hw2

March 4, 2020

1 Assignment 2 - Clustering

1.1 Visualizing High Dimensional Data

1.1.1 Michael Young / u0742759

1.1.2 Problem 1 (50pts):

(10 pts): Implement PCA as a function and provide a commented version of it. Feel free to compute eigenvectors, eigenvalues etc. using numpy or other api functionality.

[305]:
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
%matplotlib inline

# Loading iris data for use with PCA
data = load_iris()
# data points
X = data.data
# labels
Y = data.target
# label names
labels = list(data.target_names)
```

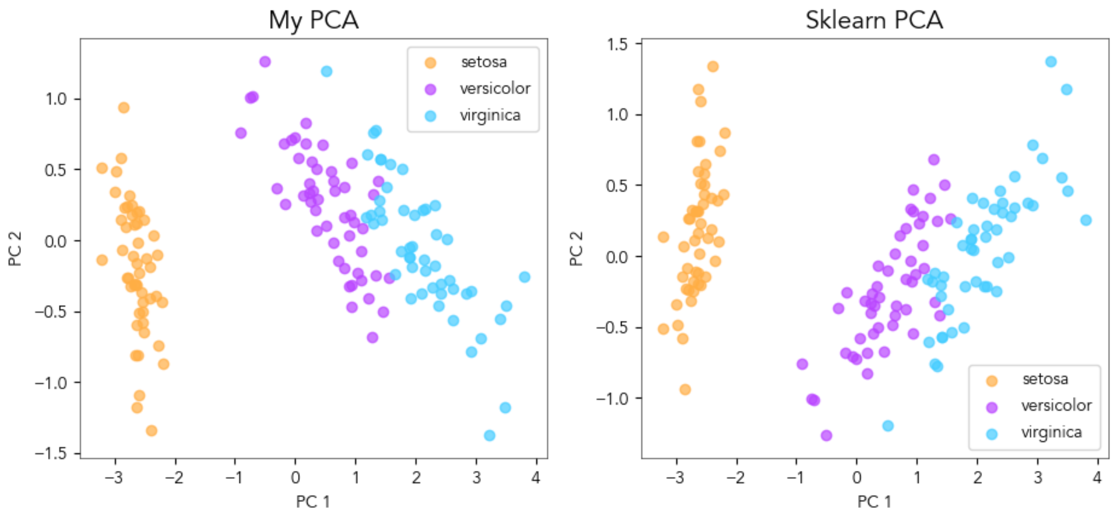
[306]:
```
# PCA function

def pca(data,n_components):
    '''Principal component analysis implementation
    input: array, number of components we want to project to
    output: coords of PCA output with corresponding nu
    '''
    ## Step 1: Zero-center data

    # init new standardized array
    data_std = np.empty((data.shape))
    for i in range(data.shape[1]):
        mean = np.mean(data[:,i])
        std = np.std(data[:,i])
        data_std[:,i] = (data[:,i] - mean)
    #print("centered data in my func:",data_std[1:5])

    ## Step 2: Covariance matrix computation
    cov = np.cov(data_std,rowvar=False)

    #print(cov)

    ## Step 3: Compute eigenvalues and eigenvectors of covariance matrix
    eigenvals, eigenvects = np.linalg.eig(cov)
    #print("vals",eigenvals,"vectors",eigenvects)

    # pair eigenvals and vectors
    eigs = list(zip(eigenvals,eigenvects.T))
    #print("my eigs",eigs)

    # sort eigs
    sorted_eigs = sorted(eigs)
    #print("sorted",sorted_eigs)

    ## Step 4: Feature vector
    feat_vect = np.empty((data.shape[1],n_components))
    for i in range(n_components):
        feat_vect[:,i] = sorted_eigs.pop()[1]
    print("My components:",feat_vect.T)

    ## Step 5: Recast the data
    return data_std.dot(feat_vect)
```

(10 pts): Plot (with a scatter plot) the iris dataset using your PCA implementation. Color each of the species differently. On a separate plot provide a scatter plot of the language api (sklearn) PCA for comparison.

[307]:
```
# Number of components I want to use for PCA
k = 2

# My PCA implementation
myPCA = pca(X,k)
# sklearn's PCA
model = PCA(n_components = k)
# Center data for skPCA
#X_std = StandardScaler().fit_transform(X)
data_cent = np.empty((X.shape))
for i in range(X.shape[1]):
    mean = np.mean(X[:,i])
    std = np.std(X[:,i])
    data_cent[:,i] = (X[:,i] - mean)
skPCA = model.fit_transform(data_cent)
print("SKlearn's components:",model.components_)
#print(X_std[1:5])
#print(skPCA)
#print(myPCA)


# Let's plot
plt.style.use('default')
plt.rcParams['font.family'] = 'Avenir'
plt.figure(figsize = (11,4.5))
colors = ['#ffae44','#ba44ff','#44cdff']

# My PCA
plt.subplot(1,2,1)
#plt.scatter(myPCA[:,0],myPCA[:,1],alpha=0.5)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(myPCA[Y==i,0], myPCA[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('My PCA',fontsize=15)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
#plt.axis('off')
# for spine in plt.gca().spines.values():
#     spine.set_visible(False)
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]


# SKLEARN PCA
plt.subplot(1,2,2)
#plt.scatter(skPCA[:,0],skPCA[:,1],alpha=0.5)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(skPCA[Y==i,0], skPCA[Y==i,1], alpha=0.7, c=c, label=lab)

plt.legend()
plt.title("Sklearn PCA",fontsize=15)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
#plt.axis('off')
# for spine in plt.gca().spines.values():
#     spine.set_visible(False)
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]
```

```
plt.show()
```

```
My components: [[ 0.3614 -0.0845  0.8567  0.3583]
 [-0.6566 -0.7302  0.1734  0.0755]]
SKlearn's components: [[ 0.3614 -0.0845  0.8567  0.3583]
 [ 0.6566  0.7302 -0.1734 -0.0755]]
```

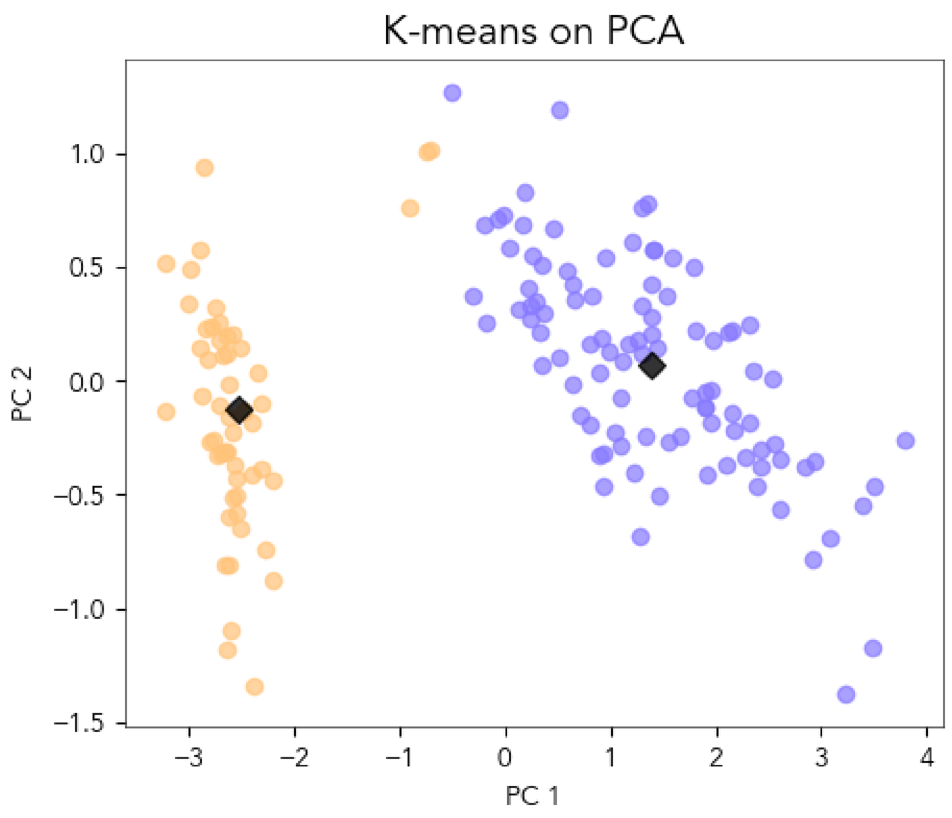


The data was centered before running PCA on it. As you can see, these outputs are the same, but the y values are flipped between them. This is due to the respective functions finding equivalent eigenvectors, but with signs flipped for the second component (see print out above). I can simply multiply the Y's by -1 to fix this, but I thought the difference was interesting.

(10 pts): Run K-Means on these results with k=2 and plot the results color according to cluster.

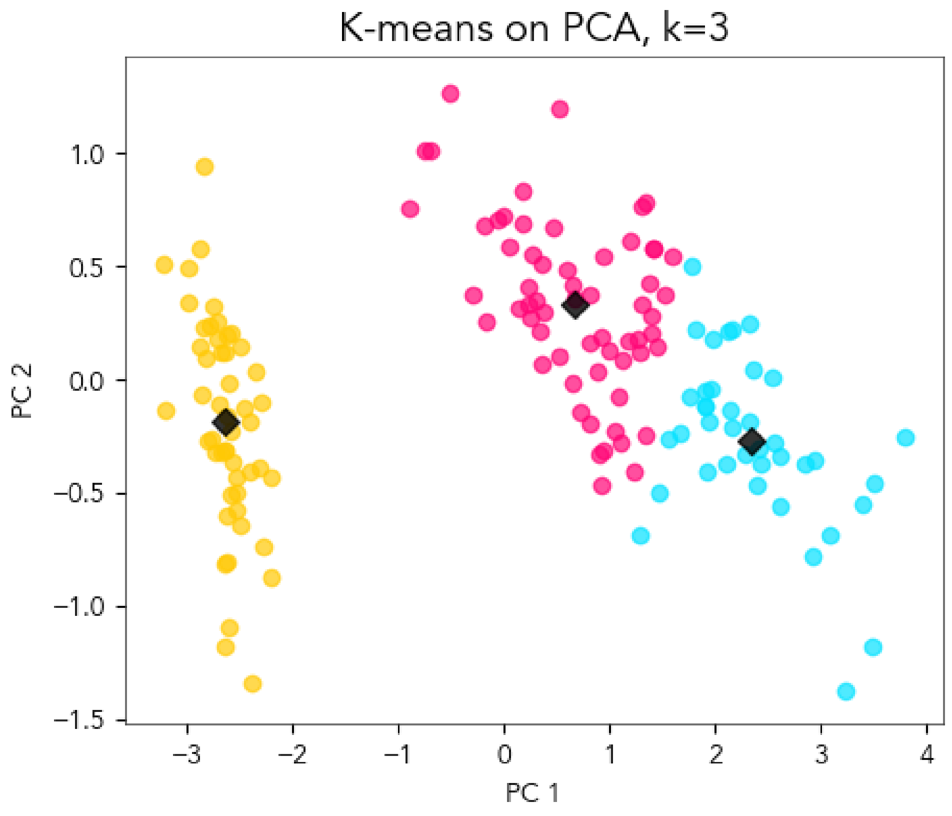
Because my PCA results are equivalent to the sklearn results, I'll just run K-means on one of them (mine).

[308]:
```
# K - Means
from sklearn.cluster import KMeans

km = KMeans(
    n_clusters=2, init='k-means++',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=42
)

y_km = km.fit_predict(myPCA)

plt.figure(figsize = (5.5,4.5))
plt.scatter(myPCA[y_km == 0, 0],myPCA[y_km == 0, 1],c="#8679ff",alpha=0.7)
plt.scatter(myPCA[y_km == 1, 0],myPCA[y_km == 1, 1],c="#ffc379",alpha=0.7)
# Plot the centers
plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1],c="k",alpha=0.
 ↪8,s=50,marker="D")
plt.title("K-means on PCA",fontsize=15)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



This k=2 k-means implementation clusters these about how we'd expect, with a bit of missclassification between setosa and the other two. It'd be more interesting to see how close a 3 clustering could get to the actual results. So, I'll do that below.

[309]:
```
# K-means, 3 clustering, for kicks

km = KMeans(
    n_clusters=3, init='k-means++',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=42
)

y_km = km.fit_predict(myPCA)

plt.figure(figsize = (5.5,4.5))
plt.scatter(myPCA[y_km == 0, 0],myPCA[y_km == 0, 1],c="#05e2ff",alpha=0.7)
plt.scatter(myPCA[y_km == 1, 0],myPCA[y_km == 1, 1],c="#ffc905",alpha=0.7)
plt.scatter(myPCA[y_km == 2, 0],myPCA[y_km == 2, 1],c="#ff0576",alpha=0.7)
# Plot the centers
plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1],c="k",alpha=0.
↪8,s=50,marker="D")
plt.title("K-means on PCA, k=3",fontsize=15)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



Well, turns out it doesn't do too well!

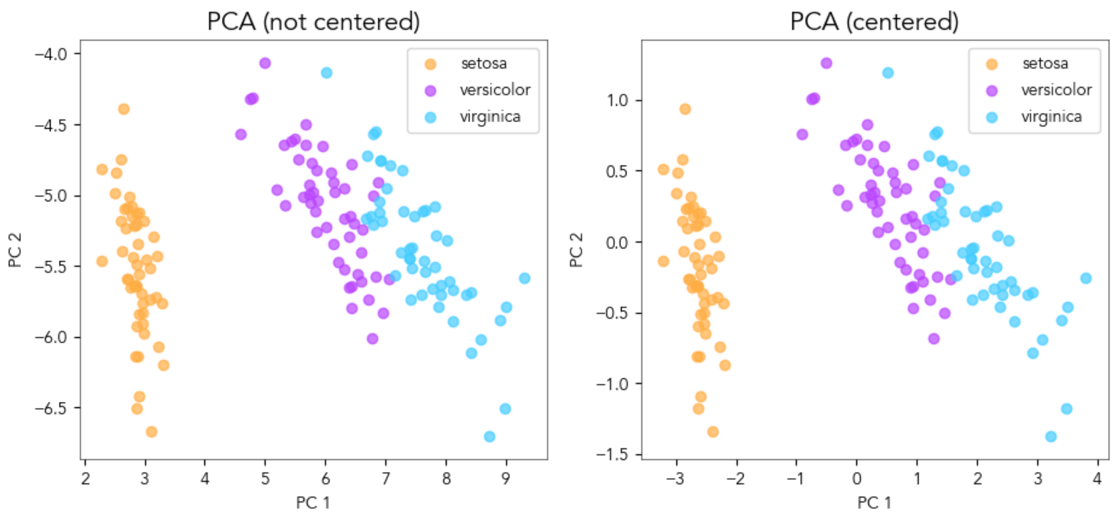
(10 pts): Now create an alternate PCA function where you do not center the data. Using a scatter plot, show the results. Again, color each of the species differently.

I'll just copy my old function and remove the centering logic from it.

[310]:
```
# PCA w/out standardization or centering



def pca_reloaded(data,n_components):
    '''Principal component analysis implementation
    input: array, number of components we want to project to
    output: coords of PCA output with corresponding nu
    '''

    ## Step 2: Covariance matrix computation
    cov = np.cov(data,rowvar=False)

    ## Step 3: Compute eigenvalues and eigenvectors of covariance matrix
    eigenvals, eigenvects = np.linalg.eig(cov)

    # pair eigenvals and vectors
    eigs = list(zip(eigenvals,eigenvects.T))

    # sort eigs
    sorted_eigs = sorted(eigs)

    ## Step 4: Feature vector
    feat_vect = np.empty((data.shape[1],n_components))
    for i in range(n_components):
        feat_vect[:,i] = sorted_eigs.pop()[1]

    ## Step 5: Recast the data
    return data.dot(feat_vect)
```

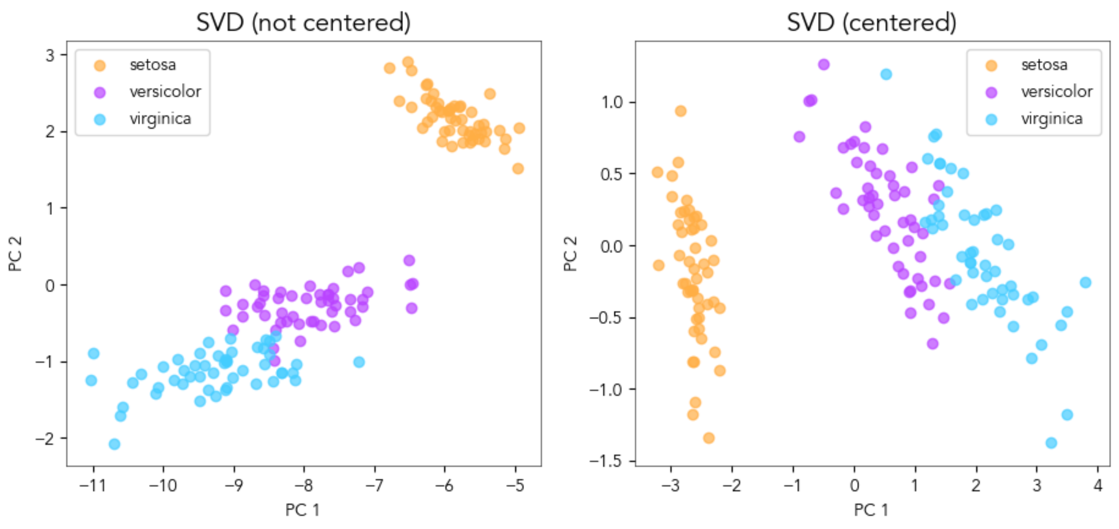
[312]:
```
# Now let's plot results and compare to original

pcaNC = pca_reloaded(X,2)
#pcaNC = model.fit_transform(X)

plt.figure(figsize = (11,4.5))
plt.subplot(1,2,1)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(pcaNC[Y==i,0], pcaNC[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('PCA (not centered)',fontsize=15)

plt.xlabel("PC 1")
plt.ylabel("PC 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]
```

```
# My PCA
plt.subplot(1,2,2)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(myPCA[Y==i,0], myPCA[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('PCA (centered)',fontsize=15)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



[336]:
```
# No differnce - I think this may be because the covariance matrix is␣
↪implicitly centering the data.
# Let's instead implement PCA using SVD and see if that makes any difference.

# This code (the svd function) is from: https://towardsdatascience.com/
↪pca-and-svd-explained-with-numpy-5d13b0d2a4d8
def svd(X):
  # Data matrix X, X doesn't need to be 0-centered
  n, m = X.shape
  # Compute full SVD
  U, Sigma, Vh = np.linalg.svd(X,
      full_matrices=False, # It's not necessary to compute the full matrix of U␣
↪or V
      compute_uv=True)
  # Transform X with SVD components
  X_svd = np.dot(U, np.diag(Sigma))
  return X_svd
```

```
X_svd = svd(X)[:,0:2]
X_svd_cent = svd(data_cent)[:,0:2]
# print(X_svd)
# print(X_svd_cent)

plt.figure(figsize = (11,4.5))
plt.subplot(1,2,1)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(X_svd[Y==i,0], X_svd[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('SVD (not centered)',fontsize=15)

plt.xlabel("PC 1")
plt.ylabel("PC 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

# My PCA
plt.subplot(1,2,2)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(X_svd_cent[Y==i,0], X_svd_cent[Y==i,1], alpha=0.7, c=c,␣
 ↪label=lab)
plt.legend()
plt.title('SVD (centered)',fontsize=15)
plt.xlabel("PC 1")
plt.ylabel("PC 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



(10 pts): What is the effect of neglecting to center the data? What type of data would not work well for PCA and why?

Ok, we've got a few different things going on here. As seen when comparing my implementation of PCA, which uses the eigenvalues of the covariance matrix, there's virtually no difference between using the centered vs. the uncentered data. There's a difference in that the actual coordinates between the centered and non-centered data are different, but the overall clustering looks identical.

However, when I implement PCA using SVD, which is a more computationally efficient way to do it, there is a clear difference between using the centered and uncentered data. It doesn't matter if the data is centered vs uncentered when using the covariance matrix, - the variance will be the same either way. Intrinsic to computing the covariance matrix is zero centering the data. When doing PCA the SVD route however, it matters that the data is centered because if it's not, the components may not correspond to the actual principal components.

Data that wouldn't work very well with PCA would be data that has several dimensions contributing equally to the variance. If we can't represent the principal components of our data with 3 or less dimensions, then this really isn't an effective technique for dimensionality reduction. Additionally, data that has dimensions that aren't intuitively related to eachother / in similar, comparable units.

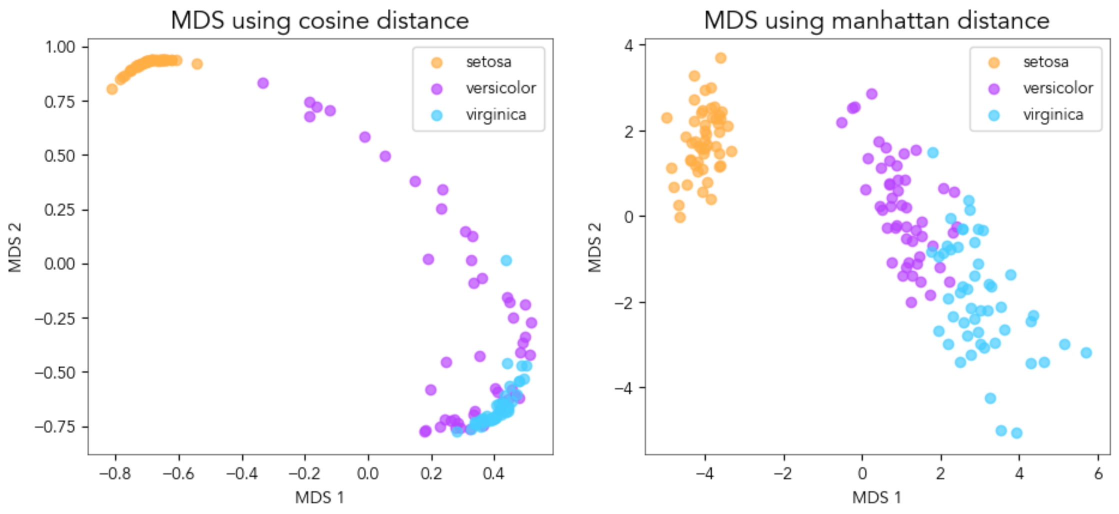
1.1.3 Problem 2 (20 pts):

(10 pts): Using MDS plot (scatter plot) the same iris data set using 2 different metrics for the dissimilarity matrix: (cosine, manhattan) coloring by label.

Because this doesn't explicitly say to implement ourselves, I'll use an API.

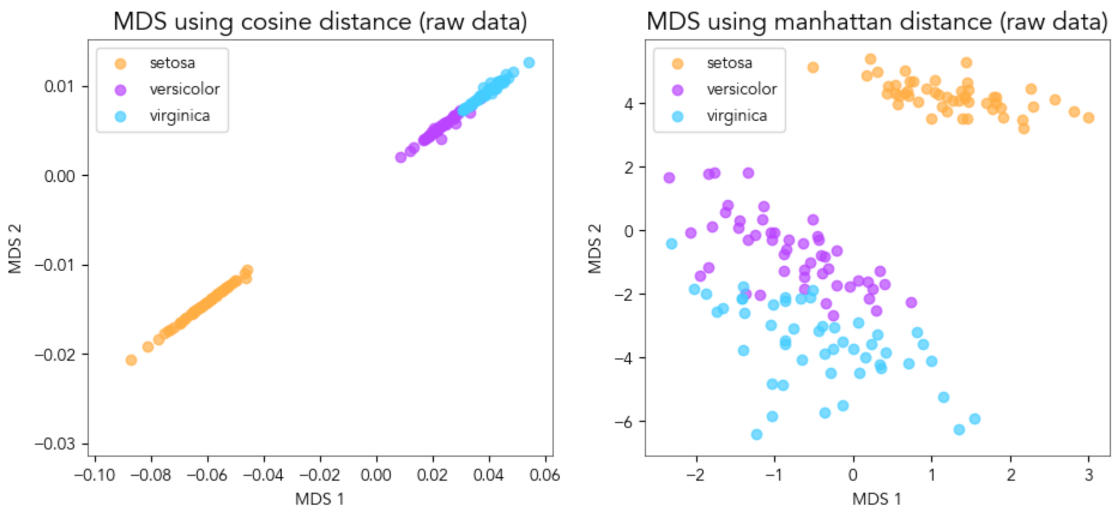
[324]:
```python
# MDS (multi-dimensional scaling)

from sklearn.manifold import MDS
from sklearn.metrics.pairwise import pairwise_distances

# Will use centered data as input
# data_cemt from above

# 2 different metrics for dissimilarity matrix:
# cosine
cos_dist = pairwise_distances(data_cent,metric='cosine')
# manhattan
man_dist = pairwise_distances(data_cent,metric='manhattan')

# MDS
mds = MDS(n_components=2, max_iter=3000, eps=1e-9, random_state=None,
                   dissimilarity="precomputed")
cos_mds = mds.fit(cos_dist).embedding_
man_mds = mds.fit(man_dist).embedding_

# Plotting
plt.figure(figsize = (11,4.5))
plt.subplot(1,2,1)
```

```
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(cos_mds[Y==i,0], cos_mds[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('MDS using cosine distance',fontsize=15)
plt.xlabel("MDS 1")
plt.ylabel("MDS 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

# My PCA
plt.subplot(1,2,2)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(man_mds[Y==i,0], man_mds[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('MDS using manhattan distance',fontsize=15)
plt.xlabel("MDS 1")
plt.ylabel("MDS 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



[322]:
```
# Just for kicks I'm going to run MDS on the raw data to compare with above

# centering the data
# X_cent = np.empty((X.shape))
# for i in range(X.shape[1]):
#     mean = np.mean(X[:,i])
#     X_cent[:,i] = (X[:,i] - mean)

# cosine
```

```
cos_dist = pairwise_distances(X,metric='cosine')
# manhattan
man_dist = pairwise_distances(X,metric='manhattan')

# MDS
mds = MDS(n_components=2, max_iter=3000, eps=1e-9, random_state=None,
                   dissimilarity="precomputed")
cos_mds = mds.fit(cos_dist).embedding_
man_mds = mds.fit(man_dist).embedding_

# Plotting
plt.figure(figsize = (11,4.5))
plt.subplot(1,2,1)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(cos_mds[Y==i,0], cos_mds[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('MDS using cosine distance (raw data)',fontsize=15)
plt.xlabel("MDS 1")
plt.ylabel("MDS 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

# My PCA
plt.subplot(1,2,2)
for c, i, lab in zip(colors, [0, 1, 2], labels):
   plt.scatter(man_mds[Y==i,0], man_mds[Y==i,1], alpha=0.7, c=c, label=lab)
plt.legend()
plt.title('MDS using manhattan distance (raw data)',fontsize=15)
plt.xlabel("MDS 1")
plt.ylabel("MDS 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```

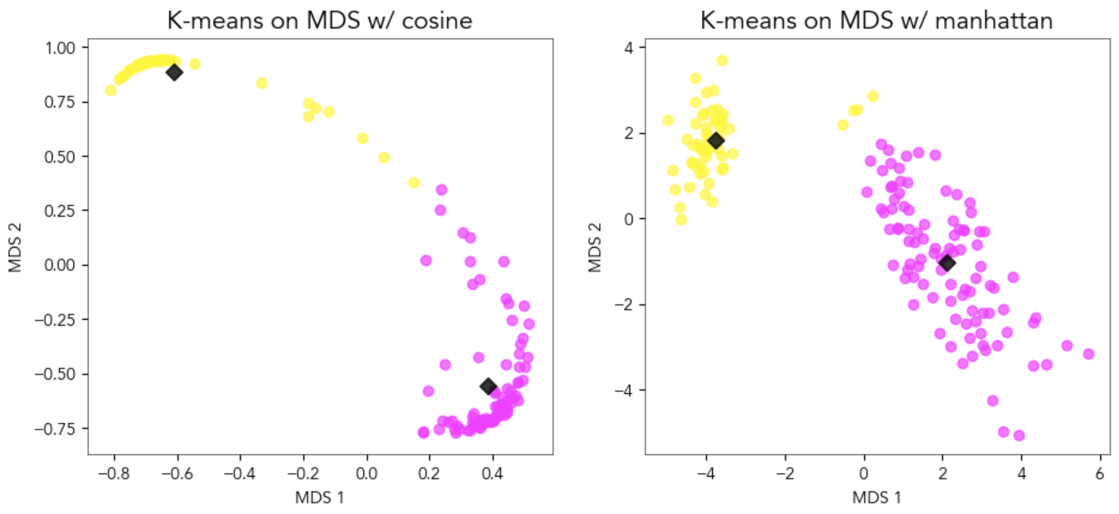


This is really interesting! When I feed in the centered data, MDS + cosine distance outputs this really fascinating circular pattern. When I instead plug in the raw data to MDS, it ouputs this linear thing seen above. This is possibly due to the cosine distance being a measure of angle, and with the uncentered data, They all have similar angles? In terms of which one does better at separating the data, it looks to me like the raw data input is a bit better.The manhattan, by contrast, looks very similar between the centeredd data and the raw data.

(10 pts): Run K-Means with k=2 on the output above and plot the results and color according to cluster.

[325]:
```
# Kmeans on MDS oututs with standardized data input

km1 = KMeans(
    n_clusters=2, init='k-means++',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=42
)
km2 = KMeans(
    n_clusters=2, init='k-means++',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=42
)

y_km_cos = km1.fit_predict(cos_mds)
y_km_man = km2.fit_predict(man_mds)

# Cosine
plt.figure(figsize = (11,4.5))
plt.subplot(1,2,1)
plt.scatter(cos_mds[y_km_cos == 0, 0],cos_mds[y_km_cos == 0,␣
 ↪1],c="#ec41ff",alpha=0.7)
plt.scatter(cos_mds[y_km_cos == 1, 0],cos_mds[y_km_cos == 1,␣
 ↪1],c="#fff641",alpha=0.7)
# Plot the centers
plt.scatter(km1.cluster_centers_[:,0],km1.cluster_centers_[:,1],c="k",alpha=0.
 ↪8,s=50,marker="D")
plt.title("K-means on MDS w/ cosine",fontsize=15)
plt.xlabel("MDS 1")
plt.ylabel("MDS 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

# Manhattan
plt.subplot(1,2,2)
plt.scatter(man_mds[y_km_man == 0, 0],man_mds[y_km_man == 0,␣
 ↪1],c="#ec41ff",alpha=0.7)
```

```
plt.scatter(man_mds[y_km_man == 1, 0],man_mds[y_km_man == 1,␣
↪1],c="#fff641",alpha=0.7)
# Plot the centers
plt.scatter(km2.cluster_centers_[:,0],km2.cluster_centers_[:,1],c="k",alpha=0.
↪8,s=50,marker="D")
plt.title("K-means on MDS w/ manhattan",fontsize=15)
plt.xlabel("MDS 1")
plt.ylabel("MDS 2")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



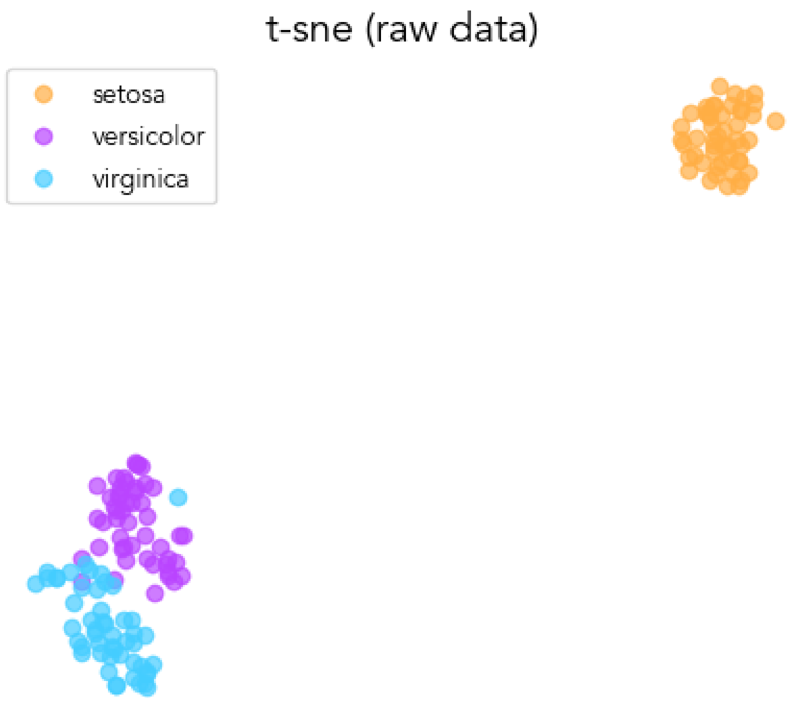
Once again, k-means clusters about how we'd expect. I'm curious why we're not being asked to do 3-clustering, considering there are actually 3 distant classes here. If I had to compare the "performance" of MDS here with the results from PCA, on this particular dataset PCA seems to perform better.

1.1.4 Problem 3

(10 pts): Using T-SNE plot (scatter plot) the same iris data set coloring by label

[364]:
```
from sklearn.manifold import TSNE

# will plot raw data for now
# Loading iris data for use with PCA
data = load_iris()
# data points
X = data.data
# labels
Y = data.target
```

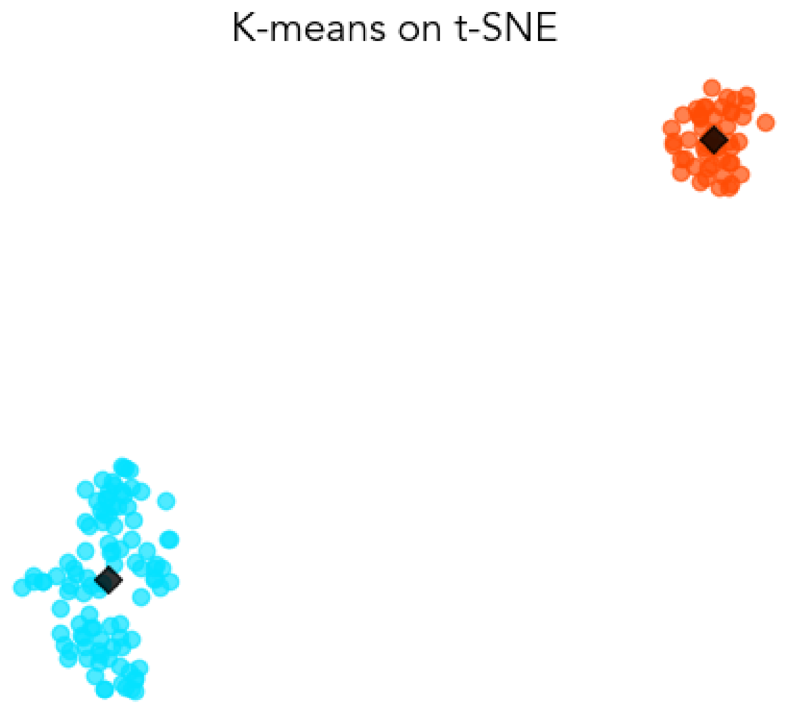
```
# label names
labels = list(data.target_names)

tsne_coords = TSNE(n_components=2,random_state=0).fit_transform(X)


plt.figure(figsize = (5.5,4.5))
for c, i, lab in zip(colors, [0, 1, 2], labels):
    plt.scatter(tsne_coords[Y==i,0], tsne_coords[Y==i,1], alpha=0.7, c=c,␣
 ↪label=lab)
plt.legend()
plt.title('t-sne (raw data)',fontsize=15)
plt.axis("off")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



I've removed the axes here becuase in t-SNE, they have no meaning

(10 pts): Run K-Means with k=2 on the output above, plot the results and color according to cluster.

[365]:
```
km = KMeans(
    n_clusters=2, init='k-means++',
    n_init=10, max_iter=300,
    tol=1e-04, random_state=42
)

y_km = km.fit_predict(tsne_coords)

plt.figure(figsize = (5.5,4.5))
plt.scatter(tsne_coords[y_km == 0, 0],tsne_coords[y_km == 0,␣
 ↪1],c="#05e2ff",alpha=0.7)
plt.scatter(tsne_coords[y_km == 1, 0],tsne_coords[y_km == 1,␣
 ↪1],c="#ff4c05",alpha=0.7)
# Plot the centers
plt.scatter(km.cluster_centers_[:,0],km.cluster_centers_[:,1],c="k",alpha=0.
 ↪8,s=50,marker="D")
plt.title("K-means on t-SNE",fontsize=15)
plt.axis("off")
[i.set_linewidth(0.4) for i in plt.gca().spines.values()]

plt.show()
```



(10 pts): Compare and contrast the results from the plots generated from all 3 problems. Also, discuss any patterns or clusters that resulted from running the K-Means algorithm on all 3 problems..

All three methods did a reasonable job at reducing this “high dimensional” data down to 2 dimensions. They all agree in that they can easily separate setosa from the other two, but they vary in the degree to which they effectively separate the other two. This is likely because versicolor and virginica are much more closely related to eachother than they are to setosa. Aesthetically, I like the clustering results from tSNE the best. The clusters are tighter and denser and I think for clustering that’s an appealing characteristic. As for distances used with MDS go, it’s clear that the cosine distance doesn’t make too much sense in this application, while the manhattan distance does reasonably well. I like the simplicity of the PCA algorithm, and think it’s generally a good place to start, but there’s something very cool about things like t-SNE and UMAP that will always keep me coming back for more.

[ ]:
```

```