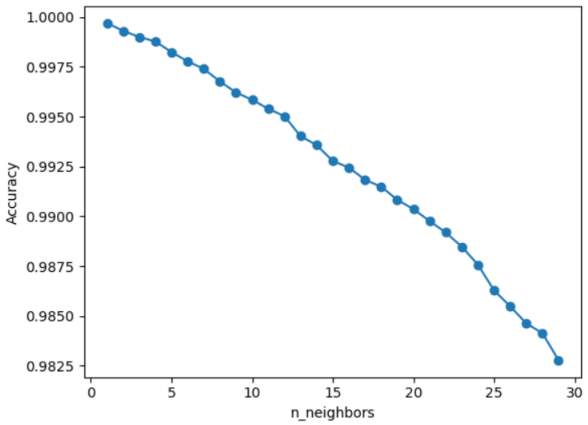

Fig. 9. Mean Cross-Validation Results with Different K Values

C. Decision Tree

Decision Tree is a supervised learning algorithm. Unlike other supervised learning algorithms, decision tree could be used for both regression and classification problems.

The goal of using a Decision Tree is to create a training model that can use to predict the class or value of the target variable by learning simple decision rules inferred from prior data (training data). Decision tree algorithm is a data mining induction techniques that recursively partitions a dataset of records using depth-first greedy approach or breadth-first approach until all the data items belong to a particular class.

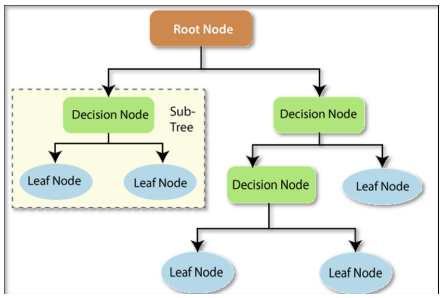

Fig. 10. Example of Decision Tree.

A decision tree structure is made of root, internal, and leaf nodes. It is a flow chart like tree structure, where every internal node denotes a test condition on an attribute, each branch represents the result of the test condition (rule), and each leaf node (or terminal node) shows an outcome (categorical or continues value). A root node is the parent of all nodes. As the name suggests, it is the topmost node in the tree. As decision tree mimics the human level thinking, it is simple to grab the data and make some good interpretations. Decision tree is constructed in a divide and conquer approach. Each path in the decision tree forms a decision rule. Generally, it utilizes greedy approach from top to bottom.

1) Classification Tree: Decision tree technique is performed in two phases: tree building and tree pruning. Tree building is performed in top-down approach. During this phase, the tree is recursively partitioned till all the data items belong to the same class label. It is very computationally intensive as the training dataset is traversed repeatedly. Tree pruning is done in a bottom-up manner. It is used to improve the prediction and classification accuracy of the algorithm by minimizing over-fitting problem of tree. Over-fitting issue in decision tree results in misclassification errors.

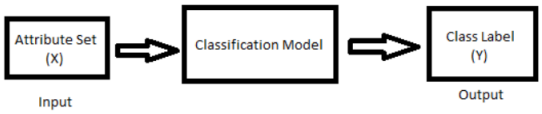

Fig. 11. Flowchart of Classification Tree.

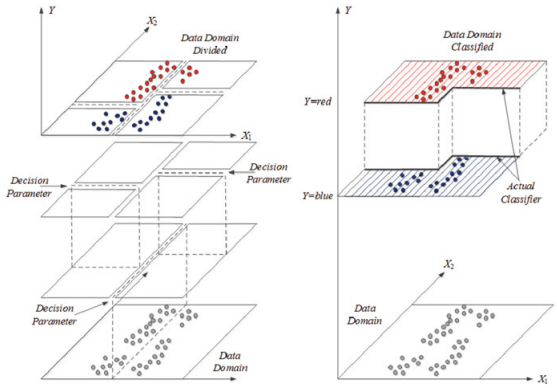

Fig. 12. The Classification Tree is illustrated in 3D using two classes with domain division properties. Response variable Y has two discrete values, red or blue.

2) Decision Tree algorithm: Decision Tree algorithm is used to split the attributes to test at any node to determine whether splitting is the "best" in individual classes. The resulting partitioned at each branch is PURE as possible, for that splitting criteria must be identical.

There are several types of Decision Tree algorithms such as: *Iterative Dichotomies 3* (ID3); *Successor of ID3* (C4.5); *Classification And Regression Tree* (CART); *Chi-squared Automatic Interaction Detector* (CHAID); *Multivariate Adaptive Regression Splines* (MARS); *Generalized, Unbiased, Interaction Detection and Estimation* (GUIDE); *Conditional Inference Trees* (CTREE); *Classification Rule with Unbiased Interaction Selection and Estimation* (CRUISE); *Quick, Unbiased, Efficient, Statistical Tree* (QUEST). Table I represents comparison between frequently used algorithms for the decision tree.

3) Attribute selection measure: : while building a decision tree, the major problem is to select the best attribute from the list of featurs of the dataset for the root and sub nodes. The selection of best attributes is being achieved with the help of a technique known as the Attribute selection measure:

a) Entropy: Entropy is a measure of the randomness in the information being processed. ID3 follows the rule — A

TABLE I
COMPARISON DECISION TREE ALGORITHMS.

Algorithms name	Classification	Description
CART (Classification and Regression Trees)	Uses Gini Index as a metric.	By applying numeric splitting, we can construct the tree based on CART.
ID3 (Iterative Dichotomister 3)	Uses Entropy function and Information gain as metrics.	The only concern with the discrete values. Therefore, the continuous dataset must be classified within the discrete dataset.
C4.5	The improved version on ID3.	Deals with both discrete as well as a continuous dataset. Also, it can handle the incomplete datasets. The technique, called "PRUNNING", solves the problem of over filtering.
C5.0	Improved version of the C4.5.	C5.0 allows to whether estimate missing values as a function of other attributes or apportions the case statistically among the results.
CHAID (Chi-square Automatic Iteration Detector)	Predates the original ID3 implementation.	For a nominal scaled variable, this type of decision tree is used. The technique detects the dependent variable from the categorized variables of a dataset.
MARS (multi-adaptive regression splines)	Used to find the best split.	In order to achieve the best split, we can use the regression tree based on MARS.

branch with an entropy of zero is a leaf node, and A branch with entropy more than zero needs further splitting.

$$E(S) = \sum_{i=1}^{c} -p_i log_2 p_i$$

where S $\rightarrow$ Current state, and Pi $\rightarrow$ Probability of an event i of state S or Percentage of class i in a node of state S.

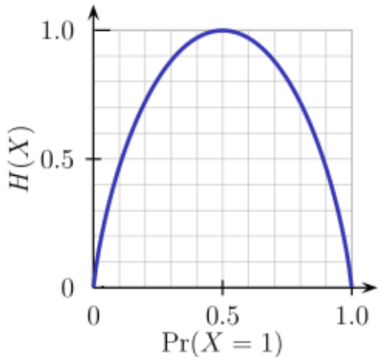

Fig. 13. Entropy graph.

b) Information Gain: Information gain is a statistical property that measures how well a given attribute separates the training examples according to their target classification. Information gain is a decrease in entropy. It computes the difference between entropy before split and average entropy after split of the dataset based on given attribute values.

$$InformationGain = Entropy(bf) - \sum_{j=1}^{k} Entropy(j, at)$$

where *"bf"* is the dataset before the split, K is the number of subsets generated by the split, and *(j, at)* is subset j after the split.

c) Gini index: Gini index is calculated by subtracting the sum of the squared probabilities of each class from one. It favors larger partitions and easy to implement, whereas information gain favors smaller partitions with distinct values. Higher value of Gini index implies higher inequality, higher heterogeneity.

$$Gini = 1 - \sum_{i=1}^{c} (p_i)^2$$

d) Gain Ratio: Gain ratio overcomes the problem of information gain by taking into account the number of branches that would result before making the split. It corrects information gain by taking the intrinsic information of a split into account.

$$GainRatio = \frac{InformationGain}{SplitInfo} = \frac{Entropy(before) - \sum_{j=1}^{k} Entropy(j, after)}{\sum_{j=1}^{k} log_2 w_j} \quad (4)$$

4) Pros and cons: Decision tree has some advantages and disadvantages

Advantages:

- Presentation is easy to understand (comprehensible).
- Be able to classify both categorical and numerical outcomes, but the generated attribute must be categorical.
- Be able to deal with noisy data.
- Quickly translated to a set of principle for production.

Disadvantages:

- Long training time.
- Too many layers makes decision tree extremely complex sometimes.
- Overfitting
- For more class labels, the computational complexity of the decision tree may increase.

5) Hyperparameters: In Decision Tree algorithms, Scikit-learn provides some parameters which support to build Decision Tree model.

a) ***criterion: "gini", "entropy", "log_loss", default="gini"****:* The function *criterion* acts as a strategy to select the features used in the division of node. It measures the split quality at each node. There are three options which are "gini", "entropy" and "log_loss". "gini" criterion is commonly used for the Gini impurity and "log_loss", "entropy" both

for the Shannon information gain. Computationally, entropy is more complex since it makes use of logarithms and consequently, the calculation of the Gini Index will be faster.

b) ***max_depth: int, default=None****:* In most tree algorithms, height, or depth is an indispensable parameter. For Decision Trees, the *max_deep* parameter is defined by the Scikit-learn library as an integer representing the maximum depth of the tree at which the leaf node will be. In the process of applying the problem, depending on the purpose of use and depending on the data set of the problem, we will fine-tune this parameter differently, if choosing a low value will help the model predict faster but less accurate, but if you choose a high value, it will easily lead to overfitting and slow down.

c) ***min_samples_split: int or float, default=2****:* The minimum sample size required to continue the division for the decision node. It is used to avoid the size of the leaf node being too small to minimize overfitting.

d) ***min_samples_leaf: int or float, default=1****:* Similar to *min_samples_split*, the *min_samples_leaf* parameter is the minimum number of leaf nodes required for a parent node to split when the tree has reached its final depth.

e) ***max_features: int, float or "auto", "sqrt", "log2", default=None****:* The number of variables selected to find the best splitter at each split. The higher the *max_feature* value, the more accurate it is, but in return it takes more time.

f) ***max_leaf_nodes: int, default=None****:* The maximum number of leaf nodes in a decision tree. It is usually set up when overfitting needs to be under control.

g) ***min_impurity_decrease: float, default=0.0****:* The tree will not generate child nodes if the impurities are more than the threshold. With the node generation threshold being a real number. The *min_impurity_decrease* parameter is used to strike a balance between overfitting and precision. Low values bring high accuracy, the risk of overfitting also increases and vice versa.

h) ***ccp_alpha: non-negative float, default=0.0****:* Subtrees of lower complexity are dropped. Used to strike a balance between overfitting and precision. Low values bring high accuracy, the risk of overfitting also increases and vice versa.

6) Tuning hyperparameters: : in this project, specifically Decision Tree model, we need to build a model which can solve the problem mentioned above with high accuracy and this model is not overfitting with training data. In order to do this, some worth-considering hyperparameters are:

- **criterion**: What criterion should I use?
- **max_depth**: What should be the maximum allowable depth for each decision tree?
- **min_samples_split** and **min_samples_leaf**: How many sample sizes should be for stopping splitting?
- **max_leaf_nodes**: How many leaf nodes should be for stopping splitting?

To find the hyperparameters for issue above, we use Grid Search Cross Validation to discover the "best" tuple of hyperparameters. With this technique, we simply build a model for each possible combination of all the hyperparameter values provided, evaluate each model, and select the set of hyperparameters which produces the best results.

We use some hyperparameters of decision tree model provided by scikit-learn and set up following:

- **criterion**: ('gini', entropy', 'log_loss')
- **max_depth**: [None] + np.arange(3,10)
- **min_samples_split**: np.arange(2,10,1)
- **min_samples_leaf**: np.arange(1,10,1)
- **max_leaf_node**: [None, 8,16,32]

7) Protocol:

- Step-1: Begin the tree with the root node, says S, which contains the complete dataset.
- Step-2: Find the best attribute in the dataset using Attribute Selection Measure (ASM).
- Step-3: Divide the S into subsets that contains possible values for the best attributes.
- Step-4: Generate the decision tree node, which contains the best attribute.
- Step-5: Recursively make new decision trees using the subsets of the dataset created in step -3. Continue this process until a stage is reached where you could not further classify the nodes and the final node is called a leaf node.

The best set of hyperparameters achieved by Grid Search Cross Validation is {'criterion': 'log_loss', 'max_depth': None, 'max_leaf_nodes': None, 'min_samples_leaf': 1, 'min_samples_split': 3}

D. Random Forest

Random Forest is a popular machine learning algorithm used for classification and regression tasks. It combines multiple decision trees to form a powerful and more stable prediction model compared to using a single decision tree.

1. Definition:
   Random Forest is a machine learning method that consists of an ensemble of many decision trees, where each decision tree is built on a random subset of the training data and then the results from these decision trees are combined to make the final prediction. This technique helps to minimize overfitting and improves the accuracy and stability of the model.
2. Basic Principle:
   Random Forest works by combining the diversity of multiple decision trees through the use of random subsets of the training data and attributes to build individual decision trees. Then, the predictions from the individual trees are combined to form the final prediction. The final decision is made using the voting method in the case of classification or the average in the case of regression.
3. Process:
   The process of building a Random Forest model includes the following steps:

Step 1: Randomly select a subset of the training data from the original dataset.

Step 2: Build a decision tree on the selected training data.

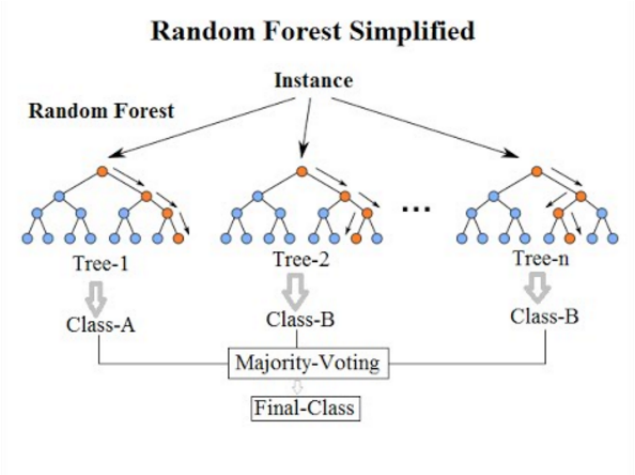

Fig. 14. Example of Decision Tree.

Step 3: Repeat steps 1 and 2 multiple times to create multiple decision trees.

Step 4: Combine the predictions from the decision trees to make the final prediction.

4. Pros and Cons:
   Pros of Random Forest:
   - Good handling of large datasets and noisy data.
   - Automatic handling of missing values in the data.
   - Ability to estimate the importance of features for prediction.
   - Ability to handle both numerical and categorical data.
   - Built-in ability to determine error distribution.

   Cons of Random Forest:
   - Difficult to interpret results compared to simpler algorithms.
   - Requires a large amount of training data to achieve optimal performance.
   - Training time can increase as the number of decision trees and data size increase.
   - Since it is built by multiple decision trees, most of the parameters of the decision tree are also present in the Random Forest.
5. Hyperparameters of Random Forest:
   Random Forest has several hyperparameters that need to be adjusted to optimize the model's performance, including:
   - Number of trees (`n_estimators`): The number of decision trees used in the model.
   - Number of random features (`max_features`): The number of features randomly selected for each decision tree.
   - Maximum depth of the tree (`max_depth`): The maximum depth limit of the decision tree.
   - Minimum number of samples required to split a node (`min_samples_split`).
   - Minimum number of samples required to form a leaf (`min_samples_leaf`).
6. Tuning Hyperparameters of Random Forest:
   Tuning hyperparameters is an important step in optimizing the performance of a Random Forest model. While the default values for hyperparameters in Random Forest are often chosen to work well in many cases, fine-tuning these hyperparameters can further improve the model's accuracy. In order to do this, some worthconsidering hyperparameters are:

   - **n_estimators** = [3,5,7,9,11,15]
   - **min_samples_split** = [20,50,100,150,200,300]
   - **max_depth** = [None, 5, 10]

Tuning the hyperparameters `n_estimators`, `min_samples_split` and `max_depth` in a Random Forest model is crucial for optimizing its performance. Here's why tuning these specific hyperparameters is important:

1. n_estimators:
   The n_estimators hyperparameter determines the number of decision trees in the Random Forest ensemble. Increasing the number of trees can improve the model's performance by reducing overfitting and increasing stability. However, adding too many trees can also increase computational time and memory requirements. Tuning the n_estimators value allows you to strike a balance between model accuracy and computational efficiency.
2. min_samples_split:
   The min_samples_split hyperparameter determines the minimum number of samples required to split an internal node during the construction of a decision tree. Setting a higher value for min_samples_split can prevent overfitting by preventing the trees from splitting nodes with a small number of samples. However, setting it too high can lead to underfitting and decrease model performance. Tuning min_samples_split helps find the optimal value that balances bias and variance in the model.
3. max_depth:
   The max_depth hyperparameter restricts the maximum depth of each decision tree in the Random Forest. Limiting the depth of the trees can help prevent overfitting and reduce computational time. However, setting the value too low can result in underfitting, while setting it too high can lead to overfitting. Tuning the max_depth value helps control the complexity of the trees and find the right balance between model accuracy and generalization.

   By tuning n_estimators, min_samples_split, and max_depth, you can customize the Random Forest model to the specific characteristics of your dataset, achieving better generalization and higher accuracy. It allows you to find the optimal trade-off between model complexity, variance, and bias, resulting in a more robust and effective model.

IV. Experiment

A. Compare the results of the models

In this report, we analyze and present 4 algorithms are KNN, Random forest, SVM and Decision Tree. We use measures: accuracy, f1-score, presicion and recall. The results are shown in the table below (results are rounded to 2 decimal places). In the table, we can see the results of

TABLE II
EVALUATION OF CLASSIFICATION ALGORITHMS(%)

Algorithms	**Acc**	**P**	**R**	**F1**
KNN	100	100	100	100
Tuned KNN	100	100	100	100
Random Forest	100	100	100	100
Tuned Random Forest	100	100	100	100
Support Vector Machines	66	58	64	53
Tuned Support Vector Machines	100	100	100	100
Decision Tree	99.64	99.59	99.59	99.59
Tuned Decision Tree	99.66	99.65	99.59	99.62

the experiments with various machine learning algorithms, namely KNN, Random Forest, SVM, and Decision Tree, for the classification of Edible and Poisonous Mushrooms. The performance of each algorithm is evaluated based on metrics such as accuracy, precision, recall, and F1 score.

Looking at the table, we observe that KNN, Random Forest, and Decision Tree all achieve superior results compared to SVM before tuning. KNN performs exceptionally well with 100% accuracy, precision, recall, and F1 score, suggesting that the dataset has a well-structured nature with close proximity among data points of the same class.

Similarly, Random Forest also achieves 100% accuracy, precision, recall, and F1 score, indicating that the ensemble of decision trees is effective in capturing the complex relationships present in the data, leading to excellent classification performance.

Decision Tree, with almost 100% accuracy, precision, recall, and F1 score, is able to create simple and accurate classification rules based on the clear structure of the data.

On the other hand, SVM falls behind other algorithms in terms of performance before tuning, achieving 66% accuracy, 58% precision, 64% recall, and 53% F1 score. This suggests that SVM may not be well-suited for the given dataset with its current hyperparameters.

However, after tuning its hyperparameters, SVM's performance improves significantly, matching other algorithms with 100% accuracy, precision, recall, and F1 score, highlighting the importance of hyperparameter tuning in enhancing SVM's capabilities.

In summary, KNN, Random Forest and Decision Tree demonstrate superior performance both before and after tuning, while SVM's performance improves substantially after hyperparameter tuning, making it competitive with other algorithms.

V. Conclusion

In conclusion, from analysis on comparison among classification algorithms (KNN, Random Forest, Decision tree and Support Vector Machines) after tuning, it shows that KNN, Random Forest and Support Vector Machines are the most accurate and precise in terms of the accuracy and precision respectively. Also, they all gain highest recall and f1 scores. Interestingly, after tuning, SVM's scores sharply jump to 100%.

References

[1] Support Vector Machine (SVM) Algorithm - GeeksforGeeks
[2] Efficient Learning Machine – Mariette Awad and Rahul Khanna
[3] Decision Trees - scikit-learn
https://scikit-learn.org/stable/modules/tree.html
[4] Decision Tree Algorithm
https://www.analyticsvidhya.com/blog/2021/08/decision-tree-algorithm/
[5] Hyperparameter Tuning of Decision Tree Classifier Using GridSearchCV
https://plainenglish.io/blog/hyperparameter-tuning-of-decision-tree-classifier-using-gridsearchcv-2a6ebcaffeda
[6] KNN Basics
https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761
[7] Random Forest With Examples
https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/
[8] Mushroom species drawn from source book: Patrick ardin.Mushrooms Toadstools.Zondervan, 1999
[9] Y. Wang, J. Du, H. Zhang, and X. Yang, Mushroomxtoxicity recognition basedxon multigrained cascade forest", Scientific Programming, (Special Issue), 2020.
[10] 2022 11th MEDITERRANEAN CONFERENCEONEMBEDDED-COMPUTING (MECO),7-10JUNE2022,BUDVA,MONTENEGRO
[11] Ensemble Machine Learning: Methods and Applications (pp.157-176)