Theoretical

In [ ]:
```
#What is K-Nearest Neighbors (KNN) and how does it work?
'''
K-Nearest Neighbors (KNN) is a supervised machine learning algorithm used for class
non-parametric, instance-based (lazy) learning algorithm, meaning it doesn’t explic
based on stored training data.

How it works
1.Choose the number of neighbors (K)
K is a hyperparameter that defines how many nearest data points will be considered
2.Calculate the distance
For a given test data point, KNN calculates the distance to every point in the trai
Common distance metrics:
Euclidean Distance (most commonly used)
Manhattan Distance
Minkowski Distance
3.Find the K nearest neighbors
It selects the K closest points from the training dataset
4.Make a prediction
For Classification: The majority class among the K neighbors determines the class o
For Regression: The prediction is the average (or weighted average) of the K neares

Advantages of KNN
Simple and easy to implement.
No need for training; just store the dataset.
Works well with small datasets and low-dimensional data.
'''
#What are the disadvantages of KNN?
'''
Disadvantages of KNN
Computationally expensive for large datasets (since it calculates distances for eve
Sensitive to irrelevant or highly correlated features.
Choosing the right K value can be tricky
'''
# What is the difference between KNN Classification and KNN Regression?
'''
KNN Classification
Output: A category/class label (e.g., "Dog" or "Cat").
Prediction Method:
Finds the K nearest neighbors of the test point.
Uses majority voting—the most common class among the K neighbors is assigned to the
Example Use Case:
Email spam detection (Spam/Not Spam).
Handwriting recognition (Digits 0-9).

KNN Regression
Output: A continuous numerical value (e.g., price, temperature).
Prediction Method:
Finds the K nearest neighbors of the test point.
Computes the average (or weighted average) of their values.
Example Use Case:
Predicting house prices based on nearby houses.
Estimating temperature based on past weather data.
```

```
'''
# What is the role of the distance metric in KNN?
'''
Role of Distance Metric in KNN
The distance metric in K-Nearest Neighbors (KNN) plays a crucial role in determinin
measured. Since KNN makes predictions based on the K nearest neighbors, the way we
accuracy and effectiveness of the model.

Why is Distance Important?
It helps in finding the most relevant neighbors for classification or regression.
A poor choice of distance metric can lead to wrong neighbor selection, reducing mod
It affects the model’s sensitivity to different feature scales (e.g., height in cm

Choosing the Right Distance Metric
Euclidean Distance: Best for continuous data with well-scaled features.
Manhattan Distance: Works well for high-dimensional or grid-based data.
Hamming Distance: Used for categorical data.
Cosine Similarity: Ideal for text or high-dimensional sparse data
'''
# What is the Curse of Dimensionality in KNN?
'''
Curse of Dimensionality in KNN
The Curse of Dimensionality refers to the problem where K-Nearest Neighbors (KNN) a
inefficient and less accurate as the number of features (dimensions) increases.

Why Does It Happen?
In high-dimensional spaces, data points become more sparse (far apart).
The distance between all points becomes almost the same, making it difficult to ide
KNN relies on distance measurements, so if all points are equally far, classificati

Effects of High Dimensionality on KNN
1.Loss of Meaningful Distance
As dimensions increase, Euclidean distance becomes less reliable for finding neares
Most points end up being equidistant, making KNN less effective.
2.Increased Computational Cost
More dimensions mean more distance calculations, leading to longer processing times
Storing high-dimensional data requires more memory.
3.Overfitting
High-dimensional data often contains irrelevant features (noise).
KNN might consider these irrelevant features, leading to poor generalization on new
'''
#How can we choose the best value of K in KNN?
'''
Choosing the right value of K in K-Nearest Neighbors (KNN) is crucial for achieving
controls the balance between bias and variance in the model.

Steps to Choose the Best K
1. Try Different Values of K (Grid Search)
Start with small values like K = 1, 3, 5, 7, … and evaluate performance.
Use a validation set or cross-validation to find the best K.

2. Check for Overfitting & Underfitting
Low K (e.g., K=1, 3):
High variance, sensitive to noise, model may overfit.
Decision boundary is too complex.
High K (e.g., K=15, 20):
```

```
High bias, model is too smooth and may underfit.
Model ignores local patterns.

3. Use the Elbow Method
Plot K vs. Error Rate (or Accuracy).
Look for the "elbow point" where the error starts to flatten out—this is usually th

4. Consider Odd K for Classification
If the number of classes is 2, an odd value of K prevents ties.
For multi-class problems, choose a value that divides evenly among data.

5. Use Cross-Validation (K-Fold)
Use K-Fold Cross-Validation to test different K values on different subsets of data

This prevents biased selection of K based on a single train-test split.
'''
#What are KD Tree and Ball Tree in KNN?When should you use KD Tree vs. Ball Tree?
'''
In K-Nearest Neighbors (KNN), finding the nearest neighbors requires computing dist
for large datasets. To speed up this process, KD-Tree and Ball-Tree are used as eff
search.
KD-Tree (K-Dimensional Tree)
What is it?
A binary tree data structure used for fast nearest neighbor search in low to modera
It recursively splits the dataset along the dimension with the highest variance at

How it Works?
Choose a splitting dimension (e.g., x-axis, y-axis).
Select a median value in that dimension as the split point.
Recursively divide points into left (less than median) and right (greater than medi
Perform KNN search efficiently by only checking relevant branches of the tree.

When to Use KD-Tree?
 Works well for low-dimensional data (<50 dimensions).
 Faster than brute-force search in small-to-medium datasets.
 Becomes inefficient in high-dimensional spaces (Curse of Dimensionality).

2. Ball-Tree
What is it?
A hierarchical tree structure used for nearest neighbor search, especially in high-
Instead of splitting along a single dimension like KD-Tree, it groups points into h

How it Works?
The dataset is divided into clusters (balls) based on proximity.
Each ball contains a center and a radius that encloses a subset of points.
When searching for nearest neighbors, the tree prunes entire balls that are too far

When to Use Ball-Tree?
 Better for high-dimensional data (>50 dimensions).
 More efficient when Euclidean distance is not ideal.
 Slower than KD-Tree in low-dimensional data.

KD-Tree vs. Ball-Tree: Which One to Use?
                        Feature KD Tree
Data Type        Low to medium dimensions (<50)                              Hig
Structure        Splits along dimensions                                     Gro
Efficiency   Fast for small datasets                                    Better
Distance      Metric    Works best with Euclidean Distance               Support

When Not to Use KD-Tree or Ball-Tree?
If the dataset is very high-dimensional (>100 dimensions), both KD-Tree and Ball-Tr
In such cases, Approximate Nearest Neighbor (ANN) methods like FAISS, LSH, or HNSW
'''
# How does feature scaling affect KNN?
'''
Feature scaling is crucial in K-Nearest Neighbors (KNN) because KNN relies on dista
Manhattan distance). If features are not scaled properly, the model may become bias

Why is Feature Scaling Important in KNN?
1.Prevents Dominance of Large-Scale Features
Example:
Feature 1: Age (range: 18–70)
Feature 2: Salary (range: 20,000–200,000)
Since salary values are much larger, they will dominate the distance calculations,
neighbors.

2.Improves Distance Calculation Accuracy
KNN relies on Euclidean Distance (or other metrics), which are sensitive to differe
Proper scaling ensures that all features contribute equally to neighbor selection.

3.Leads to Better Performance & Faster Convergence
Without scaling, KNN might select wrong neighbors, leading to lower accuracy.
Feature scaling makes the algorithm more stable and consistent
'''
#What is PCA (Principal Component Analysis)?
'''
Principal Component Analysis (PCA) is a dimensionality reduction technique used in
data into a lower-dimensional form while preserving the most important patterns.

Why Use PCA?
Reduces Dimensionality – Helps in handling datasets with many features, preventing
Removes Redundant Features – Eliminates correlated and less important features.
Improves Computational Efficiency – Makes algorithms like KNN, Logistic Regression,
Better Visualization – Helps in plotting high-dimensional data in 2D or 3D
'''
#How does PCA work?
'''
How PCA Works (Step-by-Step)
1. Standardize the Data
Convert features to the same scale (Z-score normalization).
PCA is sensitive to large-scale features, so feature scaling is essential.

2. Compute the Covariance Matrix
Measures how features vary with each other.
Helps find relationships between features.

3. Compute Eigenvalues and Eigenvectors
Eigenvectors represent the directions of new axes (Principal Components).
Eigenvalues represent the amount of variance captured by each component.

4. Select the Top K Principal Components
Sort principal components by eigenvalues (higher eigenvalue = more variance).
```

```
Choose the top K components that explain most of the variance (e.g., 95%).

5. Transform Data
Project the original data onto the new PCA space with reduced dimensions.
'''
#What is the geometric intuition behind PCA?
'''
Principal Component Analysis (PCA) can be understood geometrically as a process of
capture the variance in the data.

Step-by-Step Geometric Intuition
1. Data as Points in High-Dimensional Space
Imagine your data as a cloud of points in an N-dimensional space (where N is the nu
Each data point is represented as a vector in this space.

2. Finding the Best New Axes (Principal Components)
PCA rotates the coordinate system to align it with the directions of maximum varian
The new axes (Principal Components) are orthogonal (perpendicular to each other).
The first Principal Component (PC1) is the direction where the data varies the most
The second Principal Component (PC2) is perpendicular to PC1 and captures the next

3. Projecting Data onto the New Axes
Once we have the new principal components, we project data onto them.
This reduces the number of dimensions while keeping as much variance as possible.
Think of it as casting shadows of high-dimensional data onto a lower-dimensional pl

4. Dropping Less Important Dimensions
PCA removes dimensions with low variance, keeping only the most informative ones.
This helps reduce noise and redundancy in the data.
'''
# What is the difference between Feature Selection and Feature Extraction?
'''
Feature Selection
Definition: It involves selecting a subset of relevant features (predictors) from t
features themselves.
Purpose: To remove irrelevant, redundant, or less significant features, improving m
Methods:
Filter Methods (e.g., correlation, chi-square test, mutual information)
Wrapper Methods (e.g., recursive feature elimination, forward/backward selection)
Embedded Methods (e.g., LASSO, decision trees)
Example: In a dataset with 100 features, feature selection might keep only 20 of th

Feature Extraction
Definition: It involves transforming the original features into a new set of featur
transformed features usually have lower dimensionality while preserving essential i
Purpose: To create a more compact representation of the data while reducing redunda
Methods:
Principal Component Analysis (PCA)
Linear Discriminant Analysis (LDA)
Autoencoders (Deep Learning)
t-SNE (for visualization)
Example: Instead of selecting individual features from an image dataset, feature ex
variables that capture the most important variations in the data
'''
#What are Eigenvalues and Eigenvectors in PCA?
'''
Principal Component Analysis (PCA) is a dimensionality reduction technique that tra
-dimensional space while preserving as much variance as possible. The concepts of e
in PCA.

What are Eigenvalues and Eigenvectors?
Eigenvectors are directions in the data space along which the variance is maximized
components) in the transformed space.
Eigenvalues represent the amount of variance carried by each eigenvector. A larger
eigenvector captures more information (variance) from the dataset.

How are they used in PCA?
Compute the Covariance Matrix
The covariance matrix captures the relationships (correlations) between features in
Find Eigenvalues and Eigenvectors
Solve the characteristic equation
(A-λI)𝑣 = 0
where:
A  is the covariance matrix
λ  are the eigenvalues
𝑣  are the eigenvectors
I  is the identity matrix

Sort Eigenvalues and Eigenvectors
The eigenvectors are sorted in descending order based on their eigenvalues. The top
Select Principal Components
Choose the top
k eigenvectors that explain the most variance (based on eigenvalues) and project th
'''
# How do you decide the number of components to keep in PCA?
'''
Explained Variance (Eigenvalues) – Scree Plot
Concept: Each principal component has an associated eigenvalue, which indicates how

How to use:
Plot a Scree Plot (graph of explained variance vs. number of components).
Identify the "elbow point" where adding more components does not significantly incr
Choose the number of components before the elbow.
Example: If the elbow occurs at 3 components, selecting those 3 is ideal.

2. Retaining a Certain Percentage of Variance
Concept: Retain enough components to capture a high percentage (e.g., 95%) of total

How to use:
Compute cumulative explained variance.
Choose the smallest number of components that retain at least 90-95% of the varianc
Example: If 5 components explain 95% of the variance, keep those 5.

3. Kaiser Criterion (Eigenvalues > 1)
Concept: Keep only components with eigenvalues greater than 1, as they contribute m
feature.

How to use:
Compute eigenvalues of principal components.
Keep only those with eigenvalues > 1.
Example: If components 1, 2, and 3 have eigenvalues greater than 1, select those.
```

```
4. Cross-Validation with Machine Learning Models
Concept: Use cross-validation to check how PCA components impact model performance.

How to use:
Train a machine learning model (e.g., Logistic Regression, SVM) using different num
Choose the number that gives the best performance (highest accuracy, lowest error).
Example: If 10 components give the best classification accuracy, select those.

5. Parallel Analysis / Randomized Methods
Concept: Compare PCA eigenvalues with those from a randomized dataset of the same s

How to use:
Perform PCA on both real and randomized data.
Keep components where real eigenvalues are higher than random ones.
Example: If the first 7 components have significantly higher eigenvalues than rando
'''
# Can PCA be used for classification?
'''
No, PCA itself is not a classification algorithm, but it can be used as a preproces
performance.

How PCA Helps in Classification?
PCA is a dimensionality reduction technique, meaning it transforms the dataset into
important patterns. This can be beneficial for classification tasks in the followin

1.Removes Redundant or Correlated Features
PCA eliminates correlated features, making it easier for classification models to l

2.Reduces Overfitting
High-dimensional data can lead to overfitting in models like Decision Trees or Neur
PCA reduces dimensionality and prevents models from memorizing noise.

3.Speeds Up Training Time
With fewer features, algorithms like SVM, Logistic Regression, and KNN train faster

4.Handles Multicollinearity
In datasets where independent variables are highly correlated, PCA creates uncorrel
classification models more stable.
'''
#What are the limitations of PCA?
'''
1. Loss of Interpretability
PCA transforms original features into new principal components, which are linear co
These transformed components are often hard to interpret, making it difficult to ex
applications.
Example:
If a dataset has features like "Age", "Salary", and "Experience", after PCA, the ne
like "Salary Impact."

2. Assumes Linearity
PCA works best when the dataset has a linear structure.
If the data has non-linear relationships, PCA might not capture important patterns.
Alternative: Use kernel PCA (KPCA) or t-SNE for non-linear relationships.
Example:
PCA may fail in recognizing non-linear relationships in image or text classificatio
```

```
3. Sensitive to Scaling
PCA is affected by the scale of features since it relies on variance.
Features with large values (e.g., salary in dollars vs. age in years) will dominate
Solution: Always standardize the data (e.g., using StandardScaler in sklearn).

4. Sensitive to Outliers
Since PCA is based on variance, the presence of outliers can distort the principal
Solution: Detect and remove outliers before applying PCA.

5. May Discard Important Information
PCA prioritizes variance, but low-variance features may still be important for clas
If a classification task depends on a feature with low variance, PCA might remove i
Example:
In fraud detection, rare transactions (low variance) are important, but PCA may dis

6. Computational Cost in High Dimensions
PCA requires calculating eigenvalues and eigenvectors, which can be computationally
Solution: Use Incremental PCA or Randomized PCA for large datasets.

7. Assumes Gaussian Distribution
PCA works best when features follow a Gaussian (normal) distribution.
If data is skewed or multimodal, PCA may not be the best choice.
'''
#How do KNN and PCA complement each other?
'''
K-Nearest Neighbors (KNN) is a simple, non-parametric classification algorithm that
Euclidean distance). Principal Component Analysis (PCA) is a dimensionality reducti
lower-dimensional space while preserving variance.

When used together, PCA can enhance the performance of KNN in various ways:

1. Reduces Dimensionality to Avoid the Curse of Dimensionality
KNN works poorly in high-dimensional data due to the curse of dimensionality (where
dimensions increase).
PCA reduces the number of features while preserving important information, making K
 Example: If a dataset has 500 features, PCA can reduce it to 50 principal componen

2. Speeds Up KNN Computation
KNN is a lazy learner, meaning it stores all training data and computes distances a

With high-dimensional data, distance calculations become expensive.

PCA reduces dimensions, making distance calculations faster.
Example: In image recognition, where each image has thousands of pixels, PCA can re
classification.

3. Reduces Noise and Correlation Among Features
Real-world datasets often have correlated or redundant features that can mislead KN
PCA removes multicollinearity by transforming features into uncorrelated principal
This helps KNN make better distance-based decisions.
Example: In medical diagnosis, multiple lab test results may be correlated. PCA rem
classification accuracy.

4. Prevents Overfitting in High-Dimensional Data
KNN can overfit when there are too many features relative to the number of samples.
PCA reduces dimensions, preventing overfitting while keeping essential variance.
```

```
Example: In text classification, converting words into high-dimensional vectors can
the model.

5. Handles Collinearity Better
KNN struggles when features are highly correlated.
PCA converts correlated features into independent principal components, making KNN
Example: In stock market prediction, many financial indicators are correlated. PCA
'''
#5 How does KNN handle missing values in a dataset?
'''


1. Removing Rows with Missing Values (Simple but Risky )
If the dataset has few missing values, one quick approach is to remove rows with mi
However, this reduces data size and may lead to bias if missing values are not rand
 Use When: The dataset is large, and missing values are rare (<5%).
Avoid If: Missing values are widespread.

2. Imputing Missing Values Using Mean/Median/Mode (Basic Imputation)
Replace missing values with:
Mean (for numerical data)
Median (if data has outliers)
Mode (for categorical data)
Example: If Age = [25, 30, NaN, 35, 40], replace NaN with Mean(Age).
 Pros: Easy to implement.
 Cons: Ignores relationships between features.

3. KNN-Based Imputation (Best for Preserving Patterns ✅)
KNN can predict missing values by finding the nearest neighbors and averaging their
This method considers relationships between features.

How It Works:
Identify the k nearest neighbors (based on available features).
Use their values to estimate the missing value.
For numerical data → take the mean of nearest neighbors.
For categorical data → take the mode of nearest neighbors.
Pros: More accurate than simple imputation.
Cons: Computationally expensive for large datasets.

4. Using Machine Learning Models to Predict Missing Values
Train a regression model (for numerical data) or a classification model (for catego
Example: If Age is missing, train a regression model using other features like Inco
 Pros: More sophisticated than mean/mode imputation.
 Cons: Requires extra computation and model training.

5. Using KNN with Distance-Based Missing Value Handling
Some implementations of KNN can ignore missing values when computing distances.
Instead of discarding a row, KNN calculates distance using only available features.
How It Works:
If feature X is missing for a data point, KNN will calculate distances only based o
This prevents data loss but might reduce accuracy.
Pros: Avoids data loss while still using KNN.
Cons: Can lead to biased results if too many features are missing.
'''
#What are the key differences between PCA and Linear Discriminant Analysis (LDA)?
'''
Feature                  PCA (Principal Component Analysis)            LDA (Linear
Purpose                 Reduces dimensionality by preserving variance   Reduces dim

Type                             Unsupervised                             Supervise

Works On                Features (independent variables)                      Feat

Optimization Criteria       Maximizes variance of data                    Maximizes s

Interpretability             Principal components are abstract          Discriminant
                           and may not have clear meaning                   making t


Use Case                Best for reducing dimensions in datasets        Best for cl
                             without class labels

Computational Cost           Moderate (Eigenvalue decomposition)                Highe


Handles Multicollinearity?        Yes (removes correlation between features)

Output Components            At most min(n_samples, n_features)

Effect on Data         Retains total variance but does not guarantee
Distribution             separation between classes
```

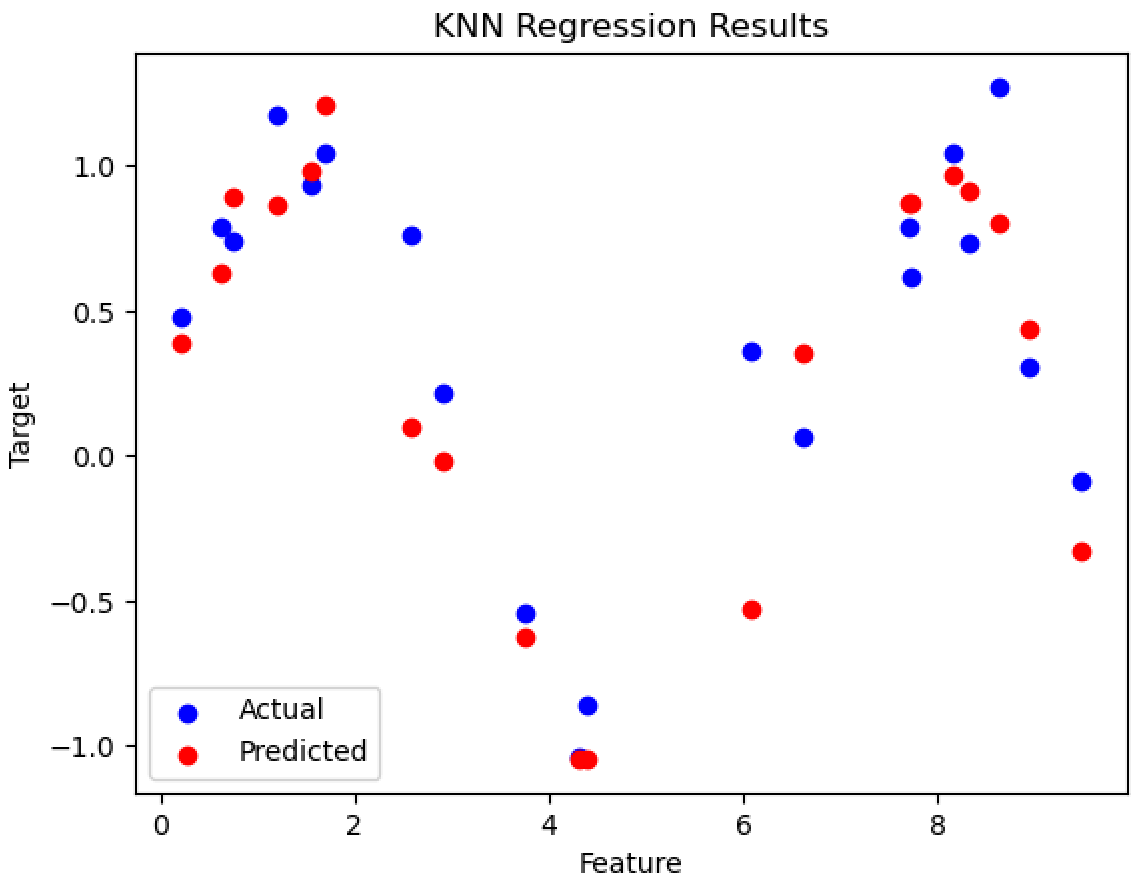
Practical

In [1]:
```python
#Train a KNN Classifier on the Iris dataset and print model accuracy
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_scaled, y_train)

y_pred = knn.predict(X_test_scaled)

accuracy = accuracy_score(y_test, y_pred)
print(f"KNN Model Accuracy on Iris Dataset: {accuracy:.4f}")
```

```
KNN Model Accuracy on Iris Dataset: 1.0000
```

In [3]:
```
#Train a KNN Regressor on a synthetic dataset and evaluate using Mean Squared Error
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error

# Generate synthetic dataset
np.random.seed(42)
X = np.random.rand(100, 1) * 10  # Features (100 samples, 1 feature)
y = np.sin(X).ravel() + np.random.normal(0, 0.3, X.shape[0])  # Target with noise

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train KNN Regressor
knn_regressor = KNeighborsRegressor(n_neighbors=5)
knn_regressor.fit(X_train_scaled, y_train)

# Predict
y_pred = knn_regressor.predict(X_test_scaled)

# Evaluate using MSE
mse = mean_squared_error(y_test, y_pred)
print(f"KNN Regressor MSE: {mse:.4f}")

# Plot results
plt.scatter(X_test, y_test, color="blue", label="Actual")
plt.scatter(X_test, y_pred, color="red", label="Predicted")
plt.xlabel("Feature")
plt.ylabel("Target")
plt.legend()
plt.title("KNN Regression Results")
plt.show()
```

```
KNN Regressor MSE: 0.0997
```

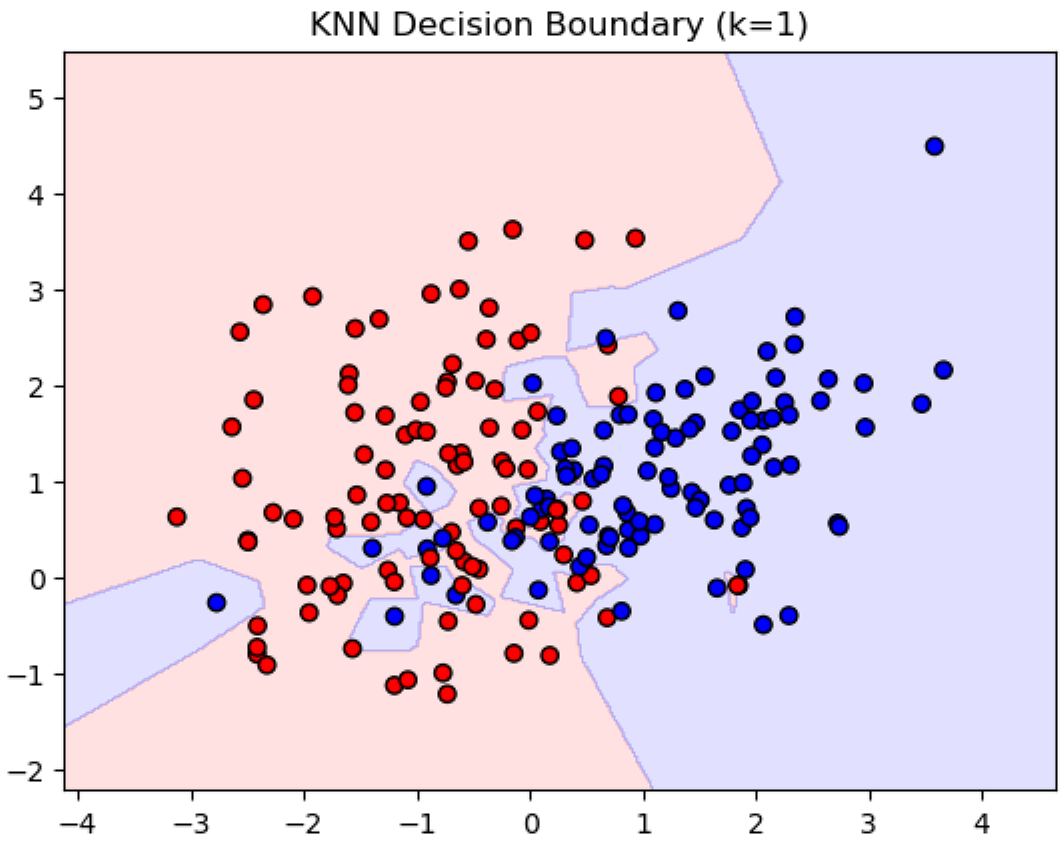


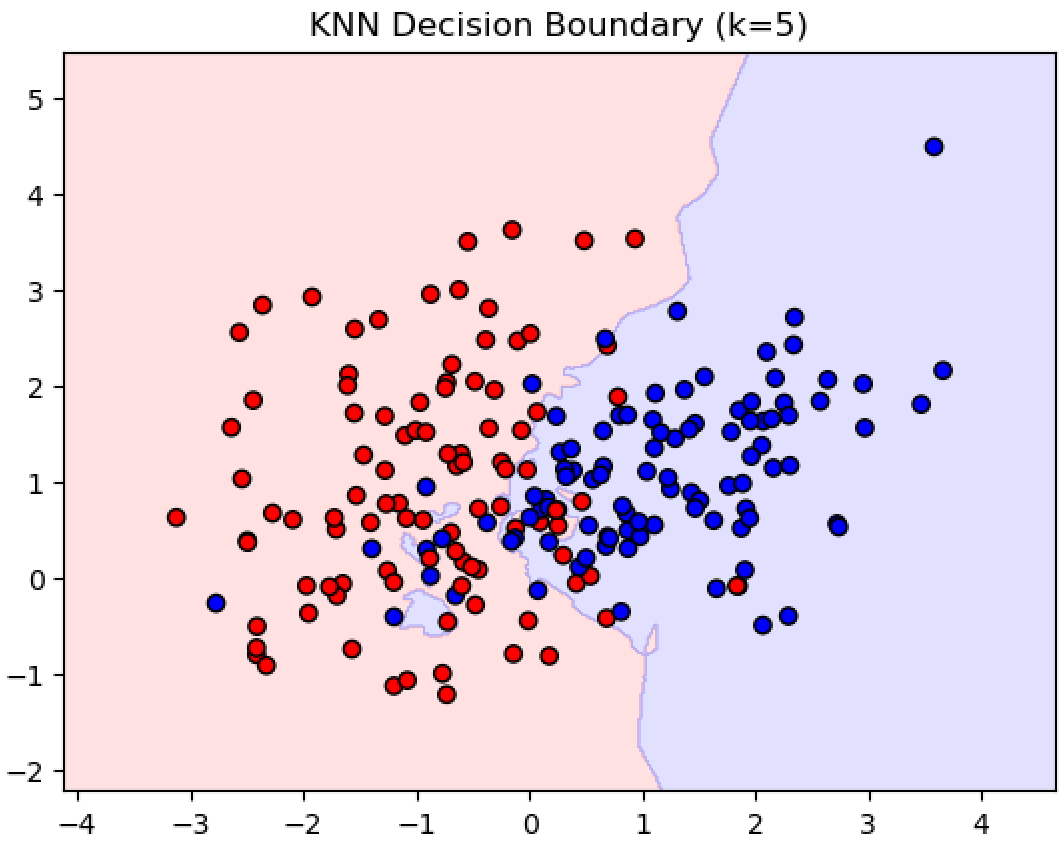
In [4]:
```
#Train a KNN Classifier using different distance metrics (Euclidean and Manhattan)


iris = load_iris()
X = iris.data
y = iris.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Standardize features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train KNN with Euclidean distance
knn_euclidean = KNeighborsClassifier(n_neighbors=5, metric='euclidean')
knn_euclidean.fit(X_train_scaled, y_train)
y_pred_euclidean = knn_euclidean.predict(X_test_scaled)
accuracy_euclidean = accuracy_score(y_test, y_pred_euclidean)

# Train KNN with Manhattan distance
knn_manhattan = KNeighborsClassifier(n_neighbors=5, metric='manhattan')
knn_manhattan.fit(X_train_scaled, y_train)
y_pred_manhattan = knn_manhattan.predict(X_test_scaled)
accuracy_manhattan = accuracy_score(y_test, y_pred_manhattan)

# Print results
```

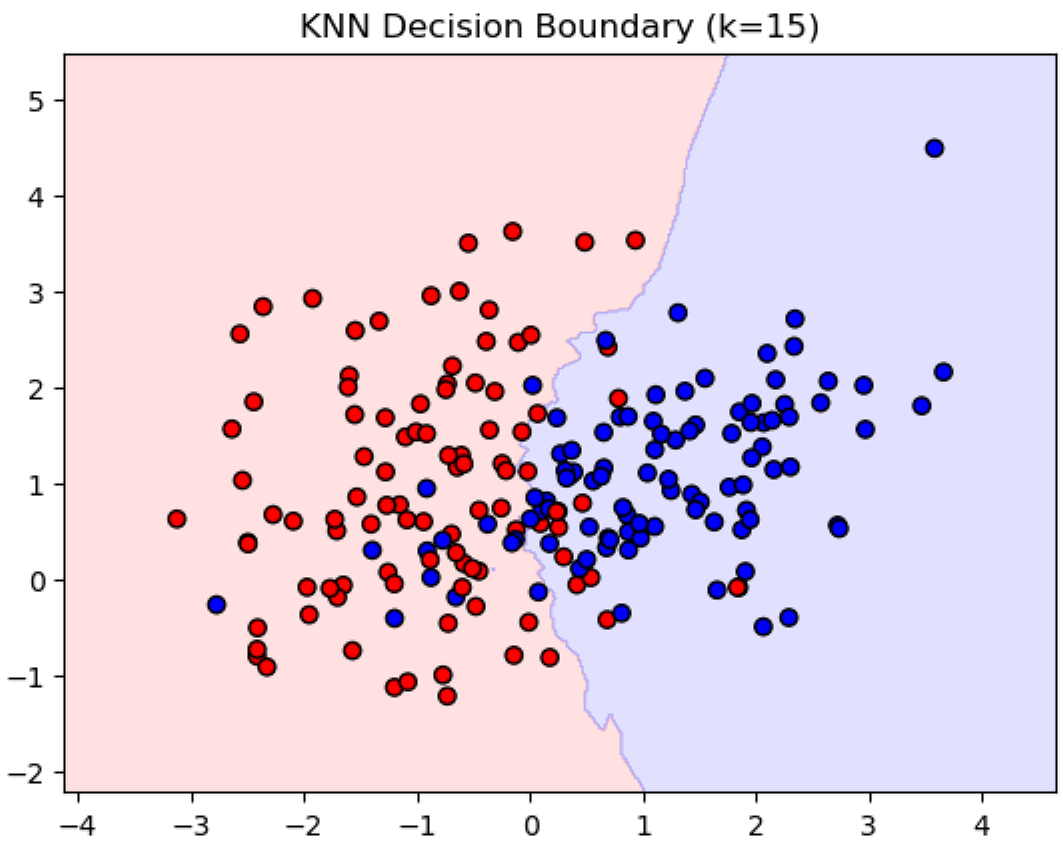
```
print(f"KNN Accuracy (Euclidean Distance): {accuracy_euclidean:.4f}")
print(f"KNN Accuracy (Manhattan Distance): {accuracy_manhattan:.4f}")
```

```
KNN Accuracy (Euclidean Distance): 1.0000
KNN Accuracy (Manhattan Distance): 1.0000
```

In [8]:
```
#Train a KNN Classifier with different values of K and visualize decision boundarie

from sklearn.datasets import make_classification

from matplotlib.colors import ListedColormap

# Generate synthetic dataset (Fix: Set n_redundant=0)
X, y = make_classification(n_samples=200, n_features=2, n_informative=2, n_redundan
                           n_classes=2, n_clusters_per_class=1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Function to plot decision boundaries
def plot_decision_boundary(X, y, model, title):
    h = 0.02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.3, cmap=ListedColormap(['#FFAAAA', '#AAAAFF']))
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', cmap=ListedColormap(['#FF000
    plt.title(title)
    plt.show()

# Train and visualize KNN for different values of K
k_values = [1, 5, 15]
for k in k_values:
    knn = KNeighborsClassifier(n_neighbors=k)
    knn.fit(X_train, y_train)
    plot_decision_boundary(X, y, knn, f"KNN Decision Boundary (k={k})")
```


KNN Decision Boundary (k=1)
5
4
3
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2
3
4


KNN Decision Boundary (k=5)
5
4
3
2
1
0
−1
−2
−4
−3
−2
−1
0
1
2
3
4

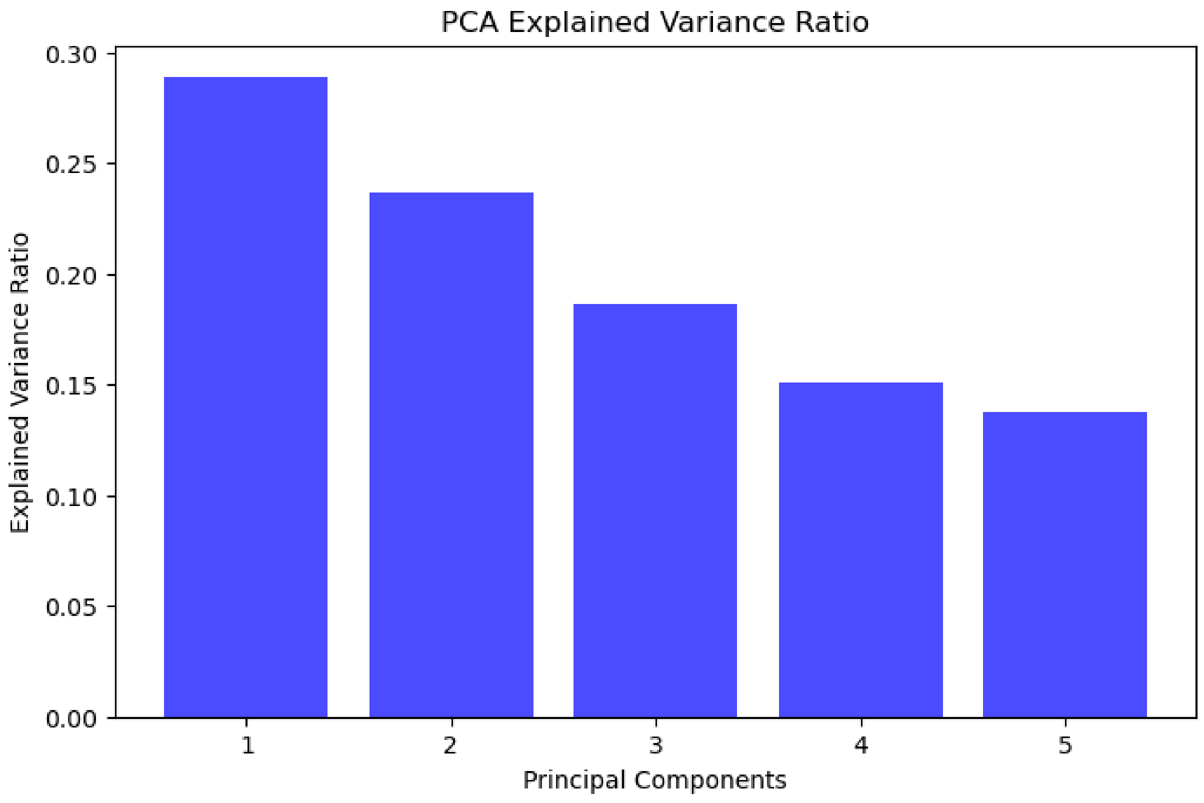


In [ ]:
```
#Apply Feature Scaling before training a KNN model and compare results with unscale


# Load dataset
iris = load_iris()
X = iris.data
y = iris.target

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Train KNN without scaling
knn_unscaled = KNeighborsClassifier(n_neighbors=5)
knn_unscaled.fit(X_train, y_train)
y_pred_unscaled = knn_unscaled.predict(X_test)
accuracy_unscaled = accuracy_score(y_test, y_pred_unscaled)

# Apply feature scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train KNN with scaling
knn_scaled = KNeighborsClassifier(n_neighbors=5)
knn_scaled.fit(X_train_scaled, y_train)
y_pred_scaled = knn_scaled.predict(X_test_scaled)
accuracy_scaled = accuracy_score(y_test, y_pred_scaled)

# Print comparison results
```

```
print(f"KNN Accuracy without Scaling: {accuracy_unscaled:.4f}")
print(f"KNN Accuracy with Scaling: {accuracy_scaled:.4f}")
```

In [10]:
```
#Train a PCA model on synthetic data and print the explained variance ratio for eac
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

# Generate synthetic data (100 samples, 5 features)
np.random.seed(42)
X = np.random.rand(100, 5) * 10  # Random values between 0 and 10

# Apply PCA
pca = PCA(n_components=5)  # Keep all components
pca.fit(X)

# Print explained variance ratio
print("Explained Variance Ratio for each Principal Component:")
for i, ratio in enumerate(pca.explained_variance_ratio_):
    print(f"Component {i+1}: {ratio:.4f}")

# Plot the explained variance ratio
plt.figure(figsize=(8, 5))
plt.bar(range(1, 6), pca.explained_variance_ratio_, alpha=0.7, color='b')
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.title('PCA Explained Variance Ratio')
plt.show()
```

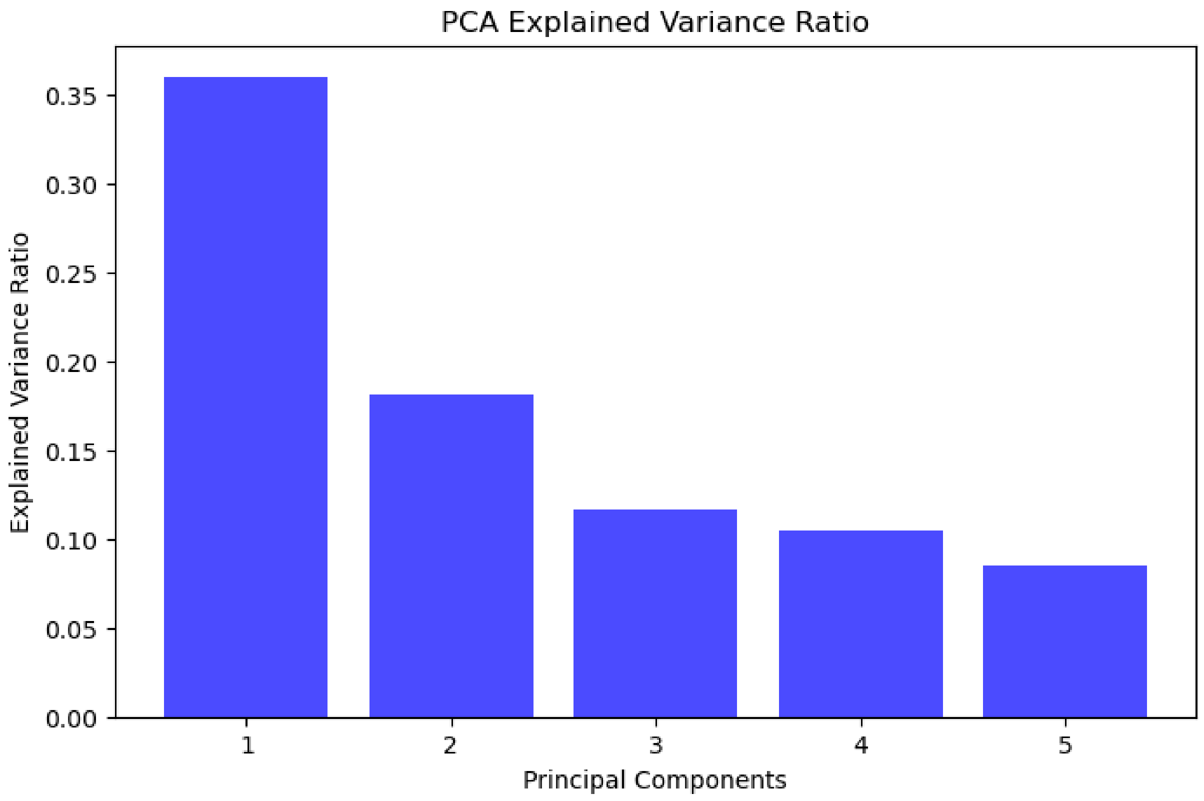
```
Explained Variance Ratio for each Principal Component:
Component 1: 0.2886
Component 2: 0.2364
Component 3: 0.1862
Component 4: 0.1510
Component 5: 0.1379
```



In [11]:
```
#Apply PCA before training a KNN Classifier and compare accuracy with and without P
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.datasets import make_classification

# Step 1: Generate synthetic dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes

# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN classifier without PCA
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
accuracy_without_pca = accuracy_score(y_test, y_pred)

# Step 4: Apply PCA (reduce to 5 components)
pca = PCA(n_components=5)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)

# Train KNN classifier on PCA-transformed data
knn_pca = KNeighborsClassifier(n_neighbors=5)
knn_pca.fit(X_train_pca, y_train)
y_pred_pca = knn_pca.predict(X_test_pca)
accuracy_with_pca = accuracy_score(y_test, y_pred_pca)
```

```
# Step 5: Compare accuracy
print(f"Accuracy without PCA: {accuracy_without_pca:.4f}")
print(f"Accuracy with PCA: {accuracy_with_pca:.4f}")

# Plot explained variance ratio
plt.figure(figsize=(8, 5))
plt.bar(range(1, 6), pca.explained_variance_ratio_, alpha=0.7, color='b')
plt.xlabel('Principal Components')
plt.ylabel('Explained Variance Ratio')
plt.title('PCA Explained Variance Ratio')
plt.show()
```

```
Accuracy without PCA: 0.7500
Accuracy with PCA: 0.6600
```



In [12]:
```
# Perform Hyperparameter Tuning on a KNN Classifier using GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import  GridSearchCV

# Step 1: Generate a synthetic dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes

# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Define KNN model
knn = KNeighborsClassifier()

# Step 4: Define the hyperparameter grid
param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],  # Number of neighbors
    'weights': ['uniform', 'distance'],  # Weighting method
    'metric': ['euclidean', 'manhattan', 'minkowski']  # Distance metric
}

# Step 5: Apply GridSearchCV
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)

# Step 6: Get best parameters and accuracy
best_params = grid_search.best_params_
best_model = grid_search.best_estimator_

# Step 7: Evaluate on the test set
y_pred = best_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Step 8: Print results
print(f"Best Hyperparameters: {best_params}")
print(f"Test Set Accuracy with Best Hyperparameters: {accuracy:.4f}")
```

```
Best Hyperparameters: {'metric': 'euclidean', 'n_neighbors': 5, 'weights': 'distanc
e'}
Test Set Accuracy with Best Hyperparameters: 0.7400
```

In [13]:

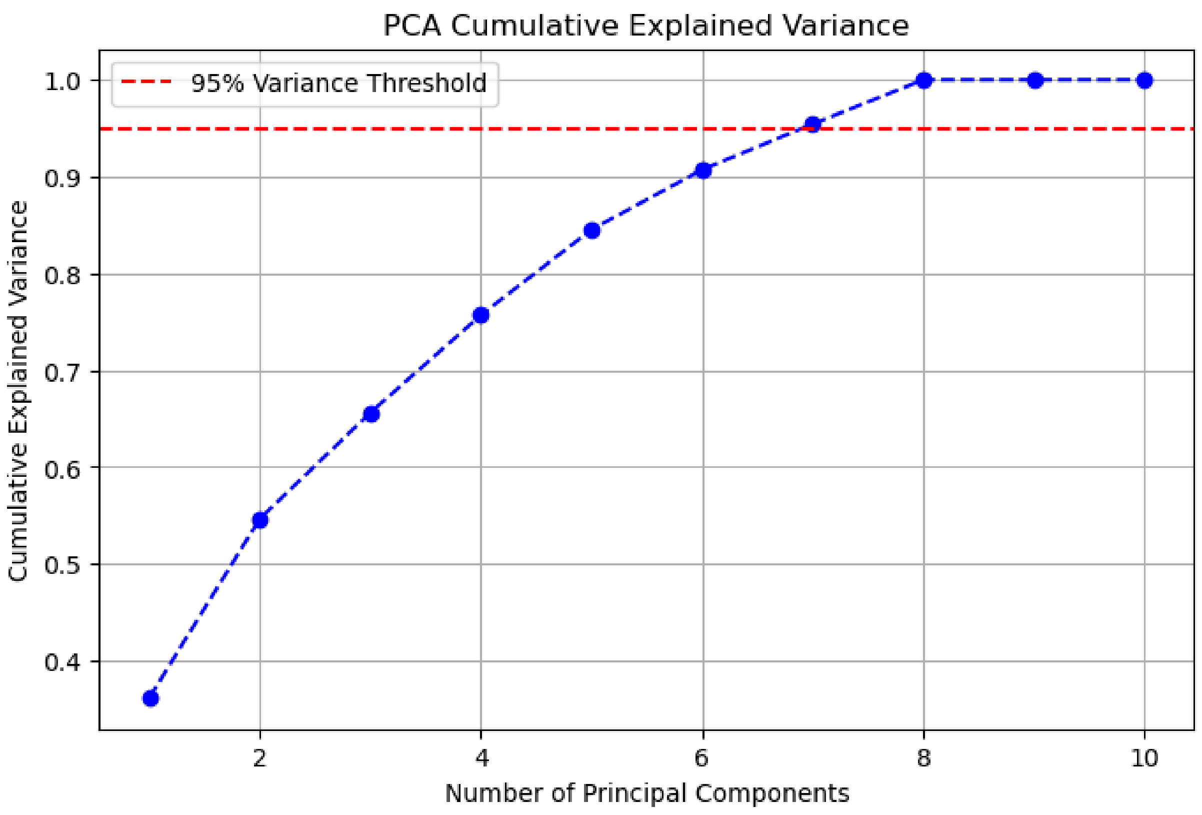
```
#Train a KNN Classifier and check the number of misclassified samples


# Step 1: Generate synthetic dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes

# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN Classifier (k=5)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Step 4: Make predictions
y_pred = knn.predict(X_test)

# Step 5: Compute accuracy
accuracy = accuracy_score(y_test, y_pred)

# Step 6: Calculate number of misclassified samples
misclassified_samples = (y_test != y_pred).sum()

# Step 7: Print results
print(f"Test Set Accuracy: {accuracy:.4f}")
print(f"Number of Misclassified Samples: {misclassified_samples} out of {len(y_test
```

```
Test Set Accuracy: 0.7500
Number of Misclassified Samples: 25 out of 100
```

In [14]:

```
#Train a PCA model and visualize the cumulative explained variance.

from sklearn.decomposition import PCA
```

```
# Step 1: Generate synthetic dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes

# Step 2: Apply PCA (keep all components)
pca = PCA()
pca.fit(X)

# Step 3: Compute cumulative explained variance
cumulative_variance = np.cumsum(pca.explained_variance_ratio_)

# Step 4: Plot cumulative explained variance
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(cumulative_variance) + 1), cumulative_variance, marker='o', l
plt.xlabel('Number of Principal Components')
plt.ylabel('Cumulative Explained Variance')
plt.title('PCA Cumulative Explained Variance')
plt.grid(True)
plt.axhline(y=0.95, color='r', linestyle='--', label="95% Variance Threshold")
plt.legend()
plt.show()
```

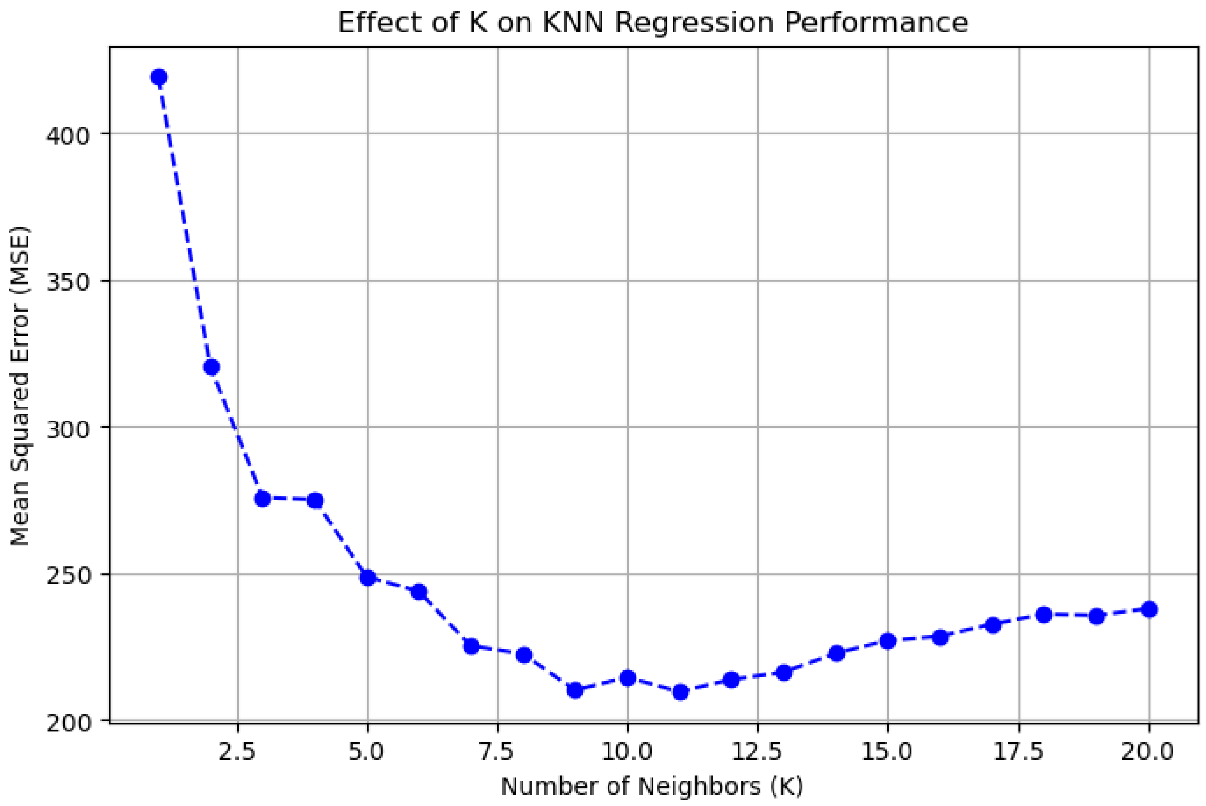


In [15]:
```
#Train a KNN Classifier using different values of the weights parameter (uniform vs


# Step 1: Generate a synthetic dataset
X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes

# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta
# Step 3: Train KNN classifier with "uniform" weights
knn_uniform = KNeighborsClassifier(n_neighbors=5, weights='uniform')
knn_uniform.fit(X_train, y_train)
y_pred_uniform = knn_uniform.predict(X_test)
accuracy_uniform = accuracy_score(y_test, y_pred_uniform)

# Step 4: Train KNN classifier with "distance" weights
knn_distance = KNeighborsClassifier(n_neighbors=5, weights='distance')
knn_distance.fit(X_train, y_train)
y_pred_distance = knn_distance.predict(X_test)
accuracy_distance = accuracy_score(y_test, y_pred_distance)

# Step 5: Compare accuracy
print(f"Accuracy with Uniform Weights: {accuracy_uniform:.4f}")
print(f"Accuracy with Distance Weights: {accuracy_distance:.4f}")
```

```
Accuracy with Uniform Weights: 0.7500
Accuracy with Distance Weights: 0.7400
```

In [16]:

```
# Train a KNN Regressor and analyze the effect of different K values on performance

from sklearn.neighbors import KNeighborsRegressor
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error

# Step 1: Generate synthetic regression data
X, y = make_regression(n_samples=500, n_features=1, noise=15, random_state=42)

# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN Regressors with different values of K
k_values = range(1, 21)  # Testing K values from 1 to 20
mse_values = []

for k in k_values:
    knn_reg = KNeighborsRegressor(n_neighbors=k)
    knn_reg.fit(X_train, y_train)
    y_pred = knn_reg.predict(X_test)
    mse = mean_squared_error(y_test, y_pred)
    mse_values.append(mse)

# Step 4: Plot K vs. MSE
plt.figure(figsize=(8, 5))
plt.plot(k_values, mse_values, marker='o', linestyle='--', color='b')
plt.xlabel("Number of Neighbors (K)")
plt.ylabel("Mean Squared Error (MSE)")
plt.title("Effect of K on KNN Regression Performance")
plt.grid(True)
plt.show()

# Step 5: Print best K value
best_k = k_values[np.argmin(mse_values)]
print(f"Best K value: {best_k} (Lowest MSE: {min(mse_values):.4f})")
```



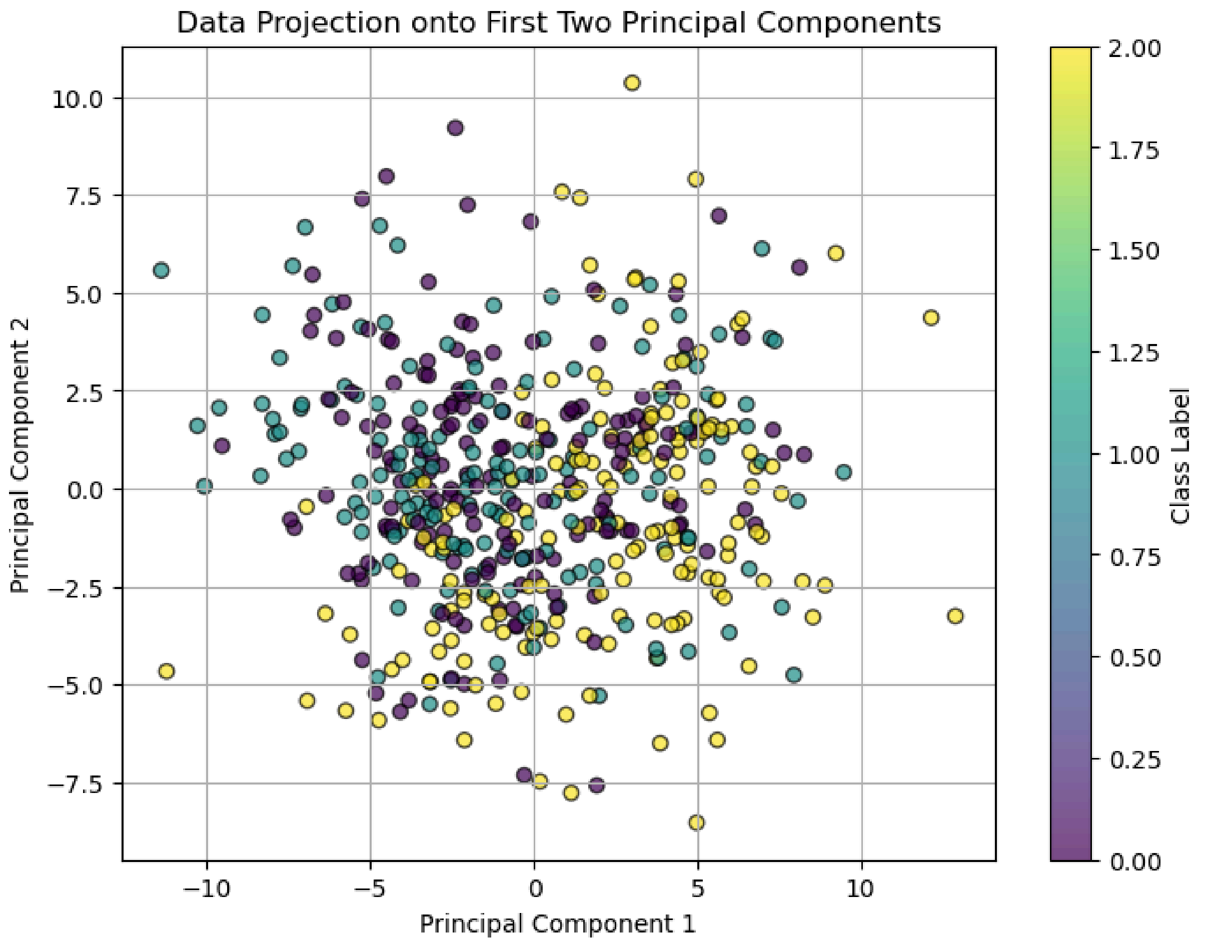
```
Best K value: 11 (Lowest MSE: 209.7065)
```

In [17]:
```python
# Implement KNN Imputation for handling missing values in a dataset

from sklearn.impute import KNNImputer

# Step 1: Create a synthetic dataset with missing values
np.random.seed(42)
data = np.random.rand(10, 5) * 10  # 10 samples, 5 features
data[2, 1] = np.nan  # Introduce missing value
data[5, 3] = np.nan
data[7, 4] = np.nan

df = pd.DataFrame(data, columns=[f'Feature_{i+1}' for i in range(5)])
print("Original Data with Missing Values:\n", df)

# Step 2: Apply KNN Imputation
knn_imputer = KNNImputer(n_neighbors=3)  # Using K=3 for imputation
imputed_data = knn_imputer.fit_transform(df)

# Step 3: Convert back to DataFrame
df_imputed = pd.DataFrame(imputed_data, columns=df.columns)
print("\nData After KNN Imputation:\n", df_imputed)
```

```
Original Data with Missing Values:
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5
0   3.745401   9.507143   7.319939   5.986585   1.560186
1   1.559945   0.580836   8.661761   6.011150   7.080726
2   0.205845        NaN   8.324426   2.123391   1.818250
3   1.834045   3.042422   5.247564   4.319450   2.912291
4   6.118529   1.394939   2.921446   3.663618   4.560700
5   7.851760   1.996738   5.142344        NaN   0.464504
6   6.075449   1.705241   0.650516   9.488855   9.656320
7   8.083973   3.046138   0.976721   6.842330        NaN
8   1.220382   4.951769   0.343885   9.093204   2.587800
9   6.625223   3.117111   5.200680   5.467103   1.848545

Data After KNN Imputation:
   Feature_1  Feature_2  Feature_3  Feature_4  Feature_5
0   3.745401   9.507143   7.319939   5.986585   1.560186
1   1.559945   0.580836   8.661761   6.011150   7.080726
2   0.205845   4.376801   8.324426   2.123391   1.818250
3   1.834045   3.042422   5.247564   4.319450   2.912291
4   6.118529   1.394939   2.921446   3.663618   4.560700
5   7.851760   1.996738   5.142344   5.324350   0.464504
6   6.075449   1.705241   0.650516   9.488855   9.656320
7   8.083973   3.046138   0.976721   6.842330   5.355188
8   1.220382   4.951769   0.343885   9.093204   2.587800
9   6.625223   3.117111   5.200680   5.467103   1.848545
```

In [18]:

```python
#Train a PCA model and visualize the data projection onto the first two principal c

X, y = make_classification(n_samples=500, n_features=10, n_informative=8, n_classes


pca = PCA(n_components=2)
X_pca = pca.fit_transform(X)


plt.figure(figsize=(8, 6))
scatter = plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', edgecolors='k'
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.title("Data Projection onto First Two Principal Components")
plt.colorbar(label="Class Label")
plt.grid(True)
plt.show()
```

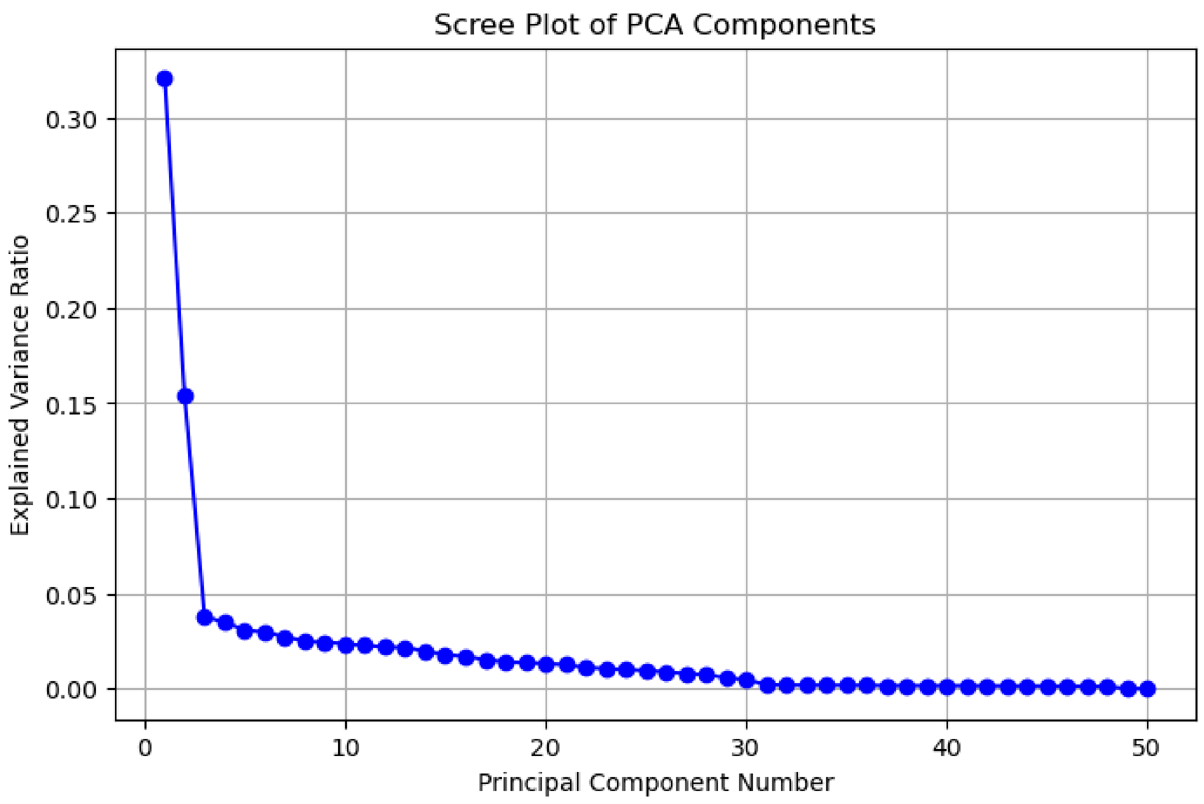


In [19]:
```
# Train a KNN Classifier using the KD Tree and Ball Tree algorithms and compare per

import time


# Step 1: Generate synthetic dataset
X, y = make_classification(n_samples=1000, n_features=10, n_informative=8, n_classe

# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN Classifier using KD Tree
start_time = time.time()
knn_kd = KNeighborsClassifier(n_neighbors=5, algorithm='kd_tree')
knn_kd.fit(X_train, y_train)
y_pred_kd = knn_kd.predict(X_test)
kd_time = time.time() - start_time
kd_accuracy = accuracy_score(y_test, y_pred_kd)

# Step 4: Train KNN Classifier using Ball Tree
start_time = time.time()
knn_ball = KNeighborsClassifier(n_neighbors=5, algorithm='ball_tree')
knn_ball.fit(X_train, y_train)
y_pred_ball = knn_ball.predict(X_test)
ball_time = time.time() - start_time
ball_accuracy = accuracy_score(y_test, y_pred_ball)
```

```
# Step 5: Compare Performance
print(f"KD Tree - Accuracy: {kd_accuracy:.4f}, Training Time: {kd_time:.4f} sec")
print(f"Ball Tree - Accuracy: {ball_accuracy:.4f}, Training Time: {ball_time:.4f} s
```

```
KD Tree - Accuracy: 0.9000, Training Time: 0.0143 sec
Ball Tree - Accuracy: 0.9000, Training Time: 0.0178 sec
```

In [20]:
```
# Train a PCA model on a high-dimensional dataset and visualize the Scree plot


# Step 1: Generate a high-dimensional dataset (500 samples, 50 features)
X, y = make_classification(n_samples=500, n_features=50, n_informative=30, random_s

# Step 2: Apply PCA (keeping all components)
pca = PCA()
pca.fit(X)

# Step 3: Explained variance ratio (Scree plot)
plt.figure(figsize=(8, 5))
plt.plot(range(1, len(pca.explained_variance_ratio_) + 1), pca.explained_variance_r
plt.xlabel("Principal Component Number")
plt.ylabel("Explained Variance Ratio")
plt.title("Scree Plot of PCA Components")
plt.grid(True)
plt.show()
```

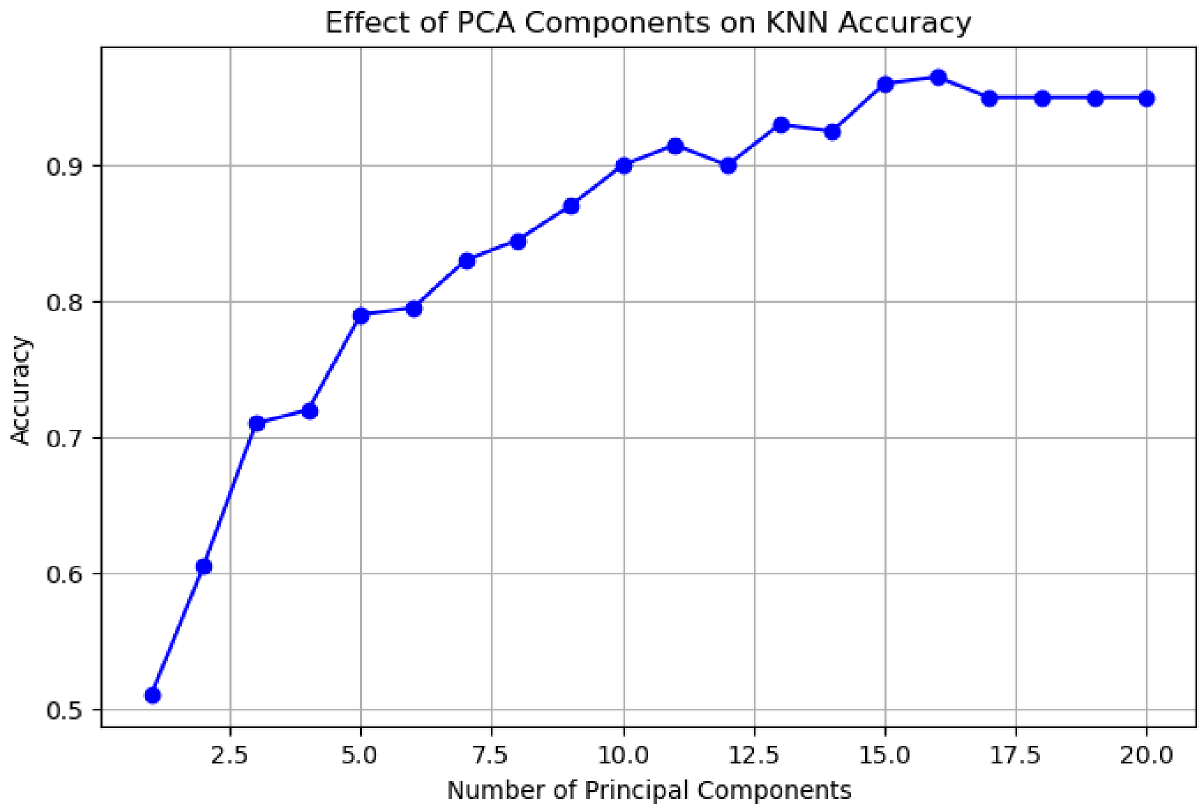


In [21]:
```
# Train a KNN Classifier and evaluate performance using Precision, Recall, and F1-S

from sklearn.metrics import precision_score, recall_score, f1_score, classification

# Step 1: Generate synthetic dataset (multi-class classification)
X, y = make_classification(n_samples=1000, n_features=10, n_informative=8, n_classe
# Step 2: Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)

# Step 4: Evaluate Performance
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')

print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1-Score: {f1:.4f}")

# Print detailed classification report
print("\nClassification Report:\n", classification_report(y_test, y_pred))
```

```
Precision: 0.9001
Recall: 0.9000
F1-Score: 0.8999

Classification Report:
               precision    recall  f1-score   support

           0       0.90      0.93      0.91        67
           1       0.91      0.89      0.90        70
           2       0.89      0.89      0.89        63

    accuracy                           0.90       200
   macro avg       0.90      0.90      0.90       200
weighted avg       0.90      0.90      0.90       200
```

In [22]:
```
# Train a PCA model and analyze the effect of different numbers of components on ac


# Step 1: Generate synthetic classification data
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, random_

# Step 2: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Test different numbers of PCA components
components_list = range(1, 21)  # Try PCA with 1 to 20 components
accuracy_scores = []

for n_components in components_list:
    # Apply PCA
    pca = PCA(n_components=n_components)
    X_train_pca = pca.fit_transform(X_train)
    X_test_pca = pca.transform(X_test)
    # Train KNN Classifier
    knn = KNeighborsClassifier(n_neighbors=5)
    knn.fit(X_train_pca, y_train)
    y_pred = knn.predict(X_test_pca)

    # Calculate accuracy
    accuracy = accuracy_score(y_test, y_pred)
    accuracy_scores.append(accuracy)

# Step 4: Plot Accuracy vs. Number of Components
plt.figure(figsize=(8, 5))
plt.plot(components_list, accuracy_scores, marker='o', linestyle='-', color='b')
plt.xlabel("Number of Principal Components")
plt.ylabel("Accuracy")
plt.title("Effect of PCA Components on KNN Accuracy")
plt.grid(True)
plt.show()

# Step 5: Print best number of components
best_components = components_list[np.argmax(accuracy_scores)]
print(f"Best number of PCA components: {best_components} (Highest Accuracy: {max(ac
```

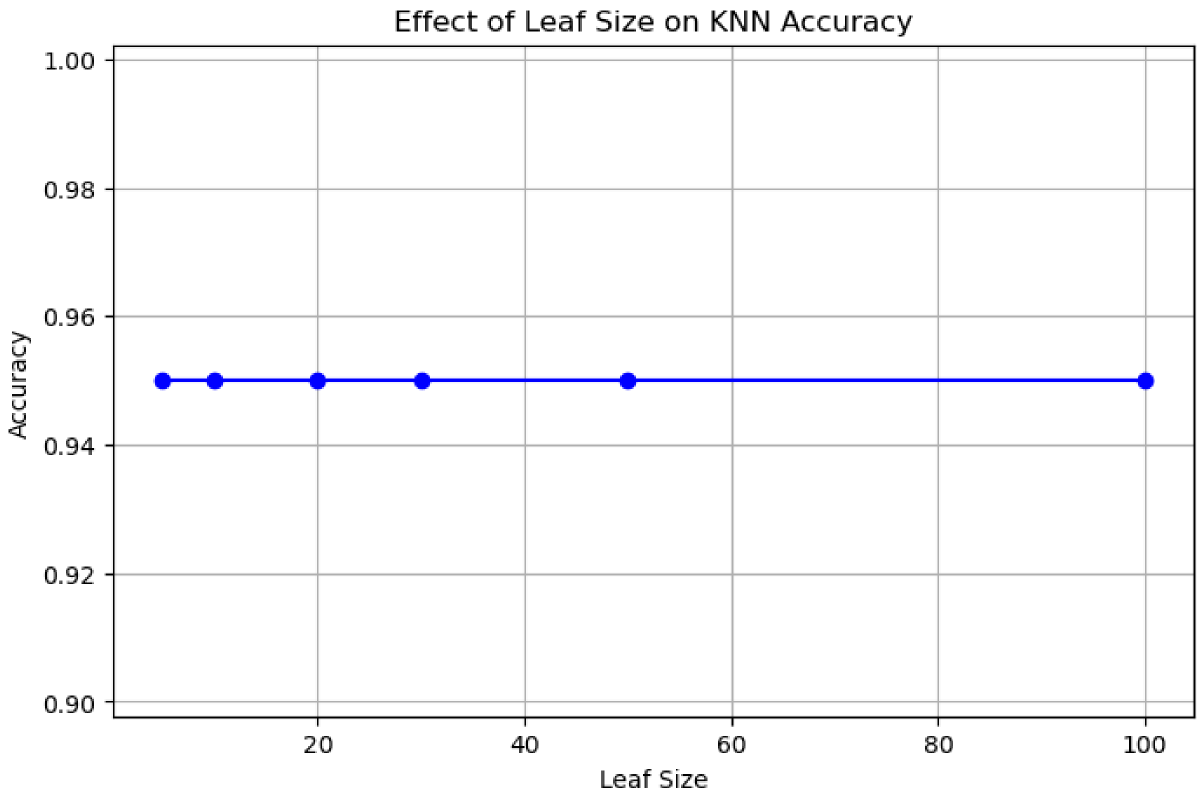
```
/opt/conda/envs/anaconda-2024.02-py310/lib/python3.10/site-packages/joblib/external
s/loky/backend/context.py:110: UserWarning: Could not find the number of physical co
res for the following reason:
found 0 physical cores < 1
Returning the number of logical cores instead. You can silence this warning by setti
ng LOKY_MAX_CPU_COUNT to the number of cores you want to use.
  warnings.warn(
  File "/opt/conda/envs/anaconda-2024.02-py310/lib/python3.10/site-packages/joblib/e
xternals/loky/backend/context.py", line 217, in _count_physical_cores
    raise ValueError(
```



```
Best number of PCA components: 16 (Highest Accuracy: 0.9650)
```

In [23]:
```
#Train a KNN Classifier with different leaf_size values and compare accuracy


# Step 1: Generate synthetic classification dataset
X, y = make_classification(n_samples=1000, n_features=20, n_informative=15, random_

# Step 2: Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN with different leaf_size values
leaf_sizes = [5, 10, 20, 30, 50, 100]  # Different values to test
accuracy_scores = []

for leaf in leaf_sizes:
    knn = KNeighborsClassifier(n_neighbors=5, leaf_size=leaf)  # Keeping k=5 consta
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    accuracy_scores.append(accuracy)
    print(f"Leaf Size {leaf} -> Accuracy: {accuracy:.4f}")

# Step 4: Plot Accuracy vs. Leaf Size
plt.figure(figsize=(8, 5))
plt.plot(leaf_sizes, accuracy_scores, marker='o', linestyle='-', color='b')
plt.xlabel("Leaf Size")
plt.ylabel("Accuracy")
plt.title("Effect of Leaf Size on KNN Accuracy")
plt.grid(True)
plt.show()

# Step 5: Print the best leaf size
best_leaf_size = leaf_sizes[np.argmax(accuracy_scores)]
print(f"\nBest leaf_size: {best_leaf_size} (Highest Accuracy: {max(accuracy_scores)
```

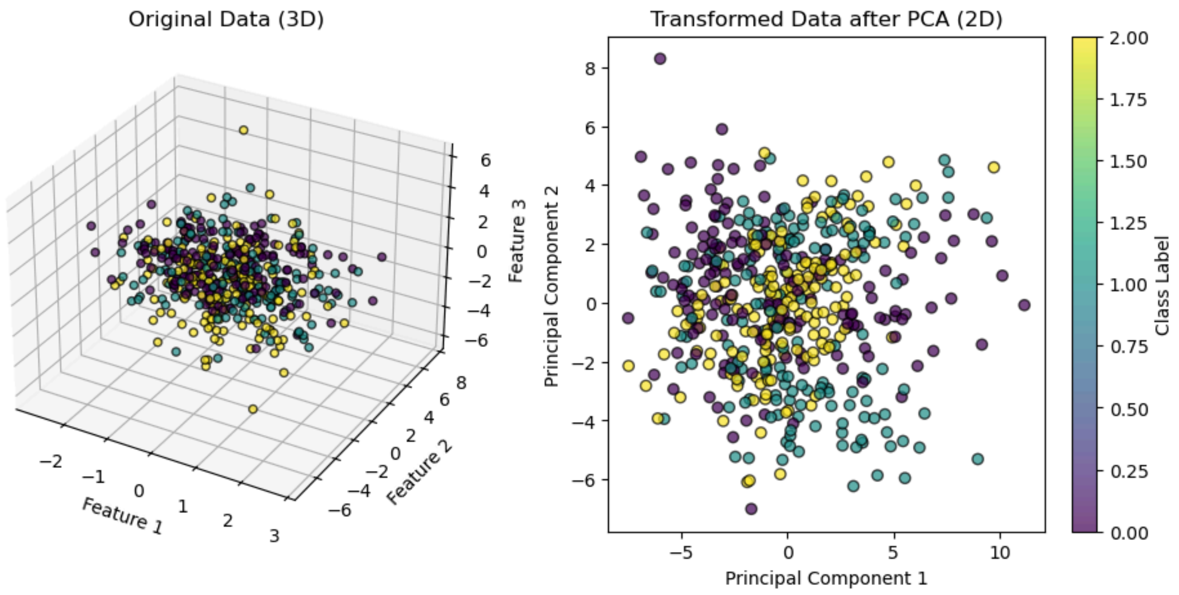
```
Leaf Size 5 -> Accuracy: 0.9500
Leaf Size 10 -> Accuracy: 0.9500
Leaf Size 20 -> Accuracy: 0.9500
Leaf Size 30 -> Accuracy: 0.9500
Leaf Size 50 -> Accuracy: 0.9500
Leaf Size 100 -> Accuracy: 0.9500
```



```
Best leaf_size: 5 (Highest Accuracy: 0.9500)
```

In [25]:
```
#Train a PCA model and visualize how data points are transformed before and after P
from sklearn.datasets import make_classification

# Step 1: Generate synthetic dataset (3 features for visualization)
X, y = make_classification(n_samples=500, n_features=10, n_informative=6, n_redunda

# Step 2: Scatter Plot of Original 3D Data
fig = plt.figure(figsize=(12, 5))

ax = fig.add_subplot(121, projection='3d')
scatter = ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap='viridis', edgecolors='k'
ax.set_xlabel("Feature 1")
ax.set_ylabel("Feature 2")
ax.set_zlabel("Feature 3")
ax.set_title("Original Data (3D)")

# Step 3: Apply PCA (reduce to 2 components)
pca = PCA(n_components=2)
```

```
X_pca = pca.fit_transform(X)

# Step 4: Scatter Plot of Transformed 2D Data
ax2 = fig.add_subplot(122)
scatter2 = ax2.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis', edgecolors='k
ax2.set_xlabel("Principal Component 1")
ax2.set_ylabel("Principal Component 2")
ax2.set_title("Transformed Data after PCA (2D)")
plt.colorbar(scatter2, ax=ax2, label="Class Label")

plt.show()
```

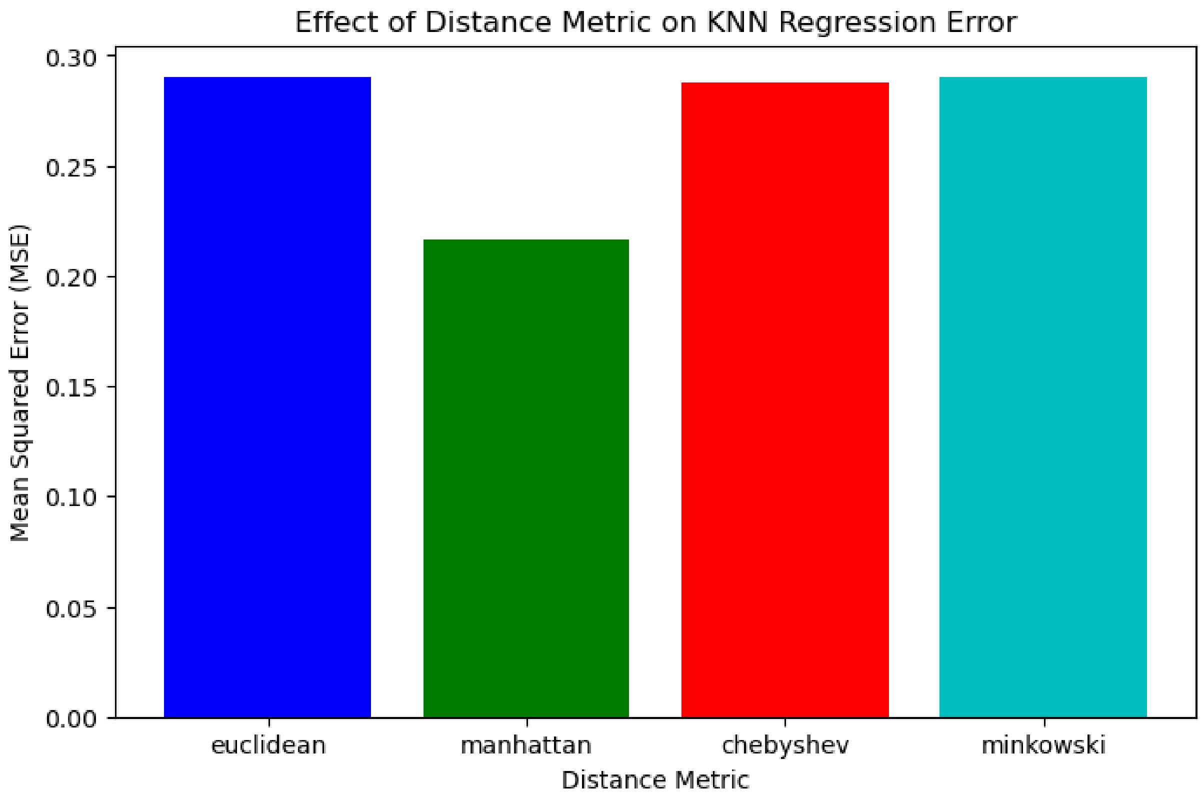


In [26]:
```
#Train a KNN Classifier on a real-world dataset (Wine dataset) and print classifica

from sklearn.datasets import load_wine
from sklearn.metrics import classification_report

# Step 1: Load the Wine dataset
wine = load_wine()
X = wine.data  # Features
y = wine.target  # Labels

# Step 2: Split dataset (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN Classifier (k=5)
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Step 4: Make Predictions
y_pred = knn.predict(X_test)

# Step 5: Print Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred, target_name
Classification Report:
               precision    recall  f1-score   support

     class_0       1.00      1.00      1.00        12
     class_1       0.77      0.71      0.74        14
     class_2       0.64      0.70      0.67        10

    accuracy                           0.81        36
   macro avg       0.80      0.80      0.80        36
weighted avg       0.81      0.81      0.81        36
```

In [28]:

```
#Train a KNN Regressor and analyze the effect of different distance metrics on pred
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_wine
from sklearn.metrics import mean_squared_error

data = load_wine()
X, y = data.data, data.target

# Step 2: Split dataset (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_sta

# Step 3: Train KNN Regressor with different distance metrics
distance_metrics = ['euclidean', 'manhattan', 'chebyshev', 'minkowski']
errors = []

for metric in distance_metrics:
    knn = KNeighborsRegressor(n_neighbors=5, metric=metric)
    knn.fit(X_train, y_train)
    y_pred = knn.predict(X_test)

    mse = mean_squared_error(y_test, y_pred)
    errors.append(mse)
    print(f"Metric: {metric} -> MSE: {mse:.4f}")

# Step 4: Plot Prediction Error vs. Distance Metric
plt.figure(figsize=(8, 5))
plt.bar(distance_metrics, errors, color=['b', 'g', 'r', 'c'])
plt.xlabel("Distance Metric")
plt.ylabel("Mean Squared Error (MSE)")
plt.title("Effect of Distance Metric on KNN Regression Error")
plt.show()

# Step 5: Print best metric
best_metric = distance_metrics[np.argmin(errors)]
print(f"\nBest distance metric: {best_metric} (Lowest MSE: {min(errors):.4f})")
```

```
Metric: euclidean -> MSE: 0.2900
Metric: manhattan -> MSE: 0.2167
Metric: chebyshev -> MSE: 0.2878
Metric: minkowski -> MSE: 0.2900
```

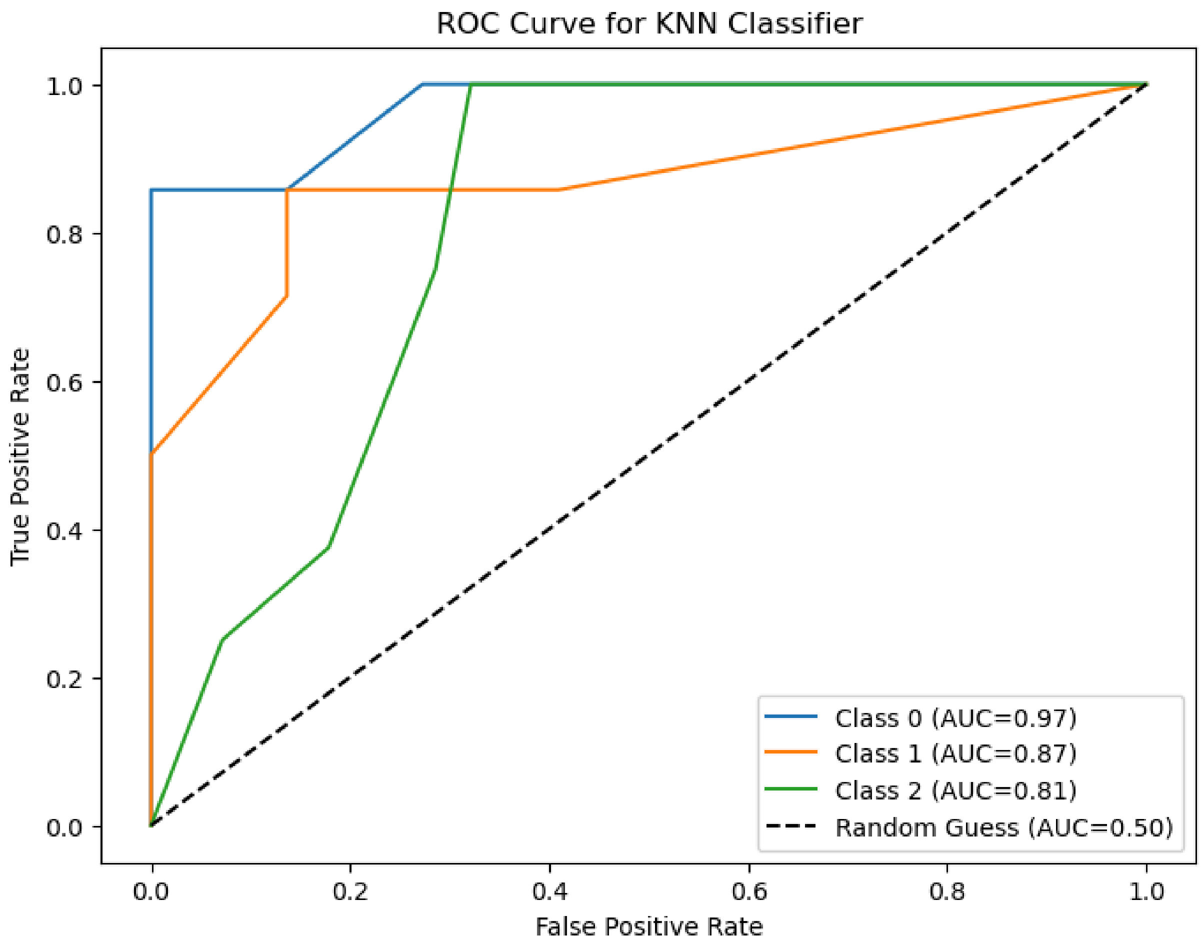


```
Best distance metric: manhattan (Lowest MSE: 0.2167)
```

In [32]:
```python
#Train a KNN Classifier and evaluate using ROC-AUC score
import numpy as np
import matplotlib.pyplot as plt
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_wine
from sklearn.metrics import roc_auc_score, roc_curve
from sklearn.preprocessing import label_binarize

# Load dataset
wine = load_wine()
X = wine.data
y = wine.target

# Convert to binary format (One-vs-Rest)
y_bin = label_binarize(y, classes=[0, 1, 2])

# Split dataset
X_train, X_test, y_train, y_test = train_test_split(X, y_bin, test_size=0.2, random

# Train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Get predicted probabilities
y_scores = np.array(knn.predict_proba(X_test))  # Convert to NumPy array

# Debugging: Print the shape
print("y_scores shape:", y_scores.shape)  # Should match y_test shape
```

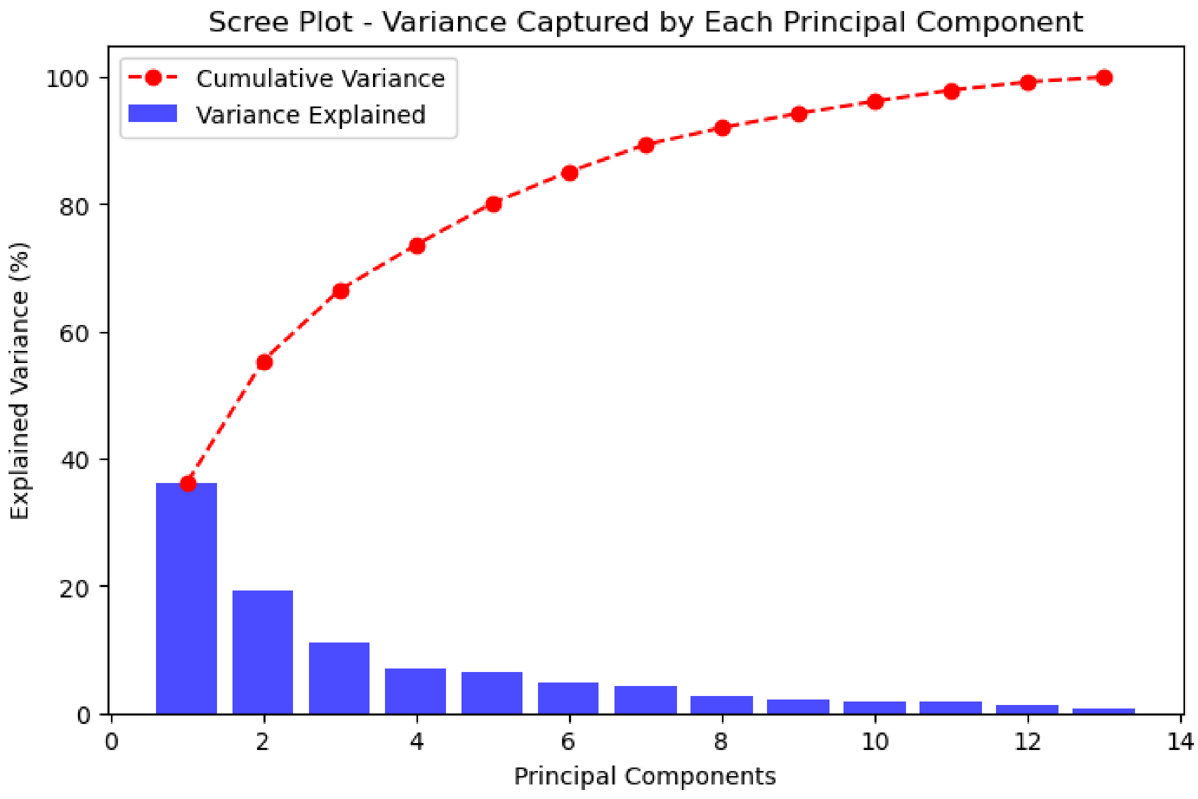
```
# Compute AUC for each class
auc_scores = []
plt.figure(figsize=(8, 6))

for i in range(y_bin.shape[1]):
    fpr, tpr, _ = roc_curve(y_test[:, i], y_scores[i][:, 1])  # Fix indexing
    auc = roc_auc_score(y_test[:, i], y_scores[i][:, 1])
    auc_scores.append(auc)
    plt.plot(fpr, tpr, label=f'Class {i} (AUC={auc:.2f})')

# Plot ROC Curve
plt.plot([0, 1], [0, 1], 'k--', label="Random Guess (AUC=0.50)")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.title("ROC Curve for KNN Classifier")
plt.legend()
plt.show()

# Print AUC scores
print(f"Mean AUC Score: {np.mean(auc_scores):.4f}")
```

```
y_scores shape: (3, 36, 2)
```



```
Mean AUC Score: 0.8828
```

In [33]:
```
#Train a PCA model and visualize the variance captured by each principal component
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_wine

# Load dataset
wine = load_wine()
X = wine.data

# Standardize the data (PCA is sensitive to scale)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train PCA model
pca = PCA(n_components=X.shape[1])  # Keep all components
X_pca = pca.fit_transform(X_scaled)

# Get explained variance ratio
explained_variance = pca.explained_variance_ratio_

# Plot Scree plot
plt.figure(figsize=(8, 5))
plt.bar(range(1, len(explained_variance) + 1), explained_variance * 100, alpha=0.7,
plt.plot(range(1, len(explained_variance) + 1), np.cumsum(explained_variance) * 100
plt.xlabel("Principal Components")
plt.ylabel("Explained Variance (%)")
plt.title("Scree Plot - Variance Captured by Each Principal Component")
plt.legend()
plt.show()
```



In [34]:
```
#Train a KNN Classifier and perform feature selection before training
import numpy as np
import matplotlib.pyplot as plt
```

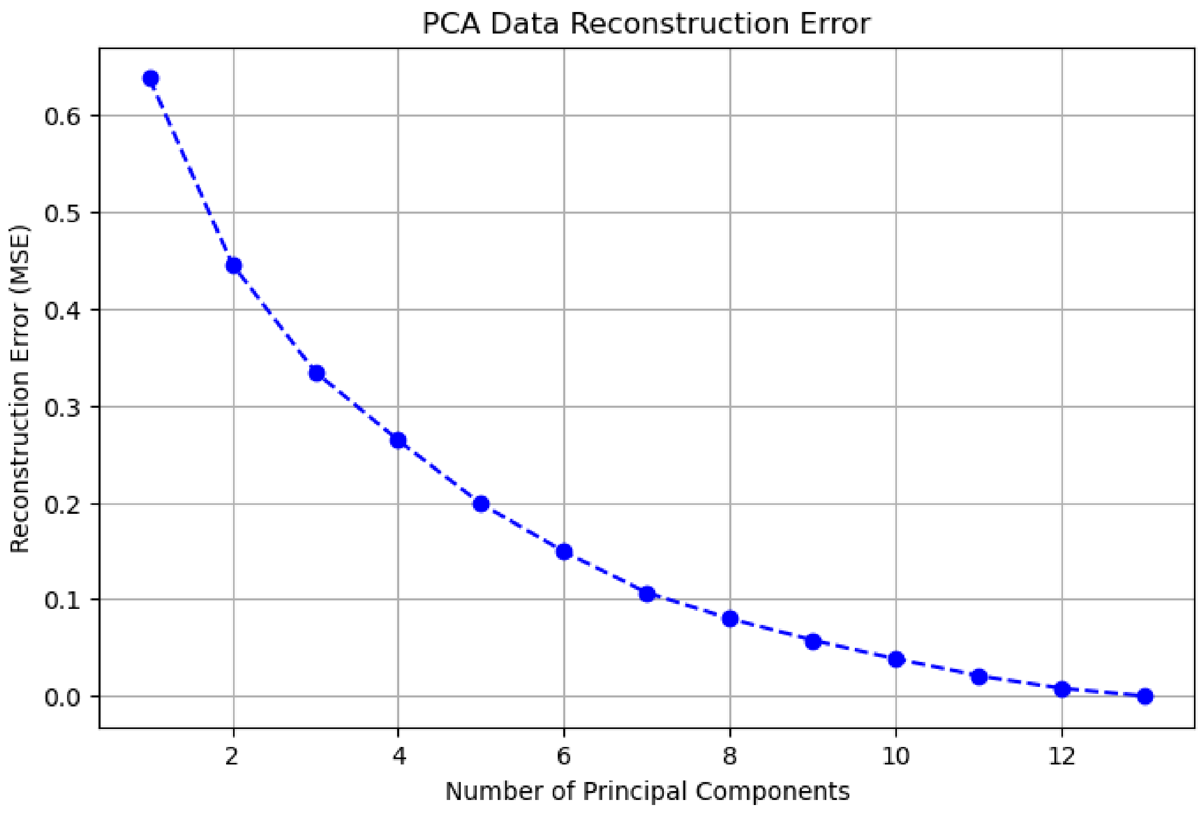
```
from sklearn.datasets import load_wine
from sklearn.preprocessing import StandardScaler
from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Load dataset
wine = load_wine()
X, y = wine.data, wine.target

# Standardize the data (KNN is distance-based, so scaling is necessary)
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Feature Selection - Select Top K Features
k = 5  # Choose top 5 best features
selector = SelectKBest(score_func=f_classif, k=k)
X_selected = selector.fit_transform(X_scaled, y)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2, r

# Train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Predict & Evaluate
y_pred = knn.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)

# Print Results
print(f"Selected Features: {np.array(wine.feature_names)[selector.get_support()]}")
print(f"Accuracy with Feature Selection: {accuracy:.4f}")
```

```
Selected Features: ['alcohol' 'flavanoids' 'color_intensity' 'od280/od315_of_diluted
_wines'
 'proline']
Accuracy with Feature Selection: 0.9722
```

In [35]:

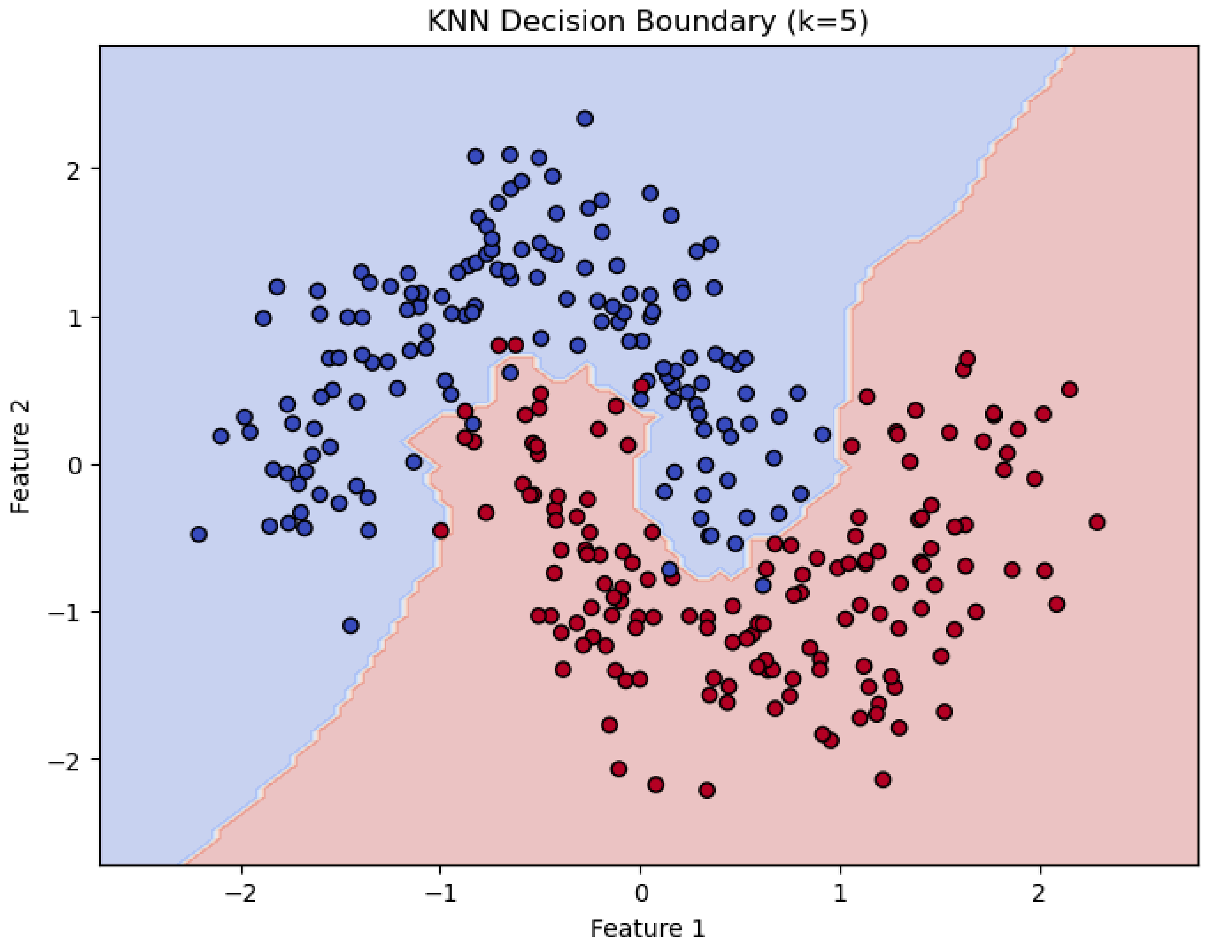
```
#Train a PCA model and visualize the data reconstruction error after reducing dimen
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_wine
from sklearn.metrics import mean_squared_error

# Load dataset
wine = load_wine()
X = wine.data

# Standardize features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
# Try different numbers of PCA components
components_range = range(1, X.shape[1] + 1)
reconstruction_errors = []

for n_components in components_range:
    pca = PCA(n_components=n_components)
    X_pca = pca.fit_transform(X_scaled)  # Reduce dimensions
    X_reconstructed = pca.inverse_transform(X_pca)  # Reconstruct data
    error = mean_squared_error(X_scaled, X_reconstructed)  # Compute reconstruction
    reconstruction_errors.append(error)

# Plot Reconstruction Error vs. Number of Components
plt.figure(figsize=(8, 5))
plt.plot(components_range, reconstruction_errors, marker='o', linestyle='--', color
plt.xlabel('Number of Principal Components')
plt.ylabel('Reconstruction Error (MSE)')
plt.title('PCA Data Reconstruction Error')
plt.grid()
plt.show()
```

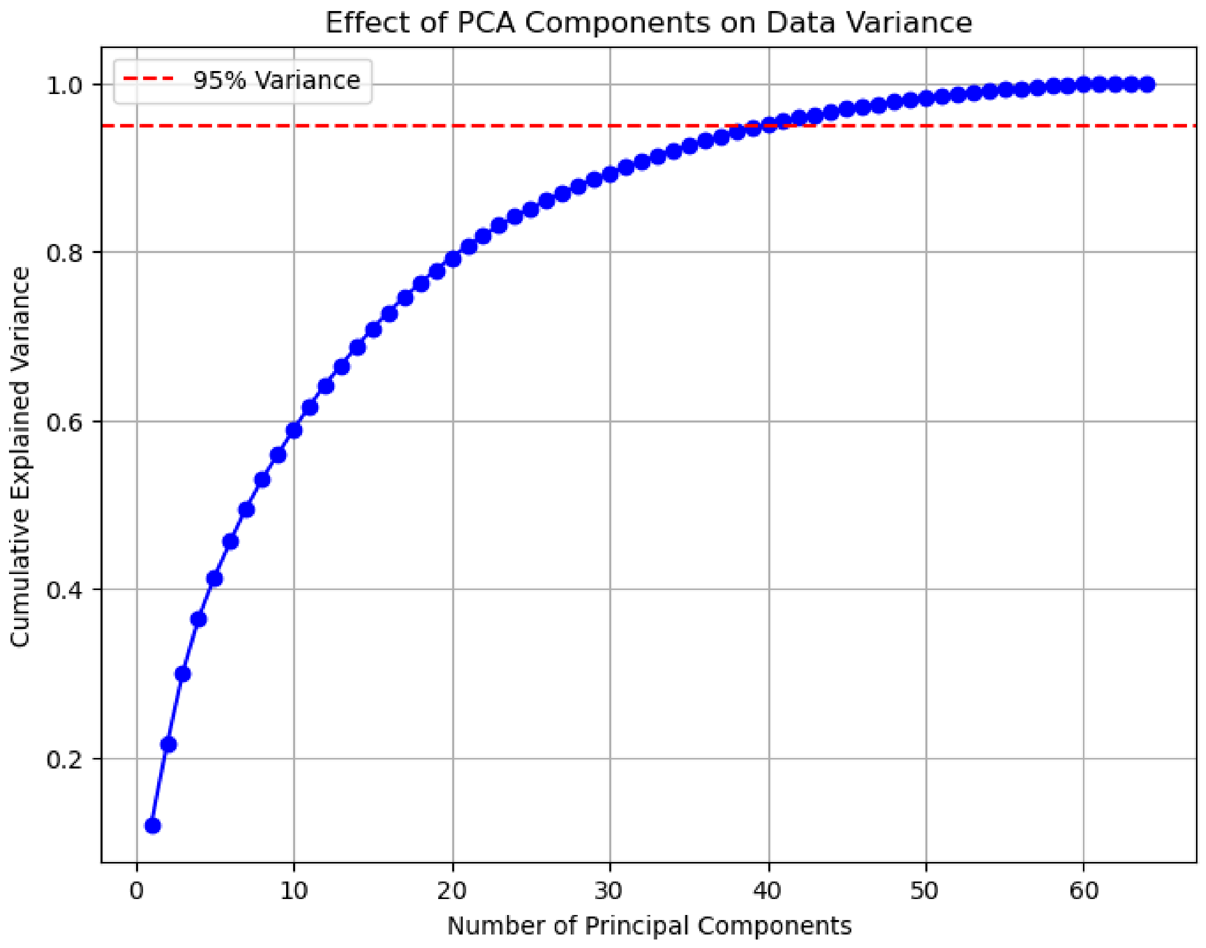


In [36]:
```
#Train a KNN Classifier and visualize the decision boundary
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_moons
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler

# Generate synthetic dataset
X, y = make_moons(n_samples=300, noise=0.2, random_state=42)

# Standardize features
```

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Train KNN Classifier
knn = KNeighborsClassifier(n_neighbors=5, weights='uniform')
knn.fit(X_scaled, y)

# Create mesh grid for visualization
x_min, x_max = X_scaled[:, 0].min() - 0.5, X_scaled[:, 0].max() + 0.5
y_min, y_max = X_scaled[:, 1].min() - 0.5, X_scaled[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100), np.linspace(y_min, y_max, 100)

# Predict on grid points
Z = knn.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# Plot decision boundary
plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.coolwarm)
plt.scatter(X_scaled[:, 0], X_scaled[:, 1], c=y, edgecolors='k', cmap=plt.cm.coolwa
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("KNN Decision Boundary (k=5)")
plt.show()
```



In [37]:
```
# Train a PCA model and analyze the effect of different numbers of components on da
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.datasets import load_digits
from sklearn.preprocessing import StandardScaler

# Load dataset (digits dataset with 64 features)
digits = load_digits()
X = digits.data

# Standardize the dataset
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Apply PCA with varying number of components
n_components = np.arange(1, X.shape[1] + 1)
explained_variances = []

for n in n_components:
    pca = PCA(n_components=n)
    pca.fit(X_scaled)
    explained_variances.append(np.sum(pca.explained_variance_ratio_))  # Cumulative

# Plot explained variance vs. number of components
plt.figure(figsize=(8, 6))
plt.plot(n_components, explained_variances, marker='o', linestyle='-', color='b')
plt.xlabel("Number of Principal Components")
plt.ylabel("Cumulative Explained Variance")
plt.title("Effect of PCA Components on Data Variance")
plt.grid(True)
plt.axhline(y=0.95, color='r', linestyle='--', label='95% Variance')
plt.legend()
plt.show()
```


Effect of PCA Components on Data Variance
95% Variance
Cumulative Explained Variance
Number of Principal Components
0.2
0.4
0.6
0.8
1.0
0
10
20
30
40
50
60

```
In [ ]:
```