Project Report: Wild Mushroom Poisoning Classification Model

1. Introduction

1.1 Problem statement

For many people the COVID-19 pandemic was a catalyst to explore new hobbies, this was especially true for outdoor hobbies. This surge in outdoor recreation paired with an increased interest in survivalism led to a growth in popularity of mushroom foraging, and with this came a rise in mushroom related poisonings [1]. Wild mushrooms are often poisonous and present a serious risk to those who consume them without proper knowledge or identification of the species. While there are applications available where you photograph a mushroom or fungus and the application informs you of the genus/species, an app that is streamlined to let you know strictly of the safety of the mushroom is novel in the space.

1.2 Proposed Solution

The starting point for solving this issue was an intuitive one. This problem lends itself well to being solved with a classification model, specifically a binary classification mode, as we are separating the mushroom results between a poisonous class and an edible class. This is represented in the data through having a binary output as the target variable. While due diligence was taken in consideration of other approaches, in the end a classification model was the only option that made sense.

1.3 Project Deliverables

1.3.1 A functioning app that accepts easy to comprehend parameters as inputs and supplies an output informing the user whether this mushroom is edible. In addition to an online deployment, the app will be available to download for local usage.

1.3.2 There is a requirement of an input to modulate the level of risk the user deems acceptable calculated from false positives and false negatives of the test set.

2. Machine Learning

2.1 EDA

2.1.1 Initial Overview

On initial examination it was observed that this dataset has many categorical features, which presented an interesting challenge for the feature engineering of this project. Encoding these categorical variables into a state in which they would be useful for our classification model took careful consideration of several factors, including the number of categories, the relationship between categories, the potential for feature interactions, and the impact of the encoding method on the performance of the model. It required a thorough understanding of the domain and the underlying mechanisms that govern the relationships between the features and the target variable.

2.1.2 Balance

The first step toward building the model was to delve into the data to see if there were any obvious trends or issues that were immediately apparent. Since the planned model is based on binary classification, it seemed as though a logical place to start was by analyzing the balance in the target variable. Having an unbalanced target variable may have presented difficulties for the project, so it was a necessary piece of information for making decisions on how we would proceed with the rest of the project. Luckily it was found that the target variable was split 55/45 toward poisonous mushrooms. This is an ideal split as having a minority feature in our target variable could have led to issues in developing a consistent model.

After analysis of the target variable focus was shifted toward the rest of the dataset. To get a high-level idea of the distribution of data in this set the rest of the features were checked for minority features and whether they skewed heavily toward one class in the target, below is a representation of initial minority feature evaluation. Below are a few select examples of this exploration on the various features concerning the 'gill' of the mushroom.

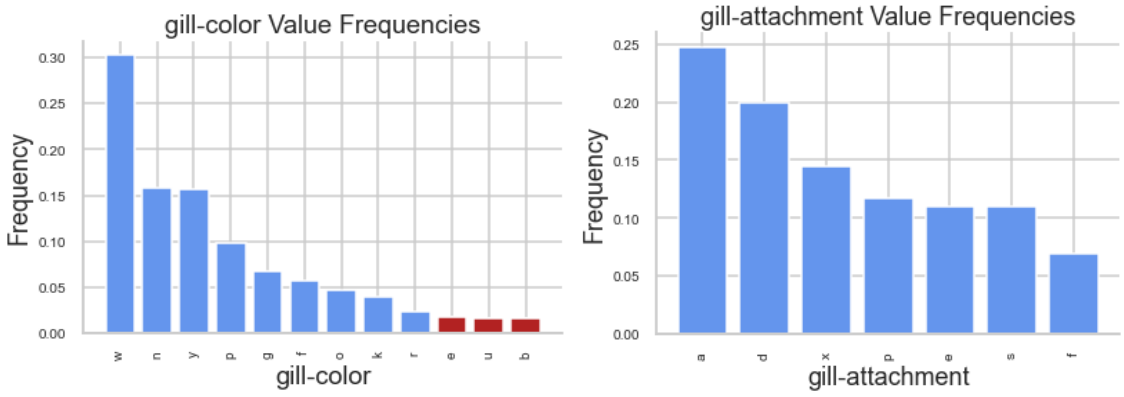

Figure 1: (left) frequency of each gill-color categorial value (dark red is feature that is less than 2%)
Figure 2: (right) frequency of each gill-attachment categorical value

2.1.3 Unique Values and Missingness

Following this examination, the data was then checked for missing values and unique observations. Below are figures depicting the percentage of NaNs present in the dataset as well as the number of unique observations for each feature.

class	2
cap-diameter	2571
cap-shape	7
cap-surface	11
cap-color	12
does-bruise-or-bleed	2
gill-attachment	7
gill-spacing	3
gill-color	12
stem-height	2226
stem-width	4630
stem-root	5
stem-surface	8
stem-color	13
veil-type	1
veil-color	6
has-ring	2
ring-type	8
spore-print-color	7
habitat	8
season	4

Figure 3: Unique Values of Each Feature

class	0.000000
cap-diameter	0.000000
cap-shape	0.000000
cap-surface	0.231214
cap-color	0.000000
does-bruise-or-bleed	0.000000
gill-attachment	0.161850
gill-spacing	0.410405
gill-color	0.000000
stem-height	0.000000
stem-width	0.000000
stem-root	0.843931
stem-surface	0.624277
stem-color	0.000000
veil-type	0.947977
veil-color	0.878613
has-ring	0.000000
ring-type	0.040462
spore-print-color	0.895954
habitat	0.000000
season	0.000000

Figure 4: Percentage of NaNs by Feature

Features with > 60% missing values were immediately dropped. This included the following features: spore-print-color, veil-type, stem-root, veil-color, and stem-surface. Features with lower percentages of missing values needed to be subjected to increased scrutiny prior to rendering a decision of whether to omit these features for the final model. The main concern present with this missing data is how the NaNs from each feature are distributed across the target variable. The first step taken was to ensure that the NaNs from each feature with missing values are roughly evenly distributed between the two classes of our target variable. This was done by plotting the normalized distribution on a bar plot. Below is an example of one of the features examined in this manner. The below plot is representative of how the rest of the features were distributed, each of the features with missing values had NaNs that were fairly evenly distributed across the classes of the target variable.

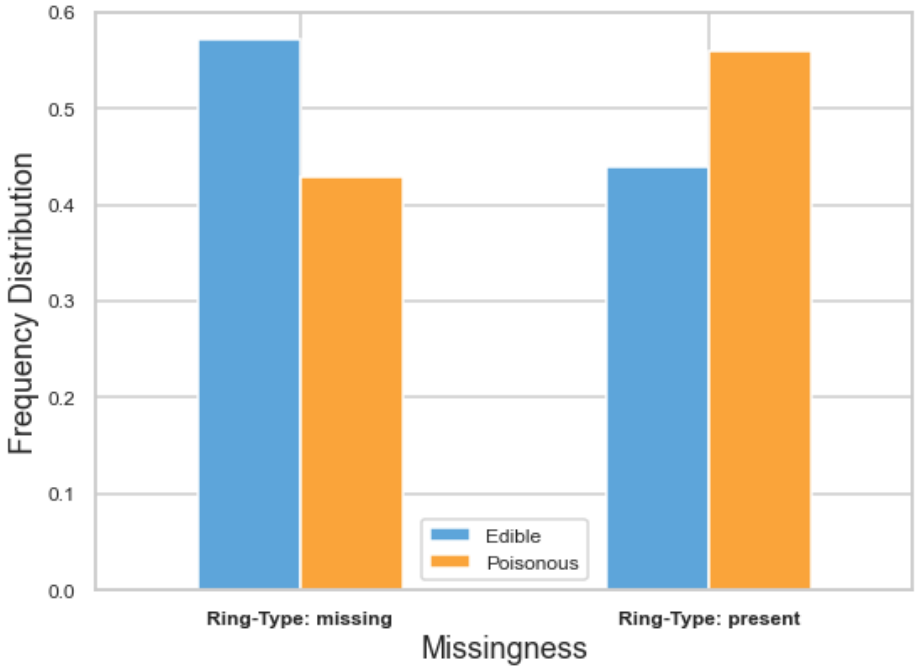

Figure 5: The above figure plots the distribution of the two target classes for the observations with missing in 'Ring-type' with missing data against those with a value.

After concluding that none of the features with missing observations had any significant skew in respect to the target class, each feature with missing data, other than ring-type, was dropped from the dataset.

2.2 Feature Engineering

2.2.1 Initial encoding

Initial encoding was very intuitive for some variables. In the initial dataset there were three variables that were already binary, these were encoded with a simple transformation. For each of these '1' was used for the affirmative response and '0' used for the negative response. Each categorical variable was encoded using one-hot-encoding for initial correlations and base model building, with plans to return to feature engineering to alter the type of encoding to enhance the finished model. There were 69 total features, including the target feature, at the end of this initial encoding. Since our dataset has close to 70,000 observations, 69 features was an appropriate number with which to move forward in

model comparison. Now that the features were in a format that lends itself to viewing correlations between each other, a plot of the correlation between each feature and the target was generated.

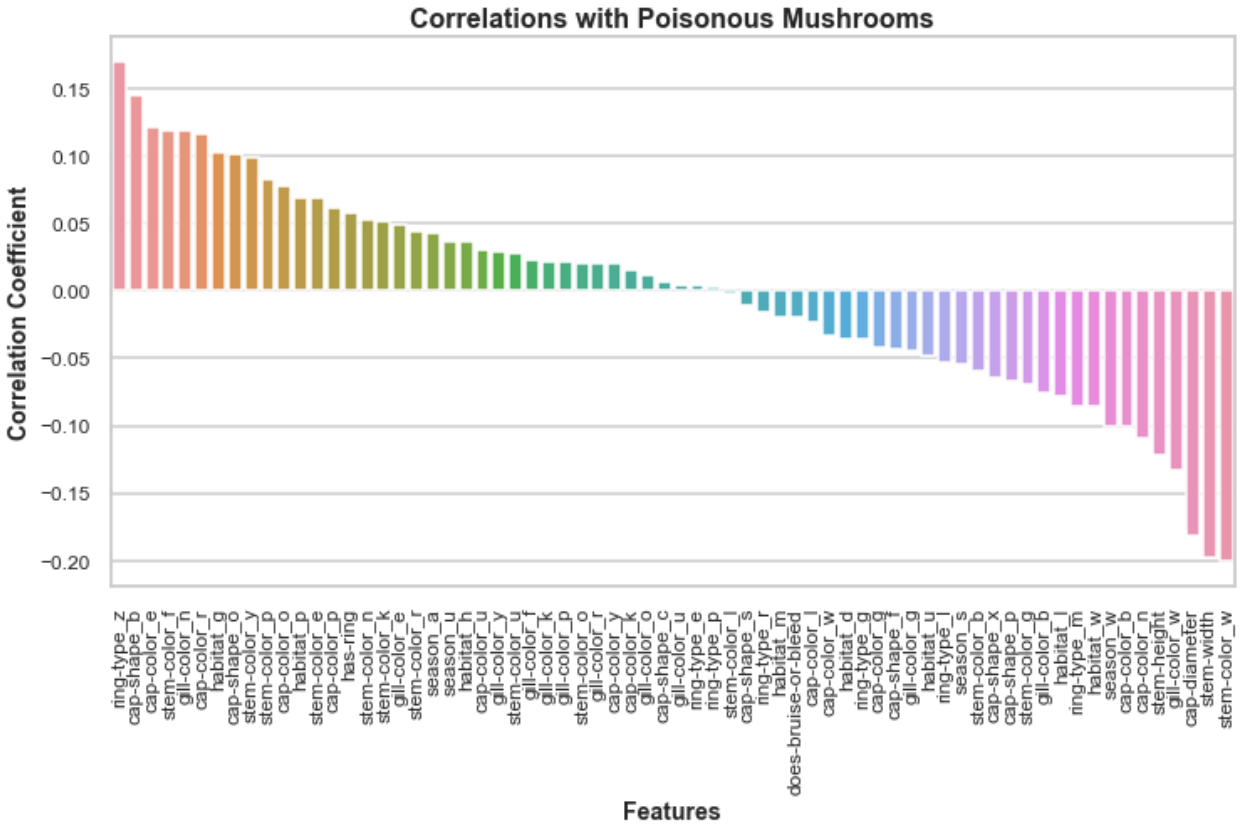

Figure 6: Correlation plot between each feature and the target variable (specifically the class 'Poisonous' in the target variable).

2.2.2 Multicollinearity

While the above plot is illustrative of the correlation each input feature has with the target variable, it is important to examine the correlation each feature shares with one another, having a large degree of multicollinearity between two features that possess a high corollary magnitude with our target feature could skew our end model by causing an invisible increase in the weight the model assigns to the colinear features.

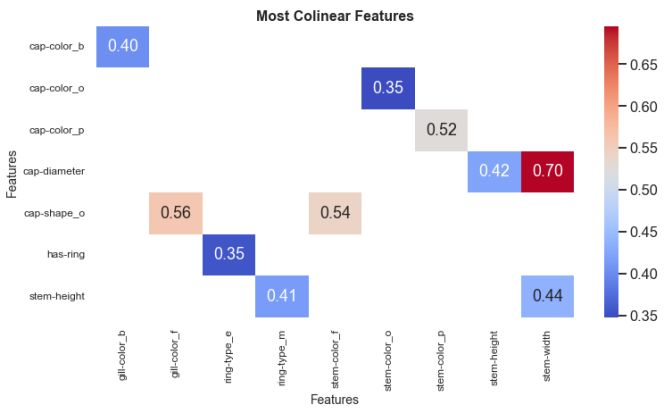

Figure 7: A plot of the features with the highest correlations

There are two pairs of variables in Figure 7 that cause concern for the overall model. The first pair is 'stem-width' and 'cap-diameter', with the second being 'stem-color_f' and 'cap-shape_o'. These draw concern due to the high corollary relationship they have with one another while also individually having a high corollary magnitude with our target feature. If nothing is done with these features, this could put the overall model at risk of overweighting these features. Due to the nature of each of these features, there will have to be different solutions implemented for each of these pairs. For the pair of 'stem-width' and 'cap-diameter' the initial solution to the multicollinearity with be deletion of one of the features. The 'stem-width' feature is a good candidate for deletion before moving onto base model building. While there are other more effective ways of dealing with collinearity in classification models, deletion of this variable makes sense in this situation because 'stem-width' is a difficult feature to measure in the field (where this app will presumably be deployed), this adds even further potential error to our model. While this is a good solution for the first pair, it will not be a potential solution to the second pair, 'stem-color_f' and 'cap-shape_o', because these variables will not be their own input. These are one-hot-encoded variables that will share an input with each other feature that was encoded from the original features. Deletion of one of a categorical feature of this type means that either the entire input of 'stem-color' or 'cap-shape' would have to be deleted as well, or that this input option would be replaced with a NaN. Because neither of these are ideal options, neither will be implemented for base modeling, rather they can be an option worth exploring, if necessary, during hyperparameter tuning.

2.3 Base Models Comparison

Prior to the base model selection, the target feature was isolated as a separate variable then data was divided into a train and test set at an 80/20 split. For base model selection a pipeline was constructed consisting of three initial models to explore: an SVC model, a logistic regression model, and a KNN model. Each of these models was initially implemented with the baseline settings from scikit-learn. The results of each model evaluated on the test set are in the table below. It is important to note that all models were tested with 5-fold cross-validation.

Model Performance

	Accuracy	Execution Time
KNN	98.7%	42.2s
Logistic Regression	74.8%	2.9s
SVC	74.0%	7m 36s

Figure 8: Table of baseline model accuracy

The baseline KNN model vastly outperformed the other models and is the model that will be moved onto hyperparameter tuning.

2.4 Return to Feature Engineering

2.4.1 Encoding continuous variables

The success of the baseline KNN model offered an opportunity to return to feature engineering to do something that may seem counter intuitive. In the baseline model we have 2 features that are continuous. These are cap-diameter and stem-height. As was briefly touched on during the initial round of feature engineering with the deletion of 'stem-width', sometimes continuous features like these can be difficult to estimate when in the field. They require some sort of measurement instrument which may not be available. Instead of dropping these features in a similar fashion to 'stem-width', instead these features will be encoded into categorical features, these categorical features will then be further encoded with one-hot-encoding. While changing these continuous features to categorical may theoretically decrease the accuracy of the end model, in practice they are more likely to increase the overall accuracy and user experience of the product. There are a few reasons for this, it will completely detether the application from the need of additional measurement tools, which will make the user experience more streamlined, this will also remove the aspect of users attempting to guess input features which could harm the model.

The choice that was then presented was: 'how to define the categories?' Below is a distribution plot of 'stem-heigh' and 'cap-diameter' with a vertical line at the 33rd and 6^{6th} percentile (the data broken into 3 equal sizes)

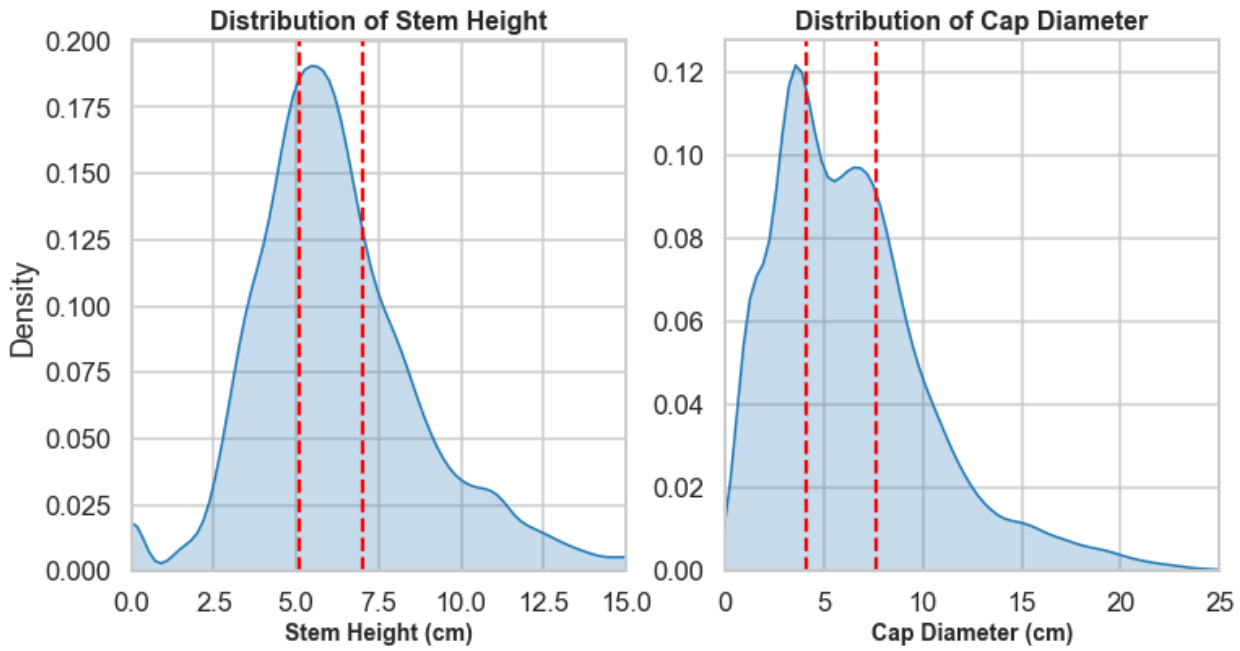

Figure 9: KDE plots of 'stem-height' (left) and 'cap-diameter'(right)

As shown in Figure 7 above it appears as though the 33rd percentile line is near 5 centimeters for each plot and the 66th percentile is near 7.5 centimeters for each plot. Though these don't exactly divide the data into thirds, for consistency 5 centimeters and 7.5 centimeters will be the dividing points for our categorical variable generation. Since one of the main reasons to change these features from continuous to categorical was the removal of the need for measurement, on the application the final input will not be in centimeters. Instead, the input will be asked in the form of everyday objects. Rather than asking the user if the feature is 'greater than 5cm' the user will be asked if the feature is 'larger than the diameter of a soda can'.

With the simple change of encoding the continuous variables to categorical the end user will no longer need to measure the exact size of the feature and will instead be asked to relate to the feature in terms of two everyday objects. (The diameter of a soda can for 5cm and the size of a baseball for 7.5cm). These terms should increase the quantity and quality of the responses for the former continuous features, as well as make the app more user friendly.

This new dataset was then run on a baseline KNN to compare to the results from the initial baseline. As expected, there was a slight drop in accuracy with the new dataset. The figures below show the results of this.

Model Performance

	Accuracy	Execution Time
KNN baseline	98.7%	42.2s
new KNN	97.9%	41.1s

Figure 10: Comparison of KNN models after encoding continuous variables.

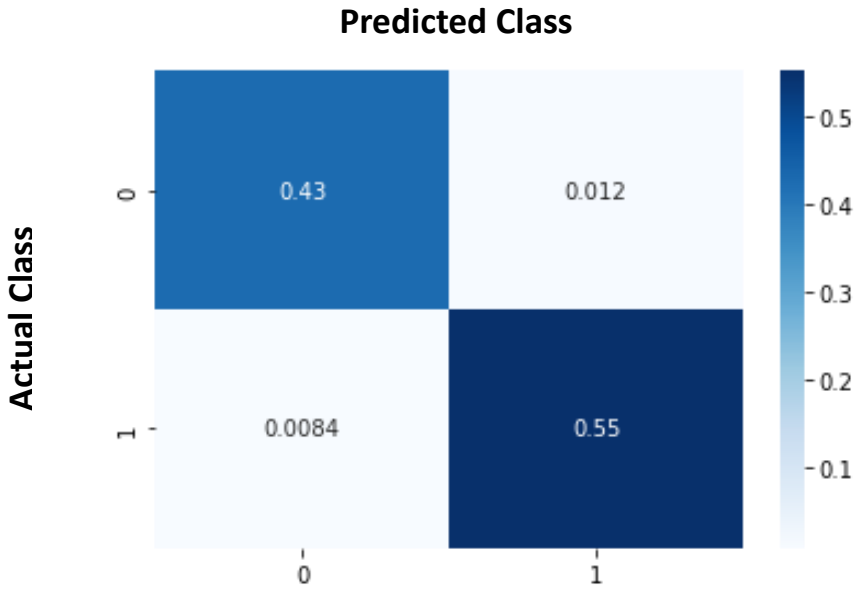

Figure 11: Confusion matrix of the KNN after encoding.

2.5 Hyperparameter Tuning

Hyperparameter tuning for this model was completed using GridSearchCV from the scikit-learn library. The parameters that were optimized with the grid search were 'n_neighbors' and 'weights' for the KNN model. The parameters used in this grid search were the following: 'n_neighbors': 1-30, 'weights': 'uniform', 'distance'. This grid search was then fit to a KNN model with a 5-fold cross-validation. While the grid search with the desired parameter grid took several hours to run, this is a relatively short amount of time for thorough grid search. This is likely a function of KNN models being faster to tune than other models. The relatively small number of parameters to iterate through is likely the only reason that outside resources, such as ICER, were not necessary.

The best parameters from the grid search, in this case 'n_neighbors' = 11, and 'weights' = 'distance', were applied to our KNN model rather than the baseline values. Below are the results presented in a table and confusion matrix of the tuned model.

Model Performance

	Accuracy	Execution Time
KNN	97.9%	41.1s
tuned KNN	98.5%	39.9s

Figure 12: Baseline KNN score vs tuned KNN score.

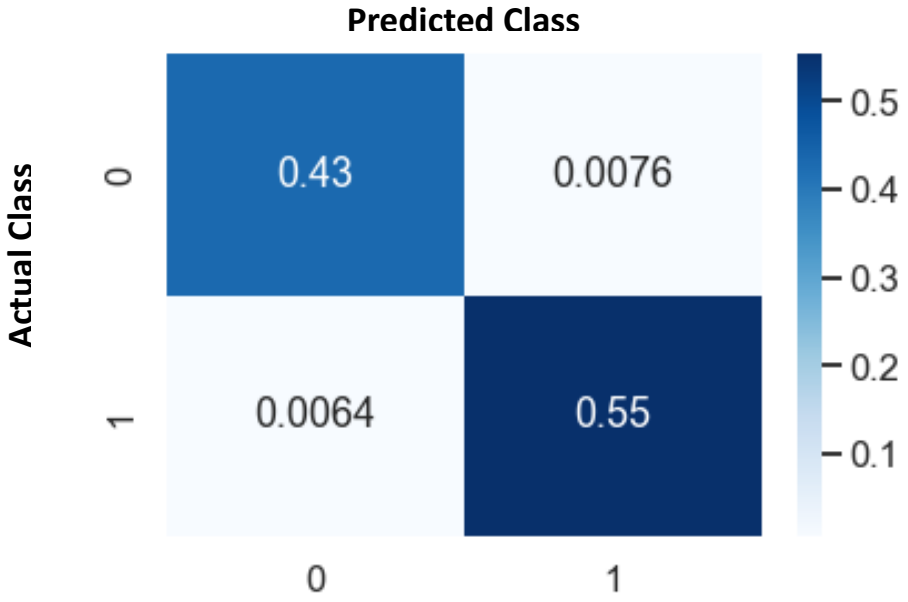

Figure 13: Confusion Matrix of the tuned KNN model.

Hyperparameter tuning improved the KNN model by a little more than half a percentage point. Upon comparison of the confusion matrix of the tuned and baseline models it appears much of the difference came from fewer instances of false positives in the tuned model. While this is a welcome change, it is much more important for our final model to have a low rate of false negatives, as false negatives have a higher cost in the form of the accidental poisonings that this project was initiated to stop.

2.6 Ensembling

To further improve the model several ensembling models were considered. The two ensembling methods that seemed to be the best fit for a KNN classification model were: a bagged classifier using the BaggingClassifer function from scikit-learn, and a stacked classifier using the scikit-learn function StackingClassifier. Between the two, the stacked classifier stood out as the more likely to succeed. This is mainly due to how well the hyperparameters obtained from GridSearchCV slot into the stacked classifier but were difficult to use with the bagged classifier. The bagged classifier was run with the base_estimator: 'KNN' and n_estimators: '10'. Unfortunately, the bagged classifier did not perform as well as the tuned KNN model. The stacked classifier was run with the top three parameter combinations

from GridSearchCV performed earlier, with a logistic regression utilized as the meta model. In this configuration the stacked classifier slightly outperformed the tuned KNN model. Below are the results presented in a table and confusion matrix of the stacked classifier ensembled model.

Model Performance

	Accuracy	Execution Time
stacked KNN	98.7%	33.3s
tuned KNN	98.5%	39.9s

Figure 14: Stacked KNN score vs tuned KNN score.

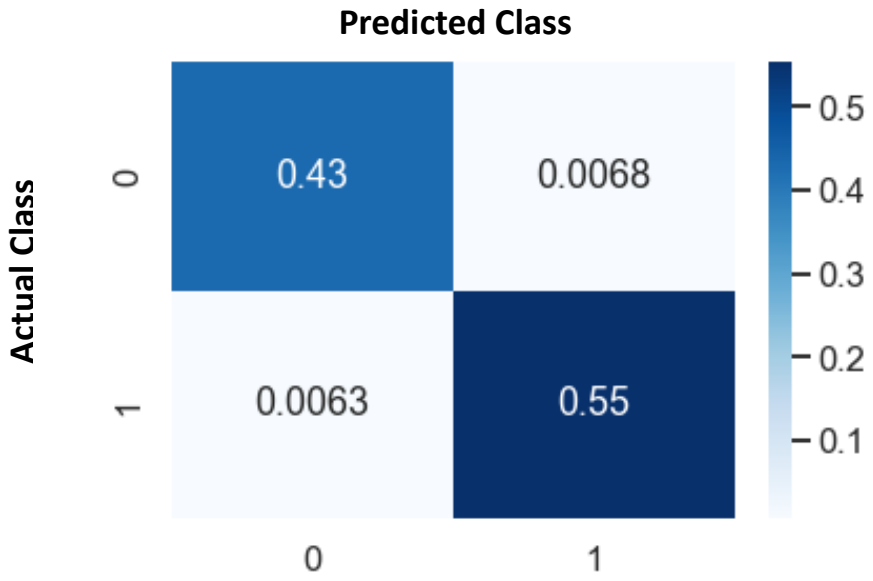

Figure 15: Confusion Matrix of the stacked KNN model.

2.7 Results

The stacked classification ensembled KNN model that utilized the top three parameter combinations from the GridSearchCV execution performed at the same level as the initial baseline KNN model. There are two main benefits to the stacked classification model versus the initial baseline model, the lack of continuous variables as input and overall model speed, the stacked classification model boasted a ~10 second faster training time which reduces the startup time for the app. These two factors make the stacked classification model a clear choice over using the baseline KNN model for development of the app.

An additional feature that has been added to the app is a 'safe mode'. The app will automatically be in 'safe mode' when booted up, the safe mode on this app changes the probability threshold for which a mushroom is classified as safe from .5 to .005. This is the lowest the threshold can go and still predicts any mushroom as safe to eat. Below is the confusion matrix of the final model in 'safe mode.'

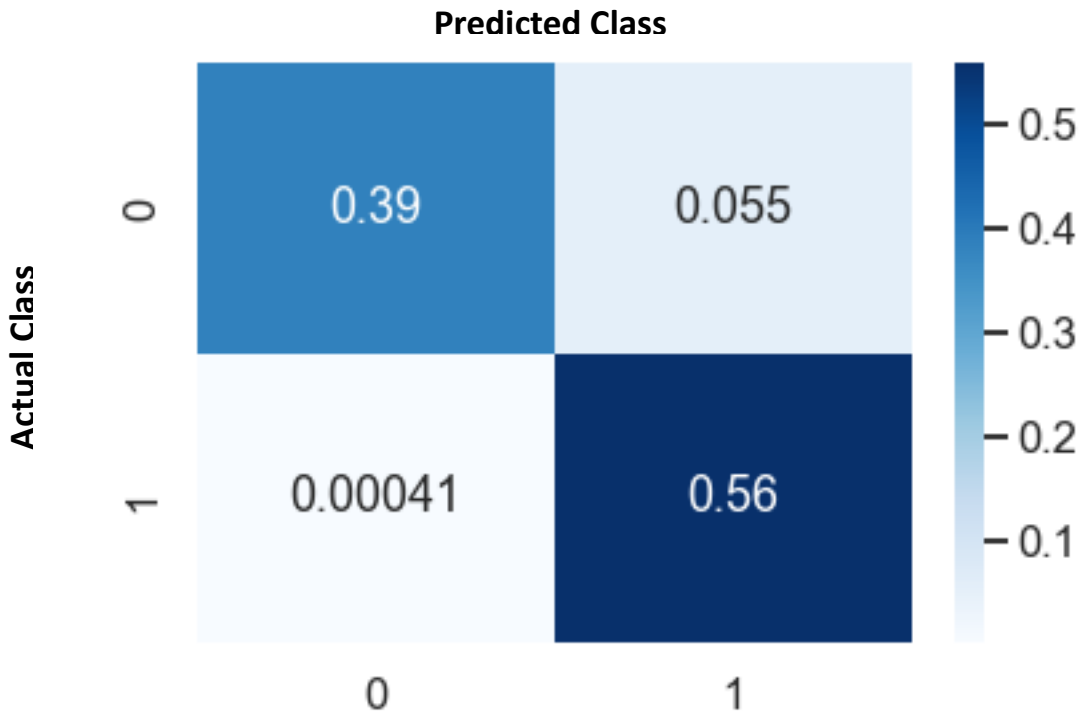

Figure 16: Confusion matrix of the 'safe mode' of the final model.

In 'safe mode' there is a 5.5% chance that a mushroom is predicted to be poisonous but is, in fact, safe to eat. Out of more than 12,000 observations in the test set there were only 5 instances of false negatives.

3. Project Summary and Goal Assessment

3.1 Summary

The final product of this project is an application that can be used both online and locally to assist in safe mushroom foraging. The final model is a stacked ensembled KNN classification model with all features encoded into categorical variables. Hyperparameter tuning was performed with the GridSearchCV function with a 5-fold cross-validation. Feature engineering mainly focused on encoding features to be usable in the final model, except for encoding continuous variables for user convenience. The final model is 98.7% accurate on the test set, with the option to almost eliminate the chance of a false negative completely while only reducing the overall accuracy of the model to 94.4%

3.2 Goal Assessment

The stated goal of this project in the approved proposal was to create an app that could predict whether a mushroom was poisonous with a greater than 95% rate and a greater than 80% rate with less than 5% false negative rate when set to 'safe mode'. Furthermore, the goal was to deploy this application in a way that would be easy to use both online and run locally. These objectives were met. The model was successful, and deployment of the described app was also successful.

3.3 Reflections and Future work

This project exhibited a reality that is important to keep in mind during all machine learning projects. Sometimes the simplest model is the best model for the situation. A KNN classifier is a relatively simple model, while it would have been a rewarding experience to develop a complex neural network that was many layers deep, it just did not make sense to pursue anything more complex when a baseline KNN is nearly as effective as possible. It made much more sense to focus on other factors, such as feature engineering and user convenience. Another benefit of starting with a successful baseline model is that it can afford opportunities to add to the model in ways that may be traditionally seen as making the model 'worse'. Some examples of this in this project are adding safety features and encoding continuous features to categorical for input simplicity. Starting near 99% with a baseline model does not give very much room for hyperparameter tuning or ensembling, but when other features are added to improve the overall model while lower accuracy slightly, you'll likely be able to recoup those gains with tuning and ensembling, thus in the end building a more robust model.

In the future I would like to improve this application by adding information from both the meta data of this study as well as other fungus studies to attempt to assign a genus and species name to each of these observations. A feature I would like to add with this is to have hyperlinks in the application to websites where mushroom selling and trading are popular. This could provide real time information about the value of edible mushrooms. While linking information, such as poison control, which could provide information about how to treat various mushroom poisonings.

4. References

Authors of the data

Patrick Hardin.Mushrooms & Toadstools. Zondervan, 1999

Author: Dennis Wagner Date: 05 September 2020

1. Nark, Jason. "CHOP Poison control center warns of spike in severe mushroom poisonings", Philadelphia inquirer, Oct. 18, 2022

5. Appendix

Github link: https://github.com/chgra1/CMSE_890

Streamlit.io link: https://chgra1-cmse-890-app-xpn3wp.streamlit.app/