New Practical exercises

1. (CS 189 Introduction to Machine Learning Spring 2018 HW5, ex.2.e-g)

Figure 1: Tennis ball pasted on top of image of St. Peter's Basillica without lighting adjustment (left) and with lighting adjustment (right)

(e) In this problem, we will use total least squares to approximately learn the lighting in a photograph, which we can then use to paste new objects into the image while still maintaining the realism of the image. You will be estimating the lighting coefficients for the interior of St. Peter's Basillica, and you will then use these coefficients to change the lighting of an image of a tennis ball so that it can be pasted into the image. In Figure 1, we show the result of pasting the tennis ball in the image without adjusting the lighting on the ball. The ball looks too bright for the scene and does not look like it would fit in with other objects in the image.

To convincingly add a tennis ball to an image, we need to need to apply the appropriate lighting from the environment onto the added ball. To start, we will represent environment lighting as a spherical function $\mathbf{f}(\mathbf{n})$ where $\mathbf{n}$ is a 3 dimensional unit vector ($||\mathbf{n}||_2 = 1$), and $\mathbf{f}$ outputs a 3 dimensional color vector, one component for red, green, and blue light intensities. Because $\mathbf{f}(\mathbf{n})$ is a spherical function, the input $\mathbf{n}$ must correspond to a point on a sphere. The function $\mathbf{f}(\mathbf{n})$ represents the total incoming light from the direction $\mathbf{n}$ in the scene. The lighting function of a spherical object $\mathbf{f}(\mathbf{n})$ can be approximated by the first 9 spherical harmonic basis functions.

Figure 2: Image of a spherical mirror inside of St. Peter's Basillica

The first 9 unnormalized sperical harmonic basis functions are given by:

$$
\begin{aligned}
L_1 &= 1 \\
L_2 &= y \\
L_3 &= x \\
L_4 &= z \\
L_5 &= xy \\
L_6 &= yz \\
L_7 &= 3z^2 - 1 \\
L_8 &= xz \\
L_9 &= x^2 - y^2
\end{aligned}
$$

where $\mathbf{n} = [x, y, z]^\top$. The lighting function can then be approximated as

$$\mathbf{f}(\mathbf{n}) \approx \sum_{i=1}^{9} \boldsymbol{\gamma}_i L_i(\mathbf{n})$$

$$
\begin{bmatrix} — & \mathbf{f}(\mathbf{n_1}) & — \\ — & \mathbf{f}(\mathbf{n_2}) & — \\ & \vdots & \\ — & \mathbf{f}(\mathbf{n_n}) & — \end{bmatrix}_{n\times 3}
=
\begin{bmatrix} L_1(\mathbf{n_1}) & L_2(\mathbf{n_1}) & \dots & L_9(\mathbf{n_1}) \\ L_1(\mathbf{n_2}) & L_2(\mathbf{n_2}) & \dots & L_9(\mathbf{n_2}) \\ & \vdots & & \\ L_1(\mathbf{n_n}) & L_2(\mathbf{n_n}) & \dots & L_9(\mathbf{n_n}) \end{bmatrix}_{n\times 9}
\begin{bmatrix} — & \boldsymbol{\gamma}_1 & — \\ — & \boldsymbol{\gamma}_2 & — \\ & \vdots & \\ — & \boldsymbol{\gamma}_9 & — \end{bmatrix}_{9\times 3}
$$

where $L_i(\mathbf{n})$ is the ith basis function from the list above.

The function of incoming light $\mathbf{f}(\mathbf{n})$ can be measured by photographing a spherical mirror placed in the scene of interest. In this case, we provide you with an image of the sphere as seen

in Figure 2. In the code provided, there is a function extractNormals(img) that will extract the training pairs $(\mathbf{n}_i, \mathrm{f}(\mathbf{n}_i))$ from the image. An example using this function is in the code.

Use the spherical harmonic basis functions to create a 9 dimensional feature vector for each sample. Use this to formulate an ordinary least squares problem and solve for the unknown coefficients γ_i. Report the estimated values for γ_i and include a visualization of the approximation using the provided code. The code provided will load the images, extracts the training data, relights the tennis ball with incorrect coefficients, and saves the results. Your task is to compute the basis functions and solve the least squares problems to provide the code with the correct coefficients. To run the starter code, you will need to use Python with `numpy` and `scipy`. Because the resulting data set is large, we reduce it in the code by taking every 50^{th} entry in the data. This is done for you in the starter code, but you can try using the entire data set or reduce it by a different amount.

Solution:

```
def leastSquaresSolve(B, vs):

  # B is n x 9
  # vs is n x 3
  XX = B.T.dot(B)
  Xy = B.T.dot(vs)
  coeff = np.linalg.solve(XX,Xy)
  res = np.linalg.lstsq(B,vs)
  return coeff

if __name__ == '__main__':

  data,tennis,target = loadImages()
  ns, vs = extractNormals(data)
  B = computeBasis(ns)

  # reduce the number of samples because computing the SVD on
  # the entire data set takes too long
  Bp = B[::50]
  vsp = vs[::50]

  coeffLSQ = leastSquaresSolve(Bp,vsp)
  imgLSQ = relightSphere(tennis,coeffLSQ)
  targetLSQ = compositeImages(imgLSQ,target)

  print('Least squares:\n'+str(coeffLSQ))

  plt.figure()
  plt.imshow(targetLSQ)
  plt.title('Least Squares')
  plt.show()
```

```
    [ 202.31845431  162.41956802  149.07075034]
    [ -27.66555164  -17.88905339  -12.92356688]
    [  -1.08629293    0.42947012    1.15475569]
    [  -5.15203925   -4.51375871   -4.24262639]
    [  -3.14053107   -3.70269907   -3.74382934]
    [  23.67671768   23.15698002   21.94638397]
    [  -3.82167171    0.57606634    1.81637483]
    [   4.7346737     1.4677692    -1.12253649]
    [  -9.72739616   -5.75691108   -4.8395598 ]
```

f) When we extract from the data the direction $\mathbf{n}$ to compute $(\mathbf{n}_i, \mathbf{f}(\mathbf{n}_i))$, we make some approximations about how the light is captured on the image. We also assume that the spherical mirror is a perfect sphere, but in reality, there will always be small imperfections. Thus, our measurement for $\mathbf{n}$ contains some error, which makes this an ideal problem to apply total least squares. **Solve this problem with total least squares by allowing perturbations in the matrix of basis functions. Report the estimated values for γ_i and include a visualization of the approximation.** The output image will be visibly wrong, and we'll explore how to fix this problem in the next part. Your implementation may only use the SVD and the matrix inverse functions from the linear algebra library in numpy as well as np.linalg.solve.

Solution: Theoretical derivation for the matrix case: As opposed to the theoretical TLS part, here we now have target vectors (per sample) instead of scalars corresponding to the rows of the matrix $\mathbf{f}(\mathbf{n}) \in \mathbb{R}^{n\times 3}$ and the original weight vector $\mathbf{w}$ is now replaced by the matrix $\gamma \in \mathbb{R}^{9\times 3}$. The feature matrix is as always $\mathbf{X} \in \mathbb{R}^{n\times 9}$.

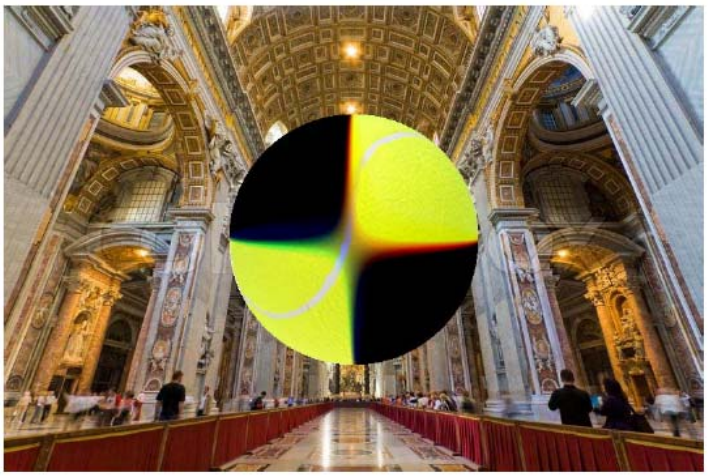
In this case again, we want $[\mathbf{X}+\epsilon_X, \mathbf{Y}+\epsilon_Y]$ (with ϵ_Y now a matrix) to be the best rank $d = 9$ approximation to the matrix $[\mathbf{X}, \mathbf{Y}] \in \mathbb{R}^{n\times 12}$, which by the Eckart-Young-Mirsky Theorem is found by setting the three smallest singular values of $[\mathbf{X}, \mathbf{Y}]$ to zero. Now by the same argumentation as in the scalar case, the matrix $\begin{bmatrix}\gamma \\ -\mathbf{I}_3\end{bmatrix}$ must lie in the nullspace of $[\mathbf{X}+\epsilon_X, \mathbf{Y}+\epsilon_Y]$. Since it is three dimensional, there is no unique answer. However, we know that, denoting the last three columns of singular matrix $\mathbf{V}$ by $\begin{bmatrix}\mathbf{V}_{xy} \\ \mathbf{V}_{yy}\end{bmatrix}$, the following holds when $\mathbf{V}_{yy}$ is invertible (which it is in this dataset)

$$\begin{aligned} 0 &= (\mathbf{X}+\epsilon_{\mathbf{X}})\mathbf{V}_{xy} + (\mathbf{Y}+\epsilon_Y)\mathbf{V}_{yy} \\ &= (\mathbf{X}+\epsilon_X)\mathbf{V}_{xy}\mathbf{V}_{yy}^{-1} + (\mathbf{Y}+\epsilon_Y)\mathbf{I}_3. \end{aligned}$$

Therefore, $\begin{bmatrix}-\mathbf{V}_{xy}\mathbf{V}_{yy}^{-1} \\ -\mathbf{I}_3\end{bmatrix}$ is in the nullspace of $[\mathbf{X}+\epsilon_X, \mathbf{Y}+\epsilon_Y]$ so that by matching matrices, one candidate solution for γ is $\gamma = -\mathbf{V}_{xy}\mathbf{V}_{yy}^{-1}$.

```
def totalLeastSquaresSolve(B, vs, scale=1):

  # Build the combined matrices
  XY = np.concatenate([B,vs/scale],-1)

  # Solve each system separately
  coeff = []
  #for xy in XY:
  u,s,v = np.linalg.svd(XY)
  v = v.T
  Vxy = v[:B.shape[1],B.shape[1]:] # 9 x 3
  Vyy = v[B.shape[1]:,B.shape[1]:] # 3 x 3
  #Vyy = v[-1,-1]
  #c = -Vxy*(1/Vyy) # 9 x 3
  coeff = -Vxy.dot(np.linalg.inv(Vyy))
  #coeff.append(c)
  #coeff = np.asarray(coeff).T
  print(str(coeff))
  return coeff*scale


if __name__ == '__main__':

  data,tennis,target = loadImages()
  ns, vs = extractNormals(data)
  B = computeBasis(ns)

  # reduce the number of samples because computing the SVD on
  # the entire data set takes too long
  Bp = B[::50]
  vsp = vs[::50]

  coeffTLS = totalLeastSquaresSolve(Bp,vsp)
  imgTLS = relightSphere(tennis,coeffTLS)
  targetTLS = compositeImages(imgTLS,target)

  print('Total least squares:\n'+str(coeffTLS))
  plt.figure()
  plt.imshow(targetTLSScaled)
  plt.title('Total Least Squares Scaled')
  plt.show()

  imsave('tls.png',targetTLS)
```

```
    [  2.13318421e+02   1.70780299e+02   1.57126297e+02]
    [ -3.23046362e+01  -2.02975310e+01  -1.45516114e+01]
    [ -4.89811386e+00  -3.37684058e+00  -1.14207091e+00]
    [ -4.31689131e+00  -3.80778081e+00  -4.83616306e+00]
    [ -7.05901066e+03  -7.39934207e+03  -4.26448732e+03]
    [ -3.05378224e+02  -1.56329401e+02   3.50285345e+02]
    [ -9.76079364e+00  -5.33182216e+00  -1.55699782e+00]
    [  7.30792588e+02   3.52130316e+02  -6.11683200e+02]
    [ -9.08887079e+00  -3.84309477e+00  -4.16456437e+00]
```



(g) In the previous part, you should have noticed that the visualization is drastically different than the one generated using least squares. Recall that in total least squares we are minimizing $||[\epsilon_{\mathbf{X}}, \epsilon_{\mathbf{y}}]||_F^2$. Intuitively, to minimize the Frobenius norm of components of both the inputs and outputs, the inputs and outputs should be on the same scale. However, this is not the case here. Color values in an image will typically be in $[0, 255]$, but the original image had a much larger range. We compressed the range to a smaller scale using tone mapping, but the effect of the compression is that relatively bright areas of the image become less bright. As a compromise, we scaled the image colors down to a maximum color value of 384 instead of 255. Thus, the inputs here are all unit vectors, and the outputs are 3 dimensional vectors where each value is in $[0, 384]$. **Propose a value by which to scale the outputs $\mathbf{f}(\mathbf{n}_i)$ such that the values of the inputs and outputs are roughly on the same scale. Solve this scaled total least squares problem, report the computed spherical harmonic coefficients and provide a rendering of the relit sphere.**

Solution:

Recall: total least squares assumes that the noise is the same in all directions. When we just have some data, it can be hard to think about what the noise model should be. The default is to take a "significant figures" mentality and assume that everything should be on the same scale, assuming that noise is roughly proportional to the size of the inputs. Because most of the basis functions lie within $[-1, 1]$, we want to scale the image pixel values so that they lie in a similar range. For these results, we can scale the values by $1/384$, but any reasonable value that scales the pixel values to a similar range is acceptable.

```
if __name__ == '__main__':

  data,tennis,target = loadImages()
  ns, vs = extractNormals(data)
  B = computeBasis(ns)

  # reduce the number of samples because computing the SVD on
  # the entire data set takes too long
  Bp = B[::50]
  vsp = vs[::50]

  coeffTLSScaled = totalLeastSquaresSolve(Bp,vsp,scale=255*1.5)
```

```
imgTLSScaled = relightSphere(tennis,coeffTLSScaled)
targetTLSScaled = compositeImages(imgTLSScaled,target)

print('Total least squares scaled:\n'+str(coeffTLSScaled))

plt.figure()
plt.imshow(targetTLSScaled)
plt.title('Total Least Squares Scaled')
plt.show()

imsave('tls_scaled.png',targetTLSScaled)
```

```
   [ 209.41539449  169.06782937  155.39642541]
   [ -30.28100667  -20.3163958   -15.21596685]
   [  -1.05621451    0.46391495    1.19212042]
   [  -5.7563859    -5.08161145   -4.78411908]
   [  -7.95607504   -8.25078526   -8.0969764 ]
   [  55.29299419   52.93568994   50.39069818]
   [  -3.84934062    0.5565465     1.80231959]
   [   7.35375998    3.85567243    1.0984583 ]
   [ -10.91282516   -6.85792251   -5.88064457]

```

2. (CS 189 Introduction to Machine Learning Spring 2018 HW5, ex.3.h-j)

(h) In the next few parts, we visualize the effect of PCA projections and random projections on a classification task. You are given 3 datasets in the data folder and a starter code.

Use the starter code to load the three datasets one by one. Note that there are two unique values in y. **Visualize the features of X these datasets using (1) Top-2 PCA components, and (2) 2-dimensional random projections. Use the code to project the features to 2 dimensions and then scatter plot the 2 features with a different color for each class.** Note that you will obtain 2 plots for each dataset (total 6 plots for this part). **Do you observe a difference in PCA vs random projections? Do you see a trend in the three datasets?**

Solution: The code for this part and the following parts is as follows:

```
import numpy as np
import matplotlib.pyplot as plt
# %matplotlib inline
import sklearn.linear_model
from sklearn.model_selection import train_test_split

######## PROJECTION FUNCTIONS ##########

## Random Projections ##
def random_matrix(d, k):
    '''
    d = original dimension
    k = projected dimension
    '''
    return 1./np.sqrt(k)*np.random.normal(0, 1, (d, k))

def random_proj(X, k):
    _, d= X.shape
    return X.dot(random_matrix(d, k))

## PCA and projections ##
def my_pca(X, k):
    '''
    compute PCA components
    X = data matrix (each row as a sample)
    k = #principal components
    '''
    n, d = X.shape
    assert(d>=k)
    _, _, Vh = np.linalg.svd(X)
    V = Vh.T
    return V[:, :k]

def pca_proj(X, k):
  '''
  compute projection of matrix X
  along its first k principal components
  '''
    P = my_pca(X, k)
    # P = P.dot(P.T)
    return X.dot(P)


######### LINEAR MODEL FITTING ############

def rand_proj_accuracy_split(X, y, k):
  '''
  Fitting a k dimensional feature set obtained
  from random projection of X, versus y
  for binary classification for y in {-1, 1}
  '''
```

```
    # test train split
    _, d = X.shape
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state
    ↪ =42)

    # random projection
    J = np.random.normal(0., 1., (d, k))
    rand_proj_X = X_train.dot(J)

    # fit a linear model
    line = sklearn.linear_model.LinearRegression(fit_intercept=False)
    line.fit(rand_proj_X, y_train)

    # predict y
    y_pred=line.predict(X_test.dot(J))

    # return the test error
    return 1-np.mean(np.sign(y_pred)!= y_test)

def pca_proj_accuracy(X, y, k):
    '''
  Fitting a k dimensional feature set obtained
  from PCA projection of X, versus y
  for binary classification for y in {-1, 1}
  '''

    # test-train split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20)

    # pca projection
    P = my_pca(X_train, k)
    P = P.dot(P.T)
    pca_proj_X = X_train.dot(P)

    # fit a linear model
    line = sklearn.linear_model.LinearRegression(fit_intercept=False)
    line.fit(pca_proj_X, y_train)

     # predict y
    y_pred=line.predict(X_test.dot(P))
```

```
    # return the test error
    return 1-np.mean(np.sign(y_pred)!= y_test)


np.random.seed(7283782)

for i in range(3):

    ######## LOADING THE DATASETS #########

    data = np.load('jl_data/data'+str(i+1)+'.npz')
    X = data['X']
    y = data['y']

    n, d = X.shape
    n_trials = 10  # to average the accuracies for random projection

    ######## CLASSIFICATION USING PROJECTED FEATURES #########

    # linear regression with projected features
    random_accuracy = np.zeros((n_trials, d))
    pca_accuracy = np.zeros(d)

    for k in range(1, d+1):
        pca_accuracy[k-1] = pca_proj_accuracy_split(X, y, k)
        for j in range(n_trials):
            random_accuracy[j, k-1] = rand_proj_accuracy_split(X, y, k)

    # take the mean of random projection performance across trials
    random_accuracy = np.mean(random_accuracy, axis=0)
```

```
125
126     ######## PLOTTING RESULTS #########
127
128     plt.figure(figsize=[15, 3])
129 #     plt.suptitle('Data'+str(i), fontsize=15)
130     plt.subplot(141)
131     p = pca_proj(X, 2)
132     plt.scatter(p[:, 0], p[:, 1], c=y)
133     plt.title('First 2 PC')
134
135     plt.subplot(142)
136     r = random_proj(X, 2)
137     plt.scatter(r[:, 0], r[:, 1], c=y)
138     plt.title('Random 2D projection')
139
140     _, S, Vh = np.linalg.svd(X)
141     plt.subplot(144)
142     plt.plot(np.arange(1, d+1), S)
143     plt.title('Singular value decay')
144
145     plt.subplot(143)
146     plt.plot(np.arange(1, d+1), random_accuracy, label="Random")
147     plt.plot(np.arange(1, d+1), pca_accuracy, label="PCA")
148     plt.legend(loc='best')
149     plt.title("Accuracy")
150     plt.xlabel("\#Dimensions")
151
152     plt.savefig('jl'+str(i+1)+'.pdf')
153    # plt.show()
```

Remark: You may notice a slight difference in the projection matrices used for visualization and for generating the features for regression. For PCA visualization we use only the projection coordinates along the principal components. But for performing regression, we need to use the corresponding directions and hence the two projection matrices are slightly different. Put simply, for visualization the code outputs a k-dimensional vector (coordinates in the principal component basis) but for generating regression features it uses the d-dimensional representation of these k-dimensional vectors. For random projections used in the regression code, there is a slight typo – the scaling is missing. However, the learned model does not change because scaling the entire feature matrix does not change the ordinary least squares solution.

For plots, refer to figures 3, 4 and 5.

For the first and second datasets it is hard to see a difference between the 2D projections with PCA or random projection.

For the third dataset, it is clear that PCA projection clearly separates the two clusters of points, while random projection still has significant overlap of the two sets of points.

Using the PCA projections, we can say that the separation between the two sets of features (corresponding to $y = 1$ and $y = -1$) is largest in data3.npz and probably smallest in data1.npz

(i) For each dataset, we will now fit a linear model on different projections of features to perform classification. The code for fitting a linear model with projected features and predicting a label for a given feature, is given to you. Use these functions and write a code that does prediction in the following way: (1) Use top k-PCA features to obtain one set of results, and (2) Use k-dimensional random projection to obtain the second set of results (take average accuracy over 10 random projections for smooth curves). Use the projection functions given in the starter code to select these features. You should vary k from 1 to d where d is the dimension of each feature $\mathbf{x}_i$. **Plot the accuracy for PCA and Random projection as a function of k. Comment on the observations on these accuracies as a function of k and across different datasets.**

Solution: For plots, refer to figures 3, 4 and 5.

PCA: We see that the accuracy of PCA increase very quickly after adding 2-3 features only and then does not increase too much. In fact, for data1.npz it decays little bit after first 2 features indicating overfitting.

Random projection: We see that the accuracy increases moderately with increase in number of dimension of the projection matrix.

The difference in increase in accuracy for PCA and random projection is intuitive as PCA captures effective dimensions – and it appears from the 2D visualization that signal strength is stronger in a few directions. In particular for data3, the data is separable if we just use one principal component. However, random projection is oblivious to the structure in the underlying data if we just use very few directions. Although as the number of dimensions increase, its performance does catch up with PCA.

Across the three datasets, we see that the accuracy for a given set of dimensions increases as we move from data1.npz to data2.npz to data3.npz, again suggesting that the signal strength = the separation between the two clusters is increasing from data1 to data3.

(j) Now plot the singular values for the feature matrices of the three datasets. **Do you observe**

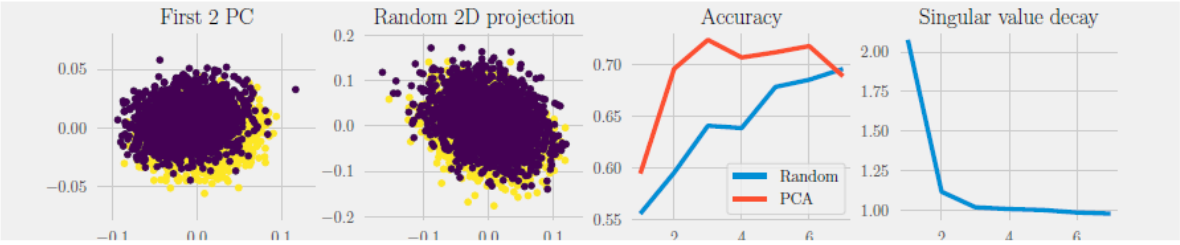


Figure 3: Plots related to data1.npz

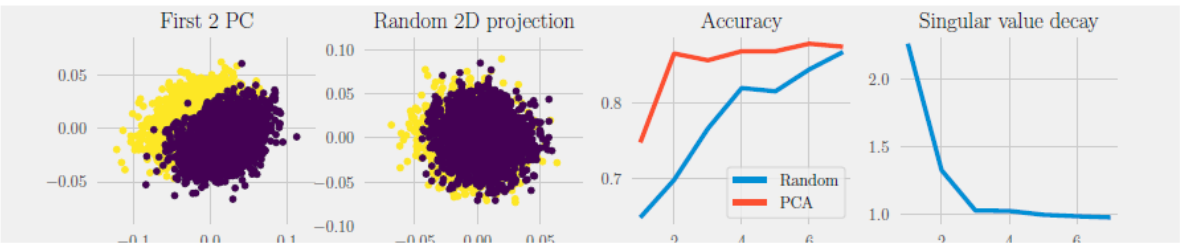


Figure 4: Plots related to data2.npz

a pattern across the three datasets? Does it help to explain the performance of PCA observed in the previous parts?

Solution: For plots, refer to figures 3, 4 and 5.

The significance of the top singular value increases from data1 to data2 with maximum value occurring for data3. For data1, first singular value accounts for approximately $2/8 \approx 25\%$ of the variance, while for data2 it explains $2.5/8.5 \approx 30\%$ variance. For Data3 this percentage is close to $4.5/11 \approx 41\%$.

Intuitively, if the labels are dependent on the direction of maximum variance, we hope to see an increase in accuracy if the ratio of variance explained goes up. PCA 2d visualizations suggest that the labeling is associated with the first principal component.

As a result, we do expect PCA with just one component do very well for data3, and in general the classification accuracy to increase from data1 to data3.

However, note that as the signal to noise ratio increases, the accuracy of random projections also increases.

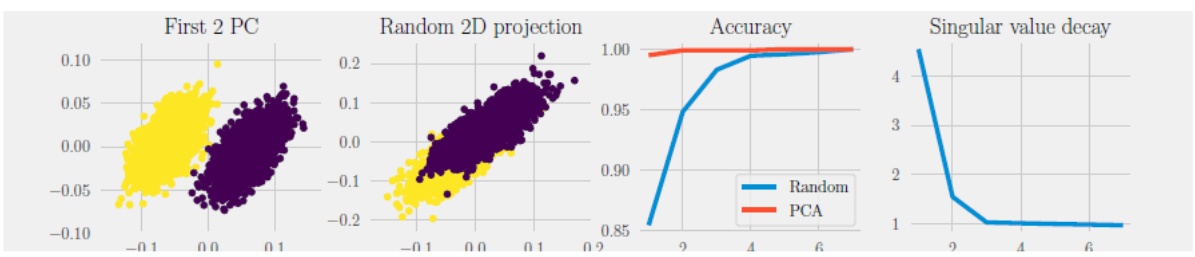


Figure 5: Plots related to data3.npz