In [1]:
```
# Import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report, accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

In [3]:
```
data = pd.read_csv(r"C:\Users\junai\OneDrive - Middlesex University\ML, Regression\
```

In [4]:
```
# Explore the dataset
print(data.head())
```

```
   alcohol  malic_acid   ash  alcalinity_of_ash  magnesium  total_phenols  \
0    13.50        1.81  2.61               20.0       96.0           2.53
1    13.50        3.12  2.62               24.0      123.0           1.40
2    13.41        3.84  2.12               18.8       90.0           2.45
3    12.77        3.43  1.98               16.0       80.0           1.63
4    13.63        1.81  2.70               17.2      112.0           2.85

   flavanoids  nonflavanoid_phenols  proanthocyanins  color_intensity   hue  \
0        2.61                  0.28             1.66             3.52  1.12
1        1.57                  0.22             1.25             8.60  0.59
2        2.68                  0.27             1.48             4.28  0.91
3        1.25                  0.43             0.83             3.40  0.70
4        2.91                  0.30             1.46             7.30  1.28

   od280/od315_of_diluted_wines  proline  target
0                          3.82    845.0       0
1                          1.30    500.0       2
2                          3.00   1035.0       0
3                          2.12    372.0       1
4                          2.88   1310.0       0
```

In [5]:
```
print(data.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 168 entries, 0 to 167
Data columns (total 14 columns):
 #   Column                        Non-Null Count  Dtype
---  ------                        --------------  -----
 0   alcohol                       168 non-null    float64
 1   malic_acid                    168 non-null    float64
 2   ash                           168 non-null    float64
 3   alcalinity_of_ash             168 non-null    float64
 4   magnesium                     168 non-null    float64
 5   total_phenols                 168 non-null    float64
 6   flavanoids                    168 non-null    float64
 7   nonflavanoid_phenols          168 non-null    float64
 8   proanthocyanins               168 non-null    float64
 9   color_intensity               168 non-null    float64
 10  hue                           168 non-null    float64
 11  od280/od315_of_diluted_wines  168 non-null    float64
 12  proline                       168 non-null    float64
 13  target                        168 non-null    int64
dtypes: float64(13), int64(1)
memory usage: 18.5 KB
None
```

In [6]:
```
print(data['target'].value_counts())
```

```
target
1    68
0    55
2    45
Name: count, dtype: int64
```

Separate feature data from target The code belo separates the data into two components:

Feature Matrix (X): Contains all the input features (independent variables) that will be used to make predictions. Target Variable (y): Contains the corresponding labels or outcomes (dependent variable) that the model will try to predict.

In [7]:
```
# Split the data into features (X) and target variable (y)
X = data.drop('target', axis=1)
y = data['target']
```

Generating train and test sets Splitting data into training and test sets is a crucial step in the machine learning workflow. It enables robust model evaluation, helps detect potential issues such as overfitting, facilitates hyperparameter tuning, and ensures that the model's performance is assessed on independent, unseen data

In [8]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
```

Logistic regression model using scikit-learn's LogisticRegression class

multi_class: This parameter determines the strategy used to handle multiple classes. 'multinomial' is used for the softmax function, which is suitable for multi-class classification problems.

solver: The optimization algorithm used to find the weights that minimize the logistic loss. 'lbfgs' is one of the solvers suitable for multiclass problems.

max_iter: The maximum number of iterations for the optimization algorithm. Increasing this value may be necessary if the algorithm doesn't converge.

random_state: Seed for the random number generator. Setting a random seed ensures that the results are reproducible.

In [9]:
```
# Create a logistic regression model
model = LogisticRegression(multi_class='multinomial', solver='lbfgs', max_iter=1000
```

Machine learning pipeline using scikit-learn's Pipeline class. This is a convenient way to chain multiple processing steps together. In this case, the pipeline consists of two steps: standardization and logistic regression.

StandardScaler: The first step in the pipeline is to standardize the features. StandardScaler scales the input features to have a mean of 0 and a standard deviation of 1. This is often important when working with machine learning models, especially those that rely on distances or gradients.

Logistic Regression Model (model): The second step in the pipeline is to apply the logistic regression model. The model was defined earlier using the LogisticRegression class with specific parameters for handling multiclass classification.

In [28]:
```
# Create a pipeline with standardization and logistic regression
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('model', model)
])
```

In [29]:
```
# Training the pipeline
pipeline.fit(X_train, y_train)

# Making predictions
y_pred = pipeline.predict(X_test)
```

Defining a grid of hyperparameter values for conducting a grid search. This grid search will be performed to find the best combination of hyperparameters for your logistic regression model within the specified parameter grid.

model__C: This parameter represents the inverse of the regularization strength. Smaller values specify stronger regularization. The grid contains a range of values from very small (0.001) to large (100) to explore the effect of regularization on the model's performance.

model__penalty: This parameter determines the type of regularization applied. 'l1' corresponds to L1 regularization (Lasso), and 'l2' corresponds to L2 regularization (Ridge). By including both options in the grid, the grid search will explore models with different types of regularization.

The 'model__' prefix is used to indicate that these hyperparameters are part of the logistic regression model within the pipeline.

In [30]:
```
# Define hyperparameters for grid search
param_grid = {
    'model__C': [0.001, 0.01, 0.1, 1, 10, 100],
    'model__penalty': ['l1', 'l2']
}
```

model = LogisticRegression(): This line initializes a logistic regression model with default hyperparameter values.

model.fit(X_train, y_train): This line fits the model to the training data, where X_train is the feature matrix and y_train is the corresponding target variable.

print("Default value of C:", model.C): This line prints the default value of the regularization parameter (C). In scikit-learn's logistic regression implementation, the default value for C is 1.0.

print("Default penalty:", model.penalty): This line prints the default penalty method. The default is 'l2', which corresponds to L2 regularization (Ridge).

In [31]:
```
# Assuming X_train and y_train are your training features and labels
model = LogisticRegression()
model.fit(X_train, y_train)
```

```
# Print the default regularization value and method
print("Default value of C:", model.C)
print("Default penalty:", model.penalty)
```

```
Default value of C: 1.0
Default penalty: l2
```

```
C:\Users\junai\anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.py:460:
ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

preds_train and preds_test will contain the model's predictions for the target variable based on the features in the training and test sets, respectively.

In [32]:
```
preds_train = model.predict(X_train)
preds_test = model.predict(X_test)
```

The code snippet below assesses how well a model is performing on the data it was trained on. It's crucial to note that evaluating the model on the training set only gives an indication of how well the model has memorised the training data

In [33]:
```
accuracy = accuracy_score(y_train, preds_train)
report = classification_report(y_train, preds_train)
print(f"Accuracy: {accuracy}")
print("Classification Report:\n", report)
```

```
Accuracy: 0.9776119402985075
Classification Report:
               precision    recall  f1-score   support

           0       1.00      0.95      0.98        44
           1       0.96      0.98      0.97        52
           2       0.97      1.00      0.99        38

    accuracy                           0.98       134
   macro avg       0.98      0.98      0.98       134
weighted avg       0.98      0.98      0.98       134
```

In [34]:
```
accuracy = accuracy_score(y_test, preds_test)
report = classification_report(y_test, preds_test)
print(f"Accuracy: {accuracy}")
print("Classification Report:\n", report)
```

```
Accuracy: 0.9705882352941176
Classification Report:
               precision    recall  f1-score   support

           0       1.00      0.91      0.95        11
           1       0.94      1.00      0.97        16
           2       1.00      1.00      1.00         7

    accuracy                           0.97        34
   macro avg       0.98      0.97      0.97        34
weighted avg       0.97      0.97      0.97        34
```

It's important to recognise that hyperparameter tuning is often employed to address issues related to overfitting or underfitting, and not all models will benefit from extensive tuning. Therefore, understanding the underlying trade-off between bias and variance is crucial for making informed decisions during the model development process.

GridSearchCV: This class performs an exhaustive search over a specified parameter grid, evaluating the model's performance using cross-validation. In your case, it's set up to use a logistic regression model within a pipeline (pipeline) and the hyperparameter grid (param_grid) you defined earlier.

grid_search.fit(X_train, y_train): This line fits the grid search to the training data, and it will evaluate the performance of different combinations of hyperparameters using 5-fold cross-validation (cv=5).

best_params = grid_search.best*params*: After the grid search is complete, you retrieve the best hyperparameters using the best*params* attribute.

final_model = grid_search.best*estimator*: This line retrieves the best estimator from the grid search. The best estimator is the combination of preprocessing steps and the model that performed the best on the validation data during the cross-validation.

final_model.fit(X_train, y_train): Finally, you train the final model using the entire training set and the best hyperparameters found during the grid search.

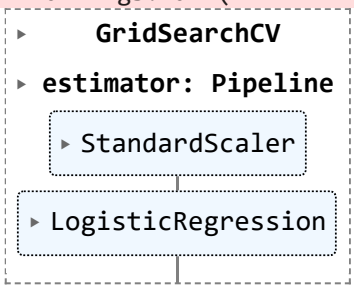
In [35]:
```
# Perform grid search with cross-validation
grid_search = GridSearchCV(pipeline, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)
```

```
C:\Users\junai\anaconda3\Lib\site-packages\sklearn\model_selection\_validation.py:
425: FitFailedWarning:
30 fits failed out of a total of 60.
The score on these train-test partitions for these parameters will be set to nan.
If these failures are not expected, you can try to debug them by setting error_sco
re='raise'.

Below are more details about the failures:
--------------------------------------------------------------------------------
30 fits failed with the following error:
Traceback (most recent call last):
  File "C:\Users\junai\anaconda3\Lib\site-packages\sklearn\model_selection\_valida
tion.py", line 732, in _fit_and_score
    estimator.fit(X_train, y_train, **fit_params)
  File "C:\Users\junai\anaconda3\Lib\site-packages\sklearn\base.py", line 1151, in
wrapper
    return fit_method(estimator, *args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\junai\anaconda3\Lib\site-packages\sklearn\pipeline.py", line 420,
in fit
    self._final_estimator.fit(Xt, y, **fit_params_last_step)
  File "C:\Users\junai\anaconda3\Lib\site-packages\sklearn\base.py", line 1151, in
wrapper
    return fit_method(estimator, *args, **kwargs)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\junai\anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.
py", line 1168, in fit
    solver = _check_solver(self.solver, self.penalty, self.dual)
             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "C:\Users\junai\anaconda3\Lib\site-packages\sklearn\linear_model\_logistic.
py", line 56, in _check_solver
    raise ValueError(
ValueError: Solver lbfgs supports only 'l2' or 'none' penalties, got l1 penalty.

  warnings.warn(some_fits_failed_message, FitFailedWarning)
C:\Users\junai\anaconda3\Lib\site-packages\sklearn\model_selection\_search.py:976:
UserWarning: One or more of the test scores are non-finite: [       nan 0.75498575
nan 0.98490028        nan 0.97720798
        nan 0.96239316        nan 0.97008547        nan 0.97008547]
  warnings.warn(
```

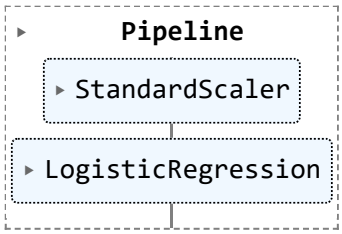
Out[35]:



In [36]:
```
# Get the best parameters
best_params = grid_search.best_params_
```

In [37]:
```
# Train the final model with the best parameters
final_model = grid_search.best_estimator_
final_model.fit(X_train, y_train)
```

Out[37]:



The code snippet below prints the intercept and coefficients of a logistic regression model.

In [38]:
```
# Print the intercept and coefficients
print("Intercept (Beta 0):", model.intercept_[0])
print("Coefficients (Beta 1 to Beta n):", model.coef_[0])
```

```
Intercept (Beta 0): -0.02725425457501436
Coefficients (Beta 1 to Beta n): [ 0.04827819  0.35462686  0.12584522 -0.39065712
-0.04070859  0.22243726
  0.47915736 -0.02486367  0.10980057  0.11965813 -0.0076989   0.36445713
  0.00975189]
```

In [ ]:
```
```

variable_names = list(X_train.columns): This line creates a list of variable names by extracting the column names from your training feature matrix X_train.

coefficients = list(model.coef_[0]): This line extracts the coefficients of the logistic regression model for each variable. The [0] index is used because model.coef_ is a 2D array, and you're interested in the coefficients for the first class (assuming a binary classification problem).

pd.DataFrame({'Variable': variable_names, 'Coefficient': coefficients}): This line creates a pandas DataFrame named beta_coeff with two columns: 'Variable' and 'Coefficient'.

print(beta_coeff): Finally, this line prints the DataFrame, displaying the variable names and their corresponding coefficients.

This DataFrame provides insights into the importance and direction of each variable in the logistic regression model. Positive coefficients indicate a positive impact on the log-odds of the target variable, while negative coefficients indicate a negative impact.

In [39]:
```
# Extract variable names and corresponding coefficients
variable_names = list(X_train.columns)
coefficients = list(model.coef_[0])
# Create a DataFrame
beta_coeff = pd.DataFrame({'Variable': variable_names, 'Coefficient': coefficients})
# Print the DataFrame
beta_coeff
```

Out[39]:

	Variable	Coefficient
0	alcohol	0.048278
1	malic_acid	0.354627
2	ash	0.125845
3	alcalinity_of_ash	-0.390657
4	magnesium	-0.040709
5	total_phenols	0.222437
6	flavanoids	0.479157
7	nonflavanoid_phenols	-0.024864
8	proanthocyanins	0.109801
9	color_intensity	0.119658
10	hue	-0.007699
11	od280/od315_of_diluted_wines	0.364457
12	proline	0.009752

y_pred = final_model.predict(X_test): This line uses the trained final_model to make predictions on the test set (X_test). The predicted values are stored in y_pred.

accuracy = accuracy_score(y_test, y_pred): This line calculates the accuracy of the model by comparing the predicted values (y_pred) with the actual labels from the test set (y_test). The result is stored in the variable accuracy.

classification_rep = classification_report(y_test, y_pred): This line generates a classification report using the classification_report function from scikit-learn. The classification report includes metrics such as precision, recall, and F1-score for each class, as well as the overall performance. The result is stored in the variable classification_rep.

After running this code, you can examine the accuracy and the detailed classification report to understand how well the model performs on the unseen test data. The classification report provides insights into the precision, recall, and F1-score for each class, helping you to evaluate the model's performance across different classes.

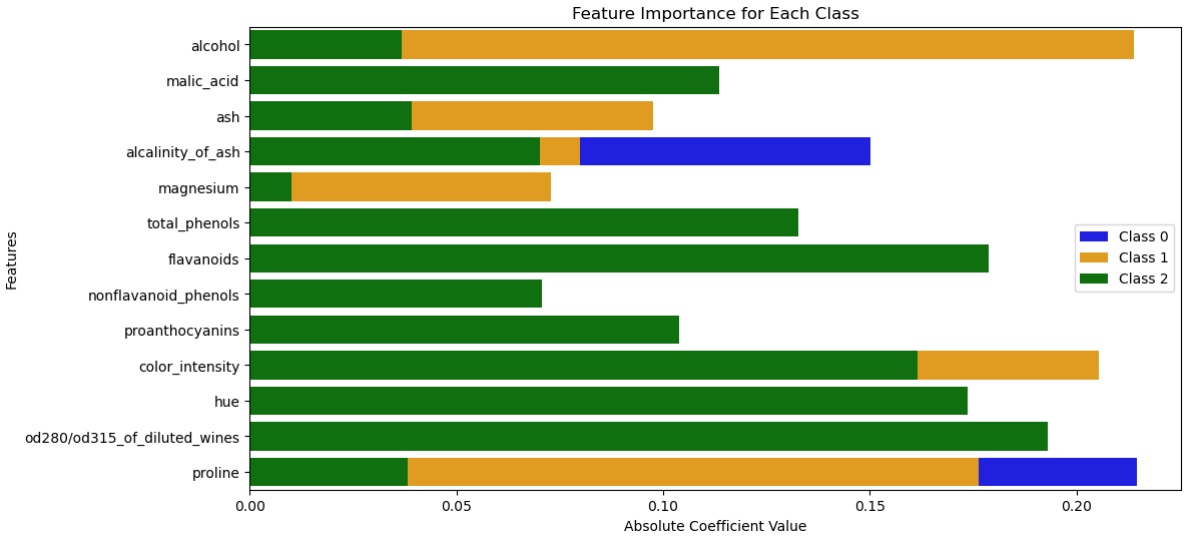
In [40]:
```
# Evaluate the model on the test set
y_pred = final_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
```

In [54]:
```
# Extract feature importance from the model
feature_importance = final_model.named_steps['model'].coef_
```

This visualization allows you to see the importance of each feature for each class in your logistic regression model. Positive and negative values indicate the direction of influence, and the absolute values provide a sense of the magnitude of the impact.

In [55]:
```
# Visualize feature importance
plt.figure(figsize=(12, 6))
sns.barplot(x=np.abs(feature_importance[0]), y=X.columns, color='blue', label='Clas
```

```
sns.barplot(x=np.abs(feature_importance[1]), y=X.columns, color='orange', label='Cl
sns.barplot(x=np.abs(feature_importance[2]), y=X.columns, color='green', label='Cla
plt.title('Feature Importance for Each Class')
plt.xlabel('Absolute Coefficient Value')
plt.ylabel('Features')
plt.legend()
plt.show()
```



In [ ]:

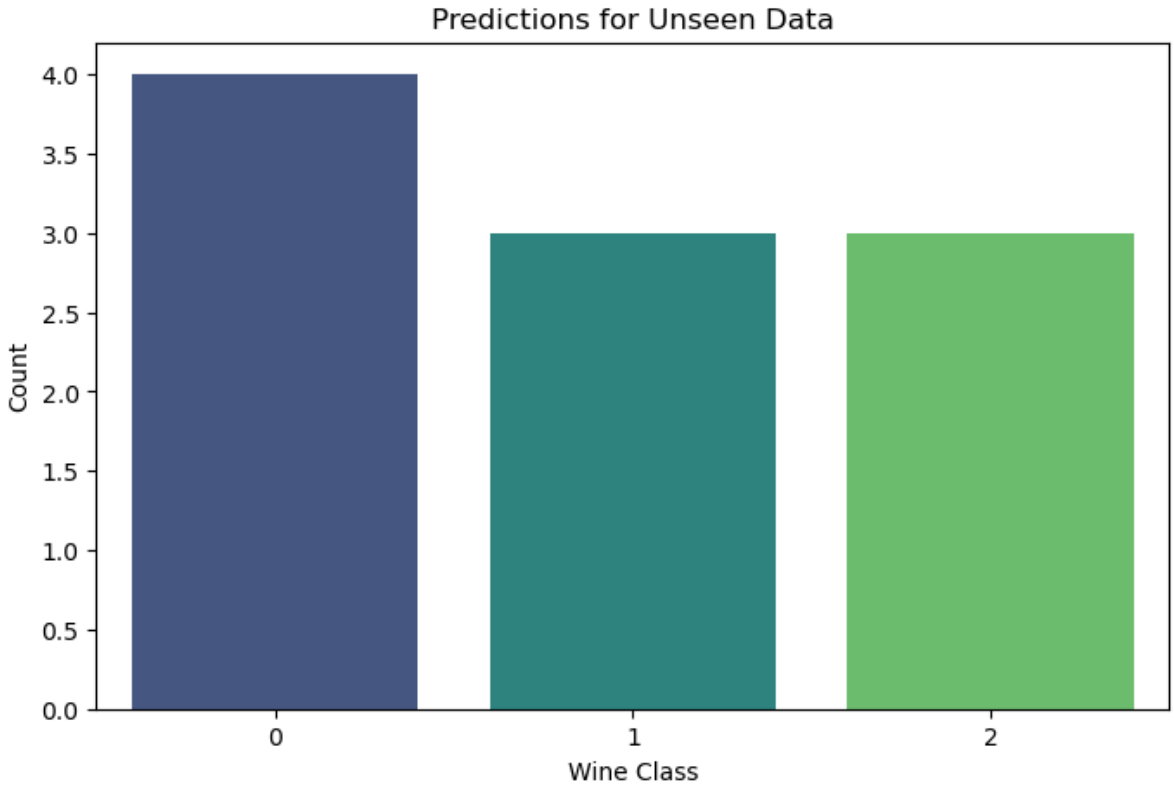
In [43]:
```
# Load the unseen data
unseen_data = pd.read_csv(r"C:\Users\junai\OneDrive - Middlesex University\ML, Regr
```

In [44]:
```
# Make predictions on unseen data
unseen_predictions = final_model.predict(unseen_data)
```

In [45]:
```
# Display the predictions
print("Predictions for Unseen Data:")
print(unseen_predictions)
```

```
Predictions for Unseen Data:
[0 0 2 0 1 0 1 2 1 2]
```

In [46]:
```
# Visualize predictions for unseen data
plt.figure(figsize=(8, 5))
sns.countplot(x=unseen_predictions, palette='viridis')
plt.title('Predictions for Unseen Data')
plt.xlabel('Wine Class')
plt.ylabel('Count')
plt.show()
```


Predictions for Unseen Data
Count
Wine Class
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
1
2

In [ ]: