Bilenkin550Week7_Exercise_7.2

April 27, 2025

1 7.2 Exercise: Dimensionality Reduction and Feature Selection

1.1 Part 1: PCA and Variance Threshold in a Linear Regression

1.1.1 1. Import the housing data as a DataFrame and ensure that the data is loaded properly

[396]:
```
import pandas as pd
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

# Defining the path to the dataset
file_path = r"C:\Users\maxim\OneDrive\Desktop\BU\DSC 550\train.csv"

# Loading the dataset
df = pd.read_csv(file_path)

# Displaying the shape and the first two rows to verify it loaded correctly
print("Shape of dataset:", df.shape)
df.head(2) # Displaying only two rows in order not to take too many pages when
 ↪converted to PDF
```

```
Shape of dataset: (1460, 81)
```

[396]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL         65.0     8450   Pave   NaN      Reg
1   2          20       RL         80.0     9600   Pave   NaN      Reg

  LandContour Utilities  … PoolArea PoolQC Fence MiscFeature MiscVal MoSold  \
0         Lvl    AllPub  …        0    NaN   NaN         NaN       0      2
1         Lvl    AllPub  …        0    NaN   NaN         NaN       0      5

  YrSold  SaleType  SaleCondition  SalePrice
0   2008        WD         Normal     208500
1   2007        WD         Normal     181500

[2 rows x 81 columns]
```

1.2 2. Drop the "Id" column and any features that are missing more than 40% of their values.

[397]:
```python
# Dropping the 'Id' column
df.drop('Id', axis=1, inplace=True)

# Dropping columns with more than 40% missing values
threshold = len(df) * 0.4
df = df.loc[:, df.isnull().sum() < threshold]

# Checking the new shape and previewing the result
print("Shape after dropping columns with >40% missing values:", df.shape)
df.head(2)
```

```
Shape after dropping columns with >40% missing values: (1460, 74)
```

[397]:
```
   MSSubClass MSZoning  LotFrontage  LotArea Street LotShape LandContour  \
0          60       RL         65.0     8450   Pave      Reg         Lvl
1          20       RL         80.0     9600   Pave      Reg         Lvl

  Utilities LotConfig LandSlope  … EnclosedPorch 3SsnPorch ScreenPorch  \
0    AllPub    Inside       Gtl  …             0         0           0
1    AllPub       FR2       Gtl  …             0         0           0

  PoolArea MiscVal  MoSold  YrSold  SaleType  SaleCondition SalePrice
0        0       0       2    2008        WD         Normal    208500
1        0       0       5    2007        WD         Normal    181500

[2 rows x 74 columns]
```

1.3 3. For numerical columns, fill in any missing data with the median value.

[398]:
```python
# Filling missing values in numerical columns with the median
numeric_cols = df.select_dtypes(include='number').columns
df[numeric_cols] = df[numeric_cols].fillna(df[numeric_cols].median())
```

1.4 4. For categorical columns, fill in missing data with the most common value (mode).

[399]:
```python
# Filling missing values in categorical columns with the most frequent value
↪(mode)
categorical_cols = df.select_dtypes(include='object').columns
for col in categorical_cols:
    df[col] = df[col].fillna(df[col].mode()[0])
```

1.5 Step 5: Convert the categorical columns to dummy variables

To prepare the dataset for the linear regression model, I need to convert the categorical columns into dummy variables (one-hot encoding). This ensures that categorical data can be used in machine learning algorithms, which require numerical inputs.

[400]:
```python
# Converting categorical columns to dummy variables (one-hot encoding)
df_encoded = pd.get_dummies(df, drop_first=True)

# Displaying the new dataframe to verify
df_encoded.head(2)
```

[400]:
```
   MSSubClass  LotFrontage  LotArea  OverallQual  OverallCond  YearBuilt  \
0          60         65.0     8450            7            5       2003
1          20         80.0     9600            6            8       1976

   YearRemodAdd  MasVnrArea  BsmtFinSF1  BsmtFinSF2  …  SaleType_ConLI  \
0          2003       196.0         706           0  …           False
1          1976         0.0         978           0  …           False

   SaleType_ConLw  SaleType_New  SaleType_Oth  SaleType_WD  \
0           False         False         False         True
1           False         False         False         True

   SaleCondition_AdjLand  SaleCondition_Alloca  SaleCondition_Family  \
0                  False                 False                 False
1                  False                 False                 False

   SaleCondition_Normal  SaleCondition_Partial
0                  True                  False
1                  True                  False

[2 rows x 230 columns]
```

1.6 6. Split the data into a training and test set, where the SalePrice column is the target.

To train and evaluate a linear regression model, the data must be split into a training set and a test set. The target variable, SalePrice, will be separated from the features.

[401]:
```python
from sklearn.model_selection import train_test_split

# Separating the target (SalePrice) and features (X)
X = df_encoded.drop('SalePrice', axis=1)  # Features
y = df_encoded['SalePrice']  # Target variable

# Splitting the data into training and test sets (80% train, 20% test)
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Displaying the shape of the resulting train and test sets
print(f"Training features shape: {X_train.shape}")
print(f"Test features shape: {X_test.shape}")
print(f"Training target shape: {y_train.shape}")
print(f"Test target shape: {y_test.shape}")
```

```
Training features shape: (1168, 229)
Test features shape: (292, 229)
Training target shape: (1168,)
Test target shape: (292,)
```

1.7 7 Run a linear regression and report the R2-value and RMSE on the test set.

To establish a baseline, I fit a linear regression model on the training data and evaluated it on the test data using R^2 (coefficient of determination) and RMSE (root mean squared error).

[402]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error
import numpy as np

# Initializing and training the model
lr = LinearRegression()
lr.fit(X_train, y_train)

# Making predictions
y_pred = lr.predict(X_test)

# Evaluating performance
r2 = r2_score(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))

print("R² Score:", r2)
print("RMSE:", rmse)
```

```
R² Score: 0.6478367401193958
RMSE: 51973.138076035466
```

Step 7: Baseline Linear Regression

After splitting the dataset into training and test sets, I trained a baseline linear regression model. The model achieved an R^2 score of approximately 0.65 (or ~65%) and an RMSE of about $52,000, suggesting a moderate level of predictive performance.

1.8 8. Fit and transform the training features with a PCA so that 90% of the variance is retained.

To reduce dimensionality while preserving most of the data's variance, I used PCA to transform the training features so that 90% of the variance is retained.

[403]:
```
from sklearn.decomposition import PCA

# Fitting PCA on the training features to retain 90% of the variance
pca = PCA(n_components=0.90, random_state=42)
X_train_pca = pca.fit_transform(X_train)

# Printing the number of principal components
print("Number of PCA components to retain 90% variance:", X_train_pca.shape[1])
```

```
Number of PCA components to retain 90% variance: 1
```

1.9 9. How many features are in the PCA-transformed matrix?

After applying PCA to the training features, I transformed the test set using the same PCA model. The PCA-transformed test set contains only 1 feature, which explains 90% of the variance in the original dataset.

[404]:
```
# Transforming the test features using the same PCA (without fitting it again)
X_test_pca = pca.transform(X_test)

# Verifying the shape of the transformed test set
print("Shape of the transformed test set:", X_test_pca.shape)
```

```
Shape of the transformed test set: (292, 1)
```

1.10 10 Transform but DO NOT fit the test features with the same PCA.

[405]:
```
# Transforming the test set using the already fitted PCA model
X_test_pca = pca.transform(X_test)

# Verifying the shape of the transformed test set
print("Shape of the transformed test set:", X_test_pca.shape)
```

```
Shape of the transformed test set: (292, 1)
```

1.11 11 Repeat step 7 with your PCA transformed data.

[406]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Fitting a linear regression model on the PCA-transformed training features
regressor_pca = LinearRegression()
regressor_pca.fit(X_train_pca, y_train)
```

```python
# Predicting on the PCA-transformed training set
y_train_pred_pca = regressor_pca.predict(X_train_pca)

# Calculating R² score and RMSE
r2_pca = r2_score(y_train, y_train_pred_pca)
rmse_pca = np.sqrt(mean_squared_error(y_train, y_train_pred_pca))

# Displaying results
print(f"R² Score (PCA-transformed data): {r2_pca}")
print(f"RMSE (PCA-transformed data): {rmse_pca}")
```

```
R² Score (PCA-transformed data): 0.07141343094474917
RMSE (PCA-transformed data): 74421.78023339862
```

Step 11: Linear Regression Using PCA-Transformed Data

After fitting a linear regression model on the PCA-transformed training data, the model performed worse than the baseline. It achieved an R^2 score of approximately 0.07 (or ~7%) and an RMSE of around $74,422. This drop in performance highlights that, although PCA retains most of the variance, it may still lose important information for predicting the target variable—especially when reduced to a single component.

1.12 12. Take your original training features (from step 6) and apply a min-max scaler to them.

[407]:
```python
from sklearn.preprocessing import MinMaxScaler

# Initializing the MinMaxScaler
scaler = MinMaxScaler()

# Fitting and transforming the training features
X_train_scaled = scaler.fit_transform(X_train)

# Printing shape to verify
print("Scaled training data shape:", X_train_scaled.shape)
```

```
Scaled training data shape: (1168, 229)
```

1.13 13. Find the min-max scaled features in your training set that have a variance above 0.1.

[408]:
```python
from sklearn.feature_selection import VarianceThreshold

# Initializing the VarianceThreshold with threshold=0.1
selector = VarianceThreshold(threshold=0.1)
X_train_high_variance = selector.fit_transform(X_train_scaled)

# Printing the shape to see how many features were retained
```

```python
print("Shape after variance thresholding (train):", X_train_high_variance.shape)
```

```
Shape after variance thresholding (train): (1168, 40)
```

1.14 14. Transform but DO NOT fit the test features with the same steps applied in steps 11 and 12.

Scaling the test set:

[409]:
```python
# Transforming the test set using the same scaler (do not fitting again)
X_test_scaled = scaler.transform(X_test)
```

Applying variance threshold:

[410]:
```python
# Applying the same variance selector to the test set
X_test_high_variance = selector.transform(X_test_scaled)

# Printing the shape to confirm
print("Shape after variance thresholding (test):", X_test_high_variance.shape)
```

```
Shape after variance thresholding (test): (292, 40)
```

1.15 Step 14: Transform but DO NOT fit the test features with the same steps applied in steps 11 and 12.

To ensure consistency between the training and test data, I applied the same transformations to the test set without fitting the scaler or variance selector again. This step involves:

1. Scaling the test features using the previously fitted min-max scaler.
2. Applying the variance threshold to keep only the features with a variance above 0.1, using the same selector applied to the training data.

The shape of the test set after these transformations is (292, 40), confirming that only the most informative features have been retained.

1.16 15. Repeat step 7 with the high variance data.

In this step, I will train a linear regression model using the high-variance features from the training set (after applying the min-max scaler and variance threshold). I will then evaluate the model's performance using R^2 and RMSE on the test set.

[411]:
```python
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Training a linear regression model on the high-variance training data
lr_high_variance = LinearRegression()
lr_high_variance.fit(X_train_high_variance, y_train)

# Predicting on the high-variance test data
y_pred_high_variance = lr_high_variance.predict(X_test_high_variance)
```

```
# Calculating R² and RMSE
r2_high_variance = r2_score(y_test, y_pred_high_variance)
rmse_high_variance = mean_squared_error(y_test, y_pred_high_variance,␣
 ↪squared=False)

# Display the results
print(f"R² Score (High Variance Data): {r2_high_variance}")
print(f"RMSE (High Variance Data): {rmse_high_variance}")
```

```
R² Score (High Variance Data): 0.648122941134226
RMSE (High Variance Data): 51952.0146512729
```

1.17 16. Summarize your findings.

I tried out different ways to reduce the number of features and see how it affects model performance:

Baseline model (all features): Worked very well with an R^2 of ~0.65 and RMSE around $52K.

PCA (90% variance): I reduced everything to just 1 component, and performance dropped substantially to R^2 ~0.07 and RMSE ~$74K). Not a good model.

High-variance features: I picked the top 40 features with most variance and got back to R^2 ~0.65 or ~65%, the same as the full model — but with fewer features.

Conclusion: PCA didn't work well here, but selecting high-variance features gave a simpler model with no loss in accuracy.

2 Part 2: Categorical Feature Selection

1. Import the data as a data frame and ensure it is loaded correctly.

[412]:
```
import pandas as pd

# Loading the dataset from your provided path
df = pd.read_csv(r"C:\Users\maxim\OneDrive\Desktop\BU\DSC 550\mushrooms.csv")

# Checking the first few rows
print(df.head(2))
print(df.info())
```

```
  class cap-shape cap-surface cap-color bruises odor gill-attachment  \
0     p         x           s         n       t    p               f
1     e         x           s         y       t    a               f

  gill-spacing gill-size gill-color  … stalk-surface-below-ring  \
0            c         n          k  …                        s
1            c         b          k  …                        s

  stalk-color-above-ring stalk-color-below-ring veil-type veil-color  \
```

```
0                      w                      w         p          w
1                      w                      w         p          w

  ring-number ring-type spore-print-color population habitat
0           o         p                 k          s       u
1           o         p                 n          n       g

[2 rows x 23 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8124 entries, 0 to 8123
Data columns (total 23 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   class                     8124 non-null   object
 1   cap-shape                 8124 non-null   object
 2   cap-surface               8124 non-null   object
 3   cap-color                 8124 non-null   object
 4   bruises                   8124 non-null   object
 5   odor                      8124 non-null   object
 6   gill-attachment           8124 non-null   object
 7   gill-spacing              8124 non-null   object
 8   gill-size                 8124 non-null   object
 9   gill-color                8124 non-null   object
 10  stalk-shape               8124 non-null   object
 11  stalk-root                8124 non-null   object
 12  stalk-surface-above-ring  8124 non-null   object
 13  stalk-surface-below-ring  8124 non-null   object
 14  stalk-color-above-ring    8124 non-null   object
 15  stalk-color-below-ring    8124 non-null   object
 16  veil-type                 8124 non-null   object
 17  veil-color                8124 non-null   object
 18  ring-number               8124 non-null   object
 19  ring-type                 8124 non-null   object
 20  spore-print-color         8124 non-null   object
 21  population                8124 non-null   object
 22  habitat                   8124 non-null   object
dtypes: object(23)
memory usage: 1.4+ MB
None
```

2. Convert the categorical features (all of them) to dummy variables.

[413]:
```python
# Dropping the target variable ('class') from the features
X = pd.get_dummies(df.drop('class', axis=1))  # Features: all predictors␣
 ↪converting to dummies
y = df['class']  # Target: 'e' (edible) or 'p' (poisonous)

# Checking the shape of the new X
```

```python
print(X.shape)
```

```
(8124, 117)
```

3. Split the data into a training and test set.

[414]:
```python
from sklearn.model_selection import train_test_split

# Splitting the data (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

# Checking the size of the splits
print(f"Training set size: {X_train.shape}")
print(f"Test set size: {X_test.shape}")
```

```
Training set size: (6499, 117)
Test set size: (1625, 117)
```

4. Fit a decision tree classifier on the training set.

[415]:
```python
from sklearn.tree import DecisionTreeClassifier

# Creating and fitting the decision tree model
tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)

# Checking the model's training accuracy
train_accuracy = tree_clf.score(X_train, y_train)
print(f"Training Accuracy: {train_accuracy:.4f}")
```

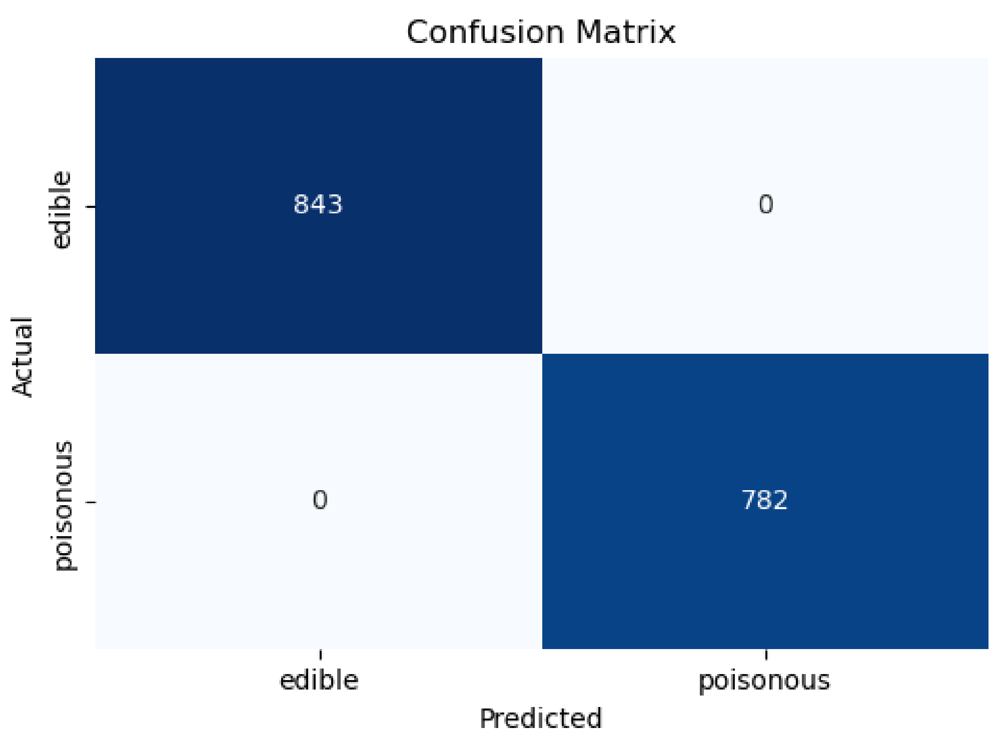
```
Training Accuracy: 1.0000
```

5. Report the accuracy and create a confusion matrix for the model prediction on the test set.

[416]:
```python
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score, confusion_matrix

# Making predictions on the test set
y_pred = tree_clf.predict(X_test)

# Calculating accuracy on the test set
test_accuracy = accuracy_score(y_test, y_pred)
print(f"Test Accuracy: {test_accuracy:.4f}")

# Creating the confusion matrix
conf_matrix = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:")
print(conf_matrix)
```

```python
# Plotting the confusion matrix as a heatmap without colorbar
plt.figure(figsize=(6, 4))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=['edible', 'poisonous'],
            yticklabels=['edible', 'poisonous'],
            cbar=False)

# Adding labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')
plt.show()
```

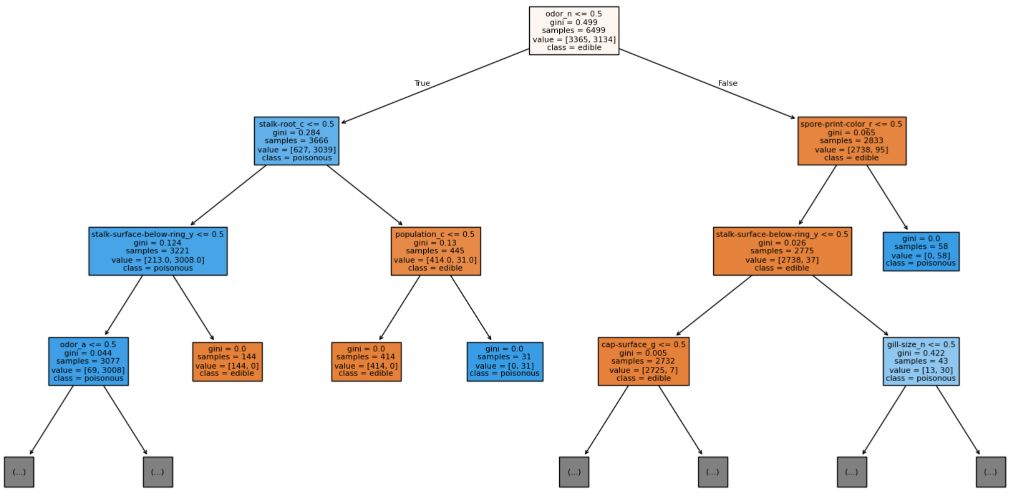
```
Test Accuracy: 1.0000
Confusion Matrix:
[[843   0]
 [  0 782]]
```



6. Create a visualization of the decision tree.

[417]:
```python
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree
```

```python
# Plotting the decision tree
plt.figure(figsize=(20, 10))
plot_tree(tree_clf, filled=True, feature_names=X.columns,␣
↪class_names=['edible', 'poisonous'], max_depth=3)
plt.show()
```



7. Use a 2-statistic selector to pick the five best features for this data.

[418]:
```python
from sklearn.feature_selection import SelectKBest, chi2

# Applying Chi-squared test to select top 5 features
chi2_selector = SelectKBest(score_func=chi2, k=5)
chi2_selector.fit(X, y)  # Fitting the selector to the data
```

[418]:
```
SelectKBest(k=5, score_func=<function chi2 at 0x000001C273730040>)
```

8. Which five features were selected in step 7? Hint: Use the get_support function.

[419]:
```python
# Getting the 5 best features selected by the  ²-statistic
selected_features = X.columns[chi2_selector.get_support()]

# Formatting the output neatly
print("Top 5 Selected Features (from Step 7):")
for idx, feature in enumerate(selected_features, 1):
    print(f"{idx}. {feature}")
```

```
Top 5 Selected Features (from Step 7):
1. odor_f
2. odor_n
3. gill-color_b
```

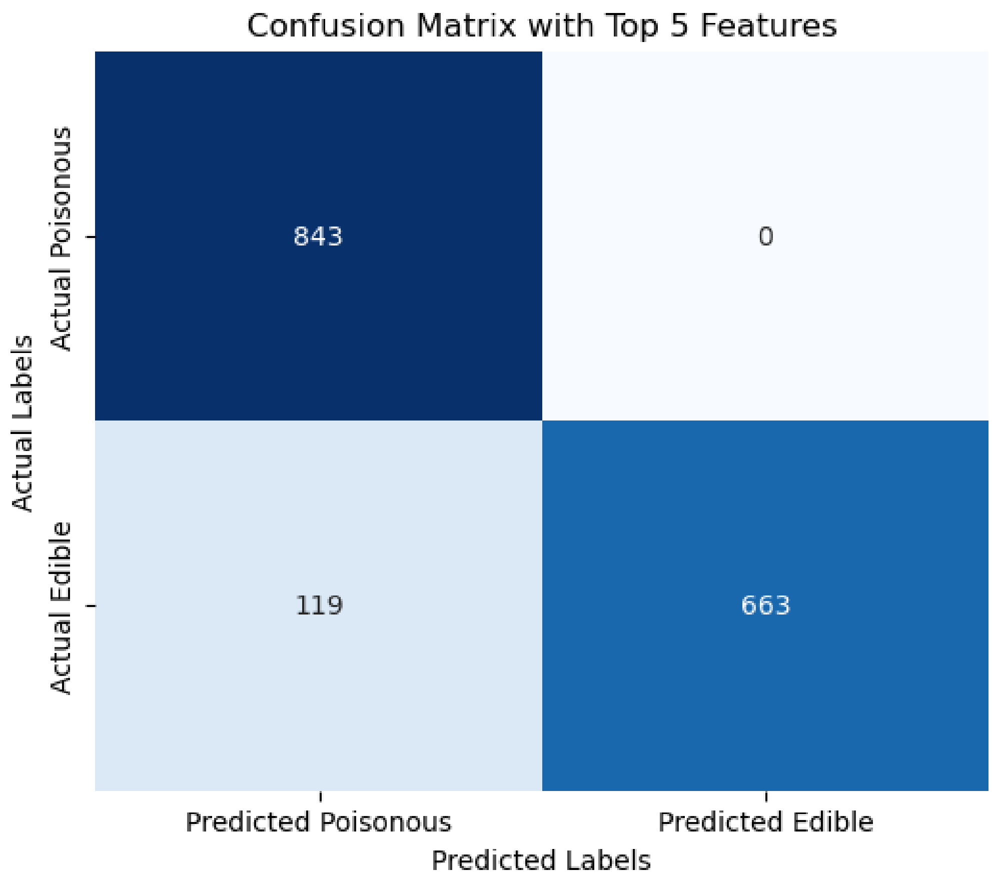
4. stalk-surface-above-ring_k
5. stalk-surface-below-ring_k

9. Repeat steps 4 and 5 with the five best features selected in step 7.

[420]:
```
from sklearn.feature_selection import SelectKBest, chi2
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Filtering the data to use only the selected features
X_top5 = X[selected_features]
print("Top 5 Features Used for the Model:")

# Splitting the data into training and test sets
X_train5, X_test5, y_train5, y_test5 = train_test_split(X_top5, y, test_size=0.
 ↪2, random_state=42)
print(f"Training set size: {X_train5.shape}")
print(f"Test set size: {X_test5.shape}")

# Fitting the decision tree model
tree_clf5 = DecisionTreeClassifier(random_state=42)
tree_clf5.fit(X_train5, y_train5)

# Evaluating the model on the test set
y_pred5 = tree_clf5.predict(X_test5)
test_accuracy5 = accuracy_score(y_test5, y_pred5)
print(f"Test Accuracy with top 5 features: {test_accuracy5:.4f}")

# Creating and printing the confusion matrix
conf_matrix5 = confusion_matrix(y_test5, y_pred5)
print("Confusion Matrix with top 5 features:")
print(conf_matrix5)

# Plotting the confusion matrix as a heatmap for better visualization
plt.figure(figsize=(6, 5))  # adjusting figure size for better clarity
sns.heatmap(conf_matrix5, annot=True, fmt="d", cmap="Blues",
            xticklabels=["Predicted Poisonous", "Predicted Edible"],
            yticklabels=["Actual Poisonous", "Actual Edible"],
            cbar=False)

# Title and labels
plt.title("Confusion Matrix with Top 5 Features")
plt.xlabel("Predicted Labels")
plt.ylabel("Actual Labels")
```

```
# Showing the plot for better readability
plt.show()
```

```
Top 5 Features Used for the Model:
Training set size: (6499, 5)
Test set size: (1625, 5)
Test Accuracy with top 5 features: 0.9268
Confusion Matrix with top 5 features:
[[843   0]
 [119 663]]
```



10. Summarize your findings.

Overall, the decision tree model demonstrated strong performance. Using the top five selected features, the model achieved a test accuracy of approximately 92.7%. The training set consisted of 6,499 mushroom samples, with the following features selected: odor_f, odor_n, gill-color_b, stalk-surface-above-ring_k, and stalk-surface-below-ring_k.

Based on the Confusion Matrix with the top five features, the model correctly classified 843 actual poisonous mushrooms and 663 actual edible mushrooms. It misclassified 119 edible mushrooms as poisonous but did not misclassify any poisonous mushrooms as edible. Overall, the model performed very well, with a relatively low number of misclassifications, indicating strong predictive capability.