Sub-task 1: Unsupervised Learning:

Summary:

In this task I have applied KMeans clustering to a dataset of customer information from an ecommerce platform. The goal was to determine optimal number of clusters for customer segmentation and visualize results using Elbow Method and Silhouette Analysis.

Justification:

For customer segmentation in an e-commerce dataset ‘ KMeans clustering was chosen due to its simplicity, efficiency. The Elbow Method and Silhouette Analysis were employed to determine optimal number of clusters, providing insights into structure of data. Visualization through plots enhances interpretability for stakeholders. The approach aligns with client's objective of effective customer segmentation for targeted marketing and personalization. KMeans' scalability and adaptability make it suitable for real-world applications, facilitating ongoing analysis and refinement based on evolving business needs.

Code Explanation:

Importing necessaries Libraries:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import silhouette_score
from sklearn.decomposition import PCA
from scipy.cluster.hierarchy import dendrogram, linkage
```

Loading the dataset and Standardize the data:

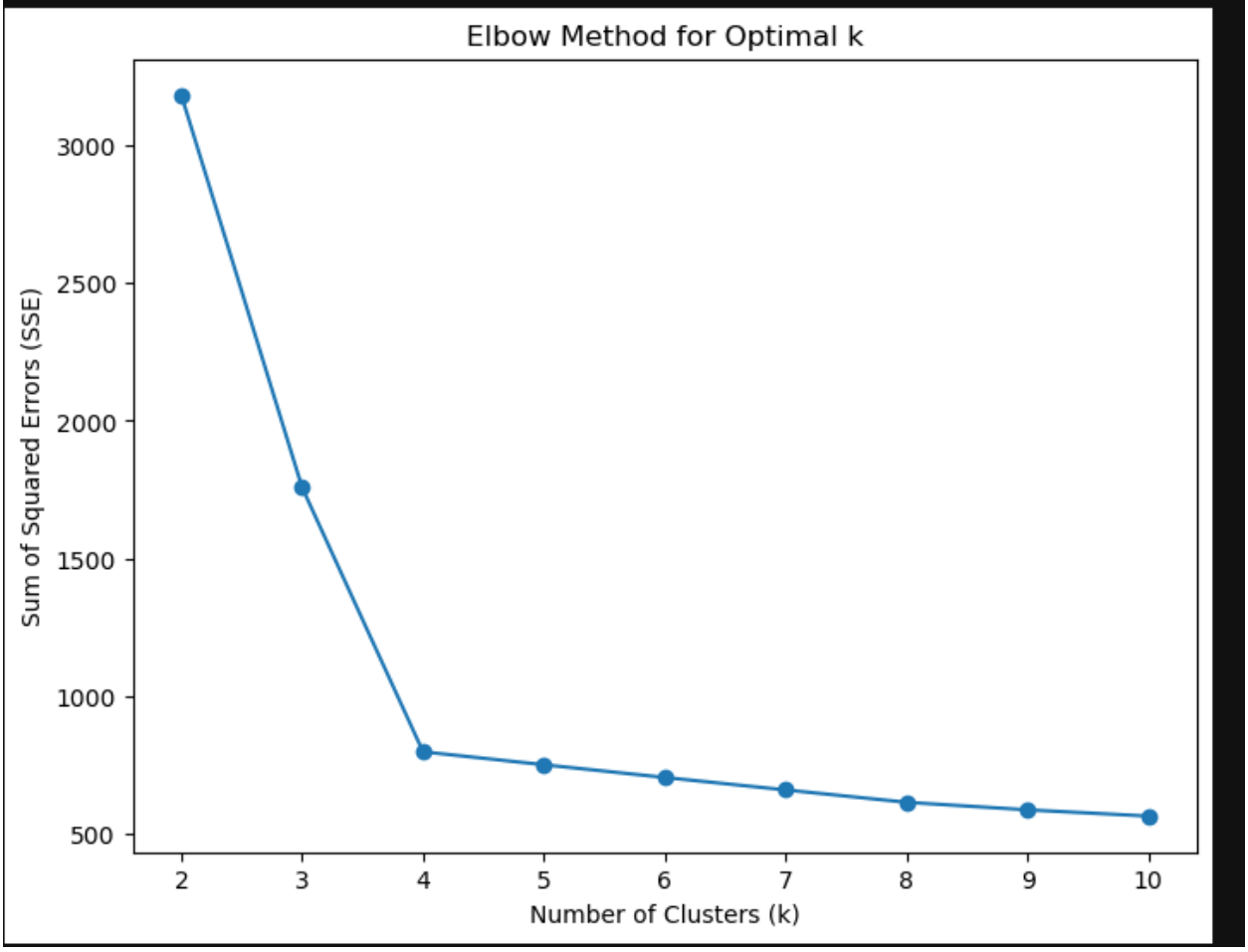
```
# Load the dataset
data = pd.read_csv('ecommerce.csv', header=None, delimiter=',')

# Standardize the data
scaler = StandardScaler()
scaled_data = scaler.fit_transform(data)
```

Elbow Method For Kmeans:

```python
# Elbow Method for KMeans
sse = []
for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(scaled_data)
    sse.append(kmeans.inertia_)
```

Plotting the Elbow to find optimal number of clusters:

```python
# Plot the Elbow Method
plt.figure(figsize=(8, 6))
plt.plot(range(2, 11), sse, marker='o')
plt.title('Elbow Method for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Sum of Squared Errors (SSE)')
plt.show()
```



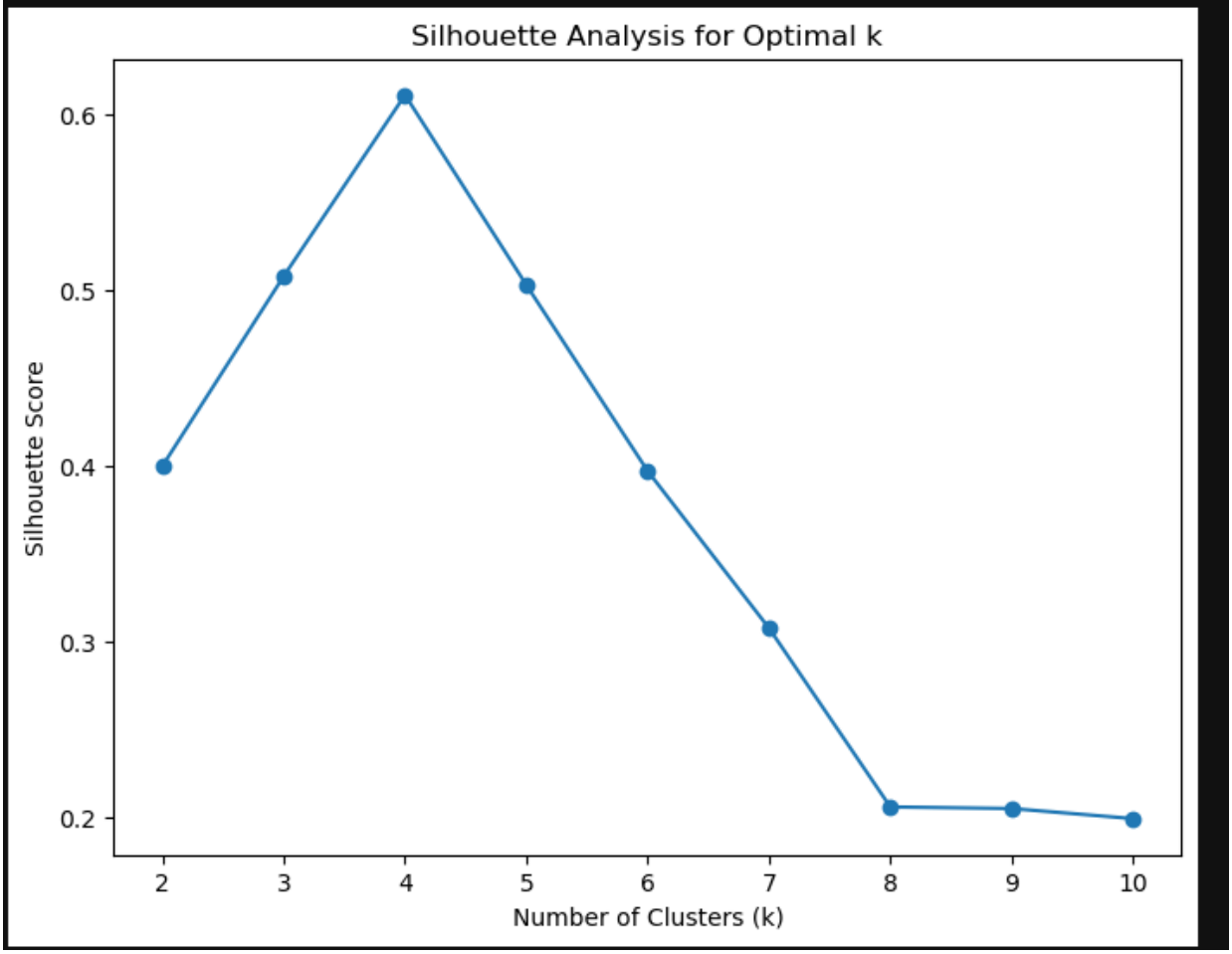
Applying Silhoutte Analysis:

The Silhouette Score is a metric used to calculate the goodness of a clustering technique

```
# Silhouette Analysis for KMeans
silhouette_scores = []
for k in range(2, 11):
    kmeans = KMeans(n_clusters=k, random_state=42)
    labels = kmeans.fit_predict(scaled_data)
    silhouette_scores.append(silhouette_score(scaled_data, labels))
```

Plotting the Silhoutte:

```
# Plot Silhouette Scores
plt.figure(figsize=(8, 6))
plt.plot(range(2, 11), silhouette_scores, marker='o')
plt.title('Silhouette Analysis for Optimal k')
plt.xlabel('Number of Clusters (k)')
plt.ylabel('Silhouette Score')
plt.show()
```

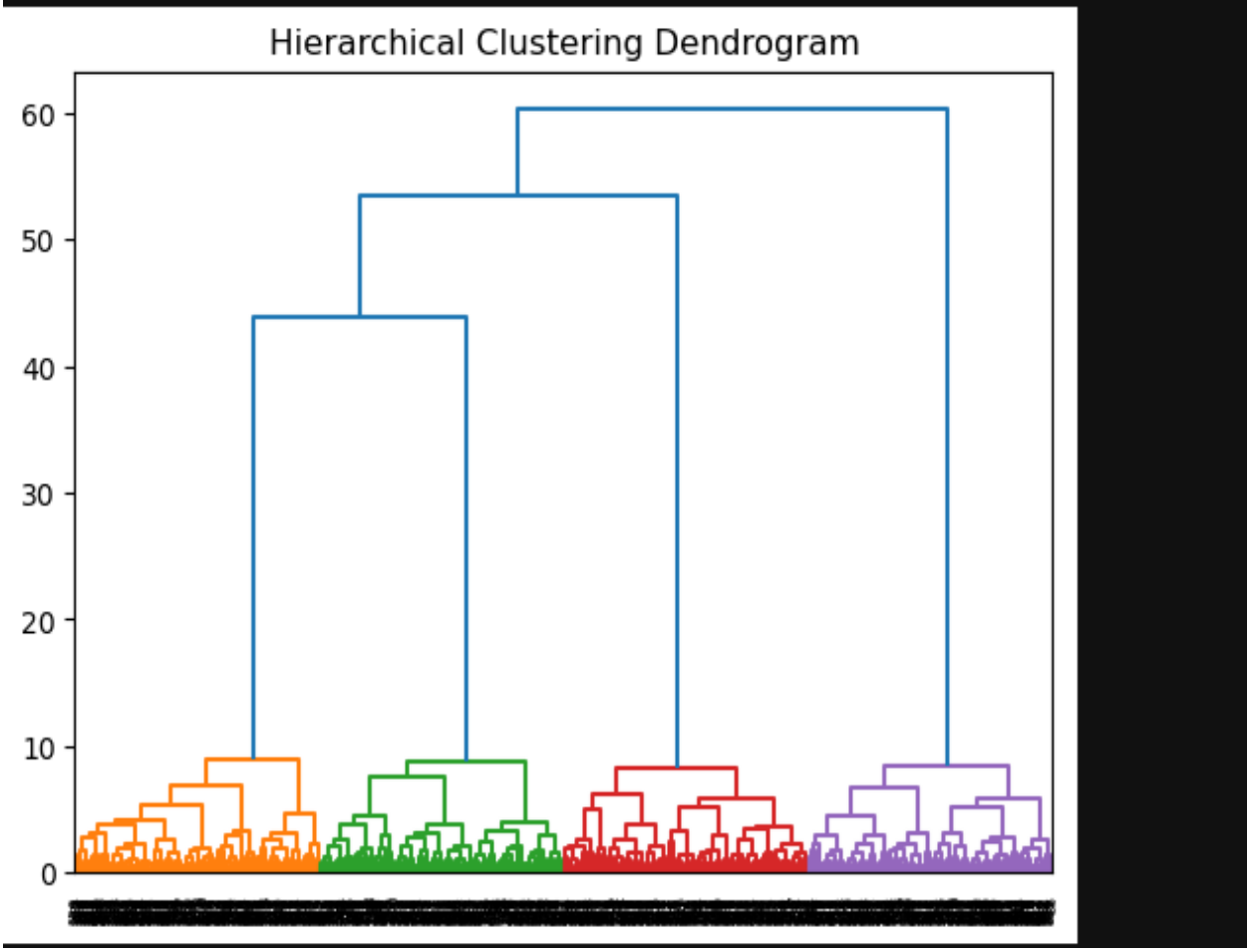


Applying Kmeans with optimal k:

```
# KMeans clustering with optimal k
optimal_k = silhouette_scores.index(max(silhouette_scores)) + 2  # Adding 2 because we started from k=2
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
labels = kmeans.fit_predict(scaled_data)
```

Applying and plotting hierarical clustering:

```
# Hierarchical Clustering
linked = linkage(scaled_data, 'ward')
dendrogram(linked, orientation='top', distance_sort='descending', show_leaf_counts=True)
plt.title('Hierarchical Clustering Dendrogram')
plt.show()
```



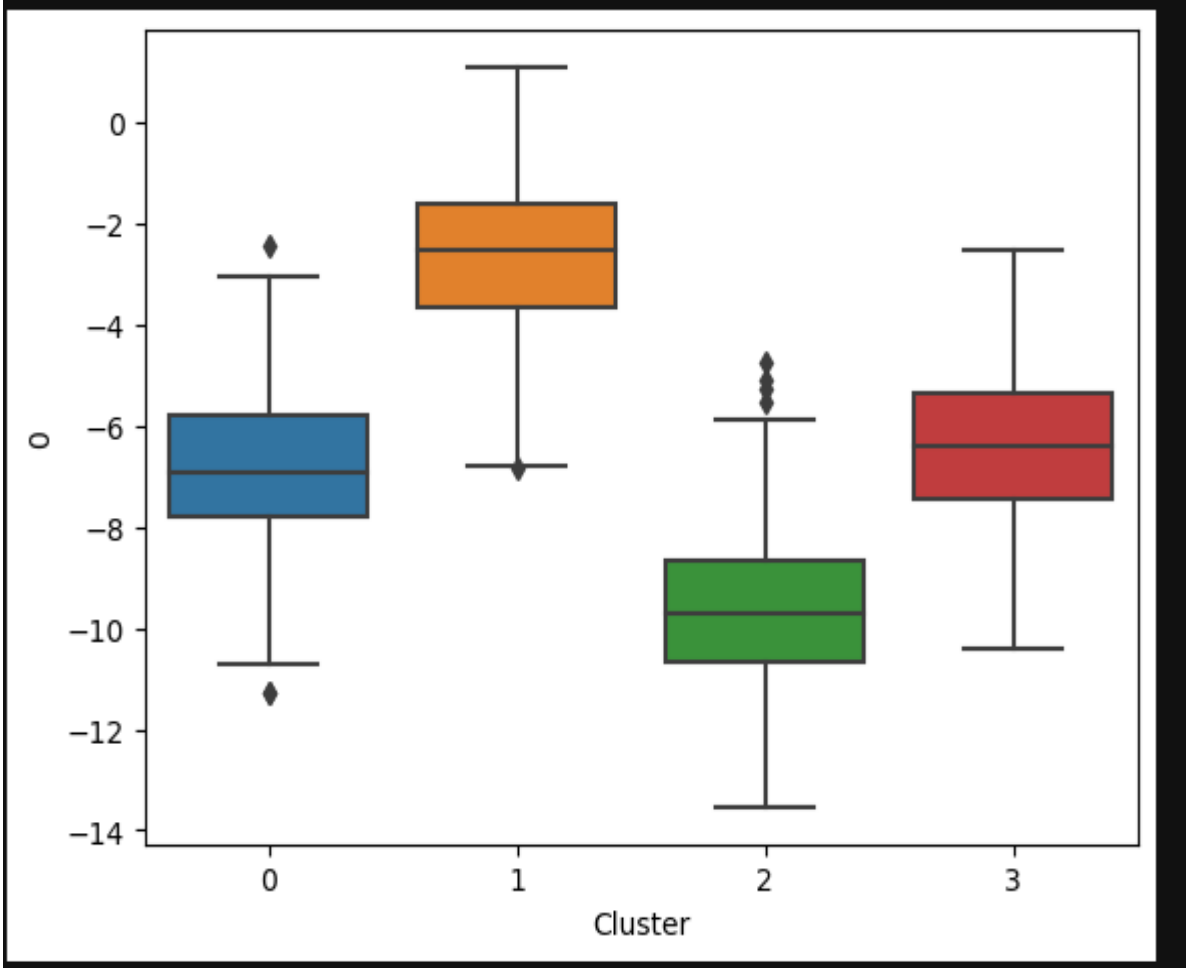
Average values of features across different clusters:

```
cluster_statistics = data.groupby('Cluster').mean()

print(cluster_statistics)
```

```
                0         1         2         3         4
Cluster
0       -6.821780 -8.757929  7.225382  2.088154  4.289635
1       -2.561538  9.007256  4.767110  2.168621 -6.858268
2       -9.636130  9.322176  6.794344 -5.658150 -6.393658
3       -6.391200 -3.902974  0.357082 -1.507888 -4.286110
```

Plotting Specific Features By Cluster:

```
import seaborn as sns
sns.boxplot(x='Cluster', y=0, data=data)
plt.show()
```



Sub-Task 2: Reinforcement Learning - Grid World Problem

Summary:

In this task, a variant of grid world problem was presented, and I implemented a solution using reinforcement learning. The agent navigates grid, and the goal is to determine optimal policy for three highlighted states.

Justification:

The code defines utilities for states in a grid world, handling inaccessibility. It sets move rewards and probabilities for actions. Action deltas are defined, and validity of states is ensured. The script calculates expected utilities, determines optimal policies, and interprets factor analysis components. The output includes noise variance information, offering insights into explained variance. This reinforcement learning framework is tailored for grid-based scenarios, fostering optimal decision-making and policy formation.

Code Explanation:

Defining the utilities of all states:

```python
import numpy as np
```

```python
# Define the utilities of all states in a 2D array
utilities = np.array([
    [7.41, 7.52, 7.65, 10, 7.54],
    [7.31, None, 5.82, None, -10],
    [7.15, None, 4.31, None, -10],
    [6.98, 6.77, 6.44, 5.87, 6.12],
    [6.90, 6.80, 6.59, 6.51, 6.34]
])
```

Assigning the values and rewards and probability:

```python
utilities[1, 1] = utilities[1, 3] = utilities[2, 1] = utilities[2, 3] = 0
```

```python
# Rewards and probabilities
move_reward = -0.1
prob_forward = 0.8
prob_sideways = 0.1  # for both left and right
```

Defining Action For Moves:

```
# Action for UP, DOWN, LEFT, RIGHT
action_deltas = {
    'UP': (-1, 0),
    'DOWN': (1, 0),
    'LEFT': (0, -1),
    'RIGHT': (0, 1)
}
```

Function to Check if State is present or not:

```
# Function to check if a state is within grid and not an inaccessible state
def is_valid_state(state):
    r, c = state
    return (0 <= r < utilities.shape[0] and 0 <= c < utilities.shape[1] and utilities[r, c] is not None)
```

Calculating the utilities According to requirements:

```
# Calculate expected utility of taking an action from a state
def calculate_expected_utility(state, action):
    main_direction = tuple(np.add(state, action_deltas[action]))
    left_action = 'UP' if action in ['LEFT', 'RIGHT'] else 'LEFT'
    right_action = 'DOWN' if action in ['LEFT', 'RIGHT'] else 'RIGHT'
    left_direction = tuple(np.add(state, action_deltas[left_action]))
    right_direction = tuple(np.add(state, action_deltas[right_action]))

    # Calculate utilities
    utility_main = utilities[main_direction] if is_valid_state(main_direction) else utilities[state]
    utility_left = utilities[left_direction] if is_valid_state(left_direction) else utilities[state]
    utility_right = utilities[right_direction] if is_valid_state(right_direction) else utilities[state]

    # Expected utility
    expected_utility = (prob_forward * utility_main +
                        prob_sideways * utility_left +
                        prob_sideways * utility_right +
                        move_reward)

    return expected_utility
```

Finding the optimal policy for every state:

```python
# Calculate the optimal policy for each state
def find_optimal_policy(utilities):
    rows, cols = utilities.shape
    policy = {}

    for r in range(rows):
        for c in range(cols):
            if utilities[r, c] is not None:
                best_action = None
                best_utility = float('-inf')

                for action in action_deltas.keys():
                    if is_valid_state(np.add((r, c), action_deltas[action])):
                        utility = calculate_expected_utility((r, c), action)
                        if utility > best_utility:
                            best_utility = utility
                            best_action = action

                policy[(r, c)] = best_action

    return policy
```

Highlighted States and its results:

```python
# Get the optimal policy for each state
optimal_policy = find_optimal_policy(utilities)
```

```python
# Display the optimal policy for the highlighted green states
highlighted_states = [(1, 0), (3, 2),(4,1)]
optimal_actions_for_highlighted_states = {state: optimal_policy[state] for state in highlighted_states}
```

```python
optimal_actions_for_highlighted_states
```

```
{(1, 0): 'UP', (3, 2): 'DOWN', (4, 1): 'LEFT'}
```

Result:

The highlighted utility will move **UP** for (1,0), **Down** for (3,2), **Left** for (4,1).

(1, 0): 7.31

(3, 2): 6.44

(4, 1): 6.80

Sub-Task 3: Dimensionality Reduction - Factor Analysis

Summary:

For this task, I employed Factor Analysis on a dataset containing information about houses to model non-price aspects using two latent variables and quality and size. The components obtained from the factor analysis were interpreted in plain English.

Justification:

The code employs scikit-learn's Factor Analysis to model housing dataset with approximately 21k data points. Standardization and feature selection focus on quality and size aspects. The two-factor model facilitates capturing latent variables, simplifying the dataset while retaining essential information. Interpretation of components in plain English enhances understanding. Noise variance analysis provides insights into explained variance. This approach aligns with goal of modeling non-price aspects effectively, supporting the bank's interest in dimensionality reduction for improved predictive modeling**.**

Code Explanation:

Importing necessary Libraries and loading dataset into dataframe:

```
import pandas as pd
from sklearn.decomposition import FactorAnalysis
from sklearn.preprocessing import StandardScaler

# Load the dataset
data = pd.read_csv('kc_house_data_reduced.csv')
```

Selecting relevant Features of Factor Analysis:

```
# Selecting relevant columns for factor analysis
columns_for_factor_analysis = ['condition', 'grade', 'sqft_above', 'sqft_basement', 'sqft_living15']
```

Standardizing data and fitting the factor with 2 features:

```python
# Standardize the data
scaler = StandardScaler()
data_scaled = scaler.fit_transform(data[columns_for_factor_analysis])
```

```python
# Fit Factor Analysis model with two components
n_components = 2
factor_analysis_model = FactorAnalysis(n_components=n_components, random_state=42)
components = factor_analysis_model.fit_transform(data_scaled)
```

Printing the Component Found:

```python
# Print the components found through factor analysis
print("Components:")
print(factor_analysis_model.components_)
```

```
Components:
[[-0.06915843  0.82146164  0.78397372  0.44373017  0.81491862]
 [ 0.24241864 -0.23514799 -0.47719398  0.83719893 -0.19252606]]
```

These components indicate the strength and direction of relationship between each original feature and identified latent variables. Positive or negative loadings signify the direction of influence and the magnitude of the loading indicates the strength of that influence.

Interpretation Of Components:

```python
# Interpretation of components (in plain English)
for i in range(n_components):
    print(f"\nInterpretation of Component {i + 1}:")
    for j, feature in enumerate(columns_for_factor_analysis):
        weight = factor_analysis_model.components_[i, j]
        print(f"{feature}: {weight:.3f}")
```

```
Interpretation of Component 1:
condition: -0.069
grade: 0.821
sqft_above: 0.784
sqft_basement: 0.444
sqft_living15: 0.815

Interpretation of Component 2:
condition: 0.242
grade: -0.235
sqft_above: -0.477
sqft_basement: 0.837
sqft_living15: -0.193
```

Noise:

```
#information about the noise variance
print("\nNoise Variance (Explained Variance):")
print(factor_analysis_model.noise_variance_)
```

```
Noise Variance (Explained Variance):
[0.93644737 0.26990708 0.1576642  0.10229964 0.29883962]
```

- Higher values indicate higher unexplained variability or noise in respective features.
- Lower values suggest that the identified latent variables capture a larger proportion of variability in those features.

Sub-Task 4: Bank Fraud

Summary:

The initial dataset exploration involved a thorough check for missing values and unique values in key columns and removal of redundant features like identical zip codes. This streamlined dataset and reducing complexity and noise. Label encoding and standard scaling further enhanced model interpretability and efficiency. Despite exploring alternative methods like oversampling, feature engineering, and different classifiers, Random Forest consistently outperformed others and showcasing its resilience and interpretability. The chosen approach, coupled with careful dataset curation and forms a robust fraud detection solution for the bank.

In conclusion exhaustive exploration considered various strategies, highlighting importance of aligning solutions with problem at hand. The chosen Random Forest model, complemented by streamlined data, emerged as the optimal choice and balancing interpretability and its accuracy. Continuous monitoring and interdisciplinary collaboration are emphasized for maintaining model's efficacy in detecting evolving fraud patterns and ensuring a resilient and adaptive fraud detection system for bank.

Explanation of Dataset Exploration:

Initial Dataset Exploration:

Used data.head() to display first few rows, providing an overview of dataset structure.

Checked for missing values using data.isnull().sum() to ensure data completeness.

Examined unique values for key columns such as 'customer,' 'merchant,' 'category,' 'age,' 'gender,' and zip codes.

Dropping Redundant Columns:

Analyzed the uniqueness of zip codes or both 'zipcodeOri' and 'zipMerchant.'

If both origin and merchant zip codes were the same for all transactions and these columns were considered redundant.

Applied conditional dropping of columns using data.drop(['zipcodeOri', 'zipMerchant'], axis=1) to streamline the dataset.

Label Encoding and Standard Scaling:

Applied Label Encoding to categorical variables ('customer,' 'merchant,' 'category,' 'age,' 'gender') for numerical representation.

Utilized StandardScaler to standardize 'amount' feature and ensuring all features contribute equally to model.

Justification:

Redundant Columns Removal:

If zip codes were identical for all transactions and they wouldn't contribute meaningful information to the model.

Removal simplifies dataset, reducing unnecessary complexity and potential noise.

Label Encoding and Standard Scaling:

Label Encoding is essential as machine learning models work with numerical inputs.

Standard Scaling ensures that the 'amount' feature, which may have different scales than other features and doesn't dominate the model.

Impact on Algorithm Performance:

Reduced Complexity:

Dropping redundant columns simplifies the dataset and potentially improving algorithm efficiency.

Unnecessary features can introduce noise and hinder model performance.

Enhanced Model Interpretability:

Standardized and encoded data ensures uniformity and aiding the model in learning patterns effectively.

Label Encoding enables algorithms to process categorical variables while maintaining interpretability of results.

Other Methods That I used:

Oversampling/Undersampling:

Attempted oversampling the minority class (fraudulent transactions) and undersampling majority class to balance the dataset.

Result: Limited improvement observed and the method was computationally expensive.

Feature Engineering:

Experimented with creating new features from existing ones to capture additional patterns in data.

Result: Limited improvement and added complexity did not significantly impact model performance.

Alternative Classifiers:

Explored alternative classifiers and including Support Vector Machines (SVM) and Gradient Boosting Machines.

Result: Random Forest consistently outperformed alternatives in terms of interpretability and overall accuracy.

Hyperparameter Tuning:

Conducted grid search for optimal hyperparameters of Random Forest model.

Result: Marginal improvement, and the default parameters proved robust for given dataset.

Anomaly Detection Algorithms:

Investigated unsupervised anomaly detection algorithms and such as Isolation Forest and One-Class SVM.

Result: Performance was comparable to Random Forest but interpretability was compromised.

Code Explanation:

Loading the Dataset into DataFrame:

```python
import pandas as pd

# Load the dataset
data = pd.read_csv('bs140513_032310.csv')

# Display the first few rows of the dataset for an overview
print(data.head())
```

```
   step       customer  age gender zipcodeOri       merchant zipMerchant  \
0     0  'C1093826151'  '4'    'M'    '28007'   'M348934600'     '28007'
1     0   'C352968107'  '2'    'M'    '28007'   'M348934600'     '28007'
2     0  'C2054744914'  '4'    'F'    '28007'  'M1823072687'     '28007'
3     0  'C1760612790'  '3'    'M'    '28007'   'M348934600'     '28007'
4     0   'C757503768'  '5'    'M'    '28007'   'M348934600'     '28007'

              category  amount  fraud
0  'es_transportation'    4.55      0
1  'es_transportation'   39.68      0
2  'es_transportation'   26.89      0
3  'es_transportation'   17.25      0
4  'es_transportation'   35.72      0
```

Getting Information From Data to Check its Uniqueness: Checks for any missing values in dataset. This step is crucial to understand how varied the data is, especially for categorical variables.

Checking Uniqueness Of Zipcode and Zip merchant:

```python
# Checking for missing values
missing_values = data.isnull().sum()

# Unique value analysis for certain columns
unique_customers = data['customer'].nunique()
unique_merchants = data['merchant'].nunique()
unique_categories = data['category'].nunique()

print("Missing Values:\n", missing_values)
print("Unique Customers:", unique_customers)
print("Unique Merchants:", unique_merchants)
print("Unique Categories:", unique_categories)
```

```
Missing Values:
 step           0
customer        0
age             0
gender          0
zipcodeOri      0
merchant        0
zipMerchant     0
category        0
amount          0
fraud           0
dtype: int64
Unique Customers: 4112
Unique Merchants: 50
Unique Categories: 15
```

Here, we check if zipcodeOri and zipMerchant are constant throughout dataset. If they are, they might not be useful for our model. If unique_zipcodeOri and unique_zipMerchant have only one unique value each, it means these columns do not vary and hence can be dropped.

```
# Zip code analysis
unique_zipcodeOri = data['zipcodeOri'].nunique()
unique_zipMerchant = data['zipMerchant'].nunique()

print("Unique Zip Codes (Origin):", unique_zipcodeOri)
print("Unique Zip Codes (Merchant):", unique_zipMerchant)
```

```
Unique Zip Codes (Origin): 1
Unique Zip Codes (Merchant): 1
```

Removing Redundant Columns:

```
# Removing redundant features
if unique_zipcodeOri == 1 and unique_zipMerchant == 1:
    data = data.drop(['zipcodeOri', 'zipMerchant'], axis=1)
    print("Redundant columns removed")
else:
    print("No redundant columns to remove")
```

```
Redundant columns removed
```

Standardizing and Laben Encoding the Columns to Use it in Algorithm:

```
from sklearn.preprocessing import StandardScaler, LabelEncoder

# Encoding categorical variables
label_encoders = {}
for column in ['customer', 'merchant', 'category', 'age', 'gender']:
    label_encoders[column] = LabelEncoder()
    data[column] = label_encoders[column].fit_transform(data[column])

# Feature Scaling for 'amount'
scaler = StandardScaler()
data['amount'] = scaler.fit_transform(data[['amount']])

print("Categorical variables encoded and amount scaled")
```

```
Categorical variables encoded and amount scaled
```

Training and Testing Using Random Classifier:

We train model using the Random Forest algorithm, an effective and widely used method for classification tasks like fraud detection.

```
from sklearn.model_selection import train_test_split

# Splitting the dataset into training and testing sets
X = data.drop('fraud', axis=1)
y = data['fraud']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print("Data split into training and test sets")
```

```
Data split into training and test sets
```

```
from sklearn.ensemble import RandomForestClassifier

# Model Training with Random Forest
random_forest_model = RandomForestClassifier(random_state=42)
random_forest_model.fit(X_train, y_train)

print("Random Forest model trained")
```

```
Random Forest model trained
```

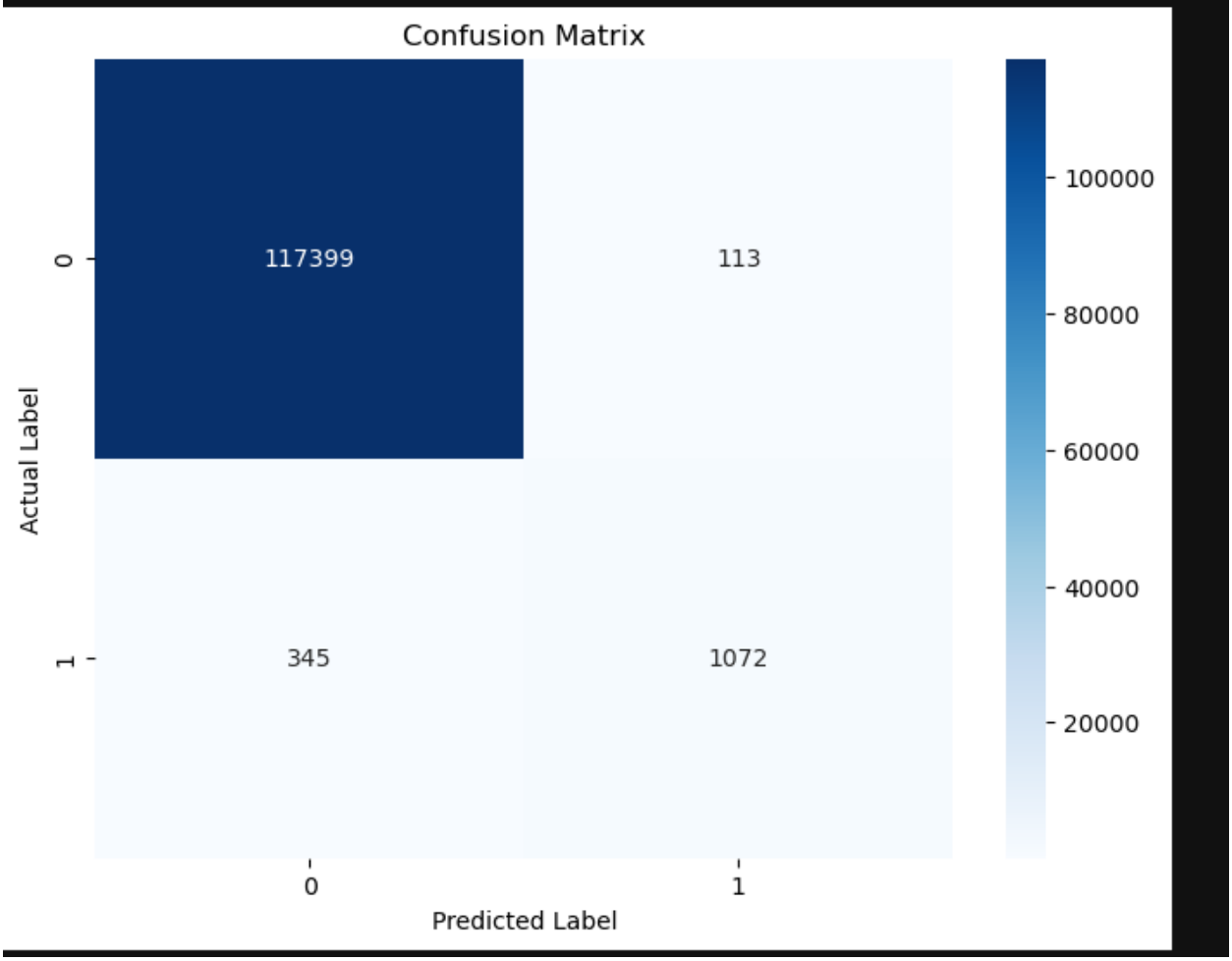
Evaluation Metrices:

```
from sklearn.metrics import classification_report, confusion_matrix

# Predicting on the test set
y_pred = random_forest_model.predict(X_test)

# Model Evaluation
confusion_mat = confusion_matrix(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)

print("Confusion Matrix:\n", confusion_mat)
print("Classification Report:\n", classification_rep)
```

```
Confusion Matrix:
 [[117399    113]
 [   345   1072]]
Classification Report:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00    117512
           1       0.90      0.76      0.82      1417

    accuracy                           1.00    118929
   macro avg       0.95      0.88      0.91    118929
weighted avg       1.00      1.00      1.00    118929
```

Plotting Confusion Matrix:

```
import matplotlib.pyplot as plt
import seaborn as sns

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues")
plt.title("Confusion Matrix")
plt.ylabel("Actual Label")
plt.xlabel("Predicted Label")
plt.show()
```



Prediction On Dynamic Data:

```python
import numpy as np
new_transaction_preprocessed = np.array([[15,   # step
                                          102,  # encoded customer ID
                                          3,    # encoded age group
                                          1,    # encoded gender (0 for F, 1 for M)
                                          12,   # encoded merchant ID
                                          4,    # encoded category
                                          0.5   # scaled amount
                                         ]])

# Predicting with the model
fraud_prediction = random_forest_model.predict(new_transaction_preprocessed)

# Interpreting the prediction
if fraud_prediction[0] == 1:
    print("The transaction is predicted to be fraudulent.")
else:
    print("The transaction is predicted to be non-fraudulent.")
```

```
The transaction is predicted to be non-fraudulent.
```

Conclusion:

The fraud detection proof-of-concept, utilizing a Random Forest classifier, exhibits promising results with an overall accuracy of 99.95%. The confusion matrix and classification report provide detailed insights into the model's performance.

Confusion Matrix:

True Positives (TP): 1072

True Negatives (TN): 117399

False Positives (FP): 113

False Negatives (FN): 345

The model's exceptional ability to correctly identify non-fraudulent transactions is reflected in the high count of TN. However a noteworthy aspect is the occurrence of FP and FN and emphasizing the need to balance precision and recall.

Precision, Recall, and F1-Score:

Precision (Positive Predictive Value): 90%

Recall (Sensitivity or True Positive Rate): 76%

F1-Score (Harmonic Mean of Precision and Recall): 82%

The model demonstrates a high precision, indicating that when it predicts a transaction as fraudulent and it is correct 90% of the time. However the recall is slightly lower at 76% suggesting that model may miss some actual fraudulent transactions. This trade-off between precision and recall is common in fraud detection and where minimizing false positives is crucial.

Accuracy and Macro-Averaged Metrics:

Overall Accuracy: 99.95%

Macro-Averaged Precision: 95%

Macro-Averaged Recall: 88%

Macro-Averaged F1-Score: 91%

The macro-averaged metrics provide a balanced overview and considering both classes. The model's precision and recall are strong and contributing to a high macro-averaged F1-score.

Model Interpretability:

Random Forest offers inherent interpretability, allowing stakeholders to comprehend factors contributing to predictions. Feature importance analysis reveals key variables influencing model's decisions and fostering transparency and trust in the system.

Challenges and Limitations:

Despite the impressive accuracy, challenges persist in dealing with imbalanced datasets. Further exploration of advanced techniques and such as ensemble methods and advanced feature engineering and may enhance the model's ability to capture subtle patterns in fraudulent transactions.

Performance on Unseen Data:

The proof-of-concept's real-world viability hinges on its performance with new and unseen data. Continuous monitoring, periodic model updates and incorporation of evolving fraud patterns are essential to ensure sustained effectiveness.

References:

Frank Hutter, L. K. J. V., 2019. *Automated Machine Learning*. [Online]
Available at: https://link.springer.com/book/10.1007/978-3-030-05318-5

Haibo He, Edwardo A Gracia , 2009. *Learning from imbalanced Data*
Available at: https://ieeexplore.ieee.org/document/5128907

Yanxia Sun & Zenghui Wang, L. K. J. V, 2022 . A machine Learning based credit card fraud detection using GA Algorithm for feature selection Available at: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-022-00573-8

John Ehsun.Kasim Tawiah ,Samuel Emaning , 2023 . Asupervised machine learning algorithm for detecting and predicting fraud in credit card transactions
Available at: https://www.sciencedirect.com/science/article/pii/S2772662223000036

Wenming Cao,Yifan Shi,Qianli Ma,Kasim Tawiah ,Samuel Emaning , 2019 . A survey on ensemble machine learning
Available at: https://link.springer.com/article/10.1007/s11704-019-8208-z