CMPSC432 PROJECT 2

December 6, 2024

[1]:
```
import pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import numpy as np
from sklearn.cluster import KMeans
import seaborn as sns


df= pd.read_csv("2024_bank_marketing.csv", sep =';')
df= df[['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate',␣
 ↪'cons.price.idx','cons.conf.idx', 'euribor3m', 'nr.employed']].dropna()
df.head()
```

[1]:
```
   age  duration  campaign  pdays  previous  emp.var.rate  cons.price.idx  \
0   56       261         1  999.0         0           1.1          93.994
1   57       149         1  999.0         0           1.1          93.994
2   37       226         1  999.0         0           1.1          93.994
3   40       151         1  999.0         0           1.1          93.994
4   56       307         1  999.0         0           1.1          93.994

   cons.conf.idx  euribor3m  nr.employed
0          -36.4      4.857       5191.0
1          -36.4      4.857       5191.0
2          -36.4      4.857       5191.0
3          -36.4      4.857       5191.0
4          -36.4      4.857       5191.0
```

0.1 TASK 1 Data Preprocessing

(a)

[2]:
```
#Standarizing with 0 mean and 1 sd.

standardized_df = StandardScaler().fit_transform(df)

standardized_df = pd.DataFrame(standardized_df, columns=df.columns)
```

[3]:
```
pca = PCA(n_components=3)
pca_result = pca.fit_transform(standardized_df)
```

```
pca_df = pd.DataFrame(pca_result, columns=['PC1', 'PC2', 'PC3'])
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = explained_variance_ratio.cumsum()
print("\nExplained Variance Ratio (Per Component):")
print(explained_variance_ratio)
print("\nCumulative Variance Ratio:")
print(f"{cumulative_variance_ratio}\n")
print("PCA Result (First 5 Rows):")
pca_df.head()
```

```
Explained Variance Ratio (Per Component):
[0.39027471 0.13608982 0.10766081]

Cumulative Variance Ratio:
[0.39027471 0.52636453 0.63402534]

PCA Result (First 5 Rows):
```

[3]:

```
        PC1       PC2       PC3
0 -1.270009  0.900235 -1.484983
1 -1.280918  0.889251 -1.500835
2 -1.275584  0.426628 -0.340350
3 -1.282628  0.476007 -0.489680
4 -1.265483  0.914746 -1.502921
```

(b)

[4]:

```
column_means = round(standardized_df.mean())
column_stds = round(standardized_df.std())
print("Column Means:")
print(column_means)

print("\nColumn Standard Deviations:")
print(column_stds)
standardized_df.head()
```

```
Column Means:
age              -0.0
duration         -0.0
campaign         -0.0
pdays             0.0
previous         -0.0
emp.var.rate      0.0
cons.price.idx    0.0
cons.conf.idx    -0.0
euribor3m         0.0
nr.employed      -0.0
dtype: float64
```

```
Column Standard Deviations:
age               1.0
duration          1.0
campaign          1.0
pdays             1.0
previous          1.0
emp.var.rate      1.0
cons.price.idx    1.0
cons.conf.idx     1.0
euribor3m         1.0
nr.employed       1.0
dtype: float64
```

[4]:
```
        age  duration  campaign     pdays  previous  emp.var.rate  \
0  1.532480  0.011198 -0.565014  0.194843  -0.35038      0.648698
1  1.628391 -0.421731 -0.565014  0.194843  -0.35038      0.648698
2 -0.289820 -0.124092 -0.565014  0.194843  -0.35038      0.648698
3 -0.002089 -0.414000 -0.565014  0.194843  -0.35038      0.648698
4  1.532480  0.189009 -0.565014  0.194843  -0.35038      0.648698

   cons.price.idx  cons.conf.idx  euribor3m  nr.employed
0        0.726328         0.8859   0.712819     0.331644
1        0.726328         0.8859   0.712819     0.331644
2        0.726328         0.8859   0.712819     0.331644
3        0.726328         0.8859   0.712819     0.331644
4        0.726328         0.8859   0.712819     0.331644
```

I was using StandardScaler from sklearn to normalized attributes by following : z = (x - u) / s. where u is the mean of the samples or zero if with_mean=False, and s is the standard deviation of the training samples or one if with_std=False. After standardization or z-score standariztion, mean of each column or each attribute is 0, and standard deviation is 1.

0.2 TASK 2 Implementation of the k-Means Algorithm

[5]:
```python
def rand_center(data,k):
    centroids=data.sample(k,random_state=None)
    return centroids.to_numpy()

def converged(centroids1, centroids2):
    return np.allclose(centroids1, centroids2)

def update_centroids(data, centroids, k):
    data= data.to_numpy()
    #create a matrix to track distance from points to centroids
    distances = np.zeros((data.shape[0],k))
    #compute distance by using
    for i, centroid in enumerate(centroids):
        distances[:, i] = np.linalg.norm(data - centroid, axis=1)

    # Assign each data point to its nearest centroid based on the Euclidean␣
 ↪distance

    # labels
    labels  = np.argmin(distances, axis=1)
    #Update centroids
    new_centroids = np.array([data[labels == i].mean(axis=0) for i in range(k)])
    #clusters
    clusters = [data[labels == i].tolist() for i in range(k)]

    return new_centroids, clusters, labels
```

[6]:
```
def kmeans(data,k=5):
    """
    >>> Main function of your k-Means implementation
    """
    # step 1:
    centroids = rand_center(data,k)
    converge = False
    while not converge:
        old_centroids = np.copy(centroids)
        # step 2 & 3; labels can be an array of labels for all the data points
        centroids, clusters_kmeans, labels = update_centroids(data,␣
 ↪old_centroids,k)
        # step 4
        converge = converged(old_centroids, centroids)
    return centroids, clusters_kmeans, labels
```

[7]:
```
centroids,clusters_kmeans, labels=kmeans(pca_df)
print(">>> final centroids")
print(centroids)

print(">>> Clusters_Kmeans")
for i, cluster in enumerate(clusters_kmeans):
    print(f"Cluster {i}:")
    print(np.array(cluster))

print(">>> Labels")
print(labels[:100])
```

```
>>> final centroids
[[-0.01368093 -0.39164516 -0.16334102]
 [ 2.40256332 -0.96384982 -0.13304566]
 [ 4.4846831   4.21128907  1.32931758]
 [-1.65179168 -0.02786154  0.79837834]
 [-1.38320225  0.44425878 -0.56882358]]
```

```
>>> Clusters_Kmeans
Cluster 0:
[[-0.01094662 -0.23461373 -0.94788106]
 [-0.11637174 -0.47416902 -0.31497909]
 [-0.04840306 -0.67301796 -0.08805813]
 …
 [-0.13355874 -0.32185334 -1.06324289]
 [ 0.50192841  0.34476882  0.40165215]
 [-0.11457988 -0.66715402 -0.2319238 ]]
Cluster 1:
[[ 2.44172683 -0.7507556  -1.54543022]
 [ 2.02515375 -1.78620966  0.71166866]
 [ 2.72981511 -0.54551666  0.3913268 ]
 …
 [ 2.05494438 -1.42218053 -0.20866034]
 [ 1.90831914 -1.88870839  1.34089248]
 [ 2.89368139 -0.4965629  -2.41242266]]
Cluster 2:
[[4.88313892 2.27347779 0.58157212]
 [3.16832527 1.82909483 0.84500712]
 [4.34458093 3.51319275 0.78636919]
 …
 [6.3369252  7.52237203 4.57466311]
 [3.97566542 2.96665671 1.26412617]
 [3.97490523 2.33980248 2.77212088]]
Cluster 3:
[[-1.64586157 -0.30018409  1.48577955]
 [-1.53846524 -0.45356985  1.22558668]
 [-1.68582461 -0.25345485  1.22274751]
 …
 [-1.48913532 -0.1909924   0.50906421]
 [-1.58715222  0.11989883  0.153085  ]
 [-1.44338183 -0.2786564   0.63720414]]
Cluster 4:
[[-1.27000932  0.90023503 -1.48498319]
 [-1.28091827  0.88925077 -1.50083497]
 [-1.27558445  0.42662811 -0.3403496 ]
 …
 [-1.28454753  0.55441149 -1.35098228]
 [-1.29058611  0.66537997 -0.98875691]
 [-1.31092484 -0.01184731 -0.06259766]]
>>> Labels
[4 4 4 4 4 1 4 4 1 1 4 3 4 0 3 4 1 3 3 3 3 3 1 1 4 4 1 1 4 3 4 1 4 1 1 3 1
 1 1 4 1 1 1 1 3 3 4 4 4 4 2 4 3 0 1 1 1 1 1 4 1 1 0 3 3 3 4 3 3 1 1 0 3 4
 1 3 3 4 3 0 4 3 4 3 0 4 4 2 4 1 2 3 4 1 3 2 3 4 1 2]
```

(b) Bisecting k-Means

[8]:
```
## calculate_sse

def calculate_sse(cluster):
    if len(cluster)==0:
        return 0
    centroid = cluster.mean(axis=0)
    sse = np.sum(np.linalg.norm(cluster - centroid, axis=1) ** 2)
    return sse

## function for bistecting cluster (split the cluster with the largest SSE)

def bisecting_step(clusters,n_init=10):
    ## find sse_largest cluster ready for bisecting
    sse_list = [calculate_sse(cluster) for cluster in clusters]
    largest_sse_index = np.argmax(sse_list)
    largest_cluster = clusters[largest_sse_index]

    ## Split sse_largest cluster by k-mean , k=2
    kmean = KMeans(n_clusters=2,random_state=None,n_init=10)
    labels=kmean.fit_predict(largest_cluster)

    ## define new clusters based on k-means results
    cluster1 = largest_cluster[labels==0]
    cluster2= largest_cluster[labels==1]

     ## Replace the original cluster with the two new clusters
    clusters.pop(largest_sse_index)
    clusters.append(cluster1)
    clusters.append(cluster2)

    return clusters

## main function
def bisecting_kmeans(data,k,random_state=None,n_init=10):
    clusters = [np.array(data)]

    while len(clusters)< k:
        clusters = bisecting_step(clusters,n_init=10)

    labels = np.zeros(len(data), dtype=int)
    for cluster_idx, cluster in enumerate(clusters):
        for point in cluster:
            point_idx = np.where((data == point).all(axis=1))[0][0]
            labels[point_idx] = cluster_idx

    return clusters, labels
```

```
[9]: clusters_Bisecting, final_labels = bisecting_kmeans(pca_df, 5,␣
      ↪random_state=None)
     print(">>>clusters_ Bisecting_kMean}")
     for i, cluster in enumerate(clusters_Bisecting):
         print(f"Cluster {i}:")
         print(cluster)
     print(">>> labels")
     print(final_labels[:100])
```

```
>>>clusters_ Bisecting_kMean}
Cluster 0:
[[4.88313892 2.27347779 0.58157212]
 [3.16832527 1.82909483 0.84500712]
 [4.34458093 3.51319275 0.78636919]
 …
 [6.3369252  7.52237203 4.57466311]
 [3.97566542 2.96665671 1.26412617]
 [3.97490523 2.33980248 2.77212088]]
Cluster 1:
[[-1.27000932  0.90023503 -1.48498319]
 [-1.28091827  0.88925077 -1.50083497]
 [-1.27558445  0.42662811 -0.3403496 ]
 …
 [-0.11457988 -0.66715402 -0.2319238 ]
 [-1.29058611  0.66537997 -0.98875691]
 [-1.31092484 -0.01184731 -0.06259766]]
Cluster 2:
[[-1.64586157 -0.30018409  1.48577955]
 [-1.53846524 -0.45356985  1.22558668]
 [-1.68582461 -0.25345485  1.22274751]
 …
 [-1.48913532 -0.1909924   0.50906421]
 [-1.58715222  0.11989883  0.153085  ]
 [-1.44338183 -0.2786564   0.63720414]]
Cluster 3:
[[ 2.02515375 -1.78620966  0.71166866]
 [ 2.72981511 -0.54551666  0.3913268 ]
 [ 2.714635   -0.70285331  0.83020182]
 …
 [ 2.09672081 -1.54120514 -0.07575902]
 [ 2.05494438 -1.42218053 -0.20866034]
 [ 1.90831914 -1.88870839  1.34089248]]
Cluster 4:
[[ 2.44172683 -0.7507556  -1.54543022]
 [ 4.07451774  1.12441346 -0.4355791 ]
 [ 2.86272445 -1.02797031 -1.1618    ]
 …
 [ 2.14747343 -0.12148466 -1.14639483]
 [ 2.01168482 -1.03059165 -1.23009044]
 [ 2.89368139 -0.4965629  -2.41242266]]
>>> labels
[1 1 1 1 1 4 1 1 3 3 1 2 1 1 2 1 3 2 2 2 2 2 4 4 1 1 3 3 1 2 1 4 1 3 3 2 3
 3 3 1 3 4 3 3 2 2 1 1 1 1 0 1 2 1 3 3 3 3 3 1 3 3 1 2 2 2 2 2 2 3 4 2 2 1
 3 2 2 1 2 1 1 2 1 2 1 1 1 0 1 3 0 2 1 4 2 0 2 1 3 0]
```

I initially treated all data points as a single cluster. Then, I applied the k-means algorithm to split this cluster into two clusters. To determine the next candidate for further splitting, I used the largest SSE (Sum of Squared Errors) rule. For comparison with standard k-means, I also fixed the number of clusters at 5.

(c) Evaluation Metrics

```
[10]: def sse_cal (clusters):
          total_sse = 0
          for cluster in clusters:
              if len(cluster) == 0:  # Skip empty clusters
                  continue
              cluster = np.array(cluster)  # Ensure cluster is a NumPy array
              centroid = cluster.mean(axis=0)  # Compute centroid
              sse = np.sum((cluster - centroid) ** 2)  # Compute SSE for this cluster
              total_sse += sse

          return total_sse
```

```
[11]: print(f"SSE of Bisecting k-Means :{sse_cal(clusters_Bisecting)}")
      print(f"SSE of Normal k-Means :{sse_cal(clusters_kmeans)}")
```

```
SSE of Bisecting k-Means :39635.6515316915
SSE of Normal k-Means :46262.56628136219
```

```
[38]: from sklearn.metrics import silhouette_score

      # calcualte Silhouette Score
      score = silhouette_score(pca_df, labels)
      print("Silhouette Score for normal_k-mean:", score)

      score2 = silhouette_score(pca_df, final_labels)
      print("Silhouette Score for Bisecting_k-mean:", score2)
```

```
Silhouette Score for normal_k-mean: 0.38438758103216225
Silhouette Score for Bisecting_k-mean: 0.42348210798146724
```

In Part C, I evaluated the cluster validity of k-means and bisecting k-means using two metrics: SSE and Silhouette Score. The SSE for Bisecting k-means is approximately 39,635, which is significantly smaller than the SSE for standard k-means (46,262), indicating better clustering performance. Additionally, the Silhouette Score further supports this conclusion, as Bisecting k-means achieves a score of 0.4235, which is closer to 1 compared to the 0.3844 achieved by standard k-means, suggesting that Bisecting k-means provides better-defined and more compact clusters.

0.3 Task 3 Cluster Analysis

(a)

[13]:
```python
def run_kmeans_multiple_times(data, k=5, num_trials=20):
    best_sse = float('inf')
    best_labels = None
    best_centroids = None
    best_clusters = None

    for _ in range(num_trials):
        centroids, clusters_kmeans, labels = kmeans(data, k)

        # Ensure clusters_kmeans is valid before calculating SSE
        if clusters_kmeans is None:
            continue  # Skip this trial if clusters_kmeans is invalid

        sse = sse_cal(clusters_kmeans)
        if sse < best_sse:
            best_sse = sse
            best_labels = labels
            best_centroids = centroids
            best_clusters = clusters_kmeans

    return best_centroids, best_labels, best_clusters, best_sse

# Run standard k-means 20 times
centroids_kmeans_best, labels_kmeans_best,clusters_kmeans_best, sse_kmeans_best␣
 ↪= run_kmeans_multiple_times(pca_df, k=5)
print(">>> best centroids of kmeans among 20 times")
print(centroids_kmeans_best)

print(">>>best clusters of kmeans among 20 times")
for i, cluster in enumerate(clusters_kmeans_best):
    print(f"Cluster {i}:")
    print(np.array(cluster))



print(">>> best labels of kmeans among 20 times")
print(labels_kmeans_best[:100])

print(">>> best SSE of kmeans among 20 times")

print(sse_kmeans_best)
```

```
>>> best centroids of kmeans among 20 times
[[ 2.26811097 -1.20427317  0.34592442]
 [-1.56208152 -0.05172737  0.77774096]
 [ 4.50915901  4.28090014  1.433013  ]
 [-1.13541713  0.30117313 -0.52802962]
 [ 2.95432989  0.05288241 -2.03937022]]
>>>best clusters of kmeans among 20 times
Cluster 0:
[[ 2.02515375 -1.78620966  0.71166866]
 [ 2.72981511 -0.54551666  0.3913268 ]
 [ 2.714635   -0.70285331  0.83020182]
 …
 [ 2.09672081 -1.54120514 -0.07575902]
 [ 2.05494438 -1.42218053 -0.20866034]
 [ 1.90831914 -1.88870839  1.34089248]]
Cluster 1:
[[-1.64586157 -0.30018409  1.48577955]
 [-1.53846524 -0.45356985  1.22558668]
 [-1.68582461 -0.25345485  1.22274751]
 …
 [-1.48913532 -0.1909924   0.50906421]
 [-1.58715222  0.11989883  0.153085  ]
 [-1.44338183 -0.2786564   0.63720414]]
Cluster 2:
[[4.88313892 2.27347779 0.58157212]
 [3.16832527 1.82909483 0.84500712]
 [4.34458093 3.51319275 0.78636919]
 …
 [6.3369252  7.52237203 4.57466311]
 [3.97566542 2.96665671 1.26412617]
 [3.97490523 2.33980248 2.77212088]]
Cluster 3:
[[-1.27000932  0.90023503 -1.48498319]
 [-1.28091827  0.88925077 -1.50083497]
 [-1.27558445  0.42662811 -0.3403496 ]
 …
 [-0.11457988 -0.66715402 -0.2319238 ]
 [-1.29058611  0.66537997 -0.98875691]
 [-1.31092484 -0.01184731 -0.06259766]]
Cluster 4:
[[ 2.44172683 -0.7507556  -1.54543022]
 [ 4.07451774  1.12441346 -0.4355791 ]
 [ 2.86272445 -1.02797031 -1.1618    ]
 …
 [ 2.40691069 -0.73388089 -1.46940561]
 [ 2.14747343 -0.12148466 -1.14639483]
 [ 2.89368139 -0.4965629  -2.41242266]]
>>> best labels of kmeans among 20 times
[3 3 3 3 3 4 3 3 0 0 3 1 3 3 1 3 0 1 1 1 1 1 4 4 3 3 0 0 3 1 3 4 3 0 0 1 0
 0 0 3 0 4 0 0 1 1 3 3 3 3 2 3 1 3 0 0 0 0 0 3 0 0 3 1 1 1 1 1 1 0 4 1 1 3
 0 1 1 3 1 3 3 1 3 1 3 3 3 2 3 0 2 1 3 4 1 2 1 3 0 2]
```

```
>>> best SSE of kmeans among 20 times
39262.99942029846
```

[14]:
```
def run_bisecting_kmeans_multiple_times(data, k=5, num_trials=20,␣
↪num_trials_per_split=10):

    best_sse = float('inf')
    best_labels = None
    best_clusters = None

    for _ in range(num_trials):
        clusters, labels= bisecting_kmeans(data, k, num_trials_per_split)
        sse = sse_cal(clusters_kmeans)
        if sse < best_sse:
            best_sse = sse
            best_labels = labels
            best_clusters = clusters

    return best_clusters, best_labels, best_sse

# Run bisect k-means 20 times
cluster_bisect_best,labels_bisect_best, best_sse_bisect, =␣
↪run_bisecting_kmeans_multiple_times(pca_df,k=5, num_trials=20,␣
↪num_trials_per_split=10)
print("\n>>> Best clusters of bisecting k-means among 20 times")
for i, cluster in enumerate(cluster_bisect_best):
    print(f"Cluster {i}:")
    print(np.array(cluster))

print("\n>>> Best labels of bisecting k-means among 20 times")
print(labels_bisect_best[:100])

print("\n>>> Best SSE of bisecting k-means among 20 times")
print(best_sse_bisect)
```

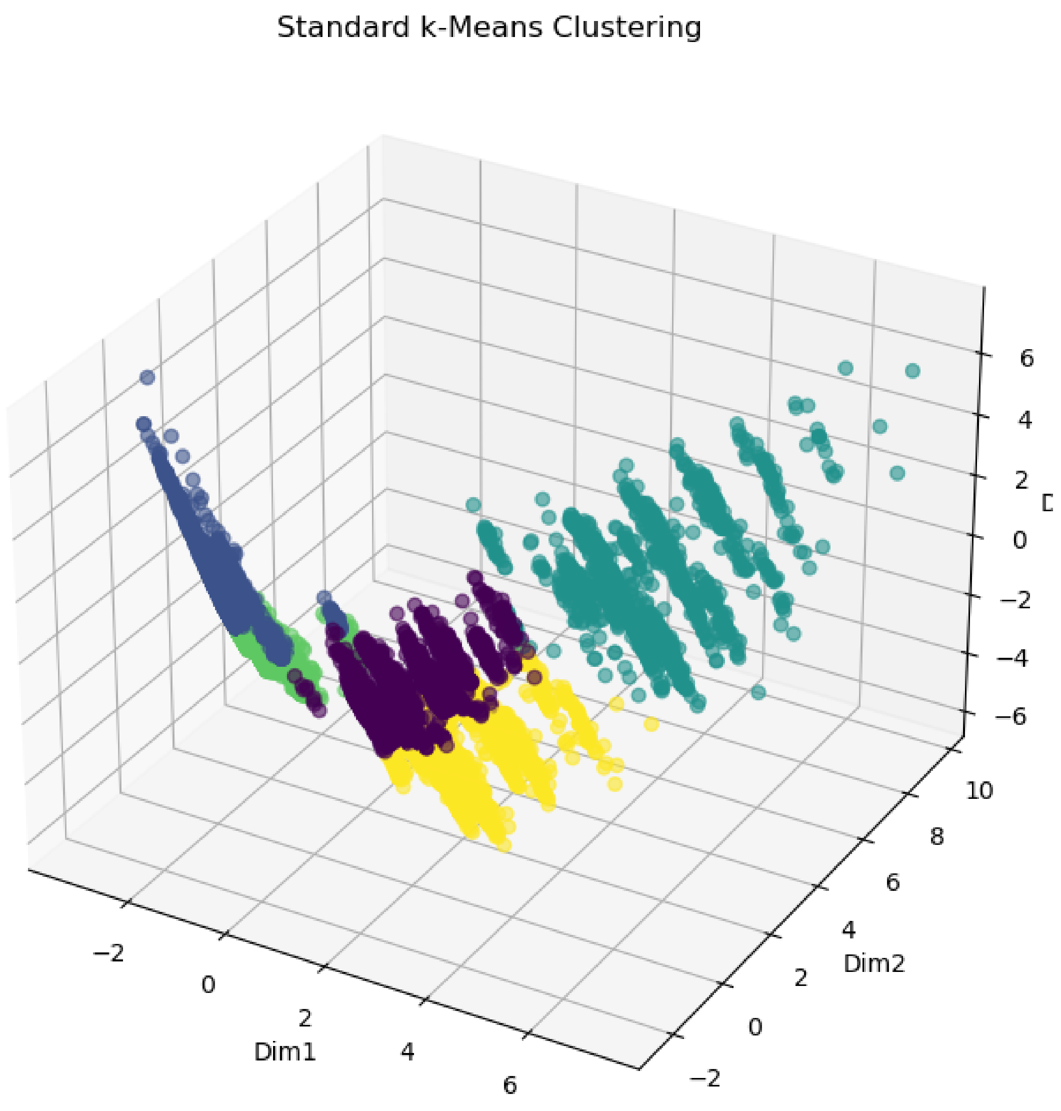
```
>>> Best clusters of bisecting k-means among 20 times
Cluster 0:
[[4.88313892 2.27347779 0.58157212]
 [3.16832527 1.82909483 0.84500712]
 [4.34458093 3.51319275 0.78636919]
 …
 [6.3369252  7.52237203 4.57466311]
 [3.97566542 2.96665671 1.26412617]
 [3.97490523 2.33980248 2.77212088]]
Cluster 1:
[[-1.27000932  0.90023503 -1.48498319]
 [-1.28091827  0.88925077 -1.50083497]
 [-1.27558445  0.42662811 -0.3403496 ]
 …
 [-0.11457988 -0.66715402 -0.2319238 ]
 [-1.29058611  0.66537997 -0.98875691]
 [-1.31092484 -0.01184731 -0.06259766]]
Cluster 2:
[[-1.64586157 -0.30018409  1.48577955]
 [-1.53846524 -0.45356985  1.22558668]
 [-1.68582461 -0.25345485  1.22274751]
 …
 [-1.48913532 -0.1909924   0.50906421]
 [-1.58715222  0.11989883  0.153085  ]
 [-1.44338183 -0.2786564   0.63720414]]
Cluster 3:
[[ 2.02515375 -1.78620966  0.71166866]
 [ 2.72981511 -0.54551666  0.3913268 ]
 [ 2.714635   -0.70285331  0.83020182]
 …
 [ 2.09672081 -1.54120514 -0.07575902]
 [ 2.05494438 -1.42218053 -0.20866034]
 [ 1.90831914 -1.88870839  1.34089248]]
Cluster 4:
[[ 2.44172683 -0.7507556  -1.54543022]
 [ 4.07451774  1.12441346 -0.4355791 ]
 [ 2.86272445 -1.02797031 -1.1618    ]
 …
 [ 2.14747343 -0.12148466 -1.14639483]
 [ 2.01168482 -1.03059165 -1.23009044]
 [ 2.89368139 -0.4965629  -2.41242266]]

>>> Best labels of bisecting k-means among 20 times
[1 1 1 1 1 4 1 1 3 3 1 2 1 1 2 1 3 2 2 2 2 2 4 4 1 1 3 3 1 2 1 4 1 3 3 2 3
 3 3 1 3 4 3 3 2 2 1 1 1 1 0 1 2 1 3 3 3 3 3 1 3 3 1 2 2 2 2 2 2 3 4 2 2 1
 3 2 2 1 2 1 1 2 1 2 1 1 1 0 1 3 0 2 1 4 2 0 2 1 3 0]

>>> Best SSE of bisecting k-means among 20 times
46262.56628136219
```

[15]:
```python
kmeans_sklearn = KMeans(n_clusters=5, random_state=None, n_init=10)
labels_sklearn_best = kmeans_sklearn.fit_predict(pca_df)
sse_sklearn = kmeans_sklearn.inertia_


print("\n>>> SSE of scikit-learn k-means ")
print(sse_sklearn)
```

```
>>> SSE of scikit-learn k-means
```

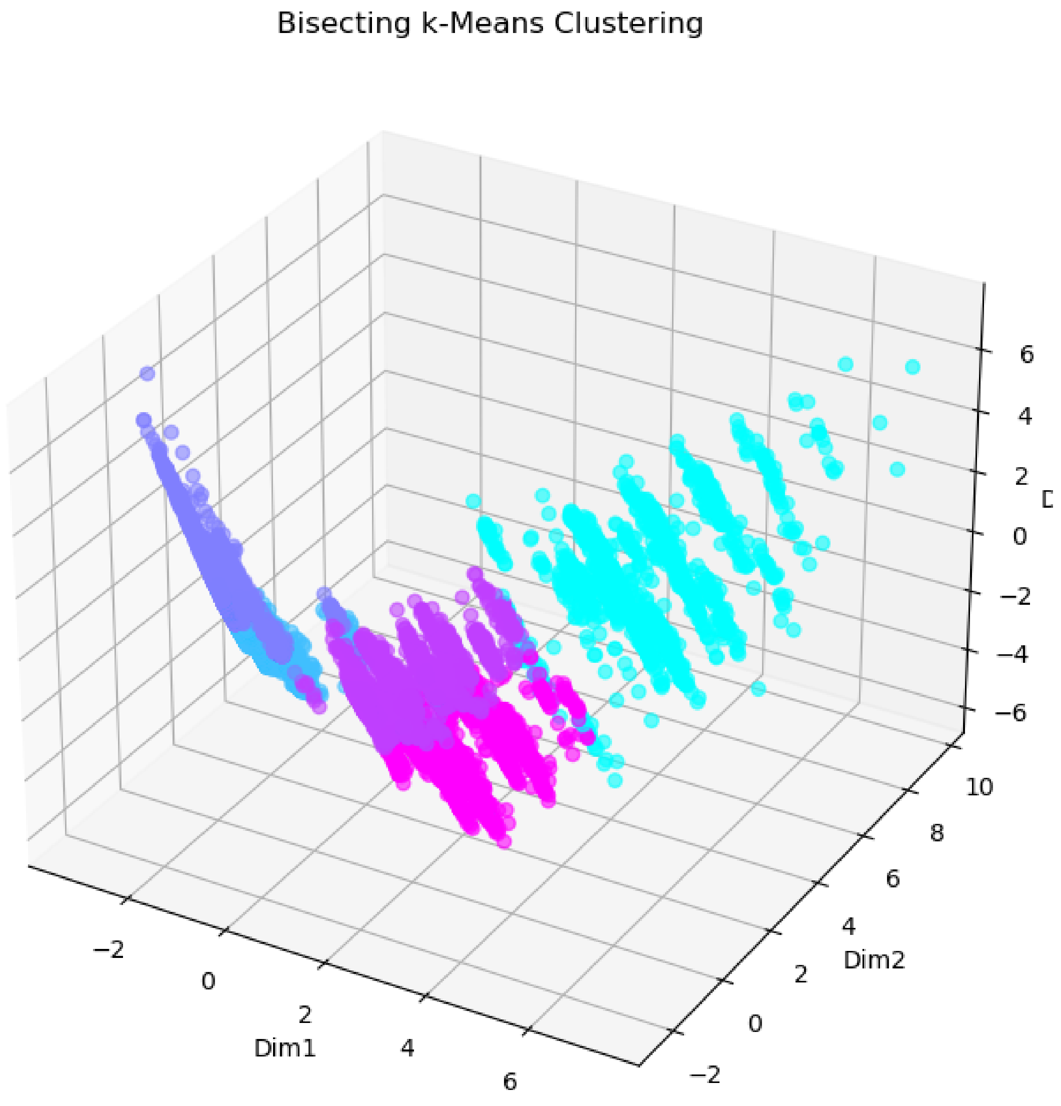
```
39263.9186668748
```

[16]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Ensure pca_df is a DataFrame or convert it to NumPy array
data_array = pca_df.values if isinstance(pca_df, pd.DataFrame) else pca_df

# Standard k-Means visualization
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for Standard k-Means
ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],␣
 ↪c=labels_kmeans_best, cmap='viridis', s=30, alpha=0.6)

# Add labels and title
ax.set_title("Standard k-Means Clustering")
ax.set_xlabel("Dim1")
ax.set_ylabel("Dim2")
ax.set_zlabel("Dim3")

plt.show()
```



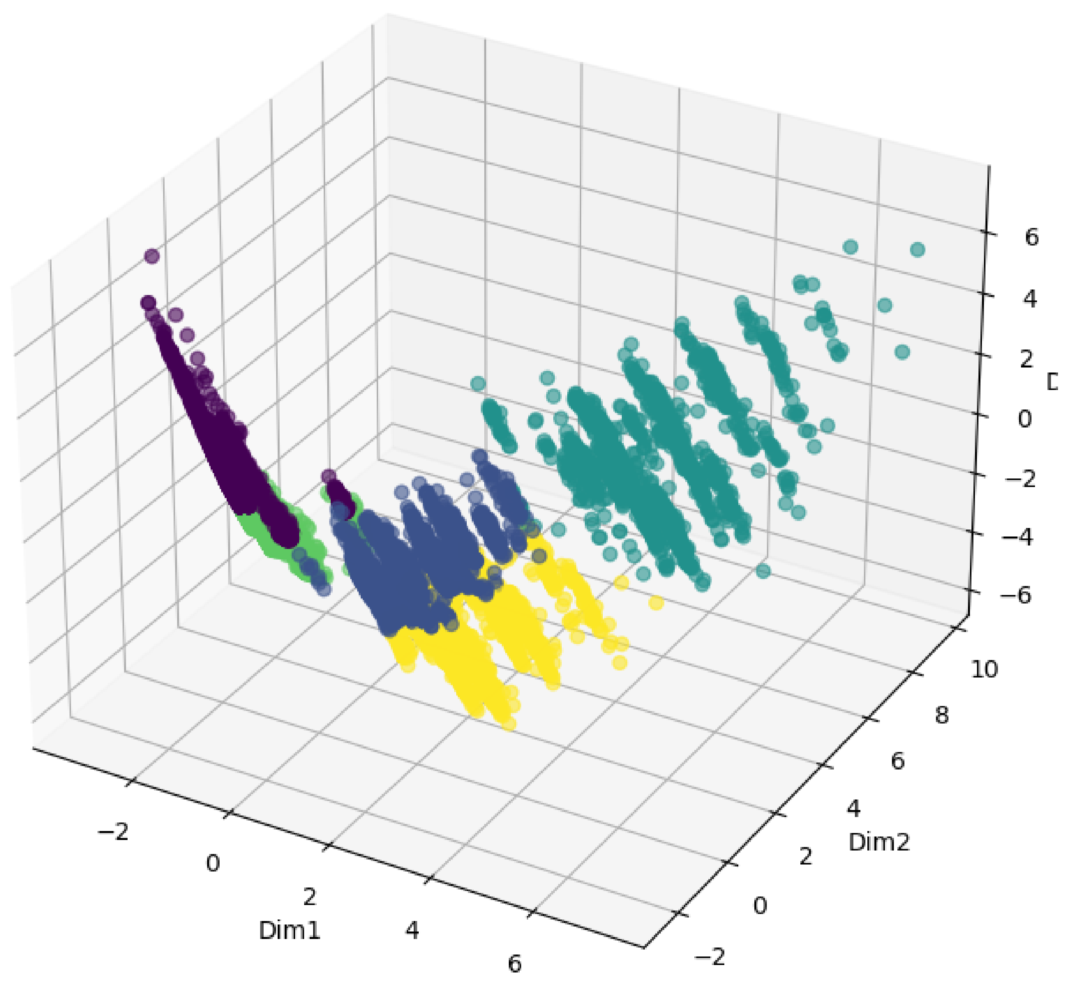
```
[17]: import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D

      # Ensure pca_df is a DataFrame or convert it to NumPy array
      data_array = pca_df.values if isinstance(pca_df, pd.DataFrame) else pca_df

      # Bisecting k-Means visualization
      fig = plt.figure(figsize=(10, 8))
      ax = fig.add_subplot(111, projection='3d')

      # Scatter plot for Bisecting k-Means
```

```
ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],␣
↪c=labels_bisect_best, cmap='cool', s=30, alpha=0.6)

# Add labels and title
ax.set_title("Bisecting k-Means Clustering")
ax.set_xlabel("Dim1")
ax.set_ylabel("Dim2")
ax.set_zlabel("Dim3")

plt.show()
```

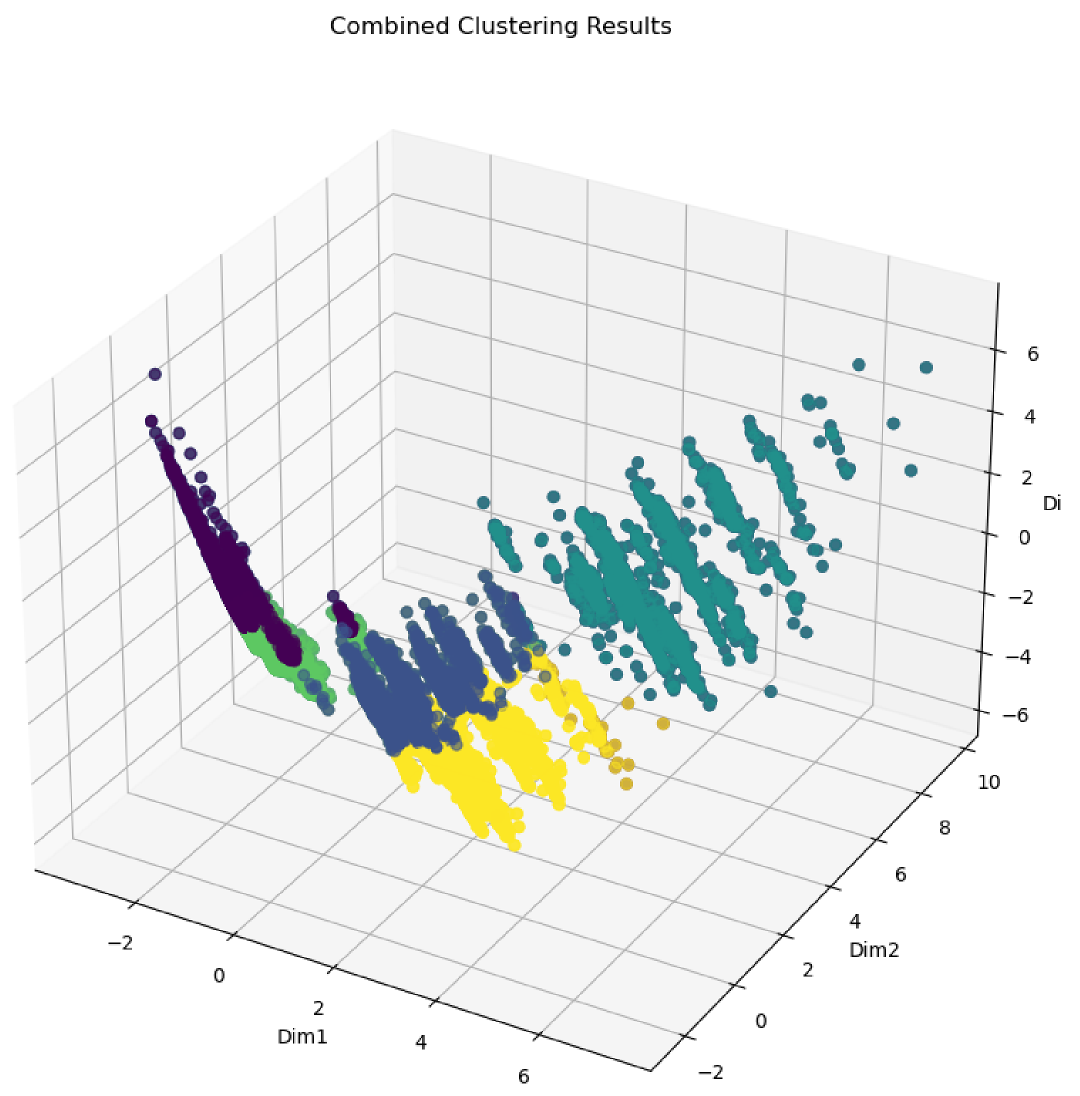


[18]:
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Ensure pca_df is a DataFrame or convert it to NumPy array
data_array = pca_df.values if isinstance(pca_df, pd.DataFrame) else pca_df

# Standard k-Means visualization
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for Standard k-Means
ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],␣
 ↪c=labels_sklearn_best, cmap='viridis', s=30, alpha=0.6)

# Add labels and title
ax.set_title("Sklearn k-Means Clustering")
ax.set_xlabel("Dim1")
ax.set_ylabel("Dim2")
ax.set_zlabel("Dim3")

plt.show()
```

Sklearn k-Means Clustering



```
[19]: import matplotlib.pyplot as plt
      from mpl_toolkits.mplot3d import Axes3D

      # Ensure pca_df is a DataFrame or convert it to NumPy array
      data_array = pca_df.values if isinstance(pca_df, pd.DataFrame) else pca_df

      # Combined visualization
      fig = plt.figure(figsize=(14, 10))
      ax = fig.add_subplot(111, projection='3d')

      # Scatter plot for Standard k-Means
      scatter1 = ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],
                             c=labels_kmeans_best, label='Standard k-Means', s=30,␣
       ↪alpha=0.6)

      # Scatter plot for Bisecting k-Means
      scatter2 = ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],
                             c=labels_bisect_best, label='Bisecting k-Means', s=30,␣
       ↪alpha=0.6)

      # Scatter plot for Scikit-learn k-Means
      scatter3 = ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],
                             c=labels_sklearn_best, label='Scikit-learn k-Means',␣
       ↪s=30, alpha=0.6)

      # Add labels, legend, and title
      ax.set_title("Combined Clustering Results")
      ax.set_xlabel("Dim1")
      ax.set_ylabel("Dim2")
      ax.set_zlabel("Dim3")

      plt.show()
```



(b)

```
[20]: print(f"SSE of best kmeans: {sse_kmeans_best}")
      print(f"SSE of best bisect-kmeans: {best_sse_bisect}")
      print(f"SSE of sklearn-kmeans: {sse_sklearn}")
```

```
SSE of best kmeans: 39262.99942029846
SSE of best bisect-kmeans: 46262.56628136219
SSE of sklearn-kmeans: 39263.9186668748
```

Through the plots from part (a) and the SSEs, we found that k-Means and Scikit-learn k-Means performed better than Bisecting k-Means, as Bisecting k-Means had the highest SSE. However, visually comparing the three clustering plots reveals no obvious pattern difference between them—most points overlap, and the clusters appear similar. The SSE of the best k-Means and Scikit-learn k-Means are nearly identical, but the difference lies in how the results were achieved: k-Means required 20 runs to obtain the best SSE, while Scikit-learn k-Means achieved the same result in just one run using the k-means++ initialization. Random initialization in k-Means can lead to inefficient first iterations, as the centroids may start far from the actual cluster centers and require significant adjustment, increasing computational cost. In contrast, k-means++ selects near-optimal initial centroids, reducing the need for multiple runs, improving convergence speed, and ensuring higher quality results. This makes Scikit-learn k-Means both computationally more efficient and more reliable than randomly initialized k-Means.

(c)

[21]:
```
df= pd.read_csv("2024_bank_marketing.csv", sep =';').dropna()
df['cluster'] = labels_sklearn_best
cluster_results=[]

for cluster_id in df['cluster'].unique():
    cluster_data = df[df['cluster'] == cluster_id]
    total_points = len(cluster_data)

    subscription_counts = cluster_data['subscription'].
  ↪value_counts(normalize=True)
    gini_index = 1 - np.sum(subscription_counts ** 2)
    yes_percentage = subscription_counts.get('yes', 0) * 100

    cluster_results.append({
        'cluster': cluster_id,
        'Total Points': total_points,
        'Gini Index': gini_index,
        'Yes Percentage (%)': yes_percentage
    })

results_df = pd.DataFrame(cluster_results)
results_df
```

[21]:
```
   cluster  Total Points  Gini Index  Yes Percentage (%)
0        3         15779    0.102500            5.418594
1        4          2362    0.464429           36.663844
2        1          9251    0.254599           14.971354
3        0         11064    0.077538            4.040130
4        2          1543    0.472327           61.762800
```

The table shows that Cluster 2, with the highest “Yes Percentage” (61.76%), indicates the highest proportion of customers who subscribed to the term deposit. However, its Gini Index is also high (0.472), suggesting that the category distribution within this cluster is more diverse.

Clusters 3 and 0 have low subscription rates (5.41% and 4.04%, respectively) and low Gini Index values (0.103 and 0.078), indicating that the category distribution in these clusters is more homogeneous, likely dominated by “No” subscriptions.

Clusters 4 and 1, with moderate subscription rates (36.67% and 14.97%, respectively). I believe that, in the context of identifying a cluster with the highest proportion of subscribers to term deposits, the “Yes Percentage” provides more valuable insights. For instance, we could explore potential optimizations or strategies for clusters with lower “Yes Percentage” values to identify opportunities for improvement or targeted interventions.

[22]:
```
# age summary in each cluster
numeric_features = ['age']
numeric_summary = df.groupby('cluster')[numeric_features].agg(['mean', 'std',␣
↪'median'])

numeric_summary.columns = ['_'.join(col).strip() for col in numeric_summary.
↪columns]
numeric_summary_df = numeric_summary.reset_index()
print(numeric_summary_df)

#martial summary in each cluster
categorical_summary = df.groupby('cluster')["marital"].
↪value_counts(normalize=True).unstack()
categorical_summary_df = categorical_summary.reset_index()

print("-----------------------------------------")
print(categorical_summary_df)
```

```
   cluster   age_mean    age_std  age_median
0        0  34.378706   6.765792        33.0
1        1  36.587612   8.807317        35.0
2        2  41.699287  15.309993        37.0
3        3  44.276190   8.432099        45.0
4        4  50.388230  16.411210        51.0
-----------------------------------------
marital  cluster  divorced   married    single   unknown
0              0  0.089389  0.551609  0.357466  0.001537
1              1  0.097071  0.522538  0.377905  0.002486
2              2  0.090732  0.524303  0.381724  0.003240
3              3  0.135813  0.694974  0.167374  0.001838
4              4  0.132091  0.635478  0.230313  0.002117
```

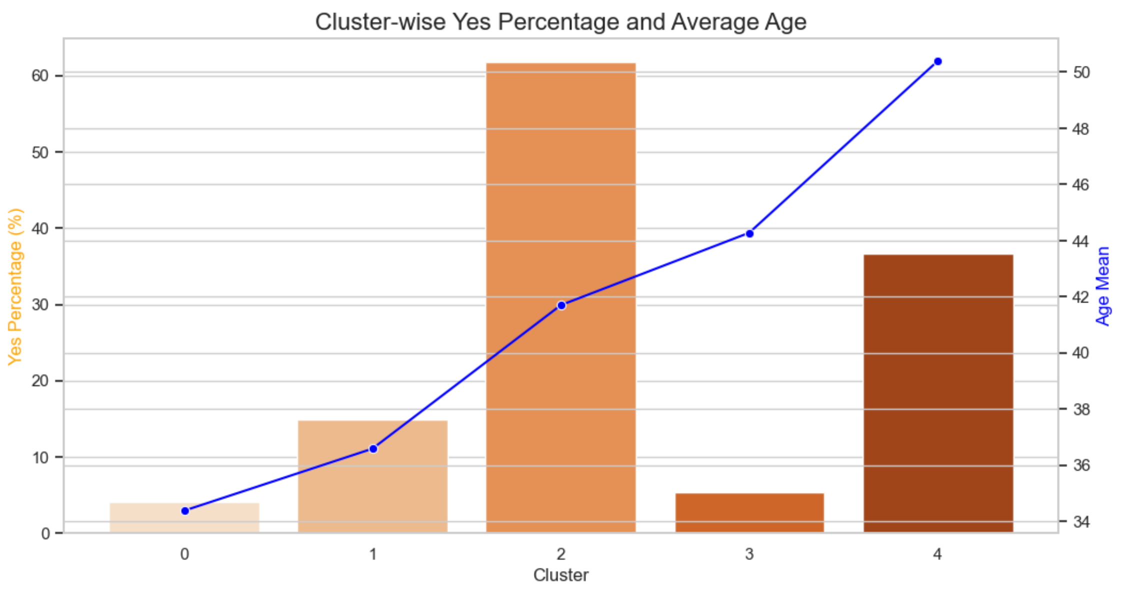
Through the “Yes Percentage,” we understand that Cluster 2 is likely dominated by “Yes” subscriptions. Curious about the characteristics of people in Cluster 2, I explored further by summarizing the mean, standard deviation, and median of age for each cluster.

The findings reveal that in Cluster 2:

The mean age of people is 41.7, which falls in the middle range of the population, suggesting that this group is neither particularly young nor particularly old.

The standard deviation of age is 15.31, indicating a relatively broad age distribution within the cluster. Another noteworthy insight is that in Cluster 2, Another noteworthy insight is that among the different clusters, single individuals are most concentrated in Cluster 2. This makes it a particularly important segment to prioritize, as it presents a valuable opportunity to target this demographic in our campaigns.

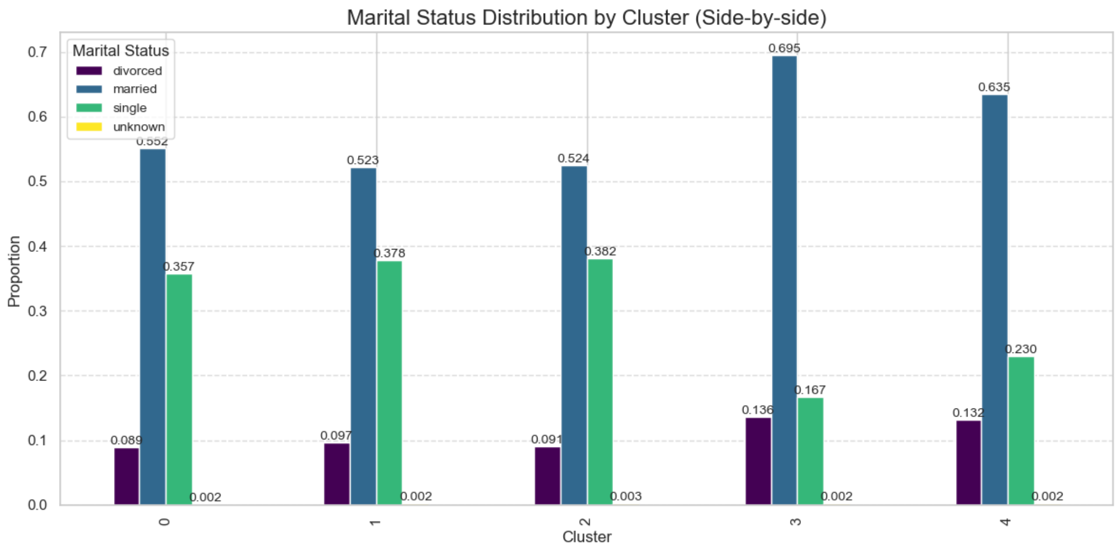
[23]:
```
merged_df = pd.merge(results_df, numeric_summary_df, on='cluster')

sns.set(style="whitegrid")


fig, ax1 = plt.subplots(figsize=(12, 6))


sns.barplot(
    x="cluster",
    y="Yes Percentage (%)",
    data=merged_df,
    palette="Oranges",
    ax=ax1
)


ax2 = ax1.twinx()
sns.lineplot(
    x="cluster",
    y="age_mean",
    data=merged_df,
    color="blue",
    marker="o",
    ax=ax2
)


ax1.set_xlabel("Cluster", fontsize=12)
ax1.set_ylabel("Yes Percentage (%)", fontsize=12, color="orange")
ax2.set_ylabel("Age Mean", fontsize=12, color="blue")
plt.title("Cluster-wise Yes Percentage and Average Age", fontsize=16)


plt.show()
```



This figure illustrates the relationship between “Yes Percentage” (subscription rate) and average age across different clusters. Cluster 4, with the highest average age (50), also has the second highest subscription rate (36.6%), suggesting that older customers are inclined to subscribe to term deposits, likely due to long-term financial planning needs. In contrast, Cluster 0, with the youngest average age (34), shows the lowest subscription rate (4%), indicating limited interest or need for such products among younger individuals. Cluster 2 demonstrates a highest subscription rate (62%) with a moderate average age (42), making it another significant group to target. Clusters 1 and 3 show moderate subscription rates and average ages, representing a balanced demographic. These insights highlight that older customers are the primary audience for term deposits, while younger clusters may require some promotion to drive engagement. This also illustrates some social imaginations, such as young people pay more attention to immediate consumption and experience, and tend to maintain high liquidity of funds.

[24]:
```
plt.figure(figsize=(12, 6))
ax = categorical_summary.plot(kind='bar', figsize=(12, 6), colormap='viridis')

for container in ax.containers:
    ax.bar_label(container, fmt='%.3f', label_type='edge', fontsize=10)

# Set chart labels and title
plt.title("Marital Status Distribution by Cluster (Side-by-side)", fontsize=16)
plt.xlabel("Cluster", fontsize=12)
plt.ylabel("Proportion", fontsize=12)
plt.legend(title="Marital Status", fontsize=10)
plt.grid(axis='y', linestyle='--', alpha=0.7)

plt.tight_layout()
plt.show()
```

```
<Figure size 1200x600 with 0 Axes>
```

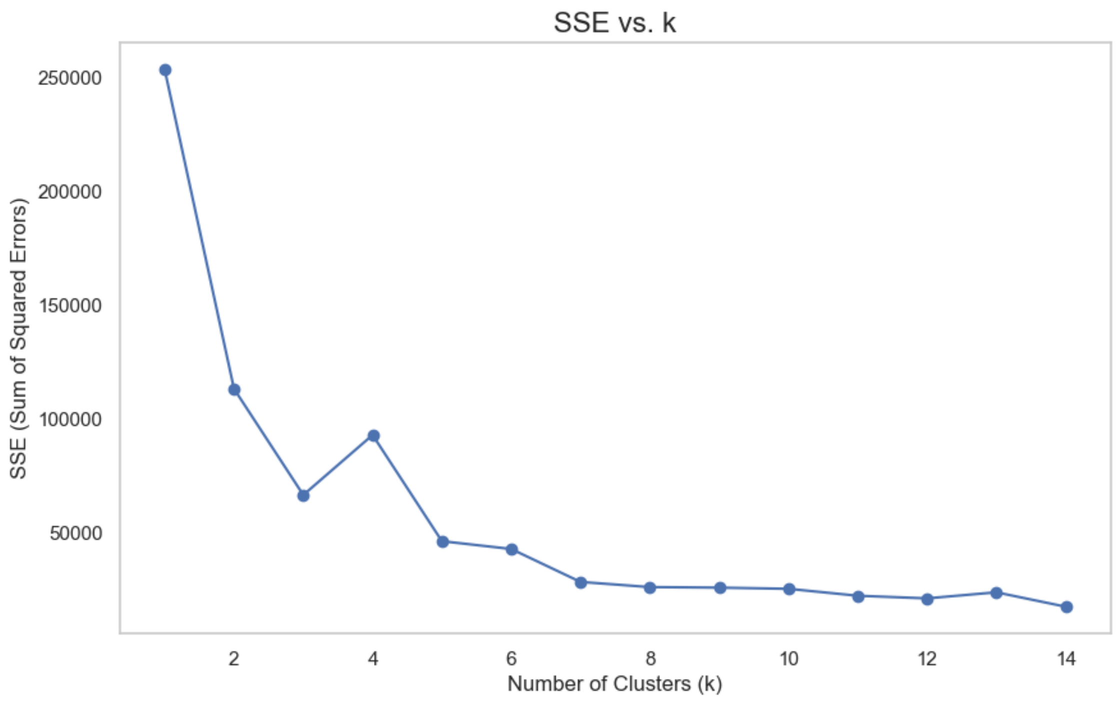


The chart illustrates that Cluster 2 has a significant proportion of single individuals (38.2%), highlighting a unique demographic characteristic. Additionally, the table shows that this cluster has the highest proportion of 'Yes' subscriptions, indicating strong engagement with term deposits. Combining these observations, I guess single people usually have no family responsibilities, have more flexibility in income allocation, and have more room for savings and investment.

(d)

[51]:
```
def kmeans(data,k=5):
    """
    >>> Main function of your k-Means implementation
    """
    # step 1:
    centroids = rand_center(data,k)
    converge = False
    while not converge:
        old_centroids = np.copy(centroids)
        # step 2 & 3; labels can be an array of labels for all the data points
        centroids, clusters_kmeans, labels = update_centroids(data,␣
↪old_centroids,k)
        # step 4
        converge = converged(old_centroids, centroids)
    return centroids, clusters_kmeans, labels



k_values = range(1, 15)
sse_values = []
```

```
label_resluts = []
# Perform k-Means for each k and calculate SSE
for k in k_values:
    centroids,clusters_kmeans, labels = kmeans(pca_df, k=k)
    sse = sse_cal(clusters_kmeans) # SSE for current k
    sse_values.append(sse)
    label_resluts.append(labels)

# Plot SSE vs. k
plt.figure(figsize=(10, 6))
plt.plot(k_values, sse_values, marker='o')
plt.title("SSE vs. k", fontsize=16)
plt.xlabel("Number of Clusters (k)", fontsize=12)
plt.ylabel("SSE (Sum of Squared Errors)", fontsize=12)
plt.grid()
plt.show()
```



[52]:
```
print(f"SSE of kmeans (k=7) :{sse_values[6]}")
```

```
SSE of kmeans (k=7) :28347.126569207794
```

[27]:
```
print(f"SSE of sklearn-kmeans: {sse_sklearn}")
```

```
SSE of sklearn-kmeans: 39263.9186668748
```

0.3.1 Summary table of k mean when k=7

[53]:
```
df['cluster_k=7'] = label_resluts[6]
cluster_results2=[]

for cluster_id in df['cluster_k=7'].unique():
    cluster_data = df[df['cluster_k=7'] == cluster_id]
    total_points = len(cluster_data)

    subscription_counts = cluster_data['subscription'].
 ↪value_counts(normalize=True)
    gini_index = 1 - np.sum(subscription_counts ** 2)
    yes_percentage = subscription_counts.get('yes', 0) * 100

    cluster_results2.append({
        'cluster': cluster_id,
        'Total Points': total_points,
        'Gini Index': gini_index,
        'Yes Percentage (%)': yes_percentage
    })

results_df2 = pd.DataFrame(cluster_results2)
results_df2
```

[53]:
```
   cluster  Total Points  Gini Index  Yes Percentage (%)
0        1          6221    0.122845            6.574506
1        3         10918    0.079539            4.149112
2        2          2363    0.464572           36.690647
3        5          9240    0.254545           14.967532
4        6          6188    0.072173            3.749192
5        0          3526    0.111521            5.927396
6        4          1543    0.472327           61.762800
```

0.3.2 Summary table of sklearn-kmeans in part c

[44]:
```
results_df
```

[44]:
```
   cluster  Total Points  Gini Index  Yes Percentage (%)
0        3         15779    0.102500            5.418594
1        4          2362    0.464429           36.663844
2        1          9251    0.254599           14.971354
3        0         11064    0.077538            4.040130
4        2          1543    0.472327           61.762800
```

[46]:
```
# calcualte Silhouette Score
score3 = silhouette_score(pca_df, labels_sklearn_best)
print("Silhouette Score for sklearn_best in part c:", score3)
```

```
score4 = silhouette_score(pca_df, label_resluts[5] )
print("Silhouette Score for k-mean when k=7:", score4)
```

```
Silhouette Score for sklearn_best in part c: 0.43917208234151777
Silhouette Score for k-mean when k=7: 0.4919972176598585
```

Based on the analysis of k-Means clustering with different k values, the optimal number of clusters is determined to be k=7. The Elbow Method identifies k=7 as the point where the reduction in SSE (Sum of Squared Errors) begins to level off, providing a good trade-off between compactness and simplicity, with an SSE of 28734 and a Silhouette Score of 0.492, slightly higher than k=5 (0.439). Compared to k=5, k=7 introduces more granular clusters that capture finer distinctions in data distribution, as reflected in the cluster characteristics (e.g., Cluster 4 with 61.76% "Yes Percentage") while maintaining interpretability. k=5 groups data into broader, simpler clusters (e.g., Cluster 2 with 61.76%) that are easier to interpret but less specific, k=7 the preferred choice for identifying distinct subgroups and k=5 more suitable for high-level insights.

0.4 Bouns

(a)

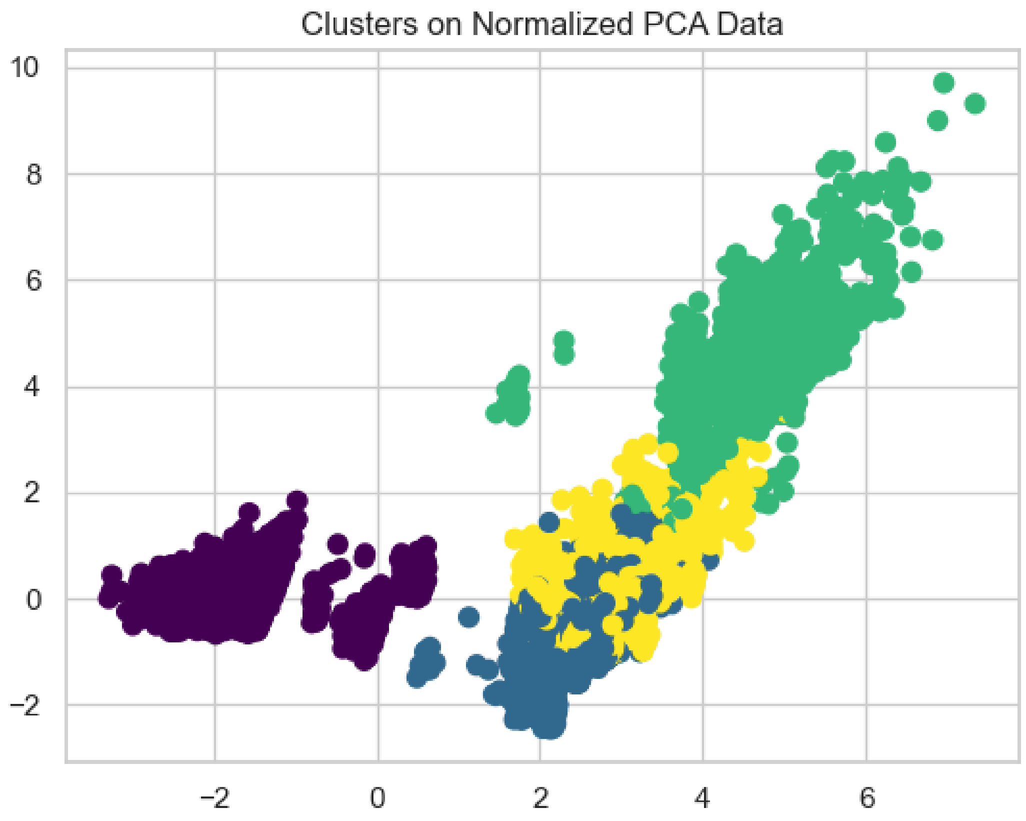
```
[31]: df= pd.read_csv("2024_bank_marketing.csv", sep =';')
df= df[['age', 'duration', 'campaign', 'pdays', 'previous', 'emp.var.rate',␣
 ↪'cons.price.idx','cons.conf.idx', 'euribor3m', 'nr.employed']].dropna()
df.head()

pca = PCA(n_components=3)
pca_result = pca.fit_transform(df)
pca_df2 = pd.DataFrame(pca_result, columns=['PC1', 'PC2', 'PC3'])
explained_variance_ratio = pca.explained_variance_ratio_
cumulative_variance_ratio = explained_variance_ratio.cumsum()
print("\nExplained Variance Ratio (Per Component):")
print(explained_variance_ratio)
print("\nCumulative Variance Ratio:")
print(f"{cumulative_variance_ratio}\n")
print("PCA Result (First 5 Rows):")
pca_df2.head()
```

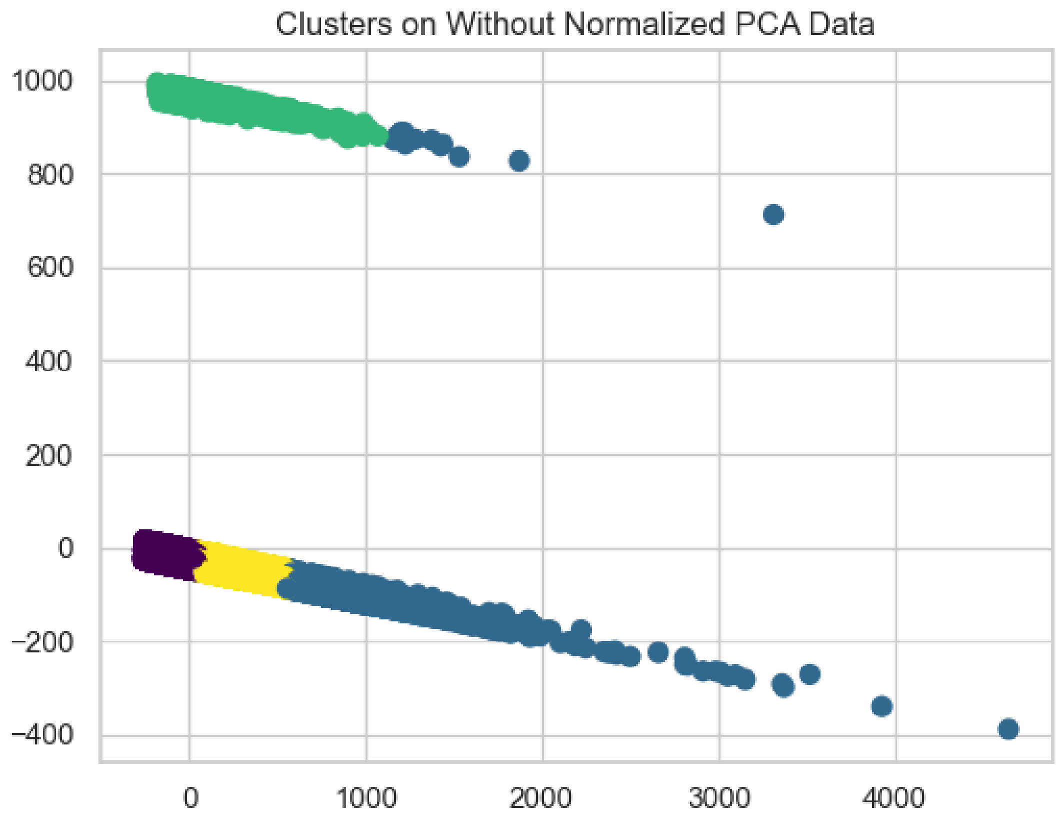
```
Explained Variance Ratio (Per Component):
[0.62711102 0.33073767 0.04086241]

Cumulative Variance Ratio:
[0.62711102 0.95784869 0.9987111 ]

PCA Result (First 5 Rows):
```

```
[31]:           PC1        PC2        PC3
     0   -0.206235 -39.770308 -17.792095
     1 -111.889897 -31.396972 -16.977814
     2  -35.108277 -37.190957 -17.545254
```

```
3 -109.896335 -31.579310 -16.999056
4   45.663859 -43.208563 -18.126370
```

[32]:
```
#normlized pca data
kmeans_normalized = KMeans(n_clusters=4, random_state=42,n_init=10).fit(pca_df)
plt.scatter(pca_df.iloc[:, 0], pca_df.iloc[:, 1], c=kmeans_normalized.labels_,␣
 ↪cmap='viridis', s=50)
plt.title("Clusters on Normalized PCA Data")
plt.show()

#without normalized pca data
kmeans_unnormalized = KMeans(n_clusters=4, random_state=42,n_init=10).
 ↪fit(pca_df2)
plt.scatter(pca_df2.iloc[:, 0], pca_df2.iloc[:, 1], c=kmeans_unnormalized.
 ↪labels_, cmap='viridis', s=50)
plt.title("Clusters on Without Normalized PCA Data")
plt.show()

sse_normalized = kmeans_normalized.inertia_
print(f"SSE of Normalized PCA Data {sse_normalized}")
sse_unnormalized = kmeans_unnormalized.inertia_
print(f"SSE of UnNormalized PCA Data: {sse_unnormalized}")

print("-------------------------------------------------------------------------")
silhouette_normalized = silhouette_score(pca_df, kmeans_normalized.labels_)


silhouette_unnormalized = silhouette_score(pca_df2, kmeans_unnormalized.labels_)

print(f"Silhouette Score for normalized data: {silhouette_normalized}")
print(f"Silhouette Score for unnormalized data: {silhouette_unnormalized}")
```


Clusters on Normalized PCA Data
10
8
6
4
2
0
−2
−2
0
2
4
6



```
SSE of Normalized PCA Data 52081.04922017116
SSE of UnNormalized PCA Data: 816333981.5692706
-----------------------------------------------------------------------
Silhouette Score for normalized data: 0.6226165255301042
Silhouette Score for unnormalized data: 0.5696532634141414
```

By comparing the clustering performance of normalized and unnormalized data, both datasets were processed with PCA for dimensionality reduction. From the two plots, it is evident that clustering on normalized data yields superior results, with the data distributed evenly after PCA, clear cluster boundaries, and well-separated clusters in space, indicating that normalization ensures equal contributions from all features to the clustering process. In contrast, clustering on unnormalized data shows stretched distributions, with most data points concentrated near the origin and some points pulled far away, revealing that feature scale differences significantly impact the clustering results, allowing certain large-scale features to dominate while overshadowing the contributions of other features.

In additional to plots, Noramlized data With significantly lower SSE (52081.05 compared to 816339381.57) and higher Silhouette Score (0.6226 compared to 0.5697), the results indicate that normalization effectively balances feature contributions, resulting in tighter and better-separated clusters, while the unnormalized data is negatively impacted by feature scale differences.

(b)

[33]:
```
selected_features = ['age', 'duration', 'pdays']
new_df = df[selected_features]
scaler = StandardScaler()
normalized_new_data = scaler.fit_transform(new_df)

normalized_new_data = pd.DataFrame(normalized_new_data, columns=new_df.columns)

normalized_new_data.head()

kmeans_new = KMeans(n_clusters=5, random_state=42,n_init=10).
  ↪fit(normalized_new_data)
sse_new = kmeans_new.inertia_
silhouette_new = silhouette_score(normalized_new_data, kmeans_new.labels_)

print(f"SSE for selected features: {sse_new}")
print(f"Silhouette Score for selected features: {silhouette_new}")
```

```
SSE for selected features: 28047.28341354733
Silhouette Score for selected features: 0.4507190212180107
```

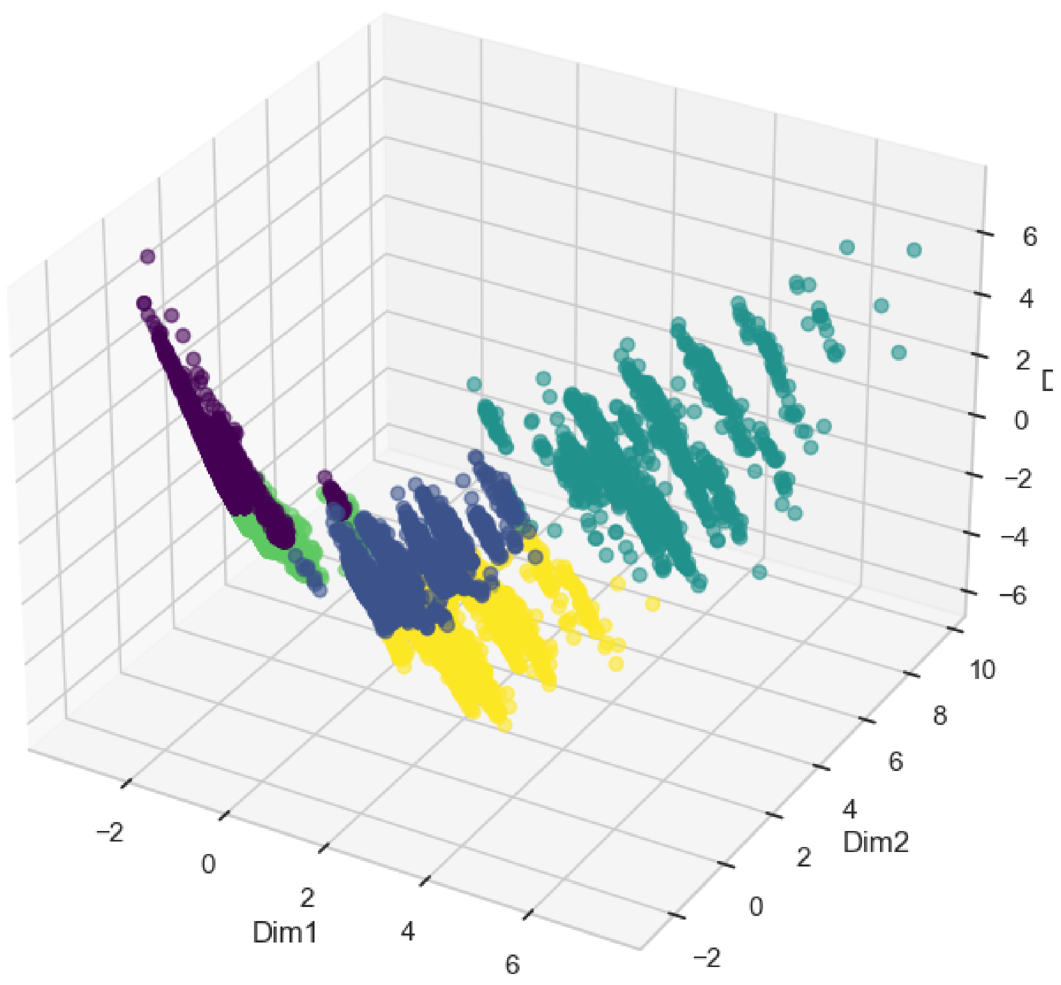
[34]:
```
print(f"SSE of sklearn-kmeans in part c: {sse_sklearn}")
print("Silhouette Score for sklearn_best in part c:", score3)
```

```
SSE of sklearn-kmeans in part c: 39263.9186668748
Silhouette Score for sklearn_best in part c: 0.4391720823415177
```

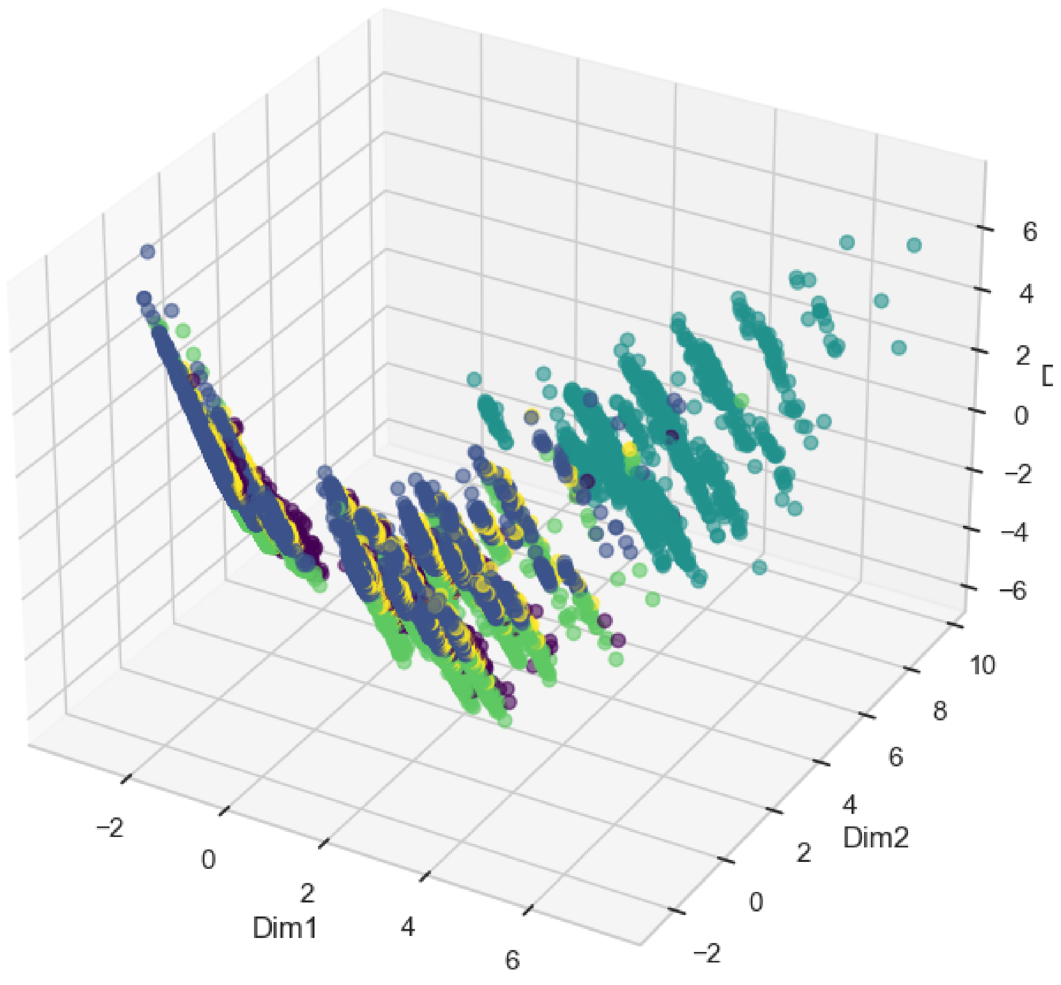
[35]:
```
# Ensure pca_df is a DataFrame or convert it to NumPy array
data_array = pca_df.values if isinstance(pca_df, pd.DataFrame) else pca_df

# Standard k-Means visualization
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for Standard k-Means
ax.scatter(data_array[:, 0], data_array[:, 1], data_array[:, 2],␣
  ↪c=labels_sklearn_best, cmap='viridis', s=30, alpha=0.6)

# Add labels and title
ax.set_title("Sklearn k-Means Clustering")
ax.set_xlabel("Dim1")
ax.set_ylabel("Dim2")
ax.set_zlabel("Dim3")

plt.show()

# Ensure pca_df is a DataFrame or convert it to NumPy array
data_array2 = pca_df.values if isinstance(normalized_new_data, pd.DataFrame)␣
  ↪else pca_df
```

```
# Standard k-Means visualization
fig = plt.figure(figsize=(10, 8))
ax = fig.add_subplot(111, projection='3d')

# Scatter plot for Standard k-Means
ax.scatter(data_array2[:, 0], data_array2[:, 1], data_array2[:, 2],␣
 ↪c=kmeans_new.labels_, cmap='viridis', s=30, alpha=0.6)

# Add labels and title
ax.set_title("Selected features k-Means Clustering")
ax.set_xlabel("Dim1")
ax.set_ylabel("Dim2")
ax.set_zlabel("Dim3")

plt.show()
```

Sklearn k-Means Clustering



Selected features k-Means Clustering



```
[36]: df= pd.read_csv("2024_bank_marketing.csv", sep =';').dropna()
      df2= df[['age', 'duration', 'pdays','subscription']]

      df2['cluster'] = kmeans_new.labels_

      cluster_results=[]

      for cluster_id in df2['cluster'].unique():
          cluster_data = df2[df2['cluster'] == cluster_id]
          total_points = len(cluster_data)
```

```
    subscription_counts = cluster_data['subscription'].
↪value_counts(normalize=True)
    gini_index = 1 - np.sum(subscription_counts ** 2)
    yes_percentage = subscription_counts.get('yes', 0) * 100

    cluster_results.append({
        'cluster': cluster_id,
        'Total Points': total_points,
        'Gini Index': gini_index,
        'Yes Percentage (%)': yes_percentage
    })

results_df3 = pd.DataFrame(cluster_results)
results_df3
```

```
C:\Users\tanxi\AppData\Local\Temp\ipykernel_2808\472510772.py:4:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df2['cluster'] = kmeans_new.labels_
```

[36]:
```
   cluster  Total Points  Gini Index  Yes Percentage (%)
0        3         12309    0.111435            5.922496
1        1         19442    0.075883            3.950211
2        4          5506    0.365283           24.046495
3        0          1281    0.485652           58.469945
4        2          1461    0.460431           64.065708
```

[37]:
```
results_df
```

[37]:
```
   cluster  Total Points  Gini Index  Yes Percentage (%)
0        3         15779    0.102500            5.418594
1        4          2362    0.464429           36.663844
2        1          9251    0.254599           14.971354
3        0         11064    0.077538            4.040130
4        2          1543    0.472327           61.762800
```

By comparing SSE and Silhouette Score, the clustering performance with the selected features (Age, Duration, and Pdays) is superior to that with all features. The SSE for the selected features is 28047.28, significantly lower than the SSE for all features, which is 39263.58, indicating tighter clusters with the selected features. Additionally, the Silhouette Score for the selected features is 0.450, slightly higher than the Silhouette Score for all features (0.439), suggesting improved cluster separation and cohesion. These results demonstrate that the selected feature combination is more effective in clustering customers, particularly when the features are highly relevant to customer behavior. From the 3D visualizations, the clustering based on all features(Sklearn k-

Means Clustering) shows a wider distribution of data points in the space, with relatively clear cluster boundaries but some overlap between clusters, especially along certain directions (e.g., Dim2 and Dim3). In contrast, the clustering with selected features shows a more concentrated distribution of data points, with significantly enhanced separation between clusters along certain dimensions. In the clustering based on all features, Cluster 4 has a “Yes Percentage” of 61.76%, indicating a high-subscription group, but it also has a high Gini Index (0.472), suggesting significant behavioral diversity within the cluster. On the other hand, in the clustering with selected features, Cluster 4 achieves a higher “Yes Percentage” of 64.07%, showing that the selected features can better identify high-subscription groups. Furthermore, Cluster 1 in the selected feature clustering is the largest cluster (19442 points), with the lowest “Yes Percentage” (3.95%) and the lowest Gini Index (0.076), indicating that it represents a consistent low-subscription group. In summary, the clustering with selected features shows slight improvements in tightness and separation compared to the clustering with all features, making it more suitable for analyzing subscription behavior. However, clustering with all features provides a more comprehensive view, which is better for capturing complex patterns and interactions between features.