Mini Project

- Jared Pek (jarpek-4@student.ltu.se)
- Randall Chiang (ranchi-4@student.ltu.se)
- Joel Tham (joetha-4@student.ltu.se)

Video Presentation

https://drive.google.com/file/d/1K4QQObYQOxo2ydEmPbwJWb3g5aPRATir/view?usp=sharing

Introduction

Our mini project will focus on these UCI machine learning datasets:

1. Abalone
2. Bank Marketing
3. Car
4. Mushrooms
5. Wine Quality

The models we will be using to train on each dataset are the unsupervised K-Means clustering algorithm and the supervised logistic regression classification model.

Results

Car:

The Car Evaluation dataset was analyzed using both unsupervised and supervised learning methods. The unsupervised learning model, K-Means Clustering, produced an Adjusted Rand Index (ARI) of **0.0007**. These results indicate that the clusters formed by K-Means do not align well with the true class labels, and the clustering quality is poor. This is likely due to the categorical nature of the dataset and the limitations of using Euclidean distance in K-Means after one-hot encoding. The dataset does not exhibit natural clusters, making K-Means unsuitable for this task.

In contrast, the supervised learning model, Logistic Regression, achieved an overall accuracy of **92.87%**, demonstrating strong performance. The model effectively classified the majority classes (good and unacc), achieving high precision and recall for these categories. The minority classes (acc and vgood) were more challenging to classify, with lower precision and recall due to their smaller sample sizes and potential feature overlap. For example, the acc class achieved a precision of **65%** and a recall of **68.4%**, reflecting some misclassification issues. Despite this, the model maintained a balanced performance across all classes, as evidenced by a macro F1-score of **84.3%** and a weighted F1-score of **92.9%.**

Overall, Logistic Regression significantly outperformed K-Means Clustering for this dataset, demonstrating that the structured and categorical nature of the data is better suited to supervised learning methods. Addressing class imbalance or exploring clustering algorithms tailored for categorical data, such as K-Prototypes, could further improve performance.

Abalone:

The performance metrics of the logistic regression model and the k-means clustering model demonstrate differences in their ability to classify the data. Logistic regression achieved an overall accuracy of 27.51%, with a weighted average precision, recall, and F1-score of 0.20, 0.27, and 0.22, respectively. The model performed moderately well for Class 7, 8, and 10, achieving F1-scores of 0.45, 0.39, and 0.32, respectively. For Class 7, precision and recall were 0.37 and 0.58, reflecting its relatively stronger performance. However, the model struggled significantly with many classes such as Class 3, 4, 12, 14, and 15, where precision, recall, and F1-scores were all 0.0. This indicates a failure to predict these classes, likely due to class imbalance and limited representation of smaller classes.

In contrast, the k-means clustering model performed slightly worse, with an overall accuracy of 26.79%. While the model demonstrated an F1-score of 0.45 for Class 8 and some recognition for Class 10 (F1-score 0.33), it struggled significantly across most other classes. Precision, recall, and F1-scores for many smaller classes were 0.0, highlighting its inability to identify these categories. The reliance on an unsupervised approach, coupled with the presence of overlapping features, likely led to poor clustering and poor classification performance.

Overall, the logistic regression model performs marginally better than the k-means clustering model, particularly for Class 7, 8, and 10, where it demonstrates higher F1-scores. However, both models exhibit severe shortcomings due to class imbalance, linear separability issues, and overlapping features. Improving the models could involve addressing class imbalance with oversampling or undersampling, engineering better features, and exploring non-linear classifiers such as decision trees or ensemble methods. The current results highlight the limitations of both approaches for this dataset and the need for further optimization.

Wine Quality:

The performance metrics of the logistic regression model and the k-means clustering model indicate distinct outcomes in their ability to classify wine quality. Logistic regression achieved an overall accuracy of 53.62%, with a weighted average precision, recall, and F1-score of 0.50, 0.54, and 0.50, respectively. The model demonstrated moderate performance for Class 5 and 6, which represent the majority of the samples, achieving F1-scores of 0.57 and 0.60, respectively. For Class 5, the precision and recall were 0.54 and 0.61, while for Class 6, precision and recall were 0.54 and 0.68, respectively. However, the model struggled significantly with minority classes such as Class 4, 8, and 9, where precision, recall, and F1-scores were 0.0, indicating its inability to predict these underrepresented classes. For Class 3, precision was perfect at 1.0, but the recall was very low at 0.17, leading to a poor F1-score of 0.29.

In contrast, the k-means clustering model performed worse, with an overall accuracy of 47.46% and a weighted average precision, recall, and F1-score of 0.36, 0.47, and 0.41, respectively. While the model showed some ability to classify Class 5 and 6, achieving F1-scores of 0.44 and 0.59, respectively, it completely failed to predict Class 3, 4, 7, 8, and 9, with precision, recall, and F1-scores of 0.0 across these classes. The macro average F1-score of 0.15 highlights the model's inability to generalize well across all classes, further underscoring its poor performance on the minority labels.

Overall, the logistic regression model outperformed the k-means clustering model, particularly for the majority classes (Class 5 and 6), where it demonstrated better precision, recall, and F1-scores. However, both models struggled significantly with the minority classes, indicating the need for improvements in handling class imbalance. Addressing this issue through techniques like resampling, cost-sensitive learning, or employing ensemble methods could improve model performance. The results suggest that while logistic regression offers a slight advantage, both models remain limited in their ability to accurately classify all wine quality levels

Bank Marketing:

The performance metrics of the logistic regression model and the k-means clustering model reveal significant differences in their ability to classify the data. Logistic regression achieved a high overall accuracy of 93%, with a strong weighted average precision, recall, and F1-score of 0.94, 0.93, and 0.93, respectively. For class "0," the model demonstrated excellent performance with precision, recall, and F1-score around 0.96-0.97, indicating a robust ability to predict this majority class. However, the model struggled with class "1," achieving only 0.41 precision, 0.50 recall, and an F1-score of 0.45, reflecting challenges in handling the minority class due to potential class imbalance.

In contrast, the k-means clustering model performed poorly, with an overall accuracy of 65% and a substantially lower weighted average F1-score of 0.76. The macro average F1-score of 0.10 highlights the model's inability to effectively classify the data across all classes. While precision and recall for class "0" were moderate at 0.95 and 0.69, respectively, the performance for class "1" was abysmal, with a precision of 0.07, recall of 0.02, and F1-score of 0.03. Additionally, the presence of multiple unused clusters (classes "2" through "7") in the k-means results suggests poor clustering and a failure to capture meaningful groupings in the data.

Overall, the logistic regression model is superior for this dataset, especially when accuracy and handling of the dominant class are critical. However, it still requires improvement in predicting the minority class. The k-means clustering model is not a suitable alternative in this context due to its inability to effectively separate the classes and its reliance on an unsupervised approach that does not leverage label information. Addressing class imbalance and optimizing the logistic regression model might further enhance its performance.

Mushrooms:

The classification results of the Logistic Regression model and the K-Means clustering model present distinct performance patterns, revealing their relative strengths and weaknesses in this dataset. For Logistic Regression, the overall accuracy is 49%, with significant imbalances between precision, recall, and F1-score for the two classes. Class 0 exhibits high precision (0.97) but poor recall (0.19), suggesting that while the model is confident in its predictions for Class 0, it fails to identify most actual instances of this class. Conversely, Class 1 shows high recall (0.99) but low precision (0.42), indicating that most actual instances of Class 1 are captured, but many false positives are included. This imbalance is reflected in the weighted F1-score of 0.42 and suggests the model struggles with proper discrimination in a likely imbalanced dataset.

For the K-Means clustering model, the performance metrics are even less consistent. The overall accuracy is slightly lower at 45%, and the metrics for individual clusters show a stark disparity. Cluster 0 achieves moderate performance with an F1-score of 0.61, supported by reasonable precision (0.56) and recall (0.68). However, Cluster 1 has a high precision of 1.00 but extremely low recall (0.04), indicating that the cluster is highly specific but captures almost none of the relevant instances. Clusters 2, 3, 5, and 7 are entirely ineffective, with zero instances classified, resulting in F1-scores of 0. The macro-average metrics (precision, recall, and F1-score) are all very low, further highlighting the uneven performance across clusters.

In comparison, Logistic Regression demonstrates better capability in identifying patterns relevant to both classes, albeit with substantial room for improvement in balancing precision and recall. K-Means clustering, on the other hand, struggles to form meaningful clusters that align well with the dataset's underlying structure, as evidenced by its poor recall and F1-scores across most clusters. This suggests that the data may not exhibit the clear separability required for K-Means to perform effectively. Additionally, the presence of empty clusters in K-Means indicates potential challenges with the choice of the number of clusters or the initialization method.

Overall, the Logistic Regression model is preferable in this case due to its comparatively better performance, albeit limited by class imbalance and misclassification issues. The clustering model might require further optimization or a different clustering technique to achieve meaningful results.

Conclusion

Based on the models created from the 5 datasets, it can be determined that unsupervised models performed much poorer than the supervised ones. This is because k-means clustering is not designed for supervised learning. Logistic regression leverages labeled data to learn the relationship between input features and the binary target variable, optimizing a decision boundary that separates the two classes based on probabilistic modeling. In contrast, k-means is an unsupervised learning algorithm that groups data points into clusters based solely on feature similarity, without considering class labels.

As a result, k-means can misidentify clusters that do not align with the true classes, especially when the data distribution is complex or the clusters are not well-separated. Additionally, k-means assumes clusters are spherical and equally sized, which may not hold in real-world classification tasks. Logistic regression, by incorporating class labels and optimizing for classification accuracy, is typically more effective at binary classification tasks where labeled training data is available.

These are evident through the results of all 5 datasets that we have done in this project.

Car Classification

In [53]:
```
from ucimlrepo import fetch_ucirepo
import pandas as pd
from sklearn.preprocessing import LabelEncoder, StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import adjusted_rand_score, accuracy_score
```

In [54]:
```
cars = fetch_ucirepo(id=19)
X = cars.data.features
y = cars.data.targets
```

In [55]:
```
print(X)
```

```
     buying  maint  doors persons lug_boot safety
0     vhigh  vhigh      2       2    small    low
1     vhigh  vhigh      2       2    small    med
2     vhigh  vhigh      2       2    small   high
3     vhigh  vhigh      2       2      med    low
4     vhigh  vhigh      2       2      med    med
...     ...    ...    ...     ...      ...    ...
1723    low    low  5more    more      med    med
1724    low    low  5more    more      med   high
1725    low    low  5more    more      big    low
1726    low    low  5more    more      big    med
1727    low    low  5more    more      big   high

[1728 rows x 6 columns]
```

In [56]:
```
print(y)
```

```
      class
0     unacc
1     unacc
2     unacc
3     unacc
4     unacc
...     ...
1723   good
1724  vgood
1725  unacc
1726   good
1727  vgood

[1728 rows x 1 columns]
```

Pre-Processing Of Data

In [78]:
```
def preprocess_car_evaluation(X, y):
    # Encode categorical features and target variable
    X_encoded = pd.get_dummies(X)  # One-hot encoding for features
    le = LabelEncoder()
    y_encoded = le.fit_transform(y)  # Encode target

    # Dynamically create a consistent class mapping
    dynamic_mapping = {index: label for index, label in enumerate(le.classes_)}

    # Standardize features
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X_encoded)

    return X_scaled, y_encoded, dynamic_mapping

# Fetch and preprocess the data
X_preprocessed, y_preprocessed, dynamic_mapping = preprocess_car_evaluation(X, y)

# Overwrite predefined class_mapping dynamically to match LabelEncoder
class_mapping = dynamic_mapping

# Print mappings to confirm consistency
print("Dynamic Mapping:", dynamic_mapping)
print("Class Mapping (Updated):", class_mapping)
```

```
Dynamic Mapping: {0: 'acc', 1: 'good', 2: 'unacc', 3: 'vgood'}
Class Mapping (Updated): {0: 'acc', 1: 'good', 2: 'unacc', 3: 'vgood'}
```

```
c:\Users\mysto\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\preprocessing\_label.py:114: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Unsupervised Learning : K-Means Clustering

In [79]:
```
# Step 3: K-Means Clustering Function
def kmeans_clustering(X, y):
    # Perform K-Means Clustering
    kmeans = KMeans(n_clusters=len(set(y)), init='k-means++', random_state=42)
    y_pred = kmeans.fit_predict(X)

    # Evaluate clustering performance
    ari = adjusted_rand_score(y, y_pred)
    silhouette = silhouette_score(X, y_pred)

    return kmeans, y_pred, ari, silhouette
```

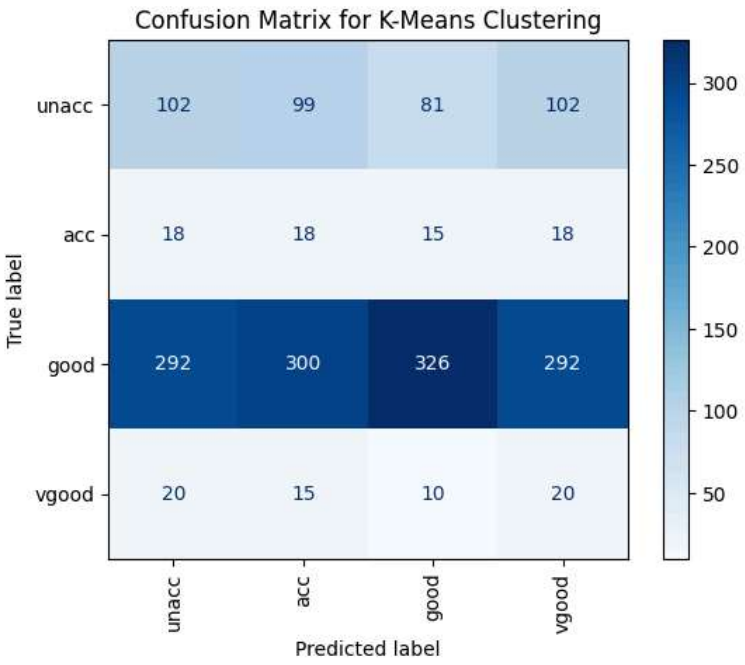
In [80]:
```
# Step 4: Apply K-Means and Evaluate
kmeans, kmeans_labels, kmeans_ari, kmeans_silhouette = kmeans_clustering(X_preprocessed, y_preprocessed)
```

In [81]:
```
# Step 5: Print Results
print("K-Means Clustering (Unsupervised) ARI:", kmeans_ari)
```

```
K-Means Clustering (Unsupervised) ARI: 0.000729652868196255
```

In [88]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from scipy.optimize import linear_sum_assignment
import numpy as np

# Map K-Means labels to actual labels using Hungarian Algorithm
def map_kmeans_labels(true_labels, kmeans_labels):
    # Create a confusion matrix
    cm = confusion_matrix(true_labels, kmeans_labels)

    # Find optimal mapping of cluster indices to true labels
    row_ind, col_ind = linear_sum_assignment(-cm)
    mapping = {col: row for row, col in zip(row_ind, col_ind)}

    # Map K-Means labels to true labels
    mapped_labels = np.array([mapping[label] for label in kmeans_labels])
    return mapped_labels, cm

# Map K-Means labels and compute confusion matrix
mapped_labels, kmeans_cm = map_kmeans_labels(y_preprocessed, kmeans_labels)

# Print confusion matrix
print("Confusion Matrix (K-Means):")
print(kmeans_cm)

# Plot confusion matrix
disp = ConfusionMatrixDisplay(confusion_matrix=kmeans_cm, display_labels=list(class_mapping.values()))
disp.plot(cmap="Blues", xticks_rotation="vertical")
plt.title("Confusion Matrix for K-Means Clustering")
plt.show()
```

```
Confusion Matrix (K-Means):
[[102  99  81 102]
 [ 18  18  15  18]
 [292 300 326 292]
 [ 20  15  10  20]]
```



The columns represent the predicted cluster labels assigned by the K-Means algorithm. Ideally, the predictions should align with the true labels, meaning the numbers would be concentrated along the diagonal of the matrix. However, in this confusion matrix, we see a very scattered distribution.

For example, in the "unacc" row (true label), the model correctly predicted 102 data points as 'unacc', but it also mistakenly placed 99, 81, and 102 data points into the other three clusters. Similarly, in the "good" row (true label), the numbers are spread across all four predicted labels, with values like 292, 300, 326, and 292. This indicates that the K-Means model has significant difficulty differentiating between the 'good' class and the other categories.

The "acc" and "vgood" rows show very small numbers scattered across the columns. This further highlights the inability of K-Means to form meaningful clusters that align with the actual class labels.

The poor performance of the K-Means algorithm can be attributed to two main reasons. First, K-Means relies on Euclidean distance to calculate similarities between data points, which works poorly with categorical features like 'buying price' or 'maintenance cost'. Second, since K-Means is an unsupervised learning algorithm, it does not have access to the true class labels during clustering. As a result, it struggles to find the correct groupings in this dataset.

In summary, the confusion matrix for K-Means clustering reveals that the algorithm fails to capture the underlying patterns in the data. The resulting clusters are poorly aligned with the true labels, making K-Means an unsuitable method for this problem.

Supervised Learning - Logistic Regression

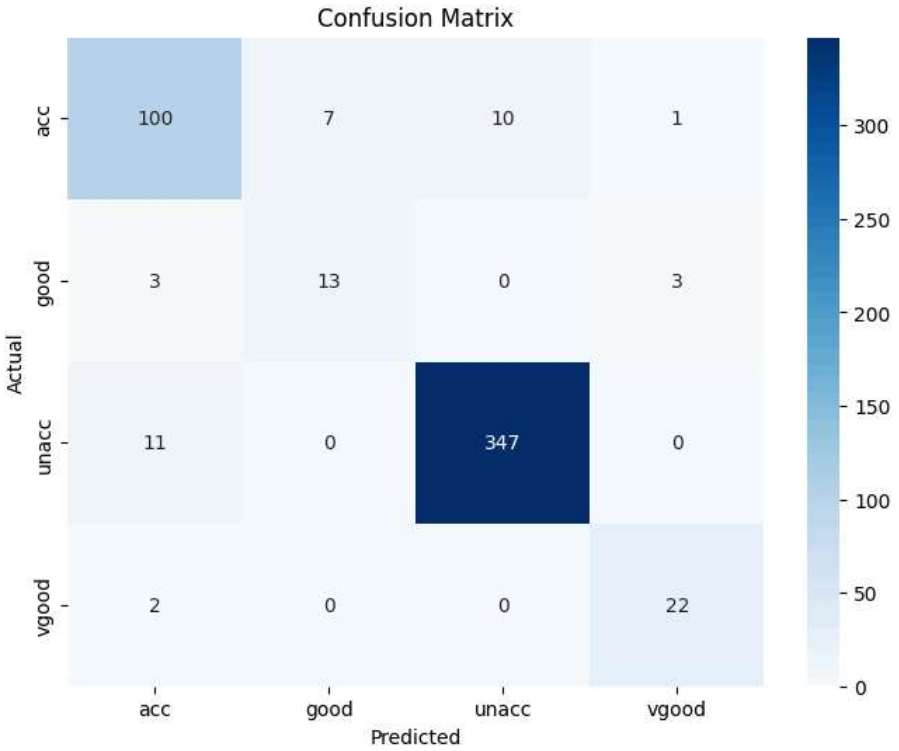
In [82]:
```
from sklearn.metrics import classification_report, confusion_matrix


def logistic_regression(X, y):
    # Split into training and test sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

    # Train logistic regression model
    clf = LogisticRegression(random_state=42, max_iter=500)
    clf.fit(X_train, y_train)

    # Predict and evaluate
    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    report = classification_report(y_test, y_pred, output_dict=True)
    confusion = confusion_matrix(y_test, y_pred)

    return clf, acc, report, confusion, X_train, X_test, y_train, y_test
```

In [83]:
```
# Call the function and get outputs
clf, logistic_accuracy, logistic_report, logistic_confusion, X_train, X_test, y_train, y_test = logistic_regression(X_prep

# Print results
print("Logistic Regression Accuracy:", logistic_accuracy)
```

```
Logistic Regression Accuracy: 0.928709055876686
```

In [84]:
```
# Step 6: Visualize Confusion Matrix
def plot_confusion_matrix(confusion, class_names):
```

```
    plt.figure(figsize=(8, 6))
    sns.heatmap(confusion, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, yticklabels=class_names)
    plt.title("Confusion Matrix")
    plt.xlabel("Predicted")
    plt.ylabel("Actual")
    plt.show()

plot_confusion_matrix(logistic_confusion, list(class_mapping.values()))
```



In this confusion matrix, the rows represent the actual class labels, while the columns represent the predicted class labels. The goal is to have the majority of values concentrated along the diagonal, which would indicate correct predictions.

Looking at the matrix, we can immediately see that Logistic Regression performs much better. For the "acc" row (actual label), 100 out of 118 cars were correctly classified as 'acc', while only a small number were misclassified into the other categories (7, 10, and 1). Similarly, in the "unacc" row, 347 out of 358 cars were correctly classified as 'unacc', with only 11 misclassifications into 'acc'. This high accuracy in the 'unacc' category demonstrates the model's ability to identify unacceptable cars with great precision.

For the "good" and "vgood" rows, the results are also promising. Most of the 'vgood' cars were perfectly classified into their correct category, with no misclassifications. The 'good' class had a small number of misclassifications, with 13 being predicted as 'acc', but overall, the errors remain minimal.

The high concentration of values along the diagonal of the matrix indicates that the Logistic Regression model is able to correctly classify the majority of the data points. This is further reflected in its high accuracy score and strong performance across all evaluation metrics.

The success of Logistic Regression can be explained by its ability to leverage the labeled data during training. Unlike K-Means, which relies on distance-based clustering, Logistic Regression uses mathematical functions to directly map the input features to the correct target labels. This makes it highly effective for supervised classification tasks, particularly when working with structured data like the Car Evaluation dataset.

In [85]:
```
print(f"Logistic Regression (Supervised) Accuracy: {logistic_accuracy}")
print("Classification Report (Logistic Regression):")
print(pd.DataFrame(logistic_report).transpose())
```

```
Logistic Regression (Supervised) Accuracy: 0.928709055876686
Classification Report (Logistic Regression):
              precision    recall  f1-score     support
0              0.862069  0.847458  0.854701  118.000000
1              0.650000  0.684211  0.666667   19.000000
2              0.971989  0.969274  0.970629  358.000000
3              0.846154  0.916667  0.880000   24.000000
accuracy       0.928709  0.928709  0.928709    0.928709
macro avg      0.832553  0.854402  0.842999  519.000000
weighted avg   0.929391  0.928709  0.928953  519.000000
```

Final Comments

The Car Evaluation dataset was analyzed using both unsupervised and supervised learning methods. The unsupervised learning model, K-Means Clustering, produced an Adjusted Rand Index (ARI) of **0.0007**. These results indicate that the clusters formed by K-Means do not align well with the true class labels, and the clustering quality is poor. This is likely due to the categorical nature of the dataset and the limitations of using Euclidean distance in K-Means after one-hot encoding. The dataset does not exhibit natural clusters, making K-Means unsuitable for this task.

In contrast, the supervised learning model, Logistic Regression, achieved an overall accuracy of **92.87%**, demonstrating strong performance. The model effectively classified the majority classes (good and unacc), achieving high precision and recall for these categories. The minority classes (acc and vgood) were more challenging to classify, with lower precision and recall due to their smaller sample sizes and potential feature overlap. For example, the acc class achieved a precision of **65%** and a recall of **68.4%**, reflecting some misclassification issues. Despite this, the model maintained a balanced performance across all classes, as evidenced by a macro F1-score of **84.3%** and a weighted F1-score of **92.9%.**

Overall, Logistic Regression significantly outperformed K-Means Clustering for this dataset, demonstrating that the structured and categorical nature of the data is better suited to supervised learning methods. Addressing class imbalance or exploring clustering algorithms tailored for categorical data, such as K-Prototypes, could further improve performance.

Abalone Classification

In [1]:
```
import numpy as np
import seaborn as sns
from ucimlrepo import fetch_ucirepo
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import confusion_matrix, classification_report
import matplotlib.pyplot as plt
```

In [2]:
```
abalone = fetch_ucirepo(id=1)
X = abalone.data.features
y = abalone.data.targets
```

In [3]:
```
print(X)
```

```
     Sex  Length  Diameter  Height  Whole_weight  Shucked_weight  \
0      M   0.455     0.365   0.095        0.5140          0.2245
1      M   0.350     0.265   0.090        0.2255          0.0995
2      F   0.530     0.420   0.135        0.6770          0.2565
3      M   0.440     0.365   0.125        0.5160          0.2155
4      I   0.330     0.255   0.080        0.2050          0.0895
...   ..     ...       ...     ...           ...             ...
4172   F   0.565     0.450   0.165        0.8870          0.3700
4173   M   0.590     0.440   0.135        0.9660          0.4390
4174   M   0.600     0.475   0.205        1.1760          0.5255
4175   F   0.625     0.485   0.150        1.0945          0.5310
4176   M   0.710     0.555   0.195        1.9485          0.9455

      Viscera_weight  Shell_weight
0             0.1010        0.1500
1             0.0485        0.0700
2             0.1415        0.2100
3             0.1140        0.1550
4             0.0395        0.0550
...              ...           ...
4172          0.2390        0.2490
4173          0.2145        0.2605
4174          0.2875        0.3080
4175          0.2610        0.2960
4176          0.3765        0.4950

[4177 rows x 8 columns]
```

In [4]:
```
print(y)
```

```
      Rings
0        15
1         7
2         9
3        10
4         7
...     ...
4172     11
4173     10
4174      9
4175     10
4176     12

[4177 rows x 1 columns]
```

Pre-processing of data

In [5]:
```
label_encoder = LabelEncoder()
if 'Sex' in X.columns:  # Ensure column name is correct
    X['Sex'] = label_encoder.fit_transform(X['Sex'])
```

```
C:\Users\randa\AppData\Local\Temp\ipykernel_19276\3566664418.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  X['Sex'] = label_encoder.fit_transform(X['Sex'])
```

In [6]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [7]:
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_test_scaled = scaler.transform(X_test)

y_train = y_train.values.flatten()
y_test = y_test.values.flatten()
```

Unsupervised K-Means Clustering Model

In [8]:
```
kmeans = KMeans()
kmeans_labels_train = kmeans.fit_predict(X_train_scaled)
```

Mapping of label to clusters (Using most common label in cluster)

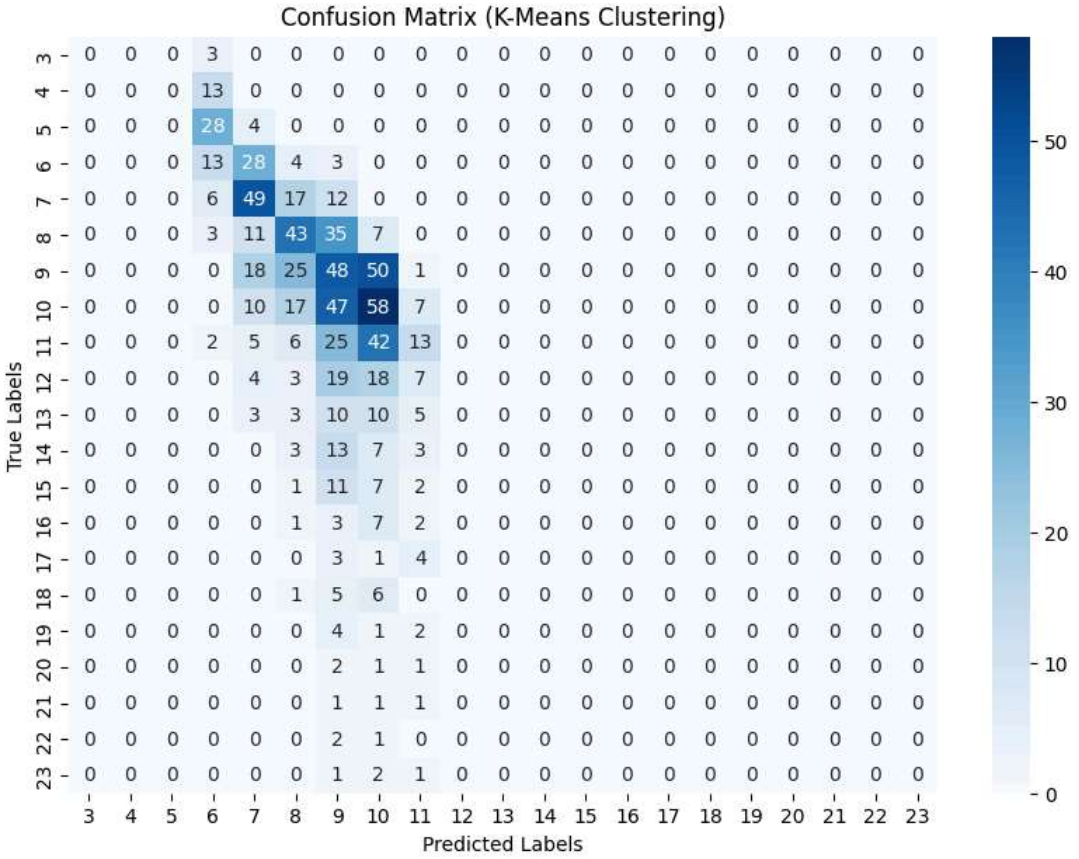
In [9]:
```
cluster_to_label = {}
for cluster in np.unique(kmeans_labels_train):
    cluster_indices = np.where(kmeans_labels_train == cluster)[0]  # Find the indices of the current cluster
    most_common_label = np.bincount(y_train[cluster_indices]).argmax()  # Get the most common label
    cluster_to_label[cluster] = most_common_label # Label each cluster

kmeans_labels_train_mapped = np.array([cluster_to_label[label] for label in kmeans_labels_train])
```

In [10]:
```
kmeans_labels_test = kmeans.predict(X_test_scaled)
kmeans_labels_test_mapped = np.array([cluster_to_label[label] for label in kmeans_labels_test])
```

In [11]:
```
correct_predictions = np.sum(kmeans_labels_test_mapped == y_test)
total_predictions = len(y_test)

model_accuracy = correct_predictions / total_predictions
print(f"Accuracy: {model_accuracy:.4f}")
```

```
Accuracy: 0.2679
```

In [12]:
```
confusion_matrix1 = confusion_matrix(y_test, kmeans_labels_test_mapped)
plt.figure(figsize=(10, 7))
sns.heatmap(confusion_matrix1, annot=True, fmt='d', cmap='Blues', xticklabels=np.unique(y_test), yticklabels=np.unique(y_t
plt.title("Confusion Matrix (K-Means Clustering)")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```



In [13]:
```
report = classification_report(y_test, kmeans_labels_test_mapped, target_names=[f"Class {label}" for label in np.unique(y_
print("Classification Report (K-Means Clustering):")
print(report)
```

```
Classification Report (K-Means Clustering):
              precision    recall  f1-score   support

     Class 3       0.00      0.00      0.00         3
     Class 4       0.00      0.00      0.00        13
     Class 5       0.00      0.00      0.00        32
     Class 6       0.19      0.27      0.22        48
     Class 7       0.37      0.58      0.45        84
     Class 8       0.35      0.43      0.39        99
     Class 9       0.20      0.34      0.25       142
    Class 10       0.26      0.42      0.32       139
    Class 11       0.27      0.14      0.18        93
    Class 12       0.00      0.00      0.00        51
    Class 13       0.00      0.00      0.00        31
    Class 14       0.00      0.00      0.00        26
    Class 15       0.00      0.00      0.00        21
    Class 16       0.00      0.00      0.00        13
    Class 17       0.00      0.00      0.00         8
    Class 18       0.00      0.00      0.00        12
    Class 19       0.00      0.00      0.00         7
    Class 20       0.00      0.00      0.00         4
    Class 21       0.00      0.00      0.00         3
    Class 22       0.00      0.00      0.00         3
    Class 23       0.00      0.00      0.00         4

    accuracy                           0.27       836
   macro avg       0.08      0.10      0.09       836
weighted avg       0.20      0.27      0.22       836
```

```
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Supervised Logistic Regression Classification Model

In [14]:
```
log_reg_model = LogisticRegression()
log_reg_model.fit(X_train_scaled, y_train)
```

```
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\linear_model\_logistic.py:465: Convergence
Warning: 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(
```

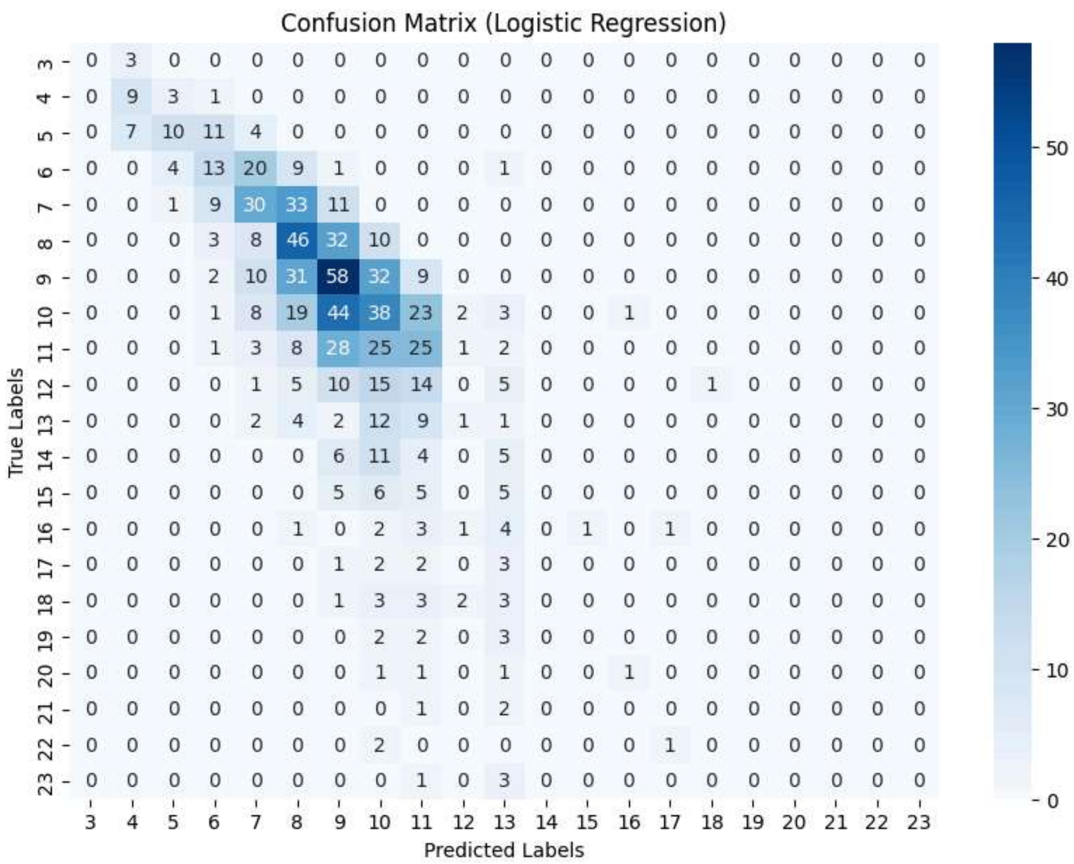
Out[14]:
```
▾ LogisticRegression
LogisticRegression()
```

In [15]:
```
y_pred = log_reg_model.predict(X_test_scaled)
```

In [16]:
```
correct_predictions = np.sum(y_pred == y_test)
total_predictions = len(y_test)

model_accuracy = correct_predictions / total_predictions
print(f"Logistic Regression Accuracy: {model_accuracy:.4f}")
```

```
Logistic Regression Accuracy: 0.2751
```

In [17]:
```
confusion_matrix2 = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(confusion_matrix2, annot=True, fmt='d', cmap='Blues', xticklabels=np.unique(y_test), yticklabels=np.unique(y_t
plt.title("Confusion Matrix (Logistic Regression)")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```



```
In [18]: report = classification_report(y_test, kmeans_labels_test_mapped, target_names=[f"Class {label}" for label in np.unique(y_
         print("Classification Report (Logistic Regresssion):")
         print(report)
```

```
Classification Report (Logistic Regresssion):
              precision    recall  f1-score   support

     Class 3       0.00      0.00      0.00         3
     Class 4       0.00      0.00      0.00        13
     Class 5       0.00      0.00      0.00        32
     Class 6       0.19      0.27      0.22        48
     Class 7       0.37      0.58      0.45        84
     Class 8       0.35      0.43      0.39        99
     Class 9       0.20      0.34      0.25       142
    Class 10       0.26      0.42      0.32       139
    Class 11       0.27      0.14      0.18        93
    Class 12       0.00      0.00      0.00        51
    Class 13       0.00      0.00      0.00        31
    Class 14       0.00      0.00      0.00        26
    Class 15       0.00      0.00      0.00        21
    Class 16       0.00      0.00      0.00        13
    Class 17       0.00      0.00      0.00         8
    Class 18       0.00      0.00      0.00        12
    Class 19       0.00      0.00      0.00         7
    Class 20       0.00      0.00      0.00         4
    Class 21       0.00      0.00      0.00         3
    Class 22       0.00      0.00      0.00         3
    Class 23       0.00      0.00      0.00         4

    accuracy                           0.27       836
   macro avg       0.08      0.10      0.09       836
weighted avg       0.20      0.27      0.22       836
```

```
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

Wine Quality Classification

In [1]:
```
import numpy as np
import seaborn as sns
from ucimlrepo import fetch_ucirepo
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
```

In [2]:
```
wine_quality = fetch_ucirepo(id=186)
X = wine_quality.data.features
y = wine_quality.data.targets
```

In [3]:
```
print(X)
```

```
      fixed_acidity  volatile_acidity  citric_acid  residual_sugar  chlorides  \
0               7.4              0.70         0.00             1.9      0.076
1               7.8              0.88         0.00             2.6      0.098
2               7.8              0.76         0.04             2.3      0.092
3              11.2              0.28         0.56             1.9      0.075
4               7.4              0.70         0.00             1.9      0.076
...             ...               ...          ...             ...        ...
6492            6.2              0.21         0.29             1.6      0.039
6493            6.6              0.32         0.36             8.0      0.047
6494            6.5              0.24         0.19             1.2      0.041
6495            5.5              0.29         0.30             1.1      0.022
6496            6.0              0.21         0.38             0.8      0.020

      free_sulfur_dioxide  total_sulfur_dioxide  density    pH  sulphates  \
0                    11.0                  34.0  0.99780  3.51       0.56
1                    25.0                  67.0  0.99680  3.20       0.68
2                    15.0                  54.0  0.99700  3.26       0.65
3                    17.0                  60.0  0.99800  3.16       0.58
4                    11.0                  34.0  0.99780  3.51       0.56
...                   ...                   ...      ...   ...        ...
6492                 24.0                  92.0  0.99114  3.27       0.50
6493                 57.0                 168.0  0.99490  3.15       0.46
6494                 30.0                 111.0  0.99254  2.99       0.46
6495                 20.0                 110.0  0.98869  3.34       0.38
6496                 22.0                  98.0  0.98941  3.26       0.32

      alcohol
0         9.4
1         9.8
2         9.8
3         9.8
4         9.4
...       ...
6492     11.2
6493      9.6
6494      9.4
6495     12.8
6496     11.8

[6497 rows x 11 columns]
```

In [4]:
```
print(y)
```

```
      quality
0           5
1           5
2           5
3           6
4           5
...       ...
6492        6
6493        5
6494        6
6495        7
6496        6

[6497 rows x 1 columns]
```

Pre-processing of data

In [5]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

```
In [6]: scaler = StandardScaler()
        X_train_scaled = scaler.fit_transform(X_train)
        X_test_scaled = scaler.transform(X_test)

        y_train = y_train.values.flatten()
        y_test = y_test.values.flatten()
```

Unsupervised K-Means Clustering Model

```
In [7]: kmeans = KMeans()
        kmeans_labels_train = kmeans.fit_predict(X_train_scaled)
```

```
In [8]: cluster_to_label = {}
        for cluster in np.unique(kmeans_labels_train):
            cluster_indices = np.where(kmeans_labels_train == cluster)[0]  # Find the indices of the current cluster
            most_common_label = np.bincount(y_train[cluster_indices]).argmax()  # Get the most common label
            cluster_to_label[cluster] = most_common_label # Label each cluster

        kmeans_labels_train_mapped = np.array([cluster_to_label[label] for label in kmeans_labels_train])
```

Unsupervised K-Means Clustering Model

```
In [9]: kmeans_labels_test = kmeans.predict(X_test_scaled)
        kmeans_labels_test_mapped = np.array([cluster_to_label[label] for label in kmeans_labels_test])
```

```
In [10]: correct_predictions = np.sum(kmeans_labels_test_mapped == y_test)
         total_predictions = len(y_test)

         model_accuracy = correct_predictions / total_predictions
         print(f"Accuracy: {model_accuracy:.4f}")
```

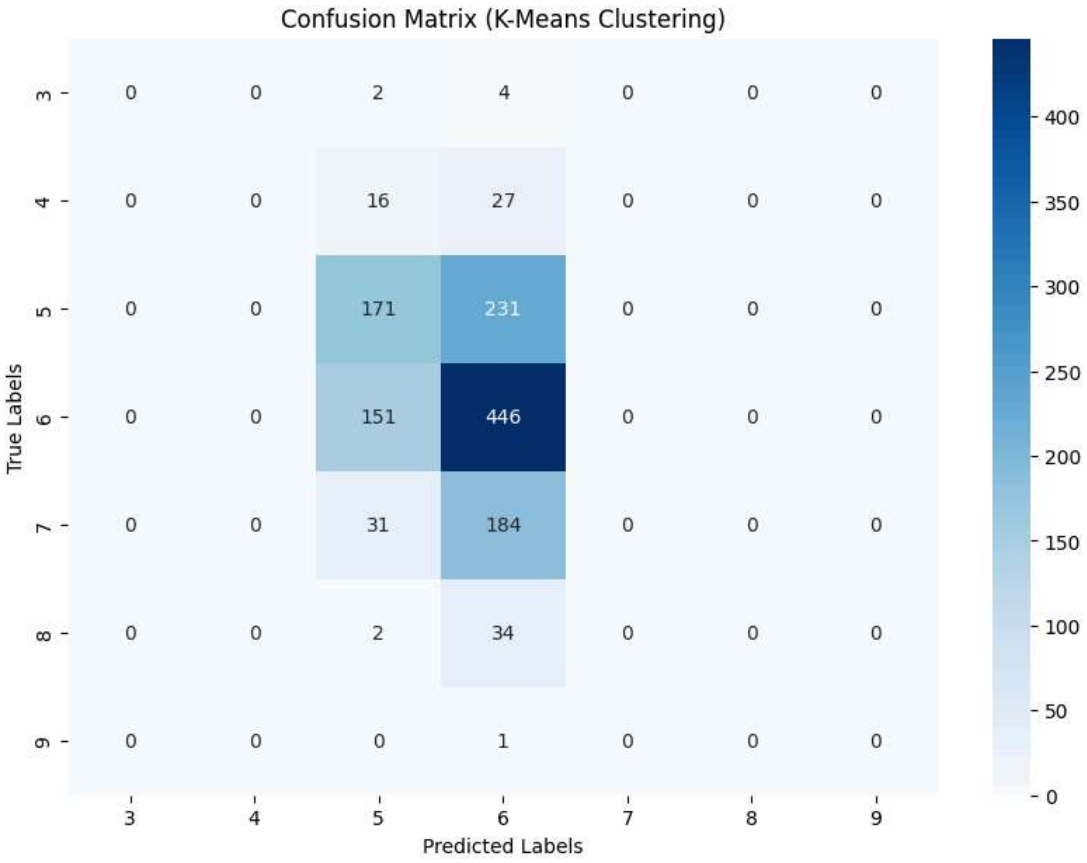
```
Accuracy: 0.4746
```

```
In [11]: report = classification_report(y_test, kmeans_labels_test_mapped, target_names=[str(i) for i in np.unique(y_test)])
         print("Classification Report (K-Means Clustering):\n", report)
```

```
Classification Report (K-Means Clustering):
               precision    recall  f1-score   support

           3       0.00      0.00      0.00         6
           4       0.00      0.00      0.00        43
           5       0.46      0.43      0.44       402
           6       0.48      0.75      0.59       597
           7       0.00      0.00      0.00       215
           8       0.00      0.00      0.00        36
           9       0.00      0.00      0.00         1

    accuracy                           0.47      1300
   macro avg       0.13      0.17      0.15      1300
weighted avg       0.36      0.47      0.41      1300
```

```
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

```
In [12]: confusion_matrix1 = confusion_matrix(y_test, kmeans_labels_test_mapped)
         plt.figure(figsize=(10, 7))
         sns.heatmap(confusion_matrix1, annot=True, fmt='d', cmap='Blues', xticklabels=np.unique(y_test), yticklabels=np.unique(y_t
         plt.title("Confusion Matrix (K-Means Clustering)")
         plt.xlabel("Predicted Labels")
         plt.ylabel("True Labels")
         plt.show()
```



Supervised Logistic Regression Classification Model

In [13]:
```
log_reg_model = LogisticRegression()
log_reg_model.fit(X_train_scaled, y_train)
```

Out[13]:

LogisticRegression

```
LogisticRegression()
```

In [14]:
```
y_pred = log_reg_model.predict(X_test_scaled)
```

In [15]:
```
correct_predictions = np.sum(y_pred == y_test)
total_predictions = len(y_test)

model_accuracy = correct_predictions / total_predictions
print(f"Logistic Regression Accuracy: {model_accuracy:.4f}")
```

```
Logistic Regression Accuracy: 0.5362
```

Results

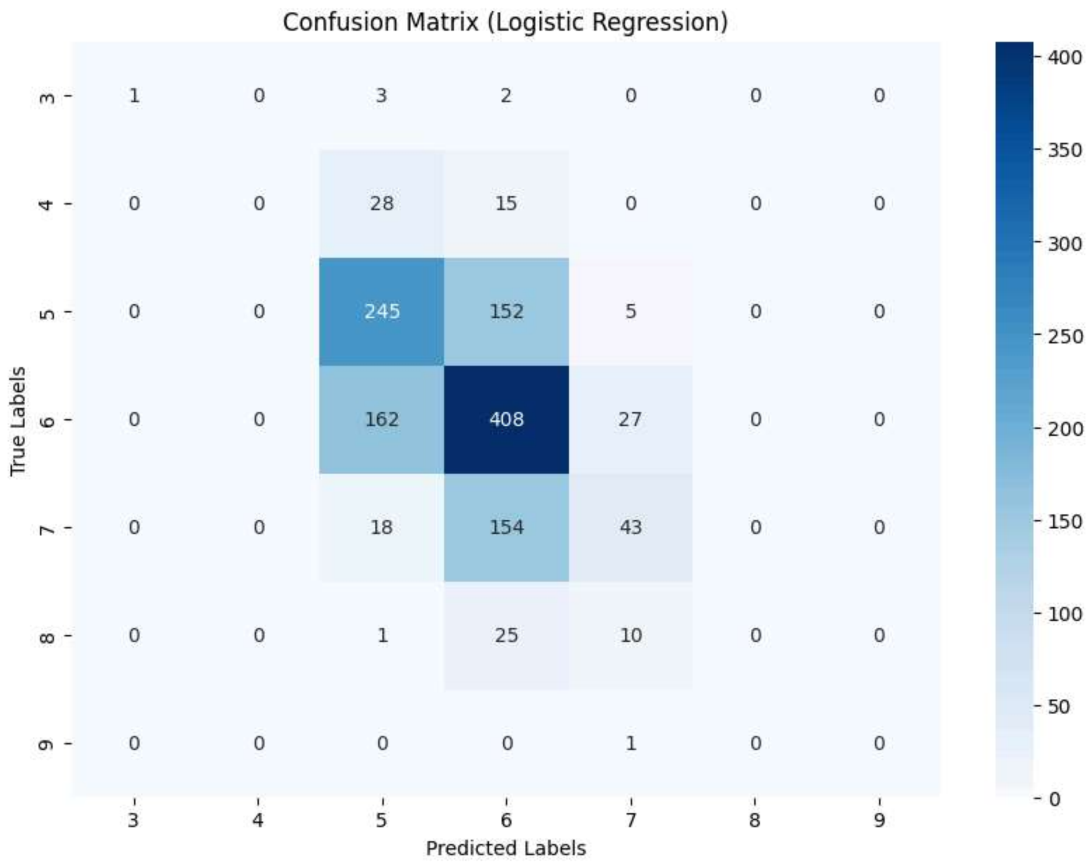
In [16]:
```
report = classification_report(y_test, y_pred, target_names=[str(i) for i in np.unique(y_test)])
print("Classification Report (Logistic Regression):\n", report)
```

```
Classification Report (Logistic Regression):
              precision    recall  f1-score   support

           3       1.00      0.17      0.29         6
           4       0.00      0.00      0.00        43
           5       0.54      0.61      0.57       402
           6       0.54      0.68      0.60       597
           7       0.50      0.20      0.29       215
           8       0.00      0.00      0.00        36
           9       0.00      0.00      0.00         1

    accuracy                           0.54      1300
   macro avg       0.37      0.24      0.25      1300
weighted avg       0.50      0.54      0.50      1300
```

```
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
c:\Users\randa\AppData\Local\Programs\Python\Python313\Lib\site-packages\sklearn\metrics\_classification.py:1565: Undefined
MetricWarning: Precision is ill-defined and being set to 0.0 in labels with no predicted samples. Use `zero_division` param
eter to control this behavior.
  _warn_prf(average, modifier, f"{metric.capitalize()} is", len(result))
```

In [17]:

```
confusion_matrix2 = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(10, 7))
sns.heatmap(confusion_matrix2, annot=True, fmt='d', cmap='Blues', xticklabels=np.unique(y_test), yticklabels=np.unique(y_t
plt.title("Confusion Matrix (Logistic Regression)")
plt.xlabel("Predicted Labels")
plt.ylabel("True Labels")
plt.show()
```



Banks Classification

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ucimlrepo import fetch_ucirepo
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

random_state = np.random.seed(12)
pd.set_option('display.max_columns', None)
```

1 - Obtaining the Initial Dataset

In [2]:
```
bank_marketing = fetch_ucirepo(id=222)
```

In [3]:
```
initial_features = bank_marketing.data.features
initial_targets = bank_marketing.data.targets

processed_features = initial_features.copy()
processed_targets = initial_targets.copy()
```

In [4]:
```
initial_features
```

Out[4]:

	age	job	marital	education	default	balance	housing	loan	contact	day_of_week	month	duration	campaign
0	58	management	married	tertiary	no	2143	yes	no	NaN	5	may	261	1
1	44	technician	single	secondary	no	29	yes	no	NaN	5	may	151	1
2	33	entrepreneur	married	secondary	no	2	yes	yes	NaN	5	may	76	1
3	47	blue-collar	married	NaN	no	1506	yes	no	NaN	5	may	92	1
4	33	NaN	single	NaN	no	1	no	no	NaN	5	may	198	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
45206	51	technician	married	tertiary	no	825	no	no	cellular	17	nov	977	3
45207	71	retired	divorced	primary	no	1729	no	no	cellular	17	nov	456	2
45208	72	retired	married	secondary	no	5715	no	no	cellular	17	nov	1127	5
45209	57	blue-collar	married	secondary	no	668	no	no	telephone	17	nov	508	4
45210	37	entrepreneur	married	secondary	no	2971	no	no	cellular	17	nov	361	2

45211 rows × 16 columns

In [5]:
```
initial_targets
```

Out[5]:

	y
0	no
1	no
2	no
3	no
4	no
...	...
45206	yes
45207	yes
45208	yes
45209	no
45210	no

45211 rows × 1 columns

2 - Processing the Dataset

2.1 - Processing Ordinal Categorical Feature Values

In [6]:
```
def ordinal(initial, key, values, is_object=False, default=0):
    data = initial.copy()
    data[key] = initial[key].apply(lambda value: values.index(value) if not is_object else values.get(value) if values.get
    return data
```

2.1.1 - Education

In [7]:
```
print(f"Initial Value Counts\n{processed_features['education'].value_counts()}\n")
processed_features = ordinal(processed_features, 'education', {"primary": 1, "secondary": 2, "tertiary": 3}, is_object=Tru
print(f"Final Value Counts\n{processed_features['education'].value_counts()}")
```

```
Initial Value Counts
education
secondary    23202
tertiary     13301
primary       6851
Name: count, dtype: int64

Final Value Counts
education
2    23202
3    13301
1     6851
0     1857
Name: count, dtype: int64
```

2.1.2 - Default

In [8]:
```
print(f"Initial Value Counts\n{processed_features['default'].value_counts()}\n")
processed_features = ordinal(processed_features, 'default', ['no', 'yes'])
print(f"Final Value Counts\n{processed_features['default'].value_counts()}")
```

```
Initial Value Counts
default
no     44396
yes      815
Name: count, dtype: int64

Final Value Counts
default
0    44396
1      815
Name: count, dtype: int64
```

2.1.3 - Housing

In [9]:
```
print(f"Initial Value Counts\n{processed_features['housing'].value_counts()}\n")
processed_features = ordinal(processed_features, 'housing', ['no', 'yes'])
print(f"Final Value Counts\n{processed_features['housing'].value_counts()}")
```

```
Initial Value Counts
housing
yes    25130
no     20081
Name: count, dtype: int64

Final Value Counts
housing
1    25130
0    20081
Name: count, dtype: int64
```

2.1.4 - Loan

In [10]:
```
print(f"Initial Value Counts\n{processed_features['loan'].value_counts()}\n")
processed_features = ordinal(processed_features, 'loan', ['no', 'yes'])
print(f"Final Value Counts\n{processed_features['loan'].value_counts()}")
```

```
Initial Value Counts
loan
no     37967
yes     7244
Name: count, dtype: int64

Final Value Counts
loan
0    37967
1     7244
Name: count, dtype: int64
```

2.1.5 - Month

In [11]:
```
print(f"Initial Value Counts\n{processed_features['month'].value_counts()}\n")
processed_features = ordinal(processed_features, 'month', ["jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep",
print(f"Final Value Counts\n{processed_features['month'].value_counts()}")
```

```
Initial Value Counts
month
may    13766
jul     6895
aug     6247
jun     5341
nov     3970
apr     2932
feb     2649
jan     1403
oct      738
sep      579
mar      477
dec      214
Name: count, dtype: int64

Final Value Counts
month
4     13766
6      6895
7      6247
5      5341
10     3970
3      2932
1      2649
0      1403
9       738
8       579
2       477
11      214
Name: count, dtype: int64
```

2.1.6 - POutcome

In [12]:
```
print(f"Initial Value Counts\n{processed_features['poutcome'].value_counts()}\n")
processed_features = ordinal(processed_features, 'poutcome', {"other": -1, "failure": 0, "success": 1}, is_object=True, de
print(f"Final Value Counts\n{processed_features['poutcome'].value_counts()}")
```

```
Initial Value Counts
poutcome
failure    4901
other      1840
success    1511
Name: count, dtype: int64

Final Value Counts
poutcome
-2    36959
 0     4901
-1     1840
 1     1511
Name: count, dtype: int64
```

2.2 - Processing Non-Ordinal Categorical Feature Values

In [13]:
```
def non_ordinal(initial, key, values):
    data = initial.copy()
    for value in values:
        data[f"{value}-{key}"] = (initial[key] == value).astype(int)
    data.drop([key], axis=1, inplace=True)
    return data
```

2.2.1 - Job

In [14]:
```
processed_features = non_ordinal(processed_features, 'job', ["admin.", "unemployed", "management", "housemaid", "entrepren
```

2.2.2 - Marital

In [15]:
```
processed_features = non_ordinal(processed_features, 'marital', ["married", "divorced", "single"])
```

2.2.3 - Contact

In [16]:
```
processed_features = non_ordinal(processed_features, 'contact', ["telephone", "cellular"])
```

2.3 - Validating the Processed Feature Values

In [17]:
```
print("Shape =", initial_features.shape)
initial_features
```

```
Shape = (45211, 16)
```

Out[17]:

	age	job	marital	education	default	balance	housing	loan	contact	day_of_week	month	duration	campaign
0	58	management	married	tertiary	no	2143	yes	no	NaN	5	may	261	1
1	44	technician	single	secondary	no	29	yes	no	NaN	5	may	151	1
2	33	entrepreneur	married	secondary	no	2	yes	yes	NaN	5	may	76	1
3	47	blue-collar	married	NaN	no	1506	yes	no	NaN	5	may	92	1
4	33	NaN	single	NaN	no	1	no	no	NaN	5	may	198	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
45206	51	technician	married	tertiary	no	825	no	no	cellular	17	nov	977	3
45207	71	retired	divorced	primary	no	1729	no	no	cellular	17	nov	456	2
45208	72	retired	married	secondary	no	5715	no	no	cellular	17	nov	1127	5
45209	57	blue-collar	married	secondary	no	668	no	no	telephone	17	nov	508	4
45210	37	entrepreneur	married	secondary	no	2971	no	no	cellular	17	nov	361	2

45211 rows × 16 columns

In [18]:
```
print("Shape =", processed_features.shape)
processed_features
```

```
Shape = (45211, 29)
```

Out[18]:

	age	education	default	balance	housing	loan	day_of_week	month	duration	campaign	pdays	previous	poutcome	adr
0	58	3	0	2143	1	0	5	4	261	1	-1	0	-2	
1	44	2	0	29	1	0	5	4	151	1	-1	0	-2	
2	33	2	0	2	1	1	5	4	76	1	-1	0	-2	
3	47	0	0	1506	1	0	5	4	92	1	-1	0	-2	
4	33	0	0	1	0	0	5	4	198	1	-1	0	-2	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
45206	51	3	0	825	0	0	17	10	977	3	-1	0	-2	
45207	71	1	0	1729	0	0	17	10	456	2	-1	0	-2	
45208	72	2	0	5715	0	0	17	10	1127	5	184	3	1	
45209	57	2	0	668	0	0	17	10	508	4	-1	0	-2	
45210	37	2	0	2971	0	0	17	10	361	2	188	11	-1	

45211 rows × 29 columns

2.4 - Processing Binary Target Values

In [19]:
```
print(f"Initial Value Counts\n{processed_targets['y'].value_counts()}\n")
processed_targets = ordinal(processed_targets, 'y', ['no', 'yes'])
print(f"Final Value Counts\n{processed_targets['y'].value_counts()}")
```

```
Initial Value Counts
y
no     39922
yes     5289
Name: count, dtype: int64

Final Value Counts
y
0    39922
1     5289
Name: count, dtype: int64
```

2.5 - Validating the Processed Target Values

In [20]:
```
print("Shape =", initial_targets.shape)
initial_targets
```

```
Shape = (45211, 1)
```

Out[20]:

	y
0	no
1	no
2	no
3	no
4	no
...	...
45206	yes
45207	yes
45208	yes
45209	no
45210	no

45211 rows × 1 columns

In [21]:
```
print("Shape =", processed_targets.shape)
processed_targets
```

```
Shape = (45211, 1)
```

Out[21]:

	y
0	0
1	0
2	0
3	0
4	0
...	...
45206	1
45207	1
45208	1
45209	0
45210	0

45211 rows × 1 columns

3 - Splitting the Dataset

In [22]:
```
split_proportion = 0.7
dataset_size = processed_features.shape[0]
train_size = int(dataset_size * split_proportion)
test_size = dataset_size - train_size
```

In [23]:
```
x_train, y_train = processed_features.iloc[train_size:], processed_targets.iloc[train_size:]['y']
x_test, y_test = processed_features.iloc[:train_size], processed_targets.iloc[:train_size]['y']

print(f"Shape of x_train = {x_train.shape}")
print(f"Shape of y_train = {y_train.shape}")
print(f"Shape of x_test = {x_test.shape}")
print(f"Shape of y_test = {y_test.shape}")
```

```
Shape of x_train = (13564, 29)
Shape of y_train = (13564,)
Shape of x_test = (31647, 29)
Shape of y_test = (31647,)
```

4 - Supervised Model

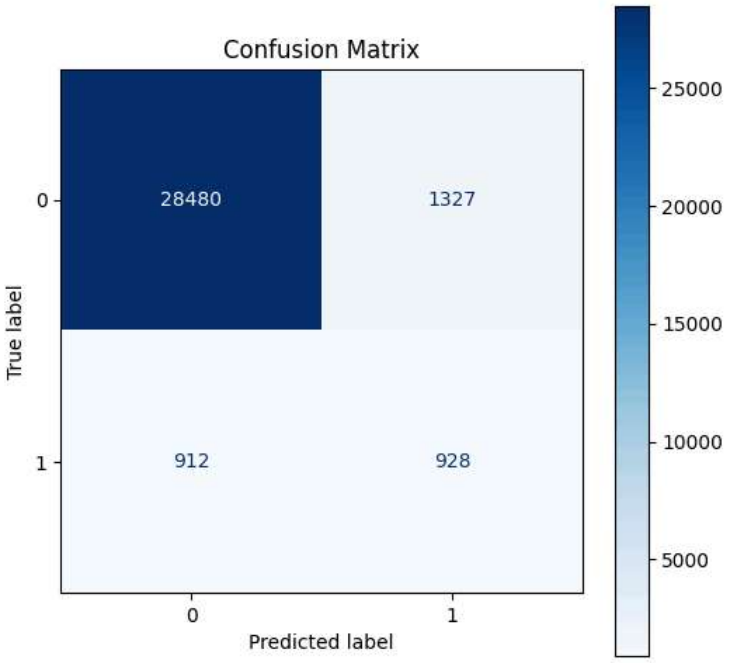
Accuracy = 0.92925

In [24]:
```
def display_results(accuracy, confusion_matrix, classification_report, predictions):
    print(f"Accuracy = {accuracy:.5f}\n")

    fig, ax = plt.subplots(figsize=(6, 6))
    disp = ConfusionMatrixDisplay(confusion_matrix=confusion_matrix, display_labels=np.unique(predictions))
    disp.plot(ax=ax, cmap='Blues', values_format='d')
    plt.title("Confusion Matrix")
    plt.show()

    print(classification_report)
```

In [25]:
```
def train_test_eval(model, x_train_set, y_train_set, x_test_set, y_test_set):
    model.fit(x_train_set, y_train_set)
    y_pred_set = model.predict(x_test_set).astype(int)
    results = (y_pred_set == y_test_set).value_counts()
    accuracy = float(results[True] / y_test_set.shape[0])
    return {
        "predictions": y_pred_set,
        "accuracy": accuracy,
        "confusion_matrix": confusion_matrix(y_test_set, y_pred_set),
        "classification_report": classification_report(y_test_set, y_pred_set)
    }
```

In [26]:
```
logistic_regression_model = LogisticRegression(n_jobs=-1)
logistic_regression_results = train_test_eval(logistic_regression_model, x_train, y_train, x_test, y_test)
display_results(**logistic_regression_results)
```

```
Accuracy = 0.92925
```



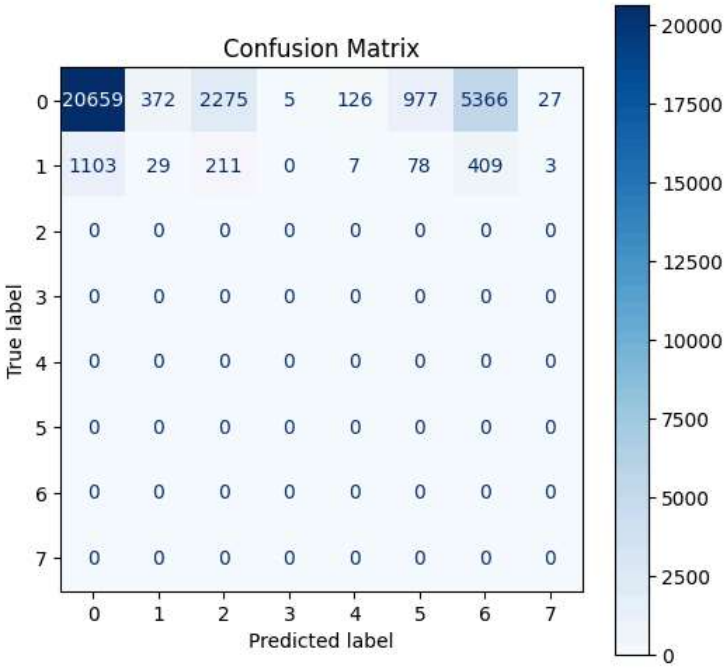

```
              precision    recall  f1-score   support

           0       0.97      0.96      0.96     29807
           1       0.41      0.50      0.45      1840

    accuracy                           0.93     31647
   macro avg       0.69      0.73      0.71     31647
weighted avg       0.94      0.93      0.93     31647
```

5 - Unsupervised Model

Accuracy = 0.65371

In [27]:
```
k_means_model = KMeans()
k_means_results = train_test_eval(k_means_model, x_train, y_train, x_test, y_test)
display_results(**k_means_results)
```

```
Accuracy = 0.65371
```



	precision	recall	f1-score	support
0	0.95	0.69	0.80	29807
1	0.07	0.02	0.03	1840
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	0.00	0.00	0.00	0
5	0.00	0.00	0.00	0
6	0.00	0.00	0.00	0
7	0.00	0.00	0.00	0
accuracy			0.65	31647
macro avg	0.13	0.09	0.10	31647
weighted avg	0.90	0.65	0.76	31647

Mushroom Classification

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from ucimlrepo import fetch_ucirepo
from sklearn.cluster import KMeans
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

random_state = np.random.seed(12)
pd.set_option('display.max_columns', None)
```

1 - Obtaining the Initial Dataset

In [2]:
```
mushrooms_dataset = fetch_ucirepo(id=73)
```

In [3]:
```
initial_features = mushrooms_dataset.data.features
initial_targets = mushrooms_dataset.data.targets

processed_features = initial_features.copy()
processed_targets = initial_targets.copy()
```

In [4]:
```
initial_features
```

Out[4]:

	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	stalk-shape	stalk-root	stalk-surface-above-ring	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil typ
0	x	s	n	t	p	f	c	n	k	e	e	s	s	w	w	
1	x	s	y	t	a	f	c	b	k	e	c	s	s	w	w	
2	b	s	w	t	l	f	c	b	n	e	c	s	s	w	w	
3	x	y	w	t	p	f	c	n	n	e	e	s	s	w	w	
4	x	s	g	f	n	f	w	b	k	t	e	s	s	w	w	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
8119	k	s	n	f	n	a	c	b	y	e	NaN	s	s	o	o	
8120	x	s	n	f	n	a	c	b	y	e	NaN	s	s	o	o	
8121	f	s	n	f	n	a	c	b	n	e	NaN	s	s	o	o	
8122	k	y	n	f	y	f	c	n	b	t	NaN	s	k	w	w	
8123	x	s	n	f	n	a	c	b	y	e	NaN	s	s	o	o	

8124 rows × 22 columns

In [5]:
```
initial_targets
```

Out[5]:

	poisonous
0	p
1	e
2	e
3	p
4	e
...	...
8119	e
8120	e
8121	e
8122	p
8123	e

8124 rows × 1 columns

2 - Processing the Dataset

2.1 - Processing Ordinal Categorical Feature Values

In [6]:
```
def ordinal(initial, key, values):
    data = initial.copy()
    data[key] = initial[key].apply(lambda value: values.index(value))
    return data
```

2.1.1 - Gill-Spacing

- c = close -> 0
- w = crowded -> 1
- d = distant -> 2

In [7]:
```
print(f"Initial Value Counts\n{processed_features['gill-spacing'].value_counts()}\n")
processed_features = ordinal(processed_features, 'gill-spacing', ['c', 'w', 'd'])
print(f"Final Value Counts\n{processed_features['gill-spacing'].value_counts()}")
```

```
Initial Value Counts
gill-spacing
c    6812
w    1312
Name: count, dtype: int64

Final Value Counts
gill-spacing
0    6812
1    1312
Name: count, dtype: int64
```

2.1.2 - Gill-Size

- n = narrow -> 0
- b = broad -> 1

In [8]:
```
print(f"Initial Value Counts\n{processed_features['gill-size'].value_counts()}\n")
processed_features = ordinal(processed_features, 'gill-size', ['n', 'b'])
print(f"Final Value Counts\n{processed_features['gill-size'].value_counts()}")
```

```
Initial Value Counts
gill-size
b    5612
n    2512
Name: count, dtype: int64

Final Value Counts
gill-size
1    5612
0    2512
Name: count, dtype: int64
```

2.1.3 - Ring-Number

- n = none -> 0
- o = one -> 1
- t = two -> 2

In [9]:
```
print(f"Initial Value Counts\n{processed_features['ring-number'].value_counts()}\n")
processed_features = ordinal(processed_features, 'ring-number', ['n', 'o', 't'])
print(f"Final Value Counts\n{processed_features['ring-number'].value_counts()}")
```

```
Initial Value Counts
ring-number
o    7488
t     600
n      36
Name: count, dtype: int64

Final Value Counts
ring-number
1    7488
2     600
0      36
Name: count, dtype: int64
```

2.1.4 - Population

- y = solitary -> 0

- v = several -> 1
- s = scattered -> 2
- n = numerous -> 3
- c = clustered -> 4
- a = abundant -> 5

In [10]:
```
print(f"Initial Value Counts\n{processed_features['population'].value_counts()}\n")
processed_features = ordinal(processed_features, 'population', ['y', 'v', 's', 'n', 'c', 'a'])
print(f"Final Value Counts\n{processed_features['population'].value_counts()}")
```

```
Initial Value Counts
population
v    4040
y    1712
s    1248
n     400
a     384
c     340
Name: count, dtype: int64

Final Value Counts
population
1    4040
0    1712
2    1248
3     400
5     384
4     340
Name: count, dtype: int64
```

2.2 - Processing Non-Ordinal Categorical Feature Values

In [11]:
```
def non_ordinal(initial, key, values):
    data = initial.copy()
    for value in values:
        data[f"{value}-{key}"] = (initial[key] == value).astype(int)
    data.drop([key], axis=1, inplace=True)
    return data
```

2.2.1 - Cap-Shape

- b = bell
- c = conical
- x = convex
- f = flat
- k = knobbed
- s = sunken

In [12]:
```
processed_features = non_ordinal(processed_features, 'cap-shape', ['b', 'c', 'x', 'f', 'k', 's'])
```

2.2.2 - Cap-Surface

- f = fibrous
- g = grooves
- y = scaly
- s = smooth

In [13]:
```
processed_features = non_ordinal(processed_features, 'cap-surface', ['f', 'g', 'y', 's'])
```

2.2.3 - Cap-Colour

- n = brown
- b = buff
- c = cinnamon
- g = gray
- r = green
- p = pink
- u = purple
- e = red
- w = white
- y = yellow

In [14]:
```
processed_features = non_ordinal(processed_features, 'cap-color', ['n', 'b', 'c', 'g', 'r', 'p', 'u', 'e', 'w', 'y'])
```

2.2.4 - Bruises

- t = bruises
- f = no bruises

In [15]:
```
processed_features = non_ordinal(processed_features, 'bruises', ['t', 'f'])
```

2.2.5 - Odour

- a = almond
- l = anise
- c = creosote
- y = fishy
- f = foul
- m = musty
- n = none
- p = pungent
- s = spicy

In [16]:
```
processed_features = non_ordinal(processed_features, 'odor', ['a', 'l', 'c', 'y', 'f', 'm', 'n', 'p', 's'])
```

2.2.6 - Gill-Attachment

- a = attached
- d = descending
- f = free
- n = notched

In [17]:
```
processed_features = non_ordinal(processed_features, 'gill-attachment', ['a', 'd', 'f', 'n'])
```

2.2.7 - Gill-Colour

- k = black
- n = brown
- b = buff
- h = chocolate
- g = gray
- r = green
- o = orange
- p = pink
- u = purple
- e = red
- w = white
- y = yellow

In [18]:
```
processed_features = non_ordinal(processed_features, 'gill-color', ['k', 'n', 'b', 'h', 'g', 'r', 'o', 'p', 'u', 'e', 'w',
```

2.2.8 - Stalk-Shape

- e = enlarging
- t = tapering

In [19]:
```
processed_features = non_ordinal(processed_features, 'stalk-shape', ['e', 't'])
```

2.2.9 - Stalk-Root

- b = bulbous
- c = club
- u = cup
- e = equal
- z = rhizomorphs
- r = rooted

In [20]:
```
processed_features = non_ordinal(processed_features, 'stalk-root', ['b', 'c', 'u', 'e', 'z', 'r'])
```

2.2.10 - Stalk-Surface-Above-Ring

- f = fibrous
- y = scaly

- k = silky
- s = smooth

In [21]:
```
processed_features = non_ordinal(processed_features, 'stalk-surface-above-ring', ['f', 'y', 'k', 's'])
```

2.2.11 - Stalk-Surface-Below-Ring

- f = fibrous
- y = scaly
- k = silky
- s = smooth

In [22]:
```
processed_features = non_ordinal(processed_features, 'stalk-surface-below-ring', ['f', 'y', 'k', 's'])
```

2.2.12 - Stalk-Colour-Above-Ring

- n = brown
- b = buff
- c = cinnamon
- g = gray
- o = orange
- p = pink
- e = red
- w = white
- y = yellow

In [23]:
```
processed_features = non_ordinal(processed_features, 'stalk-color-above-ring', ['n', 'b', 'c', 'g', 'o', 'p', 'e', 'w', 'y
```

2.2.13 - Stalk-Colour-Below-Ring

- n = brown
- b = buff
- c = cinnamon
- g = gray
- o = orange
- p = pink
- e = red
- w = white
- y = yellow

In [24]:
```
processed_features = non_ordinal(processed_features, 'stalk-color-below-ring', ['n', 'b', 'c', 'g', 'o', 'p', 'e', 'w', 'y
```

2.2.14 - Veil-Type

- p = partial
- u = universal

In [25]:
```
processed_features = non_ordinal(processed_features, 'veil-type', ['p', 'u'])
```

2.2.15 - Veil-Colour

- n = brown
- o = orange
- w = white
- y = yellow

In [26]:
```
processed_features = non_ordinal(processed_features, 'veil-color', ['n', 'o', 'w', 'y'])
```

2.2.16 - Ring-Type

- c = cobwebby
- e = evanescent
- f = flaring
- l = large
- n = none
- p = pendant
- s = sheathing
- z = zone

In [27]:
```
processed_features = non_ordinal(processed_features, 'ring-type', ['c', 'e', 'f', 'l', 'n', 'p', 's', 'z'])
```

2.2.17 - Spore-Print-Colour

- k = black
- n = brown
- b = buff
- h = chocolate
- r = green
- o = orange
- u = purple
- w = white
- y = yellow

In [28]:
```
processed_features = non_ordinal(processed_features, 'spore-print-color', ['k', 'n', 'b', 'h', 'r', 'o', 'u', 'w', 'y'])
```

2.2.18 - Habitat

- g = grasses
- l = leaves
- m = meadows
- p = paths
- u = urban
- w = waste
- d = woods

In [29]:
```
processed_features = non_ordinal(processed_features, 'habitat', ['g', 'l', 'm', 'p', 'u', 'w', 'd'])
```

2.3 - Validating the Processed Feature Values

In [30]:
```
print("Shape =", initial_features.shape)
initial_features
```

```
Shape = (8124, 22)
```

Out[30]:

	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	stalk-shape	stalk-root	stalk-surface-above-ring	stalk-surface-below-ring	stalk-color-above-ring	stalk-color-below-ring	veil typ
0	x	s	n	t	p	f	c	n	k	e	e	s	s	w	w	p
1	x	s	y	t	a	f	c	b	k	e	c	s	s	w	w	p
2	b	s	w	t	l	f	c	b	n	e	c	s	s	w	w	p
3	x	y	w	t	p	f	c	n	n	e	e	s	s	w	w	p
4	x	s	g	f	n	f	w	b	k	t	e	s	s	w	w	p
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
8119	k	s	n	f	n	a	c	b	y	e	NaN	s	s	o	o	p
8120	x	s	n	f	n	a	c	b	y	e	NaN	s	s	o	o	p
8121	f	s	n	f	n	a	c	b	n	e	NaN	s	s	o	o	p
8122	k	y	n	f	y	f	c	n	b	t	NaN	s	k	w	w	p
8123	x	s	n	f	n	a	c	b	y	e	NaN	s	s	o	o	p

8124 rows × 22 columns

In [31]:
```
print("Shape =", processed_features.shape)
processed_features
```

```
Shape = (8124, 115)
```

Out[31]:

	gill-spacing	gill-size	ring-number	population	b-cap-shape	c-cap-shape	x-cap-shape	f-cap-shape	k-cap-shape	s-cap-shape	f-cap-surface	g-cap-surface	y-cap-surface	s-cap-surface	n-cap-color	b cap colo
0	0	0	1	2	0	0	1	0	0	0	0	0	0	1	1	
1	0	1	1	3	0	0	1	0	0	0	0	0	0	1	0	
2	0	1	1	3	1	0	0	0	0	0	0	0	0	1	0	
3	0	0	1	2	0	0	1	0	0	0	0	0	1	0	0	
4	1	1	1	5	0	0	1	0	0	0	0	0	0	1	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
8119	0	1	1	4	0	0	0	0	1	0	0	0	0	1	1	
8120	0	1	1	1	0	0	1	0	0	0	0	0	0	1	1	
8121	0	1	1	4	0	0	0	1	0	0	0	0	0	1	1	
8122	0	0	1	1	0	0	0	0	1	0	0	0	1	0	1	
8123	0	1	1	4	0	0	1	0	0	0	0	0	0	1	1	

8124 rows × 115 columns

2.4 - Processing Binary Target Values

In [32]:
```
print(f"Initial Value Counts\n{processed_targets['poisonous'].value_counts()}\n")
processed_targets = ordinal(processed_targets, 'poisonous', ['e', 'p'])
print(f"Final Value Counts\n{processed_targets['poisonous'].value_counts()}")
```

```
Initial Value Counts
poisonous
e    4208
p    3916
Name: count, dtype: int64

Final Value Counts
poisonous
0    4208
1    3916
Name: count, dtype: int64
```

2.5 - Validating the Processed Target Values

In [33]:
```
print("Shape =", initial_targets.shape)
initial_targets
```

```
Shape = (8124, 1)
```

Out[33]:

	poisonous
0	p
1	e
2	e
3	p
4	e
...	...
8119	e
8120	e
8121	e
8122	p
8123	e

8124 rows × 1 columns

In [34]:
```
print("Shape =", processed_targets.shape)
processed_targets
```

```
Shape = (8124, 1)
```

Out[34]:

	poisonous
0	1
1	0
2	0
3	1
4	0
...	...
8119	0
8120	0
8121	0
8122	1
8123	0

8124 rows × 1 columns

3 - Splitting the Dataset

In [35]:
```
split_proportion = 0.7
dataset_size = processed_features.shape[0]
train_size = int(dataset_size * split_proportion)
test_size = dataset_size - train_size
```

In [36]:
```
x_train, y_train = processed_features.iloc[train_size:], processed_targets.iloc[train_size:]['poisonous']
x_test, y_test = processed_features.iloc[:train_size], processed_targets.iloc[:train_size]['poisonous']

print(f"Shape of x_train = {x_train.shape}")
print(f"Shape of y_train = {y_train.shape}")
print(f"Shape of x_test = {x_test.shape}")
print(f"Shape of y_test = {y_test.shape}")
```

```
Shape of x_train = (2438, 115)
Shape of y_train = (2438,)
Shape of x_test = (5686, 115)
Shape of y_test = (5686,)
```

4 - Supervised Model

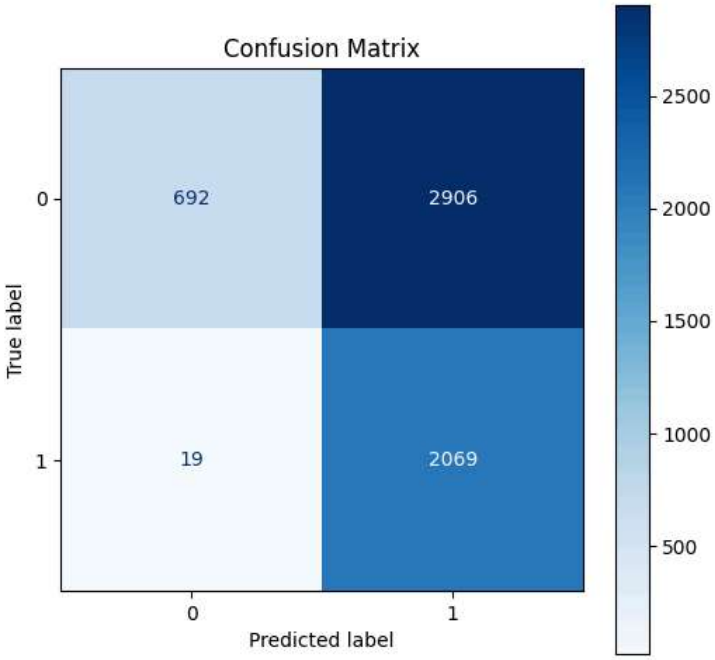
Accuracy = 0.48558

In [37]:
```
def display_results(accuracy, confusion_matrix, classification_report, predictions):
    print(f"Accuracy = {accuracy:.5f}\n")

    fig, ax = plt.subplots(figsize=(6, 6))
    disp = ConfusionMatrixDisplay(confusion_matrix=confusion_matrix, display_labels=np.unique(predictions))
    disp.plot(ax=ax, cmap='Blues', values_format='d')
    plt.title("Confusion Matrix")
    plt.show()

    print(classification_report)
```

In [38]:
```
def train_test_eval(model, x_train_set, y_train_set, x_test_set, y_test_set):
    model.fit(x_train_set, y_train_set)
    y_pred_set = model.predict(x_test_set).astype(int)
    results = (y_pred_set == y_test_set).value_counts()
    accuracy = float(results[True] / y_test_set.shape[0])
    return {
        "predictions": y_pred_set,
        "accuracy": accuracy,
        "confusion_matrix": confusion_matrix(y_test_set, y_pred_set),
        "classification_report": classification_report(y_test_set, y_pred_set)
    }
```

In [51]:
```
logistic_regression_model = LogisticRegression(n_jobs=-1)
logistic_regression_results = train_test_eval(logistic_regression_model, x_train, y_train, x_test, y_test)
display_results(**logistic_regression_results)
```

```
Accuracy = 0.48558
```



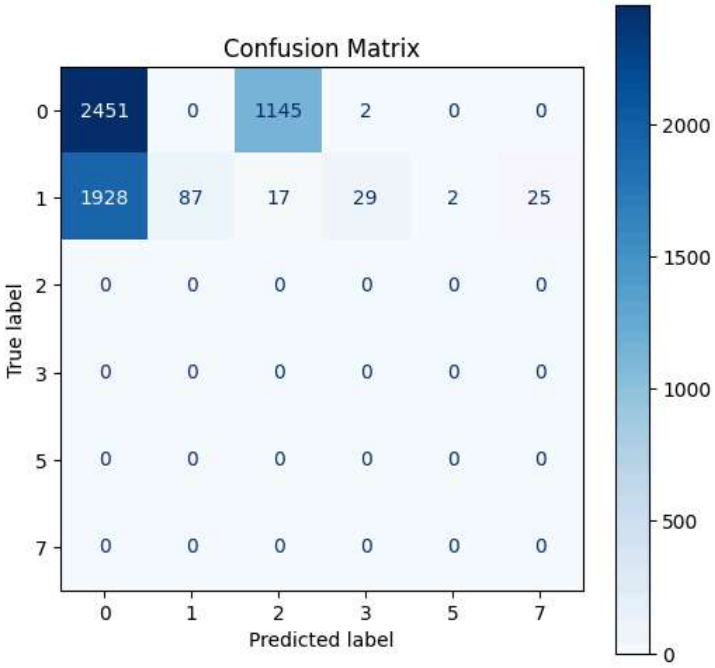
	precision	recall	f1-score	support
0	0.97	0.19	0.32	3598
1	0.42	0.99	0.59	2088
accuracy			0.49	5686
macro avg	0.69	0.59	0.45	5686
weighted avg	0.77	0.49	0.42	5686

5 - Unsupervised Model

Accuracy = 0.44636

In [50]:
```
k_means_model = KMeans()
k_means_results = train_test_eval(k_means_model, x_train, y_train, x_test, y_test)
display_results(**k_means_results)
```

```
Accuracy = 0.44636
```



	precision	recall	f1-score	support
0	0.56	0.68	0.61	3598
1	1.00	0.04	0.08	2088
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
5	0.00	0.00	0.00	0
7	0.00	0.00	0.00	0
accuracy			0.45	5686
macro avg	0.26	0.12	0.12	5686
weighted avg	0.72	0.45	0.42	5686