PCA and Variance Threshold in a Linear Regression

In [ ]:

In [1]:
```
# Import General Package
import pandas as pd
import numpy as np
```

In [2]:
```
# Import Part 1 Packages
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score, mean_squared_error
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
from sklearn.feature_selection import VarianceThreshold
```

Import the housing data as a data frame and ensure that the data is loaded properly.

In [3]:
```
# Files
file_train = '/Users/Malloryh5/Downloads/house-prices-advanced-regression-techniques/train.csv'
file_test = '/Users/Malloryh5/Downloads/house-prices-advanced-regression-techniques/test.csv'
```

In [4]:
```
# Load data
housing_data = pd.read_csv(file_train)
```

In [5]:
```
# Print data
housing_data
```

Out[5]:

	Id	MSSubClass	MSZoning	LotFrontage	LotArea	Street	Alley	LotShape	LandContour	Utilities	...	PoolArea	PoolQC	Fence	MiscFeature	MiscVal	MoSold
0	1	60	RL	65.0	8450	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	2
1	2	20	RL	80.0	9600	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	5
2	3	60	RL	68.0	11250	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	9
3	4	70	RL	60.0	9550	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	2
4	5	60	RL	84.0	14260	Pave	NaN	IR1	Lvl	AllPub	...	0	NaN	NaN	NaN	0	12
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1455	1456	60	RL	62.0	7917	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	8
1456	1457	20	RL	85.0	13175	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	MnPrv	NaN	0	2
1457	1458	70	RL	66.0	9042	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	GdPrv	Shed	2500	5
1458	1459	20	RL	68.0	9717	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	4
1459	1460	20	RL	75.0	9937	Pave	NaN	Reg	Lvl	AllPub	...	0	NaN	NaN	NaN	0	6

1460 rows × 81 columns

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

In [6]:
```
# Drop ID column
housing_data = housing_data.drop(columns = 'Id')
```

In [7]:
```
# set threshold for columns (0)
thresh_data = .4 * housing_data.shape[0]
# Drop data from columns under 40%
housing_data = housing_data.dropna(thresh = thresh_data, axis = 1)
```

In [8]:
```
# Count columns
print(f'Number of Columns: {len(housing_data.columns)}')
```

```
Number of Columns: 76
```

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

In [9]:
```
# only number columns
number_housing_column = housing_data.select_dtypes(include = ['float64','int64']).columns
```

In [10]:
```
# For loop to change NaN to median number
for x in number_housing_column:
    housing_data.loc[:,x] = housing_data[x].fillna(housing_data[x].median())
```

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

In [11]:
```
# Only categorical columns
cat_housing_column = housing_data.select_dtypes(include = ['object']).columns
```

In [12]:
```
# For loop to change NaN to most common value
for x in cat_housing_column:
    housing_data.loc[:,x] = housing_data[x].fillna(housing_data[x].mode()[0])
```

Convert the categorical columns to dummy variables.

In [13]:
```
# Make dummy values for Categorical data
housing_data = pd.get_dummies(housing_data, drop_first=True)
```

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

In [14]:
```
# Target SalePrice
X = housing_data.drop(columns = ['SalePrice'])
y = housing_data['SalePrice']
```

In [15]:
```
# Split Data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.2, random_state=42)
```

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

In [16]:
```
# Allow linear Regression
lr = LinearRegression()
```

In [17]:
```
# Fit dat
lr.fit(X_train, y_train);
```

Out[17]:
```
▾ LinearRegression
LinearRegression()
```

In [18]:
```
# Predict y
y_pred = lr.predict(X_test)
```

In [19]:
```
# R2
r2 = r2_score(y_test,y_pred)
# RMSE
rmse = mean_squared_error(y_test, y_pred, squared = False)
```

In [20]:
```
# Print results
lr_check = (f' r2: {round(r2,3)}\n RMSE:{round(rmse,3)}')
print(lr_check)
```

```
 r2: 0.643
 RMSE:52298.872
```

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

In [21]:
```
# allow PCA at 90%
pca = PCA(n_components=.9)
# fit train data
X_train_pca = pca.fit_transform(X_train)
```

Features are in the PCA-transformed matrix

In [22]:
```
print(f"Features after PCA: {X_train_pca.shape[1]}")
```

```
Features after PCA: 1
```

In [23]:
```
# Transform PCA do not fit to see how well the test set does
X_test_pca = pca.transform(X_test)
```

In [24]:
```
# fit y_test for PCA
lr.fit(X_train_pca, y_train);
```

Out[24]:
```
▾ LinearRegression
LinearRegression()
```

In [25]:
```
# Predict y_test for PCA
y_pred_pca = lr.predict(X_test_pca)
```

In [26]:
```
r2_pca = r2_score(y_test, y_pred_pca)
rmse = mean_squared_error(y_test, y_pred_pca, squared=False)
```

Use the original training features and apply a min-max scaler.

In [27]:
```
# Allow Scaler
min_max_s = MinMaxScaler()
#Run on scale test
min_max_X_train = min_max_s.fit_transform(X_train)
```

Find the min-max scaled features in training set.

In [28]:
```
# Allow varience Threshold
vt = VarianceThreshold(threshold=.1)
# Fit the modal
vt_X_train = vt.fit_transform(min_max_X_train)
```

In [29]:
```
# transform scaled test data
X_train_high_var = min_max_s.transform(X_test)
# Transform test data
X_test_high_var = vt.transform(X_train_high_var)
```

Repeat with the high variance data.

In [30]:
```
#fit linear regr. to high variance model
lr.fit(vt_X_train, y_train);
```

Out[30]:
```
▾ LinearRegression
LinearRegression()
```

In [31]:
```
# Prdict high Variance data
high_var_y_pred = lr.predict(X_test_high_var)
```

In [32]:
```
# r2
r2_highv = r2_score(y_test,high_var_y_pred)
#RMSE
rmse_highv = mean_squared_error(y_test,high_var_y_pred, squared=False)
```

In [33]:
```
# Pull results down
print(lr_check)

# Print results
print(f'\n R2 High Variance:{round(r2_highv,3)}\n RMSE High Variance: {round(rmse_highv,3)}')
```

```
r2: 0.643
RMSE:52298.872

R2 High Variance:0.656
RMSE High Variance: 51393.432
```

Findings.

Although PCA linear regression is a better indicator than running a regular linear regression model, there are better models to use than linear regression.

Part 2: Categorical Feature Selection

Predict whether or not a mushroom is edible or poisonous.

In [34]:
```
# Load Packages for part 2
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
from sklearn.feature_selection import SelectKBest, chi2
```

In [35]:
```
# Mushroom file
mushroom_file = '/Users/Malloryh5/Downloads/mushrooms.csv'
```

In [36]:
```
# Read CSV
mushroom_data = pd.read_csv(mushroom_file)
```

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

In [37]:
```
# Replace Categorical data with dummy variables
mushroom_data = pd.get_dummies(mushroom_data, drop_first=True)
```

In [38]:
```
# Replace bool type with int
for col in mushroom_data.columns:
    if mushroom_data[col].dtype == 'bool':
        mushroom_data[col] = mushroom_data[col].astype(int)
```

In [39]:
```
mushroom_data
```

Out[39]:

	class_p	cap-shape_c	cap-shape_f	cap-shape_k	cap-shape_s	cap-shape_x	cap-surface_g	cap-surface_s	cap-surface_y	cap-color_c	...	population_n	population_s	population_v	population_y	ha
0	1	0	0	0	0	1	0	1	0	0	...	0	1	0	0	
1	0	0	0	0	0	1	0	1	0	0	...	1	0	0	0	
2	0	0	0	0	0	0	0	1	0	0	...	1	0	0	0	
3	1	0	0	0	0	1	0	0	1	0	...	0	1	0	0	
4	0	0	0	0	0	1	0	1	0	0	...	0	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
8119	0	0	0	1	0	0	0	1	0	0	...	0	0	0	0	
8120	0	0	0	0	0	1	0	1	0	0	...	0	0	1	0	
8121	0	0	1	0	0	0	0	1	0	0	...	0	0	0	0	
8122	1	0	0	1	0	0	0	0	1	0	...	0	0	1	0	
8123	0	0	0	0	0	1	0	1	0	0	...	0	0	0	0	

8124 rows × 96 columns

Split the data into a training and test set.

In [40]:
```
# X - dependant
X = mushroom_data.drop(columns = ['class_p'])
# y - poisonous
y = mushroom_data['class_p']
```

In [41]:
```
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = .2, random_state=42)
```

Fit a decision tree classifier on the training set.

In [42]:
```
# Allow Decision Tree
dtc = DecisionTreeClassifier()
```

In [43]:
```
# Fit data to Decision Tree
dtc.fit(X_train, y_train);
```

Out[43]:

▾ DecisionTreeClassifier

DecisionTreeClassifier()

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

In [44]:
```
# Predict y
dtc_y_pred = dtc.predict(X_test)
```

In [45]:
```
# Accuracy
accuracy_dtc = accuracy_score(y_test, dtc_y_pred)
```

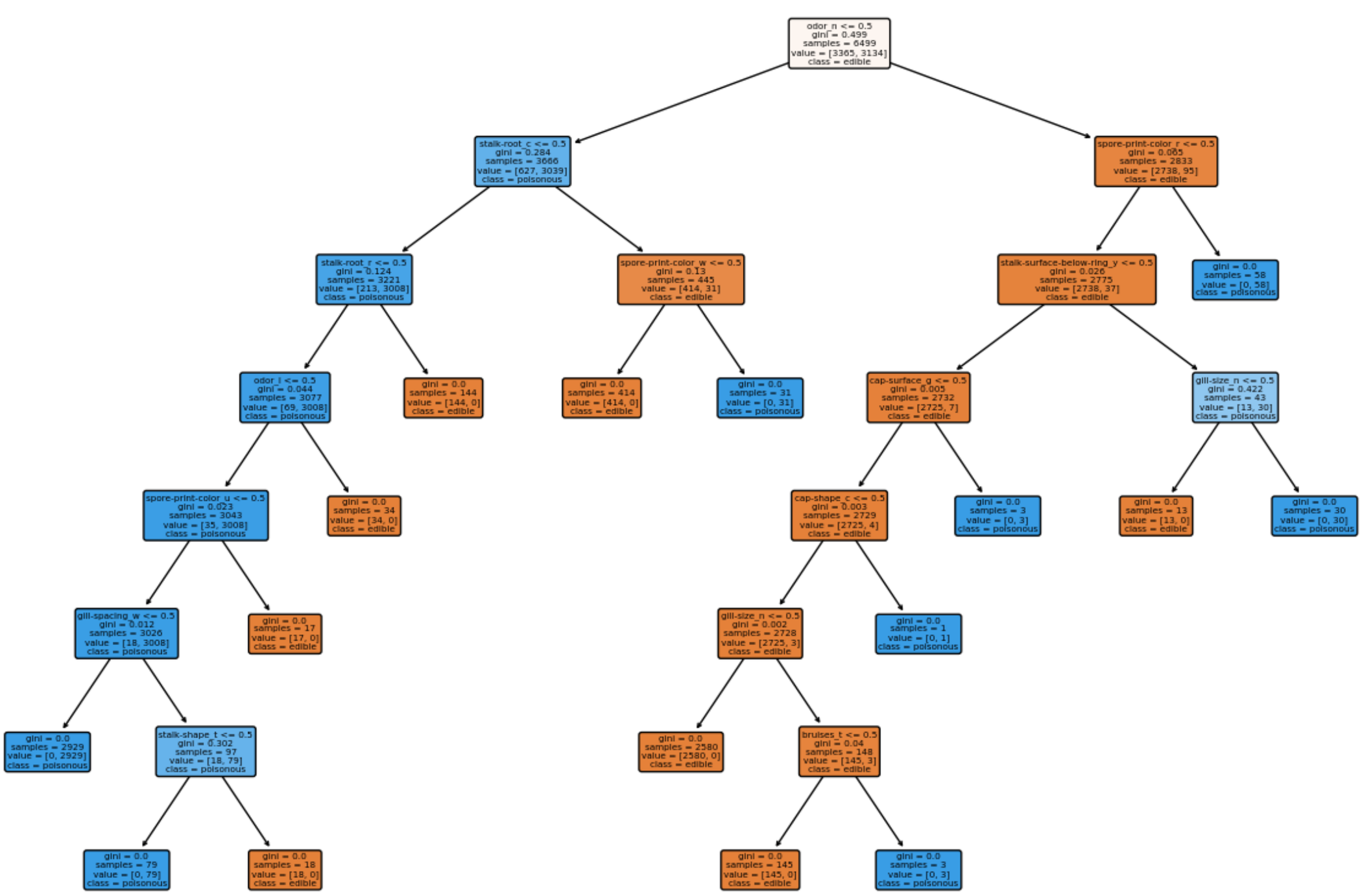
In [46]:
```
# confusion matrix
dtc_confMat = confusion_matrix(y_test, dtc_y_pred)
```

In [47]:
```
# Results
results = (f' Accuracy: {round(accuracy_dtc,3)}\n Confusion Matrix:\n {dtc_confMat}')
print(results)
```

```
 Accuracy: 1.0
 Confusion Matrix:
 [[843   0]
 [  0 782]]
```

Create a visualization of the decision tree.

In [48]:
```
# Figure size
plt.figure(figsize=(15,10))
# Make decision tree
plot_tree(dtc, filled=True, feature_names=list(X.columns), class_names= ['edible', 'poisonous'], rounded=True)
#Print graph
plt.show()
```



Used a χ2-statistic selector to pick the five best features.

In [49]:
```
# select best featues using chi-square statistical test
selector = SelectKBest(chi2, k=5)
```

In [50]:
```
# Fit data
selector.fit(X_train, y_train);
```

Out[50]:
```
SelectKBest(k=5, score_func=<function chi2 at 0x131b72fc0>)
```

Selected five features.

In [51]:
```
# selected features
select_features = X.columns[selector.get_support()]
```

In [52]:
```
# Print results
print(f"Selected features: {select_features}")
```

```
Selected features: Index(['odor_f', 'odor_n', 'gill-size_n', 'stalk-surface-above-ring_k',
       'stalk-surface-below-ring_k'],
      dtype='object')
```

In [53]:
```
#Train data
X_train_select = selector.transform(X_train)
#Test data
X_test_select = selector.transform(X_test)
```

In [54]:
```
# Fit data
dtc.fit(X_train_select, y_train);
```

Out[54]:
```
▾ DecisionTreeClassifier
DecisionTreeClassifier()
```

In [55]:
```
# Predict y
y_pred_select = dtc.predict(X_test_select)
```

In [56]:
```
# Accuracy with selected features
accuracy_select = accuracy_score(y_test, y_pred_select)
```

In [57]:
```
# Confusion Matrix with selected features
conf_matrix_select = confusion_matrix(y_test, y_pred_select)
```

Findings.

In [58]:
```
# Pull first results
print(results)

print(f"\nAccuracy with selected features: {round(accuracy_select,3)}")
print(f"Confusion Matrix with selected features:\n{conf_matrix_select}")
```

```
 Accuracy: 1.0
 Confusion Matrix:
 [[843   0]
  [  0 782]]

Accuracy with selected features: 0.974
Confusion Matrix with selected features:
[[816  27]
 [ 16 766]]
```

Although the accuracy is lower, the first five selected features explain the data well. Selecting features reduces dimensionality and generalises the data.