Assignment 5

Principal Components Analysis (50 points)

PCA and centering (15 points)

Assume we are given a matrix $S \in \mathbb{R}^{d \times d}$. The rank of S is maximized if the columns (or rows) of S are linearly independent. In this case $\text{Rank}(S) = d$. Assume that this is the case. Now consider centering the matrix. That is, we transform the matrix

$$S = \begin{pmatrix} - - - & s_1^T & - - - \\ & \vdots & \\ - - - & s_p^T & - - - \end{pmatrix}$$

Into the centered matrix

$$\tilde{S} = \begin{pmatrix} - - - & (s_1 - \bar{s})^T & - - - \\ & \vdots & \\ - - - & (s_p - \bar{s})^T & - - - \end{pmatrix}$$

Where $\bar{s} = \frac{1}{p} \sum_{i=1}^{p} s_i$ Recall that a set of vectors $(x_i)_{i \in \{1, \cdots, n\}}$ are linearly independent if and only if

$$\sum_{i=1}^{n} x_i a_i = 0$$

implies that $a_i = 0$ for all i. Now consider summing the rows of $\tilde{S}$,

$$\sum_{i=1}^{p} (s_i - \bar{s})^T = \sum_{i=1}^{p} s_i^T - p\bar{s}^T$$

Now setting this equal to 0,

$$\sum_{i=1}^{p} s_i^T - p\bar{s}^T = 0 \Leftrightarrow \frac{1}{p} \sum_{i=1}^{p} s_i^T - \bar{s}^T = \bar{s}^T - \bar{s}^T = 0$$

Which clearly always holds. Thus, the rows of $\tilde{S}$ are linearly dependent, and we can write one of the rows as a linearly combination of the others, and therefore we have that $\tilde{S} \leq d - 1$.

Explained variance and Bessel's correction (5 points)

One computes eigenvalues of a $n \times n$ matrix, X by solving the characteristic polynomial

$$\det(X - \lambda I) = 0$$

for λ. Denote the set of eigenvalues for X by λ_i. Now consider that $\det(\alpha X) = \alpha^n X$. It follows that the eigenvalues of $\tilde{X} = \alpha X$ are determined by

$$\det(\alpha X - \lambda I) = \alpha^n \det(X - \frac{\lambda}{\alpha} I) = 0$$

Considering the nondegenerate case $\alpha \neq 0$, this expression is 0, if and only if the factor with the determinant is 0, but we know what the solutions to this polynomial are. They are simply the scaled eigenvalues of X. Let $\tilde{\lambda}_i$ denote the i'th eigenvalue of $\tilde{X}$, then $\tilde{\lambda}_i = \alpha\lambda_i$. Now consider the explained variance of the first k principal components. We can set the non-Bessel corrected covariance matrix $S \cdot \frac{1}{N}$ equal to X, and the Bessel corrected matrix equal to $\tilde{X}$ (by choosing $\alpha = \frac{N}{N-1}$). Then the explained variance of the first k principal components in the non-Bessel corrected case is

$$\frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}$$

And in the Bessel-corrected case, the explained variance of the first k principal components is

$$\frac{\sum_{i=1}^{k} \tilde{\lambda}_i}{\sum_{i=1}^{p} \tilde{\lambda}_i} = \frac{\alpha \sum_{i=1}^{k} \lambda_i}{\alpha \sum_{i=1}^{p} \lambda_i} = \frac{\sum_{i=1}^{k} \lambda_i}{\sum_{i=1}^{p} \lambda_i}$$

And therefore the explained variance does not depend on whether Bessel-correction is used or not.

PCA in practice

Explained variance (15 points)

As the problem set is ambigious on whether to plot the explained variance or the eigenspectrum, we plot both. We first plot the explained variance of the principal k components, as a function of k. We first do the PCA

```
pca = PCA() #Intialize PCA
pca.fit(X) #Fit PCA to data
explained_variance = pca.explained_variance_ratio_ #Get explained variance ratio
```

We the plot the explained variance, as a cumulative sum of the number of principal components

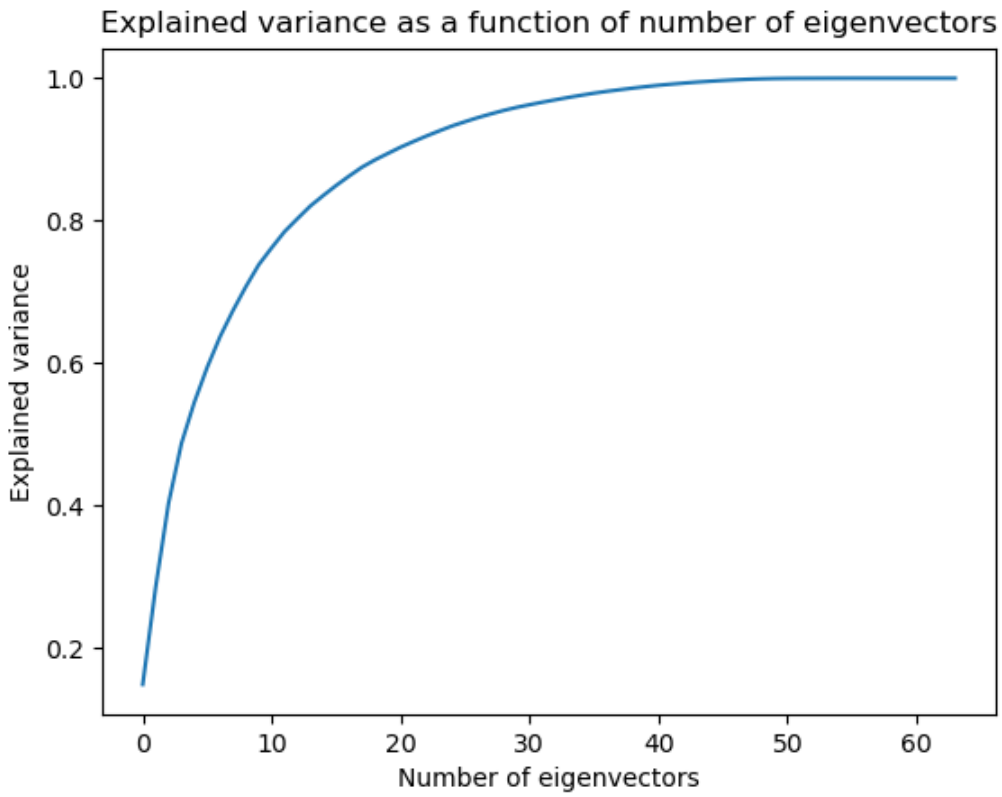


Figure 1: Explained variance as a function of number of principal components

From this plot, we get an indication that 10 principal components are not enough to explain 80 percent of variance. To confirm this, we can also calculate the explained variance of the 10 principal components,

```
np.sum(explained_variance[0:10])
```

And we see that the sum of the explained variance of the 10 principal components is 0.7382, and cleraly not enough to explain 0.8 of the variance.

Secondly we plot the eigenspectrum. That is, we divide the individual eigenvalues (sorted by magnitude) by the sum of eigenvalues.

```
eigenvalues = pca.singular_values_**2 #calculate eigenvalues
plt.plot(range(len(eigenvalues)), eigenvalues/np.sum(eigenvalues)) # plot
```

We firstly do this for every eigenvalue,

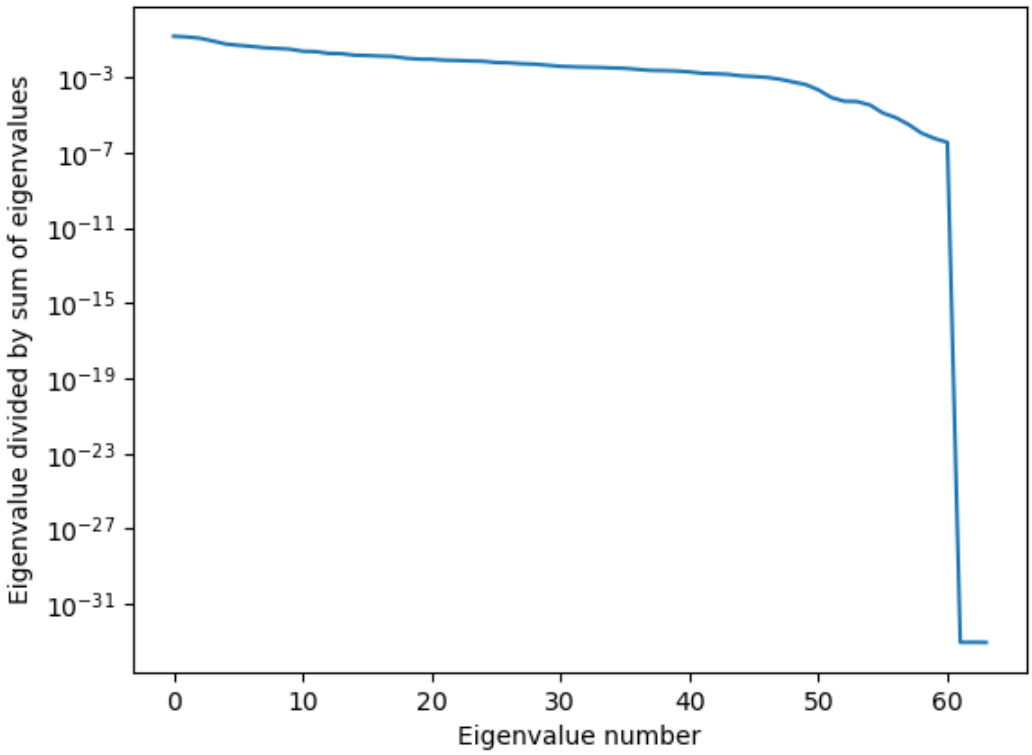


Figure 2: Eigenspectrum for all eigenvalues

As the 3 smallest eigenvalues are very small, these eigenvalues skew the plot. We therefore also plot the eigenspectrum excluding these.

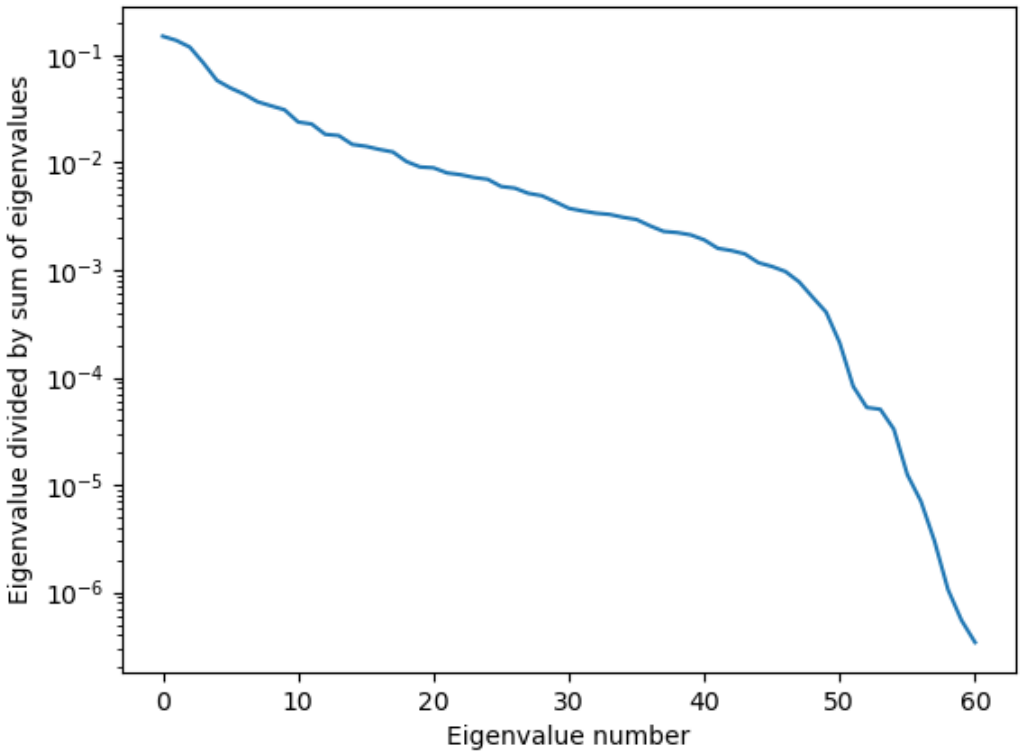


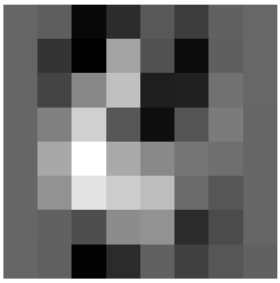
Figure 3: Eigenspectrum excluding last 3 eigenvalues

Eigendigits (15 points)

We plot the first 5 eigendigits as images. We extract the first 5 eigenvectors and reshape with the `imshape` shape provided.

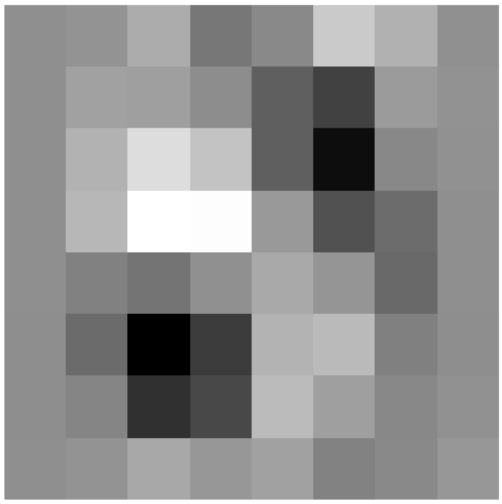
```
eigendigits_5 = pca.components_[0:5,:] ##First axis is components!
for i in range(5):
    mat = np.reshape(eigendigits_5[i,:],imshape) # reshape as imshape
    plt.imshow(mat, cmap="gray")
    plt.axis('off')
    plt.savefig("component"+str(i+1)+".png", bbox_inches='tight')
    plt.close()
```

This results in the 5 principal components



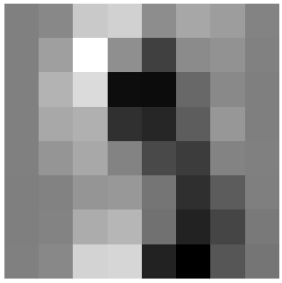
(a) 1st component

(c) 3rd component



(e) 5th component

(b) 2nd component



(d) 4th component

Logistic Regression in PyTorch (50 points)

We reimplement logistic regression from Scikit-Learn in PyTorch. We first define the model with a linear layer, and the log-softmax function as the activation function (why log-softmax is chosen instead of softmax will be apparent when we define the loss function)

```
class LogisticRegressionPytorch(nn.Module):
    def __init__(self, d, m):
        super(LogisticRegressionPytorch, self).__init__()
        # Define linear layer with d input features
        #  and m output features
        self.fc = nn.Linear(d,m)
    def forward(self, x):
        x = F.log_softmax(self.fc(x),1) # Log-softmax as activation function.
        #Apply over dim = 1
        return x
logreg_pytorch = LogisticRegressionPytorch(d, m)
```

Next we, define the loss function. We use the negative log-likelihood as our loss function. This ensures that we obtain the cross-entropy loss, as we have defined our activation function as the log-softmax function. We could have completely equivalently used a linear activation function, and then used the cross-entropy loss as the loss function. We initialize our optimizer and setup the criterion

```
optimizer = optim.Adam(logreg_pytorch.parameters(), lr=0.01)
criterion = torch.nn.NLLLoss() # Negative log likelihood loss
```

Now we get to training the model. In each epoch, we first set the gradient to 0. We apply our model

to the training dataset and feed forward through the network, to obtain outputs. We calculate our loss on the train labels, and afterwards calculate the gradient of the loss w.r.t. the model parameters, and update our parameters accordingly.

```
no_epochs = 10000  # Number of training steps
X_train_T = torch.Tensor(X_train)  # Automatically casts to float
y_train_T = torch.from_numpy(y_train)  # Does not cast to float
for epoch in range(no_epochs):  # Loop over the dataset multiple times
    # Zero the parameter gradients
    optimizer.zero_grad()

    # Forward + backward + optimize
    outputs = logreg_pytorch(X_train_T) # Apply model to train set
    loss = criterion(outputs, y_train_T.long())
    # Code wouldn't compile if y not converted to long
    loss.backward() # Backpropagate

    optimizer.step() # update parameters
```

We are now ready to visulize our data. We first get our parameters in a suitable format

```
ws_torch = logreg_pytorch.fc.weight.detach().numpy() # Convert weights to numpy array
bs_torch = logreg_pytorch.fc.bias.detach().numpy() # Convert bias to numpy array
```

And we are now ready to visualize the models. We visualize both the sklearn model and our pytorch implementation, and get the following two plots

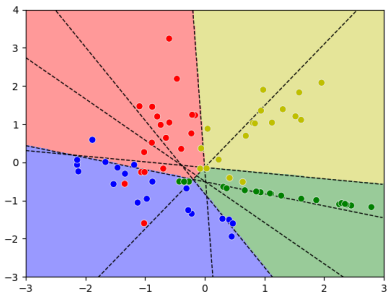


Figure 5: The Scikit-Learn model

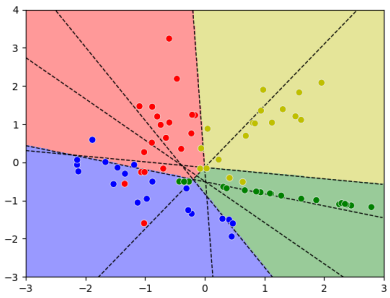


Figure 6: The pytorch model

Unsurprisingly, the 2 models look identical, and of course also have similar test and training error.