Project Report

Introduction

The application of supervised machine learning in classification is incredibly versatile. When trained properly, supervised machine learning techniques can produce models, which can classify data with high accuracy and precision. We will be looking at classification-based supervised machine learning models within the domain of mycology.

Mushrooms are diverse, plentiful, and considered a delicacy in many parts of the world. But a large percentage of them are poisonous, with reactions ranging from sickness to paralysis or even death. Considering the potential consequences of picking the wrong mushroom, it is important to be able to quickly and accurately identify whether a mushroom is toxic.

For this analysis, using a dataset of over 60000 mushrooms with 20 features, we predict whether a mushroom is poisonous or edible. We also examine whether certain characteristics of mushrooms are particularly useful in classifying the observations. There are two goals of this analysis. First, we want to use the features of this dataset to predict whether a mushroom is toxic. Next, we want to identify the features most strongly associated with toxicity.

Given that machine learning models have been demonstrated to accurately classify elements, we predict that the machine learning models we will be using for this analysis will be able to accurately classify mushrooms as edible or toxic. This tool will be useful for mushroom foragers looking for a reliable method to quickly assess their pickings.

Methods

The features cover a wide range of physical characteristics, including measurements of the cap, stem, gills, veils, and rings, and we are confident that some combination of these characteristics will be highly correlated with toxicity. Since the goal here is prediction, we judge the success of the models without regard for their explanatory power.

All data processing, analyses , and visualizations were completed in Python v3.11.7 using the *pandas, matplotlib,* and *sci-kit learn* packages. Prior to fitting our models, data preprocessing was conducted. First, we converted the levels of the class response variable to integer values, with "edible" mapped to 0 and "toxic" to 1 thereby ensuring compatibility with supervised machine learning algorithms. Following this, for similar compatibility reasons, we mapped each of the categorical predictor levels to integer values. Next, we dropped features with more than ⅓ of values being null, bringing the number of features down from 20 to 14. Null values from ‘cap-surface’, ‘gill-attachment’, and ‘ring-type’ were filtered out, bringing the total number of observations to 37065. Since we do not consider ourselves mushroom experts, we did not impute any null values.

The three numeric features were square root transformed to normalise and z-transformed to standardize the data. After transformations, outliers were examined to determine their potential impact on the models. For the numeric features ‘cap-diameter’ and ‘stem-width’, outliers were not considered problematic because they only represented 2% of the observations. For the numeric feature ‘stem-height’, outliers corresponded to ~7% of the data. These observations were kept because mushrooms with particularly long stems represent natural variation in the population and are a fair measurement of mushroom variation. Getting rid of these observations would mean removing valuable information

from the dataset. Finally, the dataset was split into separate training and testing datasets, with a 80/20% split. This resulted in 29562 training observations and 7413 testing observations.

To assess whether mushrooms can be accurately classified into toxic and edible categories, we ran 5 supervised machine learning models, each of which with 5-fold cross validation: Logistic Regression, Linear Discriminant Analysis (LDA), k-nearest neighbours (kNN), Random Forest, and Boosting. Each model was run using grid search to find the optimal hyperparameters that provide each model the highest predictive accuracy. We assessed the performance of each model using (1) performance metrics - accuracy, precision, F1, and log loss performance metrics, (2) confusion matrices, and (3) ROC curves.

Results

Logistic Regression

The logistic regression model took a total of 44 seconds to run (Table 1). Grid search was used to find the best combination of hyperparameters, obtaining a C value of 10, 300 max iterations, and the saga solver. Accuracy, precision, and the F1 score were lower than those obtained with the more complex random forests and gradient boosting classifier models (see Table 1 for metrics) and log loss was the highest of the models run. Misclassifications were an issue, with 39% of test observations being incorrectly classified (see Figure 1). Additionally, the AUC was 0.65 (see Figure 2). Overall, the predictive performance of the Logistic Regression model was well below other models we ran.

Linear Discriminant Analysis

At 38 seconds, the LDA model took the least time to run (Table 1). A grid search for linear discriminant analysis (LDA) determined the optimal hyperparameters were the Least Squares solver, auto shrinkage, and no dimensionality reduction. Scores on accuracy, precision, and F1 were similarly low to the logistic regression model and log loss was only slightly better (13.41). 37% of test observations were misclassified (Figure 1) and the AUC score was 0.62 (Figure 2). In summary, the LDA performed marginally better than the Logistic Regression model, but well below other models we ran.

k-Nearest Neighbours

The k-nearest neighbours model took 1.26 minutes to run (Table 1). Grid search returned optimal hyperparameters indicating three neighbours, uniform weights, the auto algorithm, and Manhattan distance for the power parameter. Performance metrics on the kNN model were excellent, obtaining perfect classification scores on precision, accuracy, and F1 (Table 1). Log loss was incredibly low (0.005), the AUC score was perfect (Figure 2), and only a single test observation was misclassified (Figure 1). Overall, the kNN model performed nearly perfectly and can reliably predict whether a mushroom is toxic based on the features in our dataset.

Random Forest

This technique took a total of 3.30 minutes to run (Table 1). The optimized grid search revealed the optimal parameters were: a maximum tree depth of 10, a minimum of 2 samples required for node splitting, and an ensemble of 300 trees. This fine-tuning resulted in robust performance metrics, boasting high accuracy, precision, and F1 scores exceeding 99.5% (Table 1). Additionally, the model demonstrated minimal log loss (0.131; Table 1) and achieved a significant AUC of 1 (Figure 2).

Misclassifications were minimal, with only 0.3% of test observations being incorrectly classified (Figure 1). Finally, examining feature importance through Figure 3 underscores the critical significance of attributes like stem width, gill attachment, and cap surface, which emerge as the top three predictors within the Random Forest model.

Boosting

At 3.32 minutes, it took the most time to run the boosting model (Table 1). Grid search optimization yielded the following optimal parameters: a learning rate of 1, utilization of the log loss optimization function, and an imposed maximum individual tree depth of 3. Demonstrating superior performance alongside the k nearest neighbours model, the Boosting model showcased remarkable accuracy, precision, and F1 scores, nearing 100% (Table 1). Furthermore, the model achieved a log loss close to zero (0.005; Table 1) and a perfect AUC score of 1 (Figure 2). Remarkably, misclassification was minimal, with only 1 toxic mushroom incorrectly classified as edible. Visualizing feature importance (Figure 3) further highlights the pivotal role of attributes such as stem width, gill attachment, and stem colour, emerging as the top three predictors within the Boosting model.

	Accuracy	**Precision**	**F1 Score**	**Log Loss**	**Time**
Logistic Regression	0.61	0.635	0.662	14.057	44 seconds
LDA	0.628	0.662	0.665	13.41	38 seconds
K-Nearest Neighbours	1	1	1	0.005	1.26 minutes
Random Forests	0.996	0.997	0.997	0.131	3.30 minutes
Boosting	1	1	1	0.005	3.32 minutes

Table 1: Performance metrics and the time it takes to run each Supervised Machine learning model.

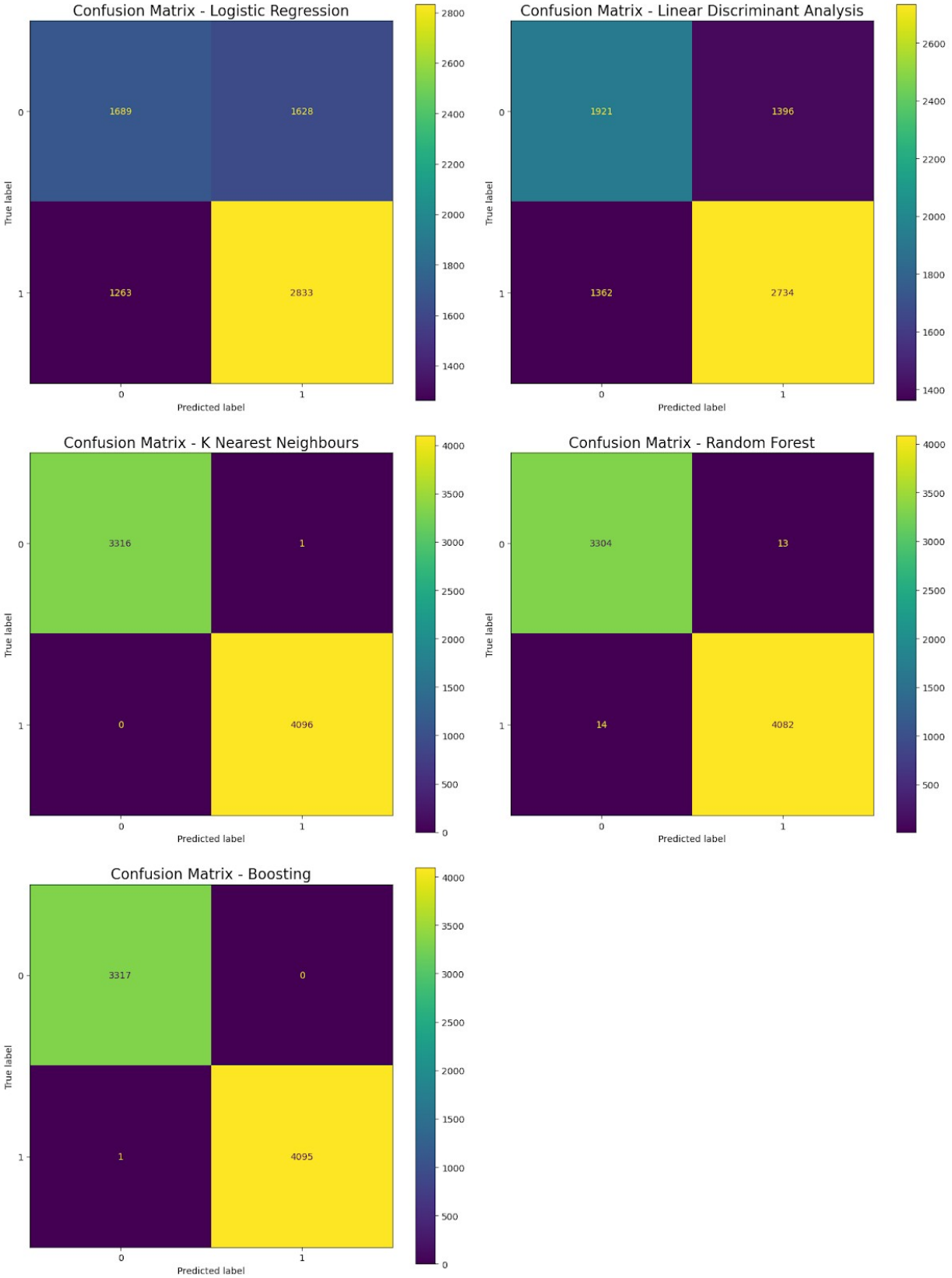

Figure 1: Confusion matrices for each Supervised Machine learning algorithm. Label 0 corresponds to "edible" and label 1 corresponds to "toxic"

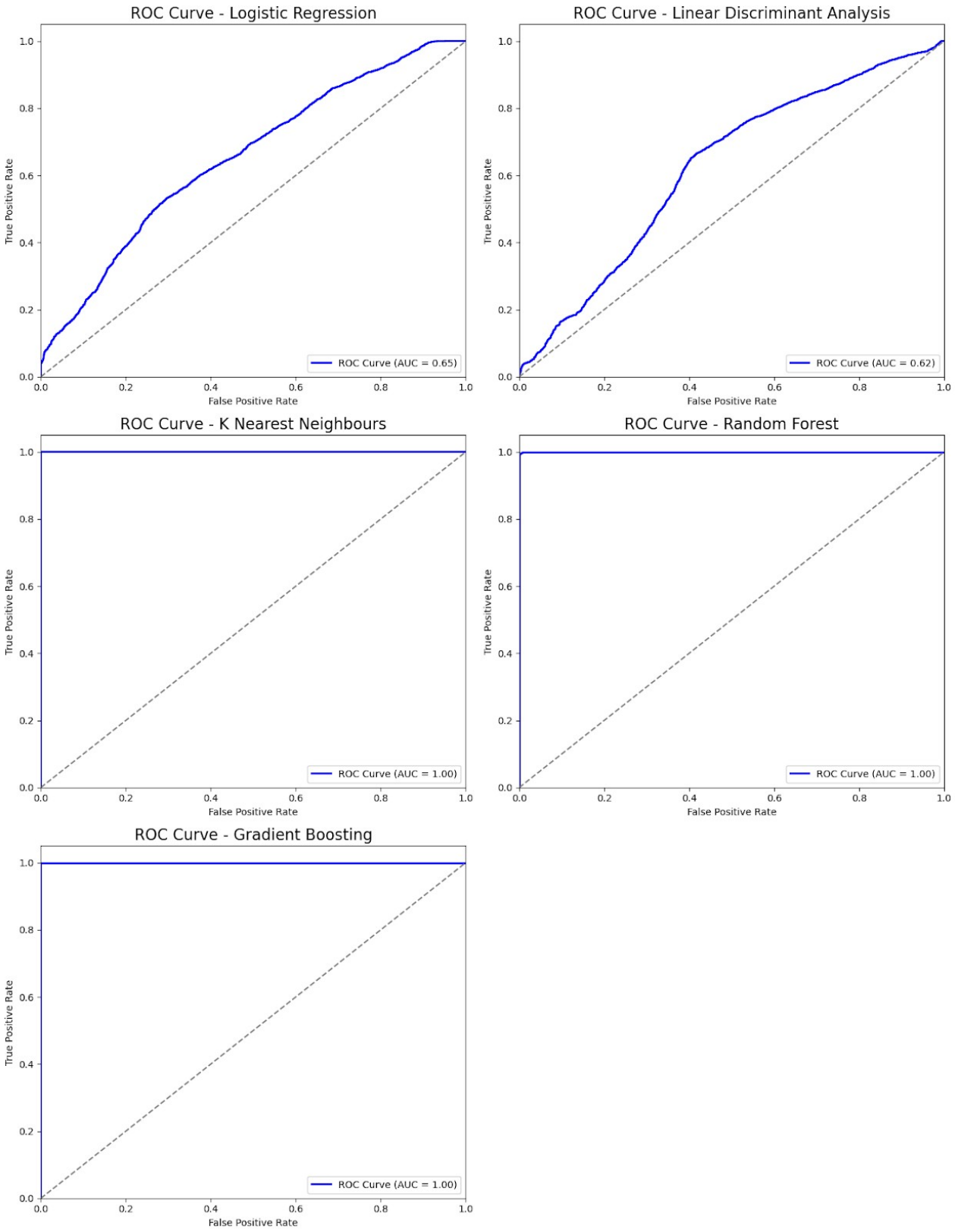

Figure 2: ROC curves for each Supervised Machine learning algorithm

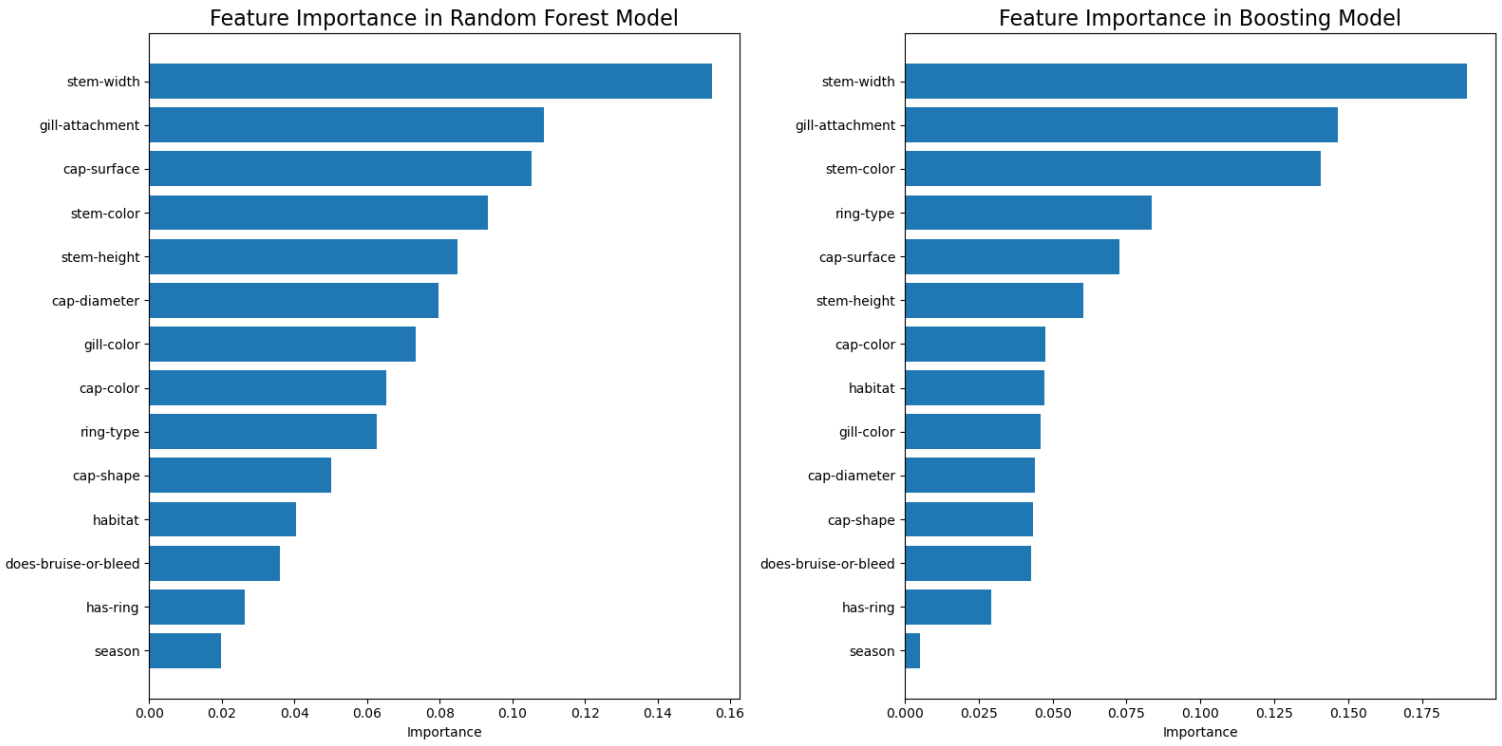

Figure 3: Features of highest importance, ranked from highest to smallest for the Random Forest and Boosting models.

Discussion and Conclusion

Best Model

The Logistic Regression and LDA models performed the worst, indicating they could not reliably predict whether a mushroom was toxic or not. Accuracy, precision, and F1 scores were all below 0.67 for both models (see Table 1), which are not acceptable classification rates when predicting something as potentially severe as mushroom poisoning.

Considering the model performance metrics presented in Table 1 and the insights gleaned from the confusion matrices (Figure 2) and ROC curves (Figure 3), it becomes evident that kNN, Boosting, and Random Forest models all exhibited strong performance across various metrics. Notably, both kNN

and Boosting models excelled, each making only one incorrect classification. However, in the context of the severe consequences associated with misclassifying a poisonous mushroom as edible, kNN emerges as the preferred model. This preference stems from kNN's ability to classify just one edible mushroom as poisonous, without making the reverse error. In contrast, Boosting misclassified a poisonous mushroom as edible, highlighting the importance of prioritizing precision in such critical classification tasks.

Feature Importance

In the discussion of feature importance, it's important to contextualize the methodology behind the analysis. Feature importance was conducted specifically for the random forest and boosting models due to the availability of this feature in Python. kNN and LDA did not provide the option for feature importance calculation. Furthermore, while it is possible to assess feature importance in logistic regression by viewing the model coefficients, we opted not to because of the poor predictive accuracy of that model.

Focusing on the results obtained from the random forest and boosting models, the consistency in the top two features of high importance—stem width and gill attachment—is particularly noteworthy. This consistent ranking across both models suggests the critical role these features play in predicting the toxicity of mushrooms.

Additionally, while stem colour emerges as the third highest priority feature in boosting, it occupies the fourth position in random forest. Despite this slight variation in ranking between the two models, the persistence of stem colour among the top features underscores its significance in discerning

toxic mushrooms. This consistent presence across models further solidifies its importance in the predictive process.

Overall, these findings emphasize the importance of stem width, gill attachment, and stem colour in determining mushroom toxicity. These insights provide valuable guidance for further research and underscore the reliability of these features in distinguishing toxic from non-toxic mushrooms.

Real-World Applicability of Mushroom Toxicity Models

The results of the analysis provide promising insights into the predictive capabilities of the models based on physical and environmental characteristics for distinguishing between edible and toxic mushrooms. However, it's crucial to recognize several limitations that may affect the generalizability of these findings to the broader mushroom population.

First, despite the high classification accuracy achieved by the models, there remains an inherent risk associated with relying solely on predictive models for mushroom consumption decisions. Even a model with perfect classification rates may not entirely eliminate the potential danger of misidentifying toxic mushrooms as edible. The consequences of consuming a toxic mushroom can be severe, underscoring the importance of caution and additional verification methods beyond predictive modelling.

Second, it's important to consider the limitations of the dataset used for the analysis. The observations in the dataset were simulated based on a smaller dataset of mushrooms, which may not fully represent the diversity of real-world mushrooms. There could be variations in physical attributes, environmental factors, and toxicity levels across different species of mushrooms that are not adequately

captured in the dataset. As a result, while the highly accurate models developed on this dataset may seem promising, caution should be exercised in generalizing these results to real-world scenarios.

In summary, while the analysis provides valuable insights and potential predictive capabilities for distinguishing between edible and toxic mushrooms, it's essential to approach the results with caution. Additional validation and verification methods, as well as consideration of the inherent risks and limitations, are necessary before applying these models to real-world mushroom identification and consumption decisions.

appendix_code

March 24, 2024

[ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score,␣
 ↪f1_score, log_loss, roc_curve, roc_auc_score, ConfusionMatrixDisplay
```

[ ]:
```
# read in mushrooms dataset
mushrooms = pd.read_csv("secondary_data.csv", sep = ";")
```

[ ]:
```
# Label encoding the response variable and categorical features
le = LabelEncoder()
mask = mushrooms.isna()
cols_to_encode = mushrooms.columns.drop(["cap-diameter", "stem-width",␣
 ↪"stem-height"])

for col in cols_to_encode:
    mushrooms[col] = le.fit_transform(mushrooms[col])

mushrooms = mushrooms.where(~ mask, np.nan)
```

[ ]:
```
# remove NaN values from dataset
mushrooms = mushrooms.drop(columns = ["veil-type", "veil-color",␣
 ↪"spore-print-color", "stem-root", "stem-surface", "gill-spacing"])
mushrooms = mushrooms[mushrooms["cap-surface"].notnull() &␣
 ↪mushrooms["gill-attachment"].notnull() & mushrooms["ring-type"].notnull()]
mushrooms.reset_index(drop = True, inplace = True)
```

[ ]:
```
# Split the dataset into X (predictors) and y (response)
X = mushrooms.drop('class', axis=1)
y = mushrooms['class']
```

[ ]:
```
# Square root transform numeric features and drop "veil-type" feature
X["stem-height"] = np.sqrt(X["stem-height"])
X["stem-width"] = np.sqrt(X["stem-width"])
X["cap-diameter"] = np.sqrt(X["cap-diameter"])
```

[ ]:
```
# Z-transform the numeric features
quantitative_vars = X[["cap-diameter", "stem-width", "stem-height"]]
X.drop(["cap-diameter", "stem-width", "stem-height"], axis=1, inplace=True)
sc = StandardScaler()
sc.fit(quantitative_vars)
x_scaled=sc.transform(quantitative_vars)

quant_scaled=pd.DataFrame(data=x_scaled,columns=["cap-diameter", "stem-width",␣
 ↪"stem-height"])
X_scaled = pd.concat([quant_scaled, X], axis = 1)
X_scaled[["cap-surface", "gill-attachment", "ring-type"]] =␣
 ↪X_scaled[["cap-surface", "gill-attachment", "ring-type"]].astype(int)
```

[ ]:
```
# Train-test split dataset
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,␣
 ↪random_state=42)
```

[ ]:
```
# Running Logistic Regression model

# Initialize logistic regression model
model_lr = LogisticRegression()

# param grid for Logistic Regression
param_grid = {
    'penalty': ['l1', 'l2'],
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'solver': ['liblinear', 'saga'],
    'max_iter': [100, 200, 300, 500]
}
lr_grid = GridSearchCV(model_lr, param_grid, cv=5)
lr_grid.fit(X_train, y_train)
print(lr_grid.best_params_)

# Predict on the test data using the best model
lr_grid_predictions = lr_grid.predict(X_test)

# Calculate model performance metrics
lr_acc = accuracy_score(y_test, lr_grid_predictions)
lr_prec = precision_score(y_test, lr_grid_predictions)
lr_f1 = f1_score(y_test, lr_grid_predictions)
lr_logloss = log_loss(y_test, lr_grid_predictions)
```

```
# Print the performance metrics
print("Accuracy:", round(lr_acc, ndigits = 3))
print("Precision:", round(lr_prec, ndigits = 3))
print("F1 Score:", round(lr_f1, ndigits = 3))
print("Log Loss:", round(lr_logloss, ndigits = 3))
```

[ ]:
```
# Running Linear Discriminant Analysis model

# Initialize LDA model
model_lda = LinearDiscriminantAnalysis()

# param grid for LDA
param_grid = {
    'solver': ['lsqr', 'eigen'],
    'shrinkage': [None, 'auto'] + list(np.linspace(0, 1, 50)),
    'n_components': [None, 1],
    'store_covariance': [True, False]
}

lda_grid = GridSearchCV(model_lda, param_grid, cv=5)
lda_grid.fit(X_train, y_train)
print(lda_grid.best_params_)

# Predict on the test data using the best model
lda_grid_predictions = lda_grid.predict(X_test)

print('-----------------------------------------------------')

# Calculate model performance metrics
lda_acc = accuracy_score(y_test, lda_grid_predictions)
lda_prec = precision_score(y_test, lda_grid_predictions)
lda_f1 = f1_score(y_test, lda_grid_predictions)
lda_logloss = log_loss(y_test, lda_grid_predictions)

# Print the performance metrics
print("Accuracy:", round(lda_acc, ndigits = 3))
print("Precision:", round(lda_prec, ndigits = 3))
print("F1 Score:", round(lda_f1, ndigits = 3))
print("Log Loss:", round(lda_logloss, ndigits = 3))
```

[ ]:
```
# Running k-Nearest Neighbours model

# Initialize kNN model
model_knn = KNeighborsClassifier()

# param_grid for k-Nearest Neighbours
param_grid = {
```

```
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'p': [1, 2]
}
knn_grid = GridSearchCV(model_knn, param_grid, cv=5)
knn_grid.fit(X_train, y_train)
print(knn_grid.best_params_)

# Predict the model using the best parameters
knn_grid_predictions = knn_grid.predict(X_test)

print('--------------------------------------------------')

knn_acc = accuracy_score(y_test, knn_grid_predictions)
knn_prec = precision_score(y_test, knn_grid_predictions)
knn_f1 = f1_score(y_test, knn_grid_predictions)
knn_logloss = log_loss(y_test, knn_grid_predictions)

print("Accuracy:", round(knn_acc, ndigits = 3))
print("Precision:", round(knn_prec, ndigits = 3))
print("F1 Score:", round(knn_f1, ndigits = 3))
print("Log Loss:", round(knn_logloss, ndigits = 3))
```

[ ]:
```
# Running Random Forest model

# Initialize random forest classifier
model_rf = RandomForestClassifier()

# param_grid for random forest classifier
param_grid = {
    'n_estimators': [100, 200, 300],
    'max_depth': [1, 5, 10],
    'min_samples_split': [2, 5, 10]
}
rf_grid = GridSearchCV(model_rf, param_grid, cv=5)
rf_grid.fit(X_train, y_train)
print(rf_grid.best_params_)

# Predict the model using the best parameters
rf_grid_predictions = rf_grid.predict(X_test)

print('--------------------------------------------------')

rf_acc = accuracy_score(y_test, rf_grid_predictions)
rf_prec = precision_score(y_test, rf_grid_predictions)
rf_f1 = f1_score(y_test, rf_grid_predictions)
```

```
rf_logloss = log_loss(y_test, rf_grid_predictions)

print("Accuracy:", round(rf_acc, ndigits = 3))
print("Precision:", round(rf_prec, ndigits = 3))
print("F1 Score:", round(rf_f1, ndigits = 3))
print("Log Loss:", round(rf_logloss, ndigits = 3))
```

[ ]:
```
# Running Boosting model

# Initialize boosting classifier
gbc_model = GradientBoostingClassifier()

# param_grid for boosting classifier
param_grid = {
    'loss': ['log_loss', 'exponential'],
    'learning_rate': [0.01, 0.5, 1],
    'max_depth': [1, 3, 5],
}
gbc_grid = GridSearchCV(gbc_model, param_grid, cv=5)
gbc_grid.fit(X_train, y_train)
print(gbc_grid.best_params_)

# Predict the model using the best parameters
gbc_grid_predictions = gbc_grid.predict(X_test)

print('-------------------------------------------------')

gbc_acc = accuracy_score(y_test, gbc_grid_predictions)
gbc_prec = precision_score(y_test, gbc_grid_predictions)
gbc_f1 = f1_score(y_test, gbc_grid_predictions)
gbc_logloss = log_loss(y_test, gbc_grid_predictions)

print("Accuracy:", round(gbc_acc, ndigits = 3))
print("Precision:", round(gbc_prec, ndigits = 3))
print("F1 Score:", round(gbc_f1, ndigits = 3))
print("Log Loss:", round(gbc_logloss, ndigits = 3))
```

[ ]:
```
# Confusion Matrices

fig, axes = plt.subplots(3, 2, figsize=(15, 20))

# Logistic Regression
cm_lr = confusion_matrix(y_test, lr_grid_predictions)
disp_lr = ConfusionMatrixDisplay(cm_lr)
disp_lr.plot(ax=axes[0, 0])
axes[0, 0].set_title('Confusion Matrix - Logistic Regression', fontsize = 16)
```

```
# LDA
cm_lda = confusion_matrix(y_test, lda_grid_predictions)
disp_lda = ConfusionMatrixDisplay(cm_lda)
disp_lda.plot(ax=axes[0, 1])
axes[0, 1].set_title('Confusion Matrix - Linear Discriminant Analysis',␣
 ↪fontsize = 16)

# kNN
cm_knn = confusion_matrix(y_test, knn_grid_predictions)
disp_knn = ConfusionMatrixDisplay(cm_knn)
disp_knn.plot(ax=axes[1, 0])
axes[1, 0].set_title('Confusion Matrix - K Nearest Neighbours', fontsize = 16)

# Random Forest
cm_rf = confusion_matrix(y_test, rf_grid_predictions)
disp_rf = ConfusionMatrixDisplay(cm_rf)
disp_rf.plot(ax=axes[1, 1])
axes[1, 1].set_title('Confusion Matrix - Random Forest', fontsize = 16)

# Boosting
cm_gbc = confusion_matrix(y_test, gbc_grid_predictions)
disp_gbc = ConfusionMatrixDisplay(cm_gbc)
disp_gbc.plot(ax=axes[2, 0])
axes[2, 0].set_title('Confusion Matrix - Boosting', fontsize = 16)

# Hide empty subplot
axes[2, 1].axis('off')
plt.tight_layout()
plt.show()
```

[ ]:
```
# ROC curves
lr_grid_probabilities = lr_grid.predict_proba(X_test)[:, 1]
fpr_lr, tpr_lr, thresholds_lr = roc_curve(y_test, lr_grid_probabilities)
roc_auc_lr = roc_auc_score(y_test, lr_grid_probabilities)

lda_grid_probabilities = lda_grid.predict_proba(X_test)[:, 1]
fpr_lda, tpr_lda, thresholds_lda = roc_curve(y_test, lda_grid_probabilities)
roc_auc_lda = roc_auc_score(y_test, lda_grid_probabilities)

knn_grid_probabilities = knn_grid.predict_proba(X_test)[:, 1]
fpr_knn, tpr_knn, thresholds_knn = roc_curve(y_test, knn_grid_probabilities)
roc_auc_knn = roc_auc_score(y_test, knn_grid_probabilities)

rf_grid_probabilities = rf_grid.predict_proba(X_test)[:, 1]
fpr_rf, tpr_rf, thresholds_rf = roc_curve(y_test, rf_grid_probabilities)
roc_auc_rf = roc_auc_score(y_test, rf_grid_probabilities)
```

```
gbc_grid_probabilities = gbc_grid.predict_proba(X_test)[:, 1]
fpr_gbc, tpr_gbc, thresholds_gbc = roc_curve(y_test, gbc_grid_probabilities)
roc_auc_gbc = roc_auc_score(y_test, gbc_grid_probabilities)

plt.figure(figsize=(14, 18))

# Logistic Regression
plt.subplot(3, 2, 1)
plt.plot(fpr_lr, tpr_lr, color='blue', lw=2, label=f'ROC Curve (AUC =␣
 ↪{roc_auc_lr:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Logistic Regression', fontsize = 16)
plt.legend(loc='lower right')

# Linear Discriminant Analysis
plt.subplot(3, 2, 2)
plt.plot(fpr_lda, tpr_lda, color='blue', lw=2, label=f'ROC Curve (AUC =␣
 ↪{roc_auc_lda:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Linear Discriminant Analysis', fontsize = 16)
plt.legend(loc='lower right')

# K Nearest Neighbours
plt.subplot(3, 2, 3)
plt.plot(fpr_knn, tpr_knn, color='blue', lw=2, label=f'ROC Curve (AUC =␣
 ↪{roc_auc_knn:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - K Nearest Neighbours',  fontsize = 16)
plt.legend(loc='lower right')

# Random Forest
plt.subplot(3, 2, 4)
plt.plot(fpr_rf, tpr_rf, color='blue', lw=2, label=f'ROC Curve (AUC =␣
 ↪{roc_auc_rf:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
```

```
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Random Forest',  fontsize = 16)
plt.legend(loc='lower right')

# Boosting
plt.subplot(3, 2, 5)
plt.plot(fpr_gbc, tpr_gbc, color='blue', lw=2, label=f'ROC Curve (AUC =␣
 ↪{roc_auc_gbc:.2f})')
plt.plot([0, 1], [0, 1], color='gray', linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Gradient Boost',  fontsize = 16)
plt.legend(loc='lower right')

# Adjust layout
plt.tight_layout()
plt.show()
```

[ ]:
```
# feature importance in random forest and boosting models


feature_importances_rf = rf_grid.best_estimator_.feature_importances_
feature_importance_df_rf = pd.DataFrame({'Feature': X_scaled.columns,␣
 ↪'Importance': feature_importances_rf})
feature_importance_df_rf = feature_importance_df_rf.
 ↪sort_values(by='Importance', ascending=False).reset_index(drop = True)

feature_importances_gbc = gbc_grid.best_estimator_.feature_importances_
feature_importance_df_gbc = pd.DataFrame({'Feature': X_scaled.columns,␣
 ↪'Importance': feature_importances_gbc})
feature_importance_df_gbc = feature_importance_df_gbc.
 ↪sort_values(by='Importance', ascending=False).reset_index(drop = True)

fig, ax = plt.subplots(1,2, figsize = (16,8))
ax[0].barh(feature_importance_df_rf['Feature'],␣
 ↪feature_importance_df_rf['Importance'])
ax[0].set_xlabel('Importance')
ax[0].set_ylabel(None)
ax[0].set_title('Feature Importance in Random Forest Model', fontsize = 16)
ax[0].invert_yaxis()
```

```
ax[1].barh(feature_importance_df_gbc['Feature'],␣
 ↪feature_importance_df_gbc['Importance'])
ax[1].set_xlabel('Importance')
ax[1].set_ylabel(None)
ax[1].set_title('Feature Importance in Boosting Model', fontsize = 16)
ax[1].invert_yaxis()

plt.tight_layout()
plt.show()
```