AAI-510-01: Assignment 2.1 - Ian Feekes
https://github.com/USD-AAI/aai-510-01-su22-new-ianfeekes-sandiego

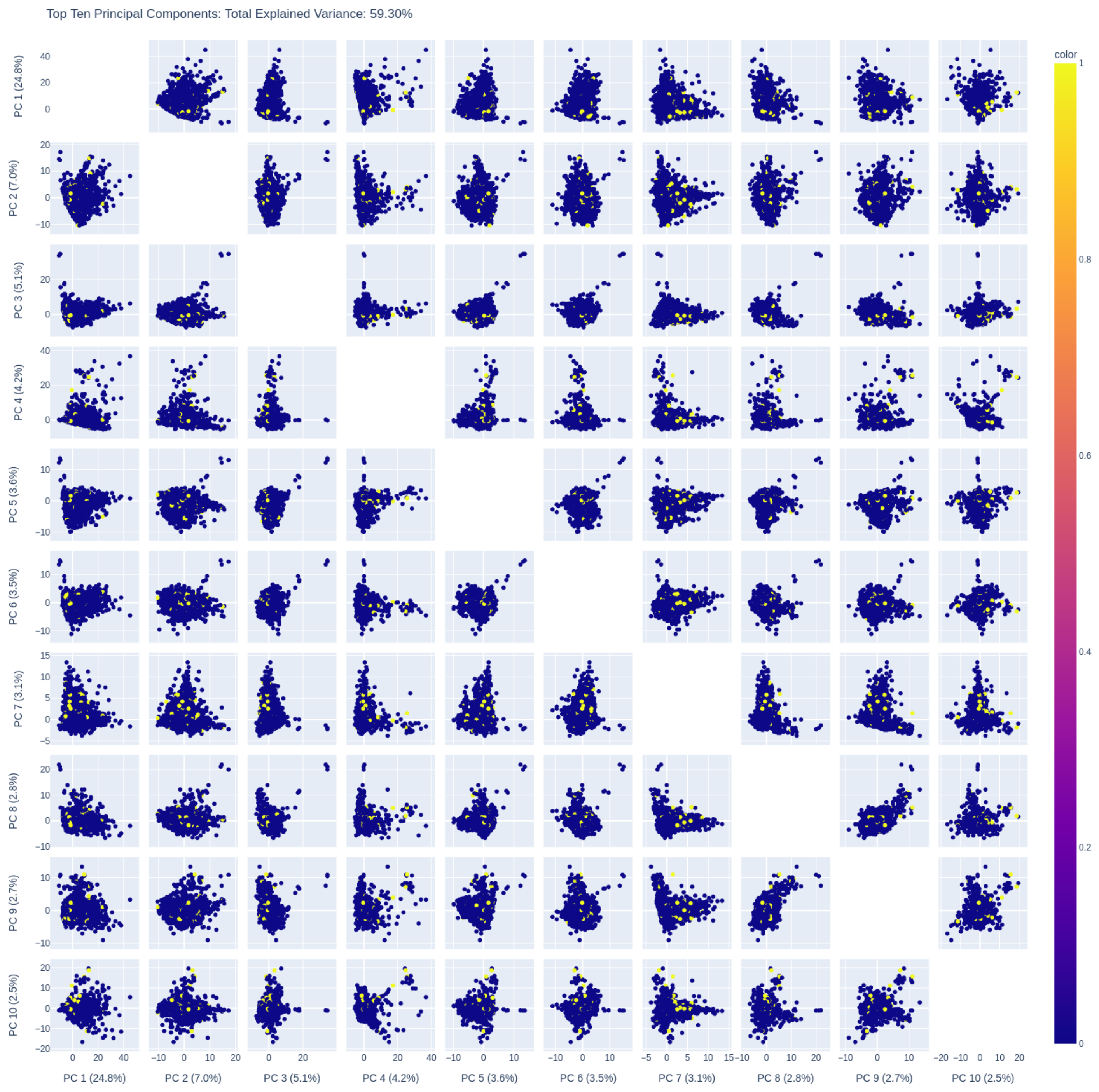
PCA Creation and Plotting

This snippet will show the shape of the data, create a PCA with the top 10 components, and plot a scatter matrix of each of the 10 Principal Components and save it to /images/pca.png

```python
# Very small value to ensure standardization success
deltaMean = .001

# Number of Principal Components we want to create
numComponents = 40
# Flag for either dropping rows with null values or filling in mean values
deleteRows = True
deleteDescription = ""

# Clean data and print useful output
if deleteRows:
   deleteDescription = "deleted_rows"
   if debug == True:
       print("Deleting missing entries: previous shape: ", df_num.shape)
   df_num = df_num.dropna()
   if debug == True:
       print("New shape after removing missing entries: ", df_num.shape)
else:
   deleteDescription = "mean_values_filled"
   df_num = df_num.fillna(df_num.mean(axis=1))
   if debug == True:
       print("Filled missing values with mean column values")

# Show the number of independant continuous variables before running PCA which would bring us down to 10
if debug == True:
   print("Number of continuous variables before PCA: ", len(df_num.columns))

# PCA has difficulty computing distance with different scaling - this normalizes
scaler = StandardScaler()
# Save attributes of dataframe to be reconverted
df_num_columns = df_num.columns
df_num_nparr = scaler.fit_transform(df_num)
# Move back into a data frame
df_num = pd.DataFrame(df_num_nparr, columns = df_num_columns)
scaled_features = df_num_nparr
# Verify standardization
assert(np.mean(scaled_features) < deltaMean)
assert(np.std(scaled_features) == 1)

# Create a PCA class with desired number of  principal components
pca = PCA(n_components = numComponents)
components = pca.fit_transform(scaled_features)
```

```python
# Plot matrix of Principal Component scatter plots
if savePlots == True:
    # Create appropriate file name and store it into clean directory
    path = "images/pca/" + deleteDescription
    if not os.path.exists(path):
        os.mkdir(path)
    # Calculates total variance as label
    total_var = pca.explained_variance_ratio_.sum() * 100
    labels = {
        str(i): f"PC {i+1} ({var:.1f}%)"
        for i, var in enumerate(pca.explained_variance_ratio_ * 100)
    }
    # Create and save plot
    fig = px.scatter_matrix(components,
                            labels=labels,
                            dimensions=range(10),
                            color = df_num[targetColumnName])
                            #color=data[targetColumnName])
    fig.update_layout(title=f' Top Ten Principal Components: Total Explained Variance:
       {total_var:.2f}%',
                      dragmode='select',
                      width=1500,
                      height=1500,
                      hovermode='closest')
    fig.update_traces(diagonal_visible=False)
    fig.write_image(path + "/pca.png")
    if debug == True:
        print("PCA scatter Plotting...success")
elif debug == True:
    print("savePlots flag set to false...skipping PCA scatter plotting")

print(pca.get_feature_names_out())

if debug == True:
    print("PCA creation...success")
```

Top 10 Principal Components:



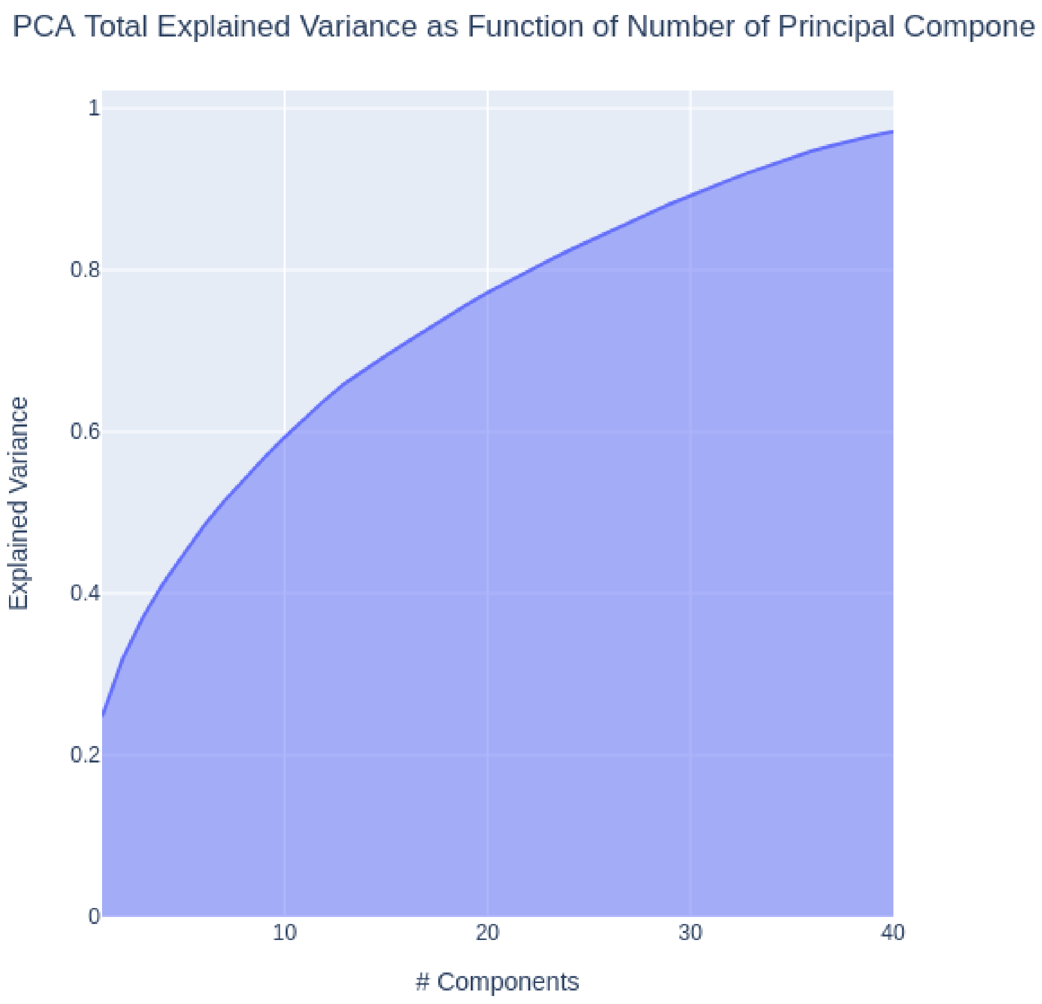
PCA Total Variance Plotting

Code

The plots are saved in /images/pca/mean_values_filled/pca_explained_variance.png and /images/pca/deleted_rows/pca_explained_variance.png. Mean values filled:

```python
# Only do this if we are saving our plots, else skip
if savePlots == True:
   # Cumulative explained variance for analysis
   exp_var_cumul = np.cumsum(pca.explained_variance_ratio_)
   # Create and save the plot with explained variance
   fig = px.area(x=range(1, exp_var_cumul.shape[0] + 1),
                 y=exp_var_cumul,
                 labels={"x": "# Components", "y": "Explained Variance"})
   fig.update_layout(title="PCA Total Explained Variance as Function of Number of Principal Components",
                     width = 600,
                     height = 600)
   fig.write_image("images/pca/" + deleteDescription + "/pca_explained_variance.png")
   if debug == True:
       print("PCA Total Explained Variance Plotting...success")
elif debug == True:
   print("Save Plots turned off...skipping plotting PCA total variance analysis")
```

Visualization:



Analysis/Density of the Data

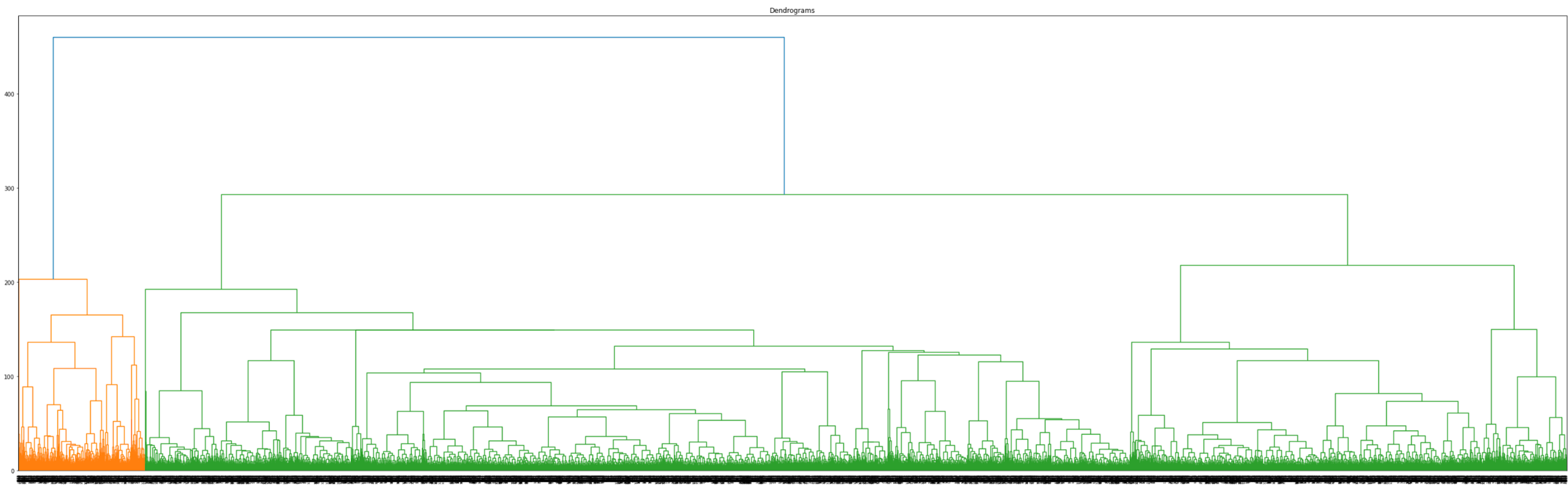
This plot shows that with standardized data, 95% variance (desired amount) is only really achieved with about 35 principal components (which is about half that of the total number of variables). The top 10 components cover about 60% of the cumulative explained variance. However selecting 10 components does meet a point of diminishing returns on incremental explained variance.

Hierarchical Clustering

```python
# Import hierarchical clustering libraries
from scipy.cluster.hierarchy import dendrogram, linkage
import scipy.cluster.hierarchy as sc
from sklearn.cluster import AgglomerativeClustering
import scipy.cluster.hierarchy as shc

# Variable to allow manipulating plot size
dendogramSize = (50, 15)
# Variable to allow manipulating the number of clusters we are thresholding heirarchical cluster
numClusters = 5

# Heirarchical clustering is computationally intensive so only do so if necessary
if True:
   # Plot dendrogram
   plt.figure(figsize=dendogramSize)
   plt.title("Dendrograms")

   # Create dendrogram
   dendrogram = sc.dendrogram(sc.linkage(df_num, 'ward'),
                              show_leaf_counts=True)
   # Save plot
   plt.savefig("images/hierarchicalClustering/hierarchicalClustering.png")

   # Output message
   if debug == True:
       print("Dendrogram Plotting...success")
elif debug == True:
   print("Save Plots turned off...skipping plotting dendrogram for hierarchical clustering")

# Perform Agglomerative Clustering for the given number of clusters
dataArr = df_num.iloc[:,:].values
cluster = AgglomerativeClustering(n_clusters=numClusters, affinity='euclidean', linkage='ward')
cluster.fit_predict(df_num)

if savePlots == True:
   # Iterate through columns for given number of clusters
   for i in range(0, len(df_num.columns), 2):
       # Create appropriate file name and store it into clean directory
       path = "images/hierarchicalClustering/" + str(numClusters)
       if not os.path.exists(path):
           os.mkdir(path)
       fileName = path + "/" + df_num.columns[i] + '_' + df_num.columns[i+1]
       # Configure the plot given the number of clusters
       plt.figure(figsize = (10, 7))
       plt.scatter(dataArr[:,i], dataArr[:,i+1], c = cluster.labels_, cmap = 'rainbow')
       plt.title("Hierarchical Clustering: " + fileName)
       plt.xlabel(df_num.columns[i], fontsize=16)
       plt.ylabel(df_num.columns[i+1], fontsize=16)
       plt.savefig(fileName)
       if debug == True:
           print("hierarchical clustering plot created: ", fileName)
   if debug == True:
       print("Finished plotting hierarchical cluster scatter plots")
elif debug == True:
   print("Save Plots turned off...skipping plotting scatter plots of continuous variables with cluster
labels")
```

Visualization

Plots output from this subsection are located in images/hierarchicalClustering



Best Cut

While there is not necessarily a clear-cut way to determine where to cut a dendrogram, analysis of the scatter plots between variables tends to show much more clear segregation between 3 and 4 clusters: I would tend to cut at a threshold leading to 4 clusters. To get a cut of 4 clusters, a Euclidean distance threshold of 210 could achieve this cut.

Distance Measure Used

Summary

Largely Euclidean distance was used for building the visualization and clustering models as the default, however “euclidean”, “l1”, “l2”, “manhattan”, “cosine”, and “precomputed” parameters were each evaluated for building the models.

Justification

Euclidean distance is fairly sufficient and uniform in constructing clusters that have been standardized.This data is looking exclusively at continuous variables, and not at correlation (1-r, like in genes), where the distance between two correlated vectors being 0 is of greater importance. Since there are still some slight outliers in the dataset, maximum distance would lead to possibly different sequences of the clustering that would be suboptimal. Even more so, manhattan distance would lead to less value in the hierarchical clustering algorithm given the high degree of dimensionality of the data set (or even simply taking our top 10 Principal Components).

Compare Branching Order to CART:

The dendrogram branching order splits off into couples at thresholds having differing numbers of point for each node until eventually all data points are accounted for. In random tree classifiers/regressors and forests, each tree is split off likewise into couples, however the number of branches varies significantly as branches are based on threshold values of variables moving down into a classification decision. This is opposed to a branch forming as the result of an iteration of the hierarchical classification algorithm grouping another point into a cluster

K-Means Clustering Visualization and Analysis

```python
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score

# For visualization of how many clusters to go with and saving runtime we threshold
maxClusters = 20
initializationType = "random"#"k-means++"

# If we are skipping the plotting & analytics, simply create our clustering with best found value
chosenOptimalClusters = 4

# K-means has difficulty computing distance with different scaling - this normalizes
scaler = StandardScaler()
scaled_features = scaler.fit_transform(df_num)

# Performs 10 runs of K-means algorithm with 3 clusters and maximum of 300 iterations.
kmeansArgs = {
   "init" : initializationType,   # Both have been tried - optimal to intelligently initialize centroids
   "n_init" : 10,                 # Performs 10 runs
   "max_iter" : 300,              # At most 300 iterations - usually doesn't go through too many
   "random_state" : 42
}

# Compute and plot k-means cluster analysis
if savePlots == True:
   # K-means metric lists
   # Sum of Squares error for Euclidean distance
   sse = []
   # How well data point fits inside its own cluster and how far it is from other clusters
   silhouetteCoefficients = []
   # Run k-means parameters with various numbers of clusters and obtain metrics
   for k in range(1, maxClusters+1):
       kmeans = KMeans(n_clusters=k, **kmeansArgs)
       kmeans.fit(scaled_features)
       sse.append(kmeans.inertia_)
       # Silhouette Score needs at least two clusters to run
       if k > 1:
           score = silhouette_score(scaled_features, kmeans.labels_)
           silhouetteCoefficients.append(score)
   # Create a nice folder to store the plots, if necessary
   path = "images/k_means/" + initializationType
   if not os.path.exists(path):
       os.mkdir(path)
   # Plot SSE as function of number of clusters
   plt.plot(range(0, len(sse)), sse)
   plt.xticks(range(0, len(sse)+1))
   plt.xlabel("Number of Clusters")
   plt.ylabel("SSE")
   plt.savefig(path + "/sse_to_clusters.png")
   plt.clf()
   # Plot silhouette coefficients as function of number of clusters
   plt.plot(range(0, len(silhouetteCoefficients)), silhouetteCoefficients)
   plt.xticks(range(0, len(silhouetteCoefficients)+1))
   plt.xlabel("Number of Clusters")
   plt.ylabel("Silhouette Coefficients")
   plt.savefig(path + "/silhouette_cofefficients_to_clusters.png")
   plt.clf()
```

```
    # Debugging output
    if debug == True:
        print("K-means cluster plots...success")
elif debug == True:
        print("Save plots off: skipping K-means cluster plots")

# After visualization we save our k-means data structure with optimal number of clusters
kmeans = KMeans(n_clusters=chosenOptimalClusters, **kmeansArgs)
#kmeans.fit(scaled_features)
y_predict= kmeans.fit_predict(scaled_features)

if savePlots == True:
    # Iterate through columns for given number of clusters
    for i in range(0, len(df_num.columns), 2):
        # Create appropriate file name and store it into clean directory
        path = "images/k_means/" + str(chosenOptimalClusters)
        if not os.path.exists(path):
            os.mkdir(path)
        fileName = path + "/" + df_num.columns[i] + '_' + df_num.columns[i+1]
        # Configure the plot given the number of clusters
        plt.figure(figsize = (10, 7))
        plt.scatter(scaled_features[:,i], scaled_features[:,i+1], c = y_predict, cmap = 'rainbow')
        plt.title("K-Means Clustering: " + fileName)
        plt.xlabel(df_num.columns[i], fontsize=16)
        plt.ylabel(df_num.columns[i+1], fontsize=16)
        plt.savefig(fileName)
        if debug == True:
            print("K-means clustering plot created: ", fileName)
    if debug == True:
        print("Finished plotting K-means cluster scatter plots")
elif debug == True:
    print("Save Plots turned off...skipping plotting scatter plots of continuous variables with k-means
cluster labels")


if debug == True:
    print("K-means initialization and visualization...success")
```

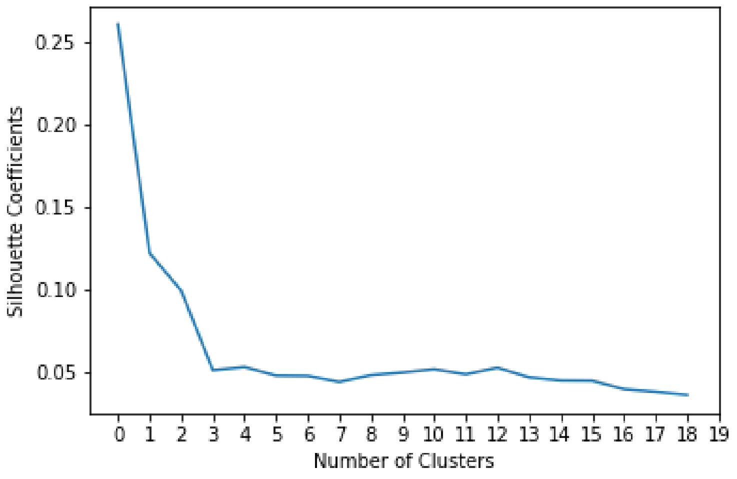
Here, necessary libraries for k-means are imported and the data-set is scaled to be normalized for Euclidean distance. Afterwards, if desired (i.e. "savePlots" flag is on), plots of functions for k-means metrics to the number of clusters are created and saved.

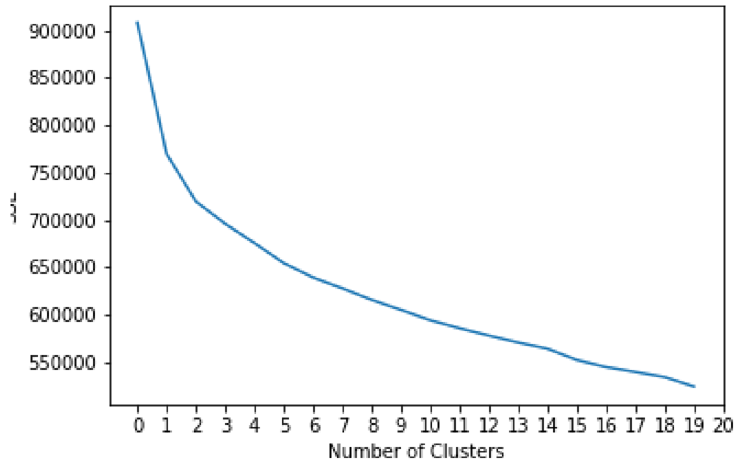
The visualization portion of this following block can simply be skipped, and k-means will be ran on our optimal number of clusters, which is 3.

The initial centroid points were ran both with random and with k-means++, which initializes the centroids. Most subsequent runs past analysis and visualization were ran with k-means++ because it is most likely to compute a competitive solution with the original k-means algorithm.

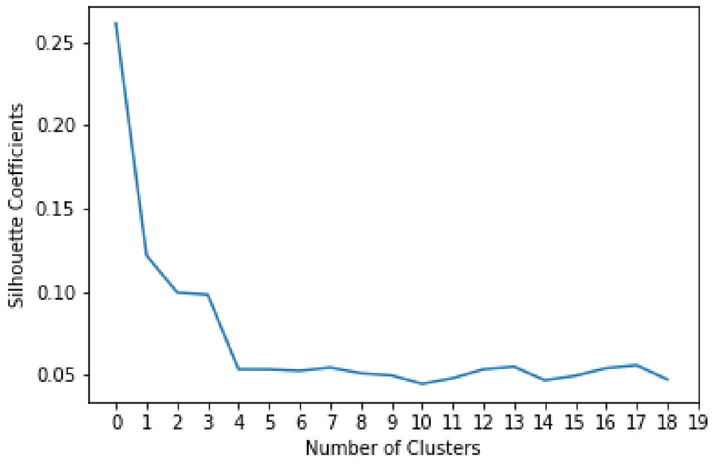
Visualization:

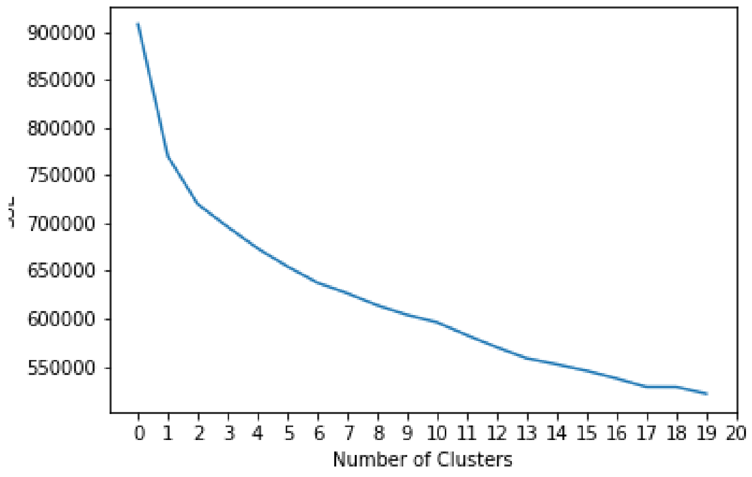
K-means++ initialization:





K-means++ initialization:





For other scatter plot graphs, please visit the repository under images/k_means/<number of clusters>

Chosen K:

I found 4 to be the optimal value for both metrics (Dunn Index & Silhouette Coefficients).

1) For SSE, even though the error remains high throughout, and the graph does not diminish as hyperbolically as desired to illustrate returns, there is still a slight elbow at 4 clusters for k-means++ initialization. Subsequent runs with Randomized initialization of centroids resulted in no elbow at 3, but a small elbow at 4 clusters as well. 2) The

silhouette coefficients show a must more significant diminish in returns for the number of clusters being created at 4 clusters for k-means++ initialization. Subsequent runs with randomized initialization of centroids resulted in a very sharp elbow at 3 clusters, rather than 4. Given this challenge in selecting an elbow given slightly conflicting metrics for slightly conflicting parameters, I opted to select a k value of 4. My reasoning in this final selection was that it is best to avoid losing significance and increasing the training time. Additionally, perhaps there exist some outliers in the dataset which have distance values that would otherwise drag centroids, but would perhaps gain value in their own clusters.

Chosen Distance Type:

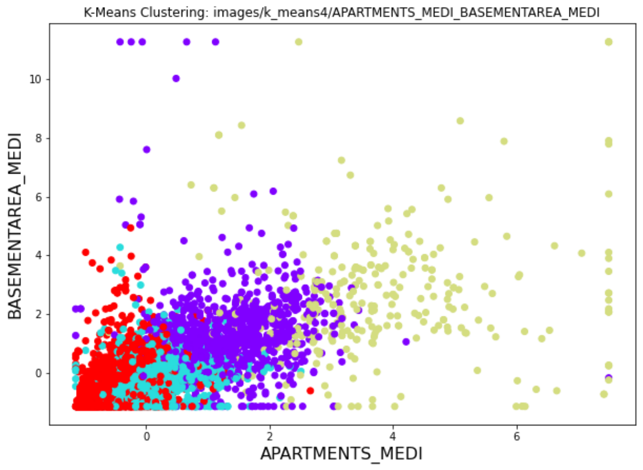
Background:

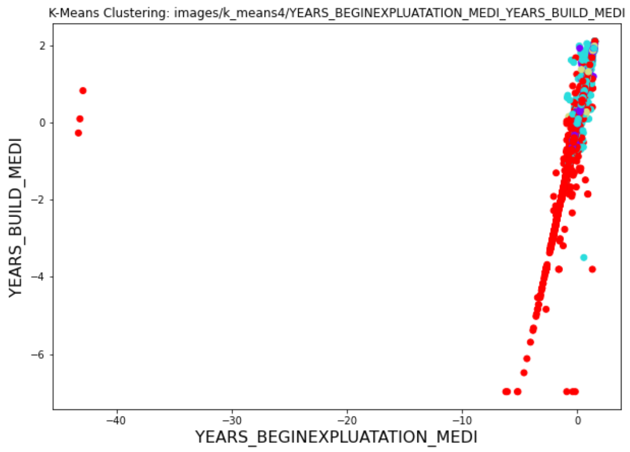
K-means algorithm most commonly supports: Euclidean distance measure, Manhattan distance measure, A squared euclidean distance measure, and Cosine distance measure. I started running the algorithm with Euclidean distance measure as it is the default and also most generally-intuitive measure.

Euclidean distance is fairly sufficient and uniform in constructing clusters that have been standardized.This data is looking exclusively at continuous variables, and not at correlation (1-r, like in genes), where the distance between two correlated vectors being 0 is of greater importance. Since there are still some slight outliers in the dataset, maximum distance would lead to possibly different sequences of the clustering that would be suboptimal. Even more so, manhattan distance would lead to less value in the hierarchical clustering algorithm given the high degree of dimensionality of the data set (or even simply taking our top 10 Principal Components).

What does this tell you about the data:

Such high MSE throughout cluster sizes suggests the sparsity of the data, particularly among the continuous variables, in regards to the importance of specific data features. Most data points can be grouped into 3 or 4 clusters and give a score suggesting that the data points are closer amongst themselves in their cluster than to any other data point - especially given the high dimensionality of the dataset.





Images such as the above illustrate how, even though there are 80-some continuous variables to evaluate, there is clear segregation in clustering, which is particularly easy to see in many of the time and home-related variables.