p3_testing

December 10, 2023

1 Qpca1 (15%) Implement PCA

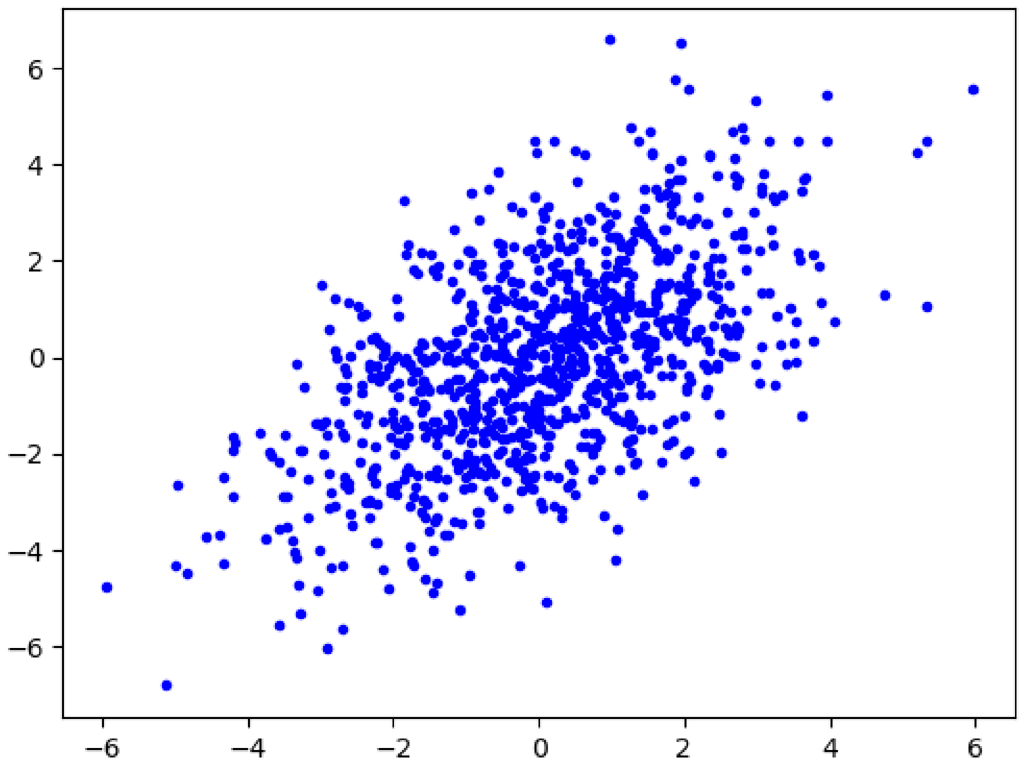
[1]:
```
from numpy import *
from matplotlib.pyplot import *
import util
```

[2]:
```
Si = util.sqrtm(array([[3,2],[2,4]]))
```

[3]:
```
x = dot(random.randn(1000,2), Si)
```

[4]:
```
plot(x[:,0], x[:,1], 'b.')
```

[4]:
```
[<matplotlib.lines.Line2D at 0x136f5d68490>]
```



[5]:
```
dot(x.T,x) / real(x.shape[0]-1) # The sample covariance matrix. Random
 ↪generated data cause result to vary
```

[5]:
```
array([[3.02255811, 2.00896613],
       [2.00896613, 4.1563332 ]])
```

[6]:
```
import dr
```

[7]:
```
(P,Z,evals) = dr.pca(x, 2)
```

[8]:
```
Z
```

[8]:
```
array([[-0.60325301, -0.79754988],
       [-0.79754988,  0.60325301]])
```

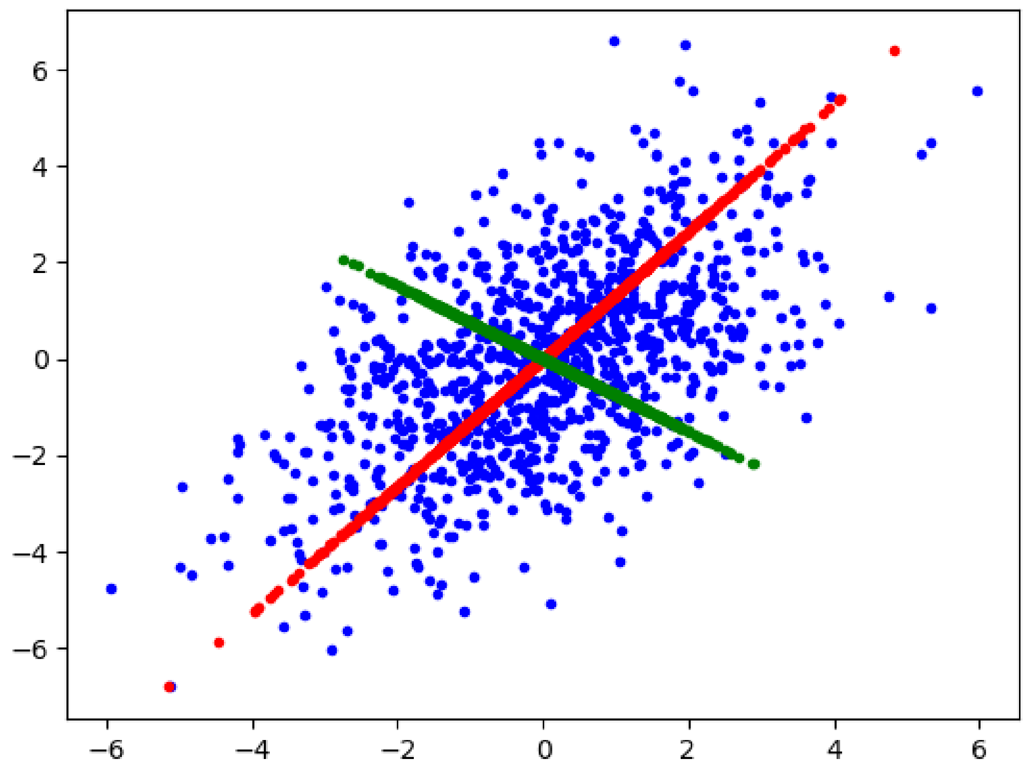
[9]:
```
evals
```

[9]:
```
array([5.6763145 , 1.49895553])
```

[10]:
```
x0 = dot(dot(x, Z[:,0]).reshape(1000,1), Z[:,0].reshape(1,2))
x1 = dot(dot(x, Z[:,1]).reshape(1000,1), Z[:,1].reshape(1,2))
```

[11]:
```
plot(x[:,0], x[:,1], 'b.', x0[:,0], x0[:,1], 'r.', x1[:,0], x1[:,1], 'g.')
```

[11]:
```
[<matplotlib.lines.Line2D at 0x136f6606dd0>,
 <matplotlib.lines.Line2D at 0x136f65cb650>,
 <matplotlib.lines.Line2D at 0x136f65cb710>]
```




```
[12]: import datasets
```

```
[13]: (X,Y) = datasets.loadDigits()
```

```
[14]: (P,Z,evals) = dr.pca(X, 784)
```

```
[15]: evals[:5]
```

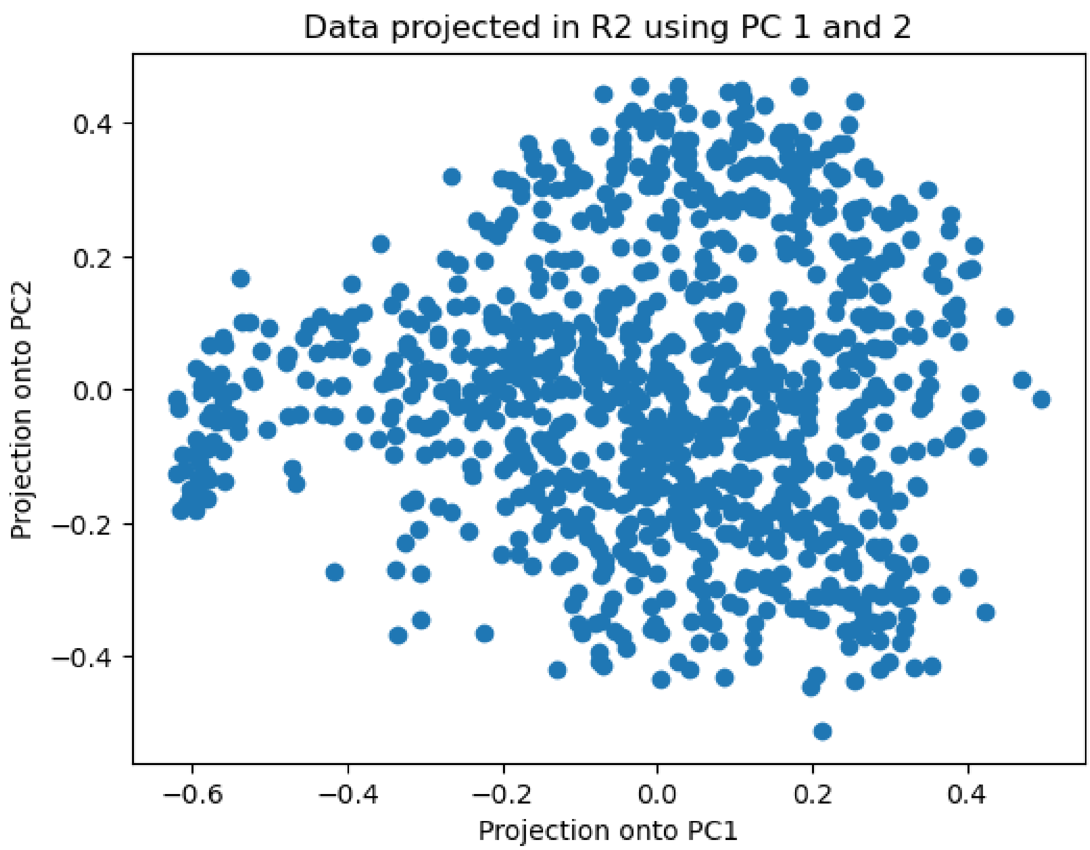
```
[15]: array([0.05471459, 0.04324574, 0.03918324, 0.03075898, 0.02972407])
```

2 Qpca2 (10%): Plot the normalized eigenvalues (include the plot in your writeup). How many eigenvectors do you have to include before you've accounted for 90% of the variance? 95%? (Hint: see function cumsum.)

Since the eigen vectors are in dimension (784 x 1) we cannot plot them. What we can plot, however, is the embedding of the data which is the data projected into a lower dimensional space using the eigen vectors as basis vectors (i.e. the projections onto the principal components). Here we can show how the more important dimensions/eigen vectors correspond to more variation in the data.

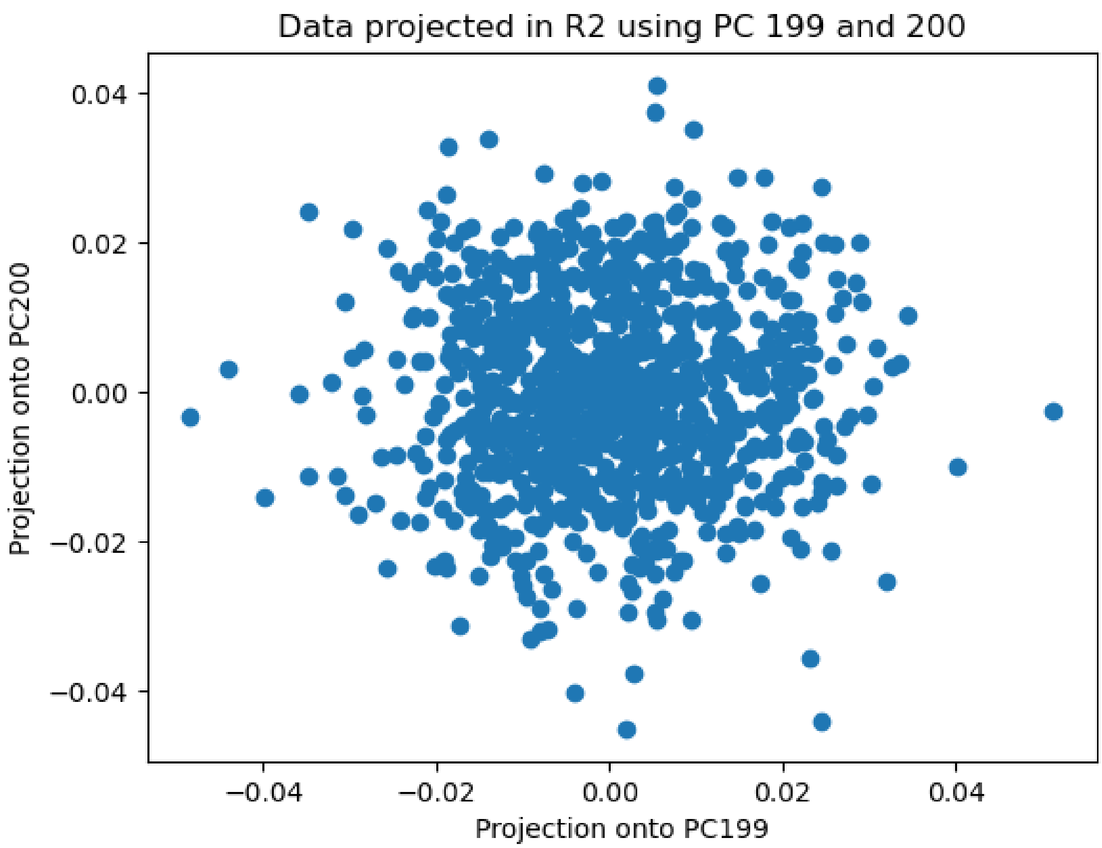
[16]:
```
scatter(P[:,0], P[:,1])
title("Data projected in R2 using PC 1 and 2")
xlabel("Projection onto PC1")
ylabel("Projection onto PC2")
```

[16]: Text(0, 0.5, 'Projection onto PC2')



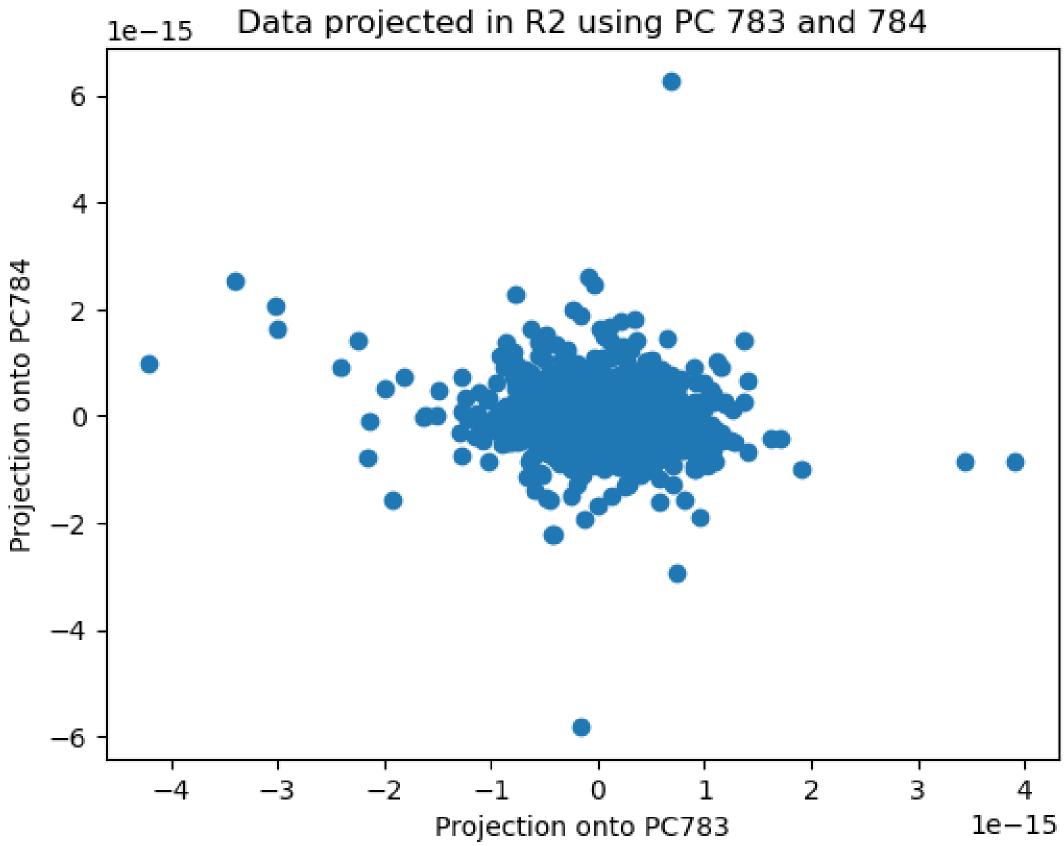
[17]:
```
scatter(P[:,198], P[:,199])
title("Data projected in R2 using PC 199 and 200")
xlabel("Projection onto PC199")
ylabel("Projection onto PC200")
```

[17]: Text(0, 0.5, 'Projection onto PC200')



[18]:
```
scatter(P[:,782], P[:,783])
title("Data projected in R2 using PC 783 and 784")
xlabel("Projection onto PC783")
ylabel("Projection onto PC784")
```

[18]:
```
Text(0, 0.5, 'Projection onto PC784')
```



[19]:
```
eval_sum = cumsum(evals)
variation = eval_sum / eval_sum[-1]
```

[20]:
```
np.argmax(variation >= 0.9)
```

[20]: 81

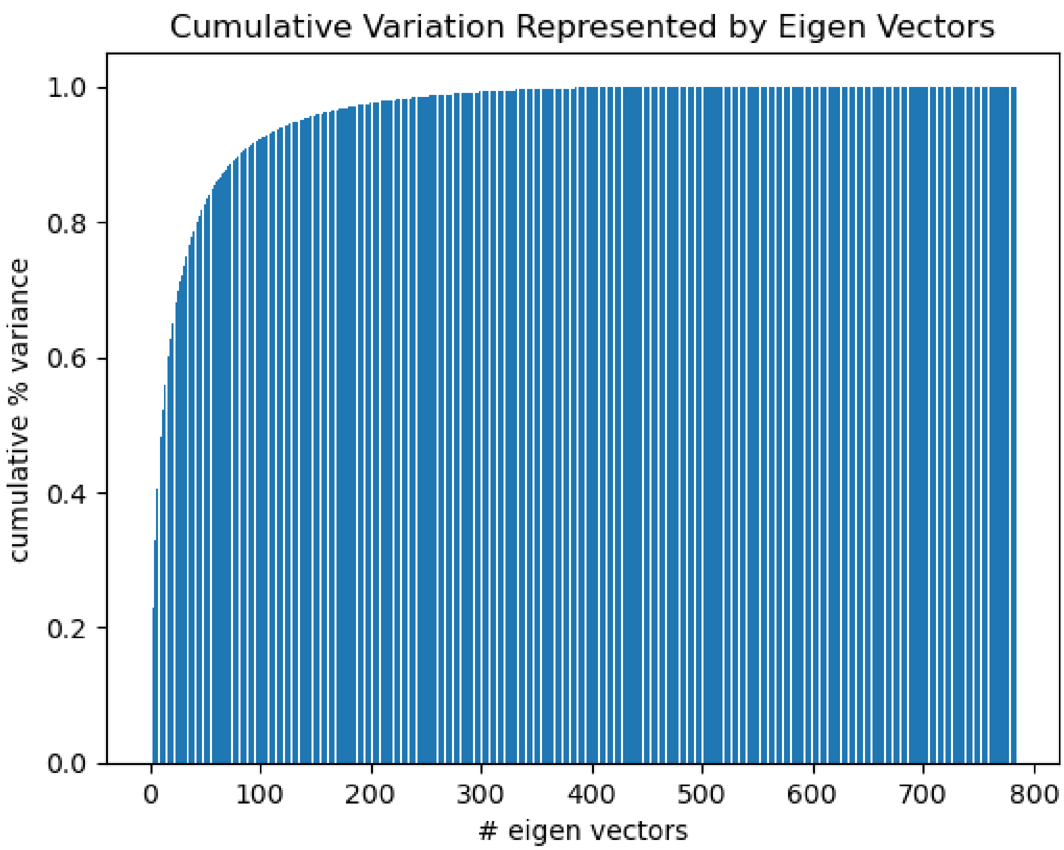
[21]:
```
np.argmax(variation >= 0.95)
```

[21]: 135

You need 82 eigen vectors (81th index) to account for 90% of the variance, and 136 eigen vectors (135th index) to account for 95% of the variance.

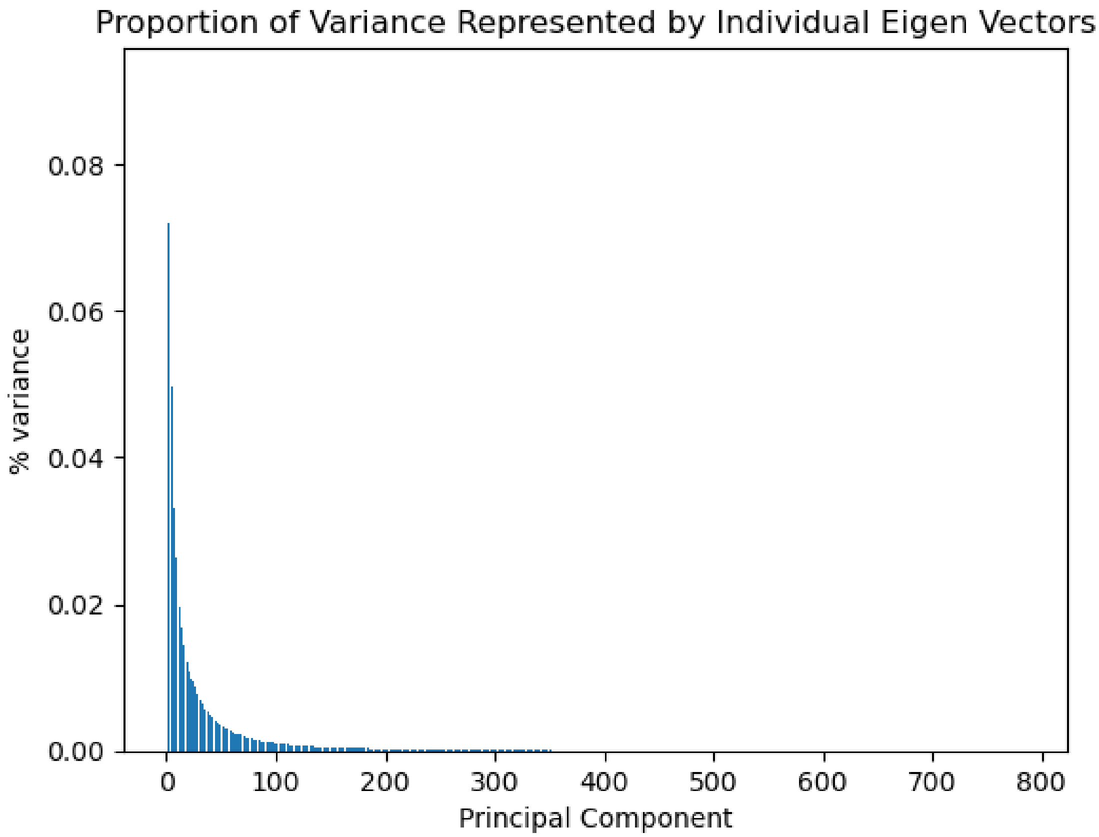
[22]:
```
bar(range(1,785), height=variation)
title("Cumulative Variation Represented by Eigen Vectors")
xlabel("# eigen vectors")
ylabel("cumulative % variance")
```

[22]: Text(0, 0.5, 'cumulative % variance')



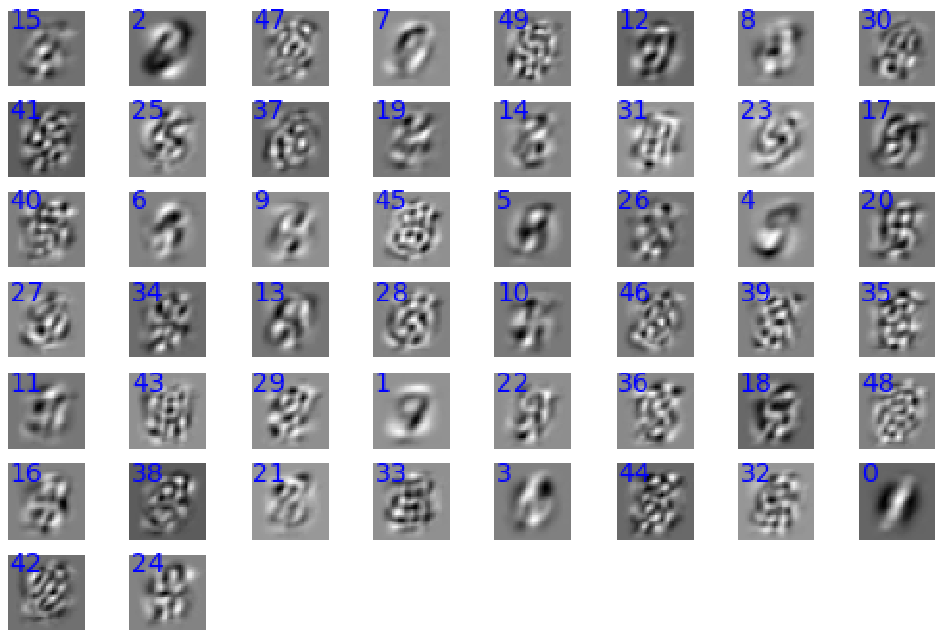
[26]:
```
bar(range(1,785), height=evals/sum(evals))
title("Proportion of Variance Represented by Individual Eigen Vectors")
xlabel("Principal Component")
ylabel("% variance")
```

[26]: `Text(0, 0.5, '% variance')`



3 Qpca3 (5%): Do these look like digits? Should they? Why or why not? (Include the plot in your write-up.) (Make sure you have got rid of the imaginary part in pca.)

[28]:
```
util.drawDigits(Z.T[:50,:], arange(50))
```



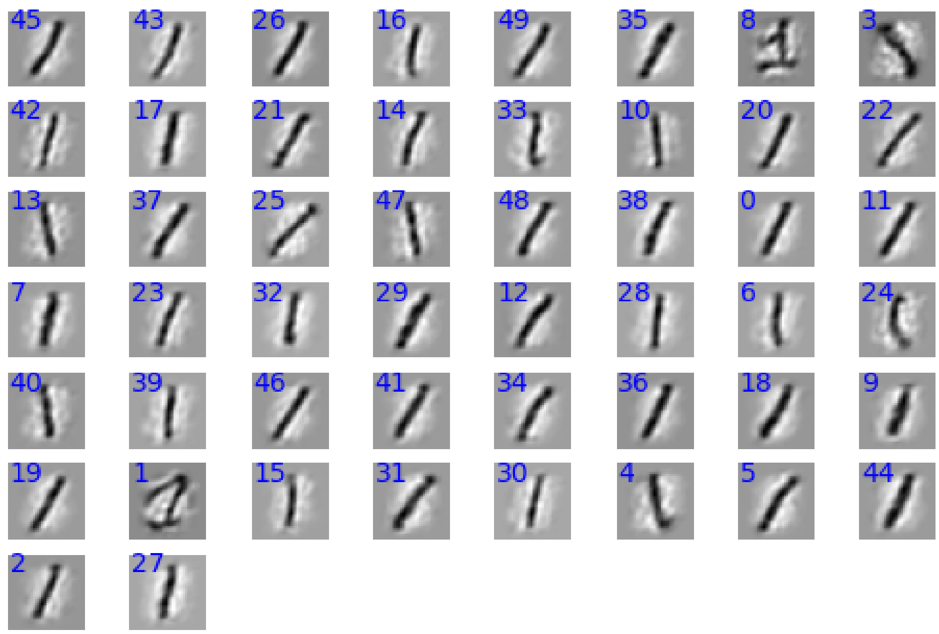
No these don't look like digits, and they shouldn't, as these are just the eigen vectors corresponding to the top 50 dimensions with the most variance (i.e. basis vectors). In order to get an image that looks like a digit, you must have data somewhat similar to what you started with (i.e. data that represents pixels in an image, not basis vectors nor embedded data).

To reconstruct the original data you need to use the embedded data (P) and the transpose of the mapping/eigen vectors (Z.T will map the embedded data back to the original form like an inverse operation). If you only want to use the top 50 eigen vectors, call pca with 50, so P is of dimension (n x 50) and Z is of dimension (d x 50) so P@Z.T is of dimension (n x d) which is a "reconstruction" of the original image (it's an approximation, since you're storing less information in P and Z compared to X).

Below, showcases the reconstruction process outlined above using 50 eigen vectors for the first 50 digits.

[29]:
```
(P,Z,evals) = dr.pca(X, 50)
```

[30]:
```
util.drawDigits((P @ Z.T)[:50,:], arange(50))
```



4 Qsr1 (10%)

4.0.1 (1) Show that the probabilities sum to 1

4.0.2 (2) What are the dimensions of W? X? WX?

(1)

$$\sum_{k=1}^{K} P[y=k] = \sum_{k=1}^{K} \frac{e^{\vec{w}_k \cdot \vec{x}}}{\sum_{j=1}^{K} e^{\vec{w}_j \cdot \vec{x}}} = \frac{\sum_{k=1}^{K} e^{\vec{w}_k \cdot \vec{x}}}{\sum_{j=1}^{K} e^{\vec{w}_j \cdot \vec{x}}} = 1$$

(2) W is of dimension (K x D) where K is the number of classes and D is the number of features. This is because each row i of W is w_i and w_i is of dimension (1 x D) when it's represented as a row since it provides weights to the D features of an x vector. X i s of dimension (D x N) where N is the number of examples. This is because each column of X is a single example x, and x is of dimension (D x 1) when represented as a column vector. Hence, the dimension of WX is (K x N) since (K x D * D x N) = (K x N). This makes sense as their should be K probabilities returned for each of the N observations since we are using softmax.

[31]:
```
from utils import *
from softmax import *

# Code adapted from https://github.com/jatinshah/ufldl_tutorial

# MNIST images are 28 * 28
exSize = 28*28
# 10 digits
```

```
numClasses = 10
# Regularizer coefficient
reg = 0.0001


X, Y = loadMNIST('data/train-images.idx3-ubyte', 'data/train-labels.idx1-ubyte')
sm = SoftmaxRegression(numClasses, exSize, opts={'maxIter':402})
sm.train(X, Y)

testX, testY = loadMNIST('data/t10k-images.idx3-ubyte', 'data/t10k-labels.
↪idx1-ubyte')
predictions = sm.predict(X)
print("Accuracy: {0:.2f}%".format(100 * np.sum(predictions == Y, dtype=np.
↪float64) / Y.shape[0]))
```

```
Accuracy: 94.02%
```

5 Qsr3 (10%)

5.0.1 In the cost function, we see the line

5.0.2 W_X = W_X - np.max(W_X)

5.0.3 This means that each entry is reduced by the largest entry in the matrix.

5.0.4 (1) Show that this does not affect the predicted probabilities.

5.0.5 (2) Why might this be an optimization over using W_X? Justify your answer.

(1)

$$\frac{e^{\vec{w}_k\cdot\vec{x}-C}}{\sum_{j=1}^{K} e^{\vec{w}_j\cdot\vec{x}-C}} = \frac{e^{\vec{w}_k\cdot\vec{x}}e^{-C}}{\sum_{j=1}^{K} e^{\vec{w}_j\cdot\vec{x}}e^{-C}} = \frac{e^{-C}e^{\vec{w}_k\cdot\vec{x}}}{e^{-C}\sum_{j=1}^{K} e^{\vec{w}_j\cdot\vec{x}}} = \frac{e^{\vec{w}_k\cdot\vec{x}}}{\sum_{j=1}^{K} e^{\vec{w}_j\cdot\vec{x}}} = P[y=k]$$

Where C is np.max(W_X) and <w_k,x> is an element of the W_X matrix (suppose x is pth observation; x=x_p). Hence, you can see that subtracting a constant C from each <w_i, x_p> component of the W_X matrix, ultimately does not impact the the final predicted probabilities as the e^-C terms will cancel out, leaving you with the original P(y=k) probability (where you use W_X elements in the exponent, without subtracting by the np.max).

(2) This is an optimization over using W_X because it will make the exponents smaller. Since e^x grows exponentially fast, we want smaller exponents to reduce the likelihood of overflow. This is especially the case for the denominator of the probability equation, since we are adding up all the exponentials. Also, working with smaller numbers is nicer because the numerical errors will be smaller too, so we get more accurate computations. For example, if your numerical method has an error of 10% and the actual result is 0.5, you could get a value like 0.55, but if your actual result is 500 then you could get a value like 550, which has a much larger error of 50.

Hence, by subtracting the max from all elements, we get to work with smaller numbers which reduces the likelihood of overflow errors and we reduce the size of numerical errors, all without affecting the final predicted probabilities.

6 Qsr4 (10%)

6.0.1 Use the learningCurve function in runClassifier.py to plot the accuracy of the classifier as a function of the number of examples seen. Include the plot in your write-up. Do you observe any overfitting or underfitting? Discuss and expain what you observe.

[38]:
```python
from runClassifier import *
```

[39]:
```python
X, Y = loadMNIST('data/train-images.idx3-ubyte', 'data/train-labels.idx1-ubyte')
testX, testY = loadMNIST('data/t10k-images.idx3-ubyte', 'data/t10k-labels.
↪idx1-ubyte')
# MNIST images are 28 * 28
exSize = 28*28
# 10 digits
numClasses = 10
sm = SoftmaxRegression(numClasses, exSize)
sizes, trainAcc, testAcc = learningCurve(sm,10,28*28,X,Y,testX,testY)
```

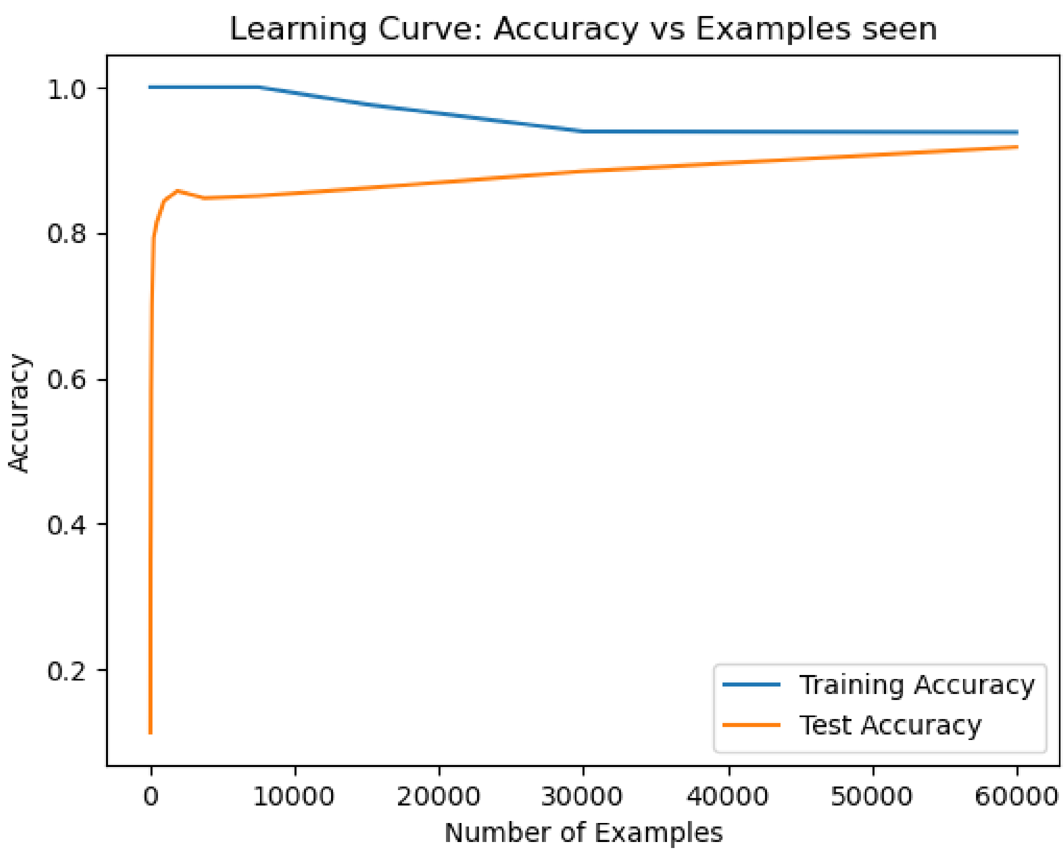
```
60000
16
1
(784, 2)
(2,)
Training classifier on 2 points…
Training accuracy 1, test accuracy 0.1139
2
(784, 4)
(4,)
Training classifier on 4 points…
Training accuracy 1, test accuracy 0.1437
3
(784, 8)
(8,)
Training classifier on 8 points…
Training accuracy 1, test accuracy 0.2692
4
(784, 15)
(15,)
Training classifier on 15 points…
Training accuracy 1, test accuracy 0.3502
5
(784, 30)
(30,)
Training classifier on 30 points…
Training accuracy 1, test accuracy 0.4004
6
(784, 59)
```

```
(59,)
Training classifier on 59 points…
Training accuracy 1, test accuracy 0.5819
7
(784, 118)
(118,)
Training classifier on 118 points…
Training accuracy 1, test accuracy 0.7018
8
(784, 235)
(235,)
Training classifier on 235 points…
Training accuracy 1, test accuracy 0.793
9
(784, 469)
(469,)
Training classifier on 469 points…
Training accuracy 1, test accuracy 0.815
10
(784, 938)
(938,)
Training classifier on 938 points…
Training accuracy 1, test accuracy 0.8431
11
(784, 1875)
(1875,)
Training classifier on 1875 points…
Training accuracy 1, test accuracy 0.8575
12
(784, 3750)
(3750,)
Training classifier on 3750 points…
Training accuracy 1, test accuracy 0.8475
13
(784, 7500)
(7500,)
Training classifier on 7500 points…
Training accuracy 1, test accuracy 0.8506
14
(784, 15000)
(15000,)
Training classifier on 15000 points…
Training accuracy 0.9762, test accuracy 0.8613
15
(784, 30000)
(30000,)
Training classifier on 30000 points…
Training accuracy 0.939167, test accuracy 0.8845
```

```
16
(784, 60000)
(60000,)
Training classifier on 60000 points…
Training accuracy 0.938083, test accuracy 0.9176
```

[37]:
```
plot(sizes,trainAcc,label='Training Accuracy')
plot(sizes,testAcc,label='Test Accuracy')
title("Learning Curve: Accuracy vs Examples seen")
xlabel("Number of Examples")
ylabel("Accuracy")
legend()
```

[37]:
```
<matplotlib.legend.Legend at 0x136805af050>
```



At first you can see the softmax regression model is severely overfitting as it has an accuracy of 100% on the training data, but it performs poorly on the test data with an accuracy of 10% - 60%. However, as we add more data we reduce the variance of our model as it starts to learn from more diverse examples rather than just a select few. This results in a lower variance model, without really compromising the complexity/bias, hence giving us a better test accuracy as you can see above. The training error decreases as the number of examples increases because the model starts to generalize rather than memorize; there are more examples to predict so it can't perfectly predict all the training data like it used to be able to, so it generalizes as best as it can which leads to some training error. There doesn't seem to be any sign of underfitting as the training accuracy is always respectable at around +90% (the model is always capturing some pattern).

Hence we can see that our model is a high complexity (low bias) model, which is made better by providing more data as it reduces the variance of our model (which in turn, results in better test accuracies of ~90%).

7 Qnn1 (20% for Qnn1.1, 1.2, 1.3 and 5% for Qnn 1.4) Implement the NN

[40]:
```
from nn import NN, Relu, Linear, SquaredLoss
from utils import data_loader, acc, save_plot, loadMNIST, onehot
```

[41]:
```
model = NN(Relu(), SquaredLoss(), hidden_layers=[128,128])
```

[42]:
```
model.print_model()
```

```
activation:Relu
loss function:SquaredLoss
Layer 1 w:(128, 784)    b:(128, 1)
Layer 2 w:(128, 128)    b:(128, 1)
Layer 3 w:(10, 128)     b:(10, 1)
```

[43]:
```
x_train, label_train = loadMNIST('data/train-images.idx3-ubyte', 'data/
 ↪train-labels.idx1-ubyte')
```

[44]:
```
x_test, label_test = loadMNIST('data/t10k-images.idx3-ubyte', 'data/t10k-labels.
 ↪idx1-ubyte')
```

[45]:
```
y_train = onehot(label_train)
```

[46]:
```
y_test = onehot(label_test)
```

[47]:
```
model = NN(Relu(), SquaredLoss(), hidden_layers=[128, 128], input_d=784,␣
 ↪output_d=10)
```

[48]:
```
model.print_model()
```

```
activation:Relu
loss function:SquaredLoss
Layer 1 w:(128, 784)    b:(128, 1)
Layer 2 w:(128, 128)    b:(128, 1)
Layer 3 w:(10, 128)     b:(10, 1)
```

[49]:
```
training_data, dev_data = {"X":x_train, "Y":y_train}, {"X":x_test, "Y":y_test}
```

[50]:
```python
from run_nn import train_1pass
```

[51]:
```python
model, plot_dict = train_1pass(model, training_data, dev_data,␣
↪learning_rate=1e-2, batch_size=64)
```

```
#Samples  6400  loss:0.49420    dev_acc:0.58390
#Samples 12800  loss:0.33410    dev_acc:0.70730
#Samples 19200  loss:0.29239    dev_acc:0.76880
#Samples 25600  loss:0.26416    dev_acc:0.80590
#Samples 32000  loss:0.24390    dev_acc:0.82370
#Samples 38400  loss:0.22800    dev_acc:0.83710
#Samples 44800  loss:0.22074    dev_acc:0.84770
#Samples 51200  loss:0.20776    dev_acc:0.85970
#Samples 57600  loss:0.19842    dev_acc:0.86700
```

[52]:
```python
import numpy as np
from nn import NN
from nn import Relu, Linear, SquaredLoss, CELoss
from utils import data_loader, acc, save_plot, loadMNIST, onehot

# Several passes of the training data
def train(model, training_data, dev_data, learning_rate, batch_size, max_epoch):
    X_train, Y_train = training_data['X'], training_data['Y']
    X_dev, Y_dev = dev_data['X'], dev_data['Y']
    for i in range(max_epoch):
        for X,Y in data_loader(X_train, Y_train, batch_size=batch_size,␣
↪shuffle=True):
            training_loss, grad_Ws, grad_bs = model.compute_gradients(X, Y)
            model.update(grad_Ws, grad_bs, learning_rate)
        dev_acc = acc(model.predict(X_dev), Y_dev)
        print("Epoch {: >3d}/{}\tloss:{:.5f}\tdev_acc:{:.5f}".
↪format(i+1,max_epoch,training_loss, dev_acc))
    return model

# One pass of the training data
def train_1pass(model, training_data, dev_data, learning_rate, batch_size,␣
↪print_every=100, plot_every=10):
    X_train, Y_train = training_data['X'], training_data['Y']
    X_dev, Y_dev = dev_data['X'], dev_data['Y']

    num_samples = 0
    print_loss_total = 0
    plot_loss_total = 0

    plot_losses = []
    plot_num_samples = []
```

```
    for idx, (X,Y) in enumerate(data_loader(X_train, Y_train,␣
↪batch_size=batch_size, shuffle=True),1):
        training_loss, grad_Ws, grad_bs = model.compute_gradients(X, Y)
        model.update(grad_Ws, grad_bs, learning_rate)
        num_samples += Y.shape[1]
        print_loss_total += training_loss
        plot_loss_total += training_loss

        if idx % print_every == 0:
            dev_acc = acc(model.predict(X_dev), Y_dev)
            print_loss_avg = print_loss_total/print_every
            print_loss_total = 0
            print("#Samples {: >5d}\tloss:{:.5f}\tdev_acc:{:.5f}".
↪format(num_samples, print_loss_avg, dev_acc))
        if idx % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_loss_total = 0
            plot_losses.append(plot_loss_avg)
            plot_num_samples.append(num_samples)

    return model, {"losses":plot_losses, "num_samples":plot_num_samples}

if __name__ == "__main__":
    x_train, label_train = loadMNIST('data/train-images.idx3-ubyte', 'data/
↪train-labels.idx1-ubyte')
    x_test, label_test = loadMNIST('data/t10k-images.idx3-ubyte', 'data/
↪t10k-labels.idx1-ubyte')
    y_train = onehot(label_train)
    y_test = onehot(label_test)

    model = NN(Relu(), SquaredLoss(), hidden_layers=[256, 256], input_d=784,␣
↪output_d=10)
    model.print_model()

    lr = 1e-2
    max_epoch = 20
    batch_size = 128
    training_data = {"X":x_train, "Y":y_train}
    dev_data = {"X":x_test, "Y":y_test}

    #model, plot_dict = train_1pass(model, training_data, dev_data, lr,␣
↪batch_size)
    #save_plot(plot_dict["num_samples"], plot_dict["losses"])

    model = train(model, training_data, dev_data, lr, batch_size, max_epoch)
```

```
activation:Relu
```

```
loss function:SquaredLoss
Layer 1 w:(256, 784)    b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.23427    dev_acc:0.84100
Epoch   2/20    loss:0.19180    dev_acc:0.88540
Epoch   3/20    loss:0.15605    dev_acc:0.90290
Epoch   4/20    loss:0.14881    dev_acc:0.91060
Epoch   5/20    loss:0.16186    dev_acc:0.91720
Epoch   6/20    loss:0.12370    dev_acc:0.92320
Epoch   7/20    loss:0.12110    dev_acc:0.92880
Epoch   8/20    loss:0.10296    dev_acc:0.93170
Epoch   9/20    loss:0.11618    dev_acc:0.93360
Epoch  10/20    loss:0.11499    dev_acc:0.93650
Epoch  11/20    loss:0.09947    dev_acc:0.93880
Epoch  12/20    loss:0.09130    dev_acc:0.94080
Epoch  13/20    loss:0.09694    dev_acc:0.94110
Epoch  14/20    loss:0.11454    dev_acc:0.94190
Epoch  15/20    loss:0.07746    dev_acc:0.94410
Epoch  16/20    loss:0.09219    dev_acc:0.94450
Epoch  17/20    loss:0.09445    dev_acc:0.94550
Epoch  18/20    loss:0.09497    dev_acc:0.94650
Epoch  19/20    loss:0.07064    dev_acc:0.94720
Epoch  20/20    loss:0.09592    dev_acc:0.94720
```

8 Qnn1.4 (No implementation needed for this question). When initializing the weight matrix, in some cases it may be appropriate to initialize the entries as small random numbers rather than all zeros. Give one reason why this may be a good idea.

One reason why you don't initialize the weight matrix to be all 0's is that (if bias is also 0, or you're not using bias), after the first forward propagation each hidden layer's output would be the same since Wx (+b if b = 0) = 0 since W is 0. Then activating the 0 could give you 0 using relu or .5 using sigmoid. Regardless of what activation function you used, each vector output of a hidden layer would be uniformly the same (i.e. all 0's or all 0.5's), so the NN only outputs uniform results, regardless of your input vector. This is bad because in back propagation the gradient of Loss/W depends on the gradient on z/W where z = (W * output_of_prev_layer + b). Clearly, this reduces to the fact that gradient of Loss/W depends on something * the output of the previous hidden layer, which is uniformly the same value if W = 0. Hence, when you update W of a given layer using gradient descent, all the values will move together so you're W's won't be very useful (and it won't move at all if you use relu since then output_of_prev_layer = 0 always). Also, doing multiple random initializations helps you find better minimas since the weight space isn't convex.

9 Qnn2 (Extra-Credit 15%) Try something new.

(1) Do dimension reduction with PCA. Try with different dimensions. Can you observe the trade-off in time and acc? Plot training time v.s. dimension, testing time v.s dimension and acc v.s. dimension. Visualize the principal components.

[53]:
```
from sklearn.decomposition import PCA
from matplotlib.pyplot import *
from timeit import default_timer as timer
```

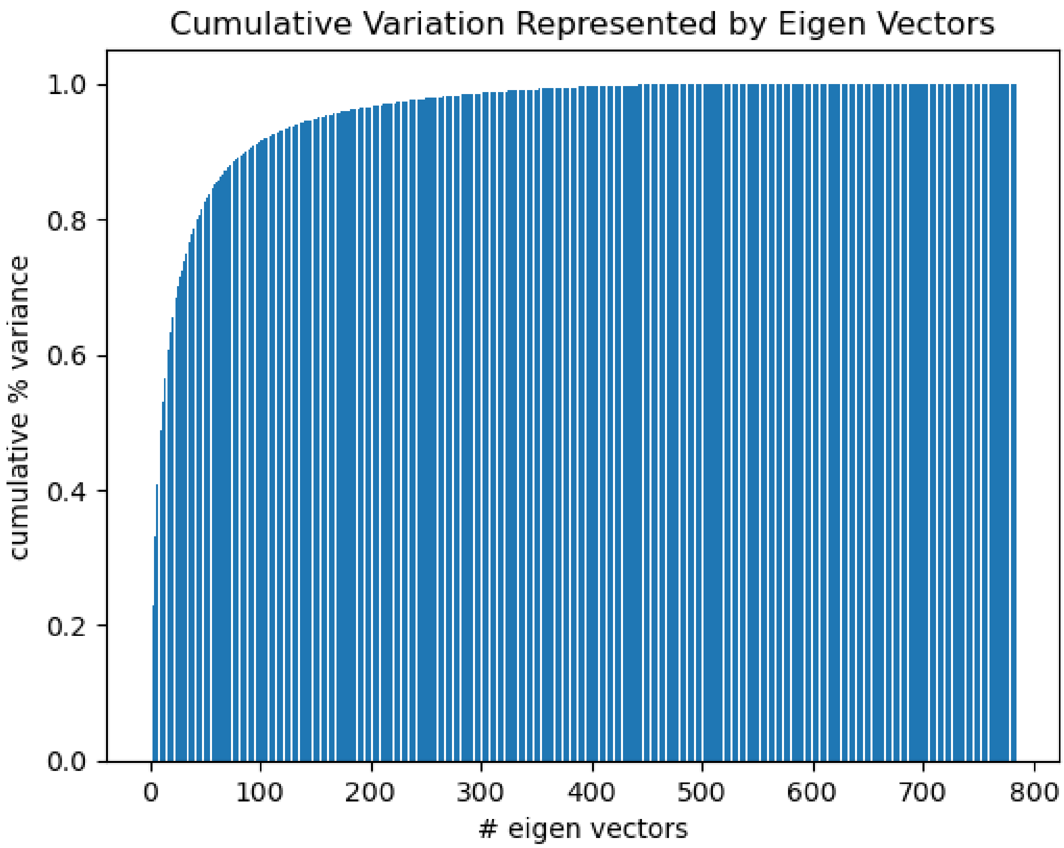
[54]:
```
import numpy as np
from nn import NN
from nn import Relu, Linear, SquaredLoss, CELoss
from utils import data_loader, acc, save_plot, loadMNIST, onehot

# Several passes of the training data
def train(model, training_data, dev_data, learning_rate, batch_size, max_epoch):
    X_train, Y_train = training_data['X'], training_data['Y']
    X_dev, Y_dev = dev_data['X'], dev_data['Y']
    for i in range(max_epoch):
        for X,Y in data_loader(X_train, Y_train, batch_size=batch_size,␣
 ↪shuffle=True):
            training_loss, grad_Ws, grad_bs = model.compute_gradients(X, Y)
            model.update(grad_Ws, grad_bs, learning_rate)
        dev_acc = acc(model.predict(X_dev), Y_dev)
        print("Epoch {: >3d}/{}\tloss:{:.5f}\tdev_acc:{:.5f}".
 ↪format(i+1,max_epoch,training_loss, dev_acc))
    return model

# One pass of the training data
def train_1pass(model, training_data, dev_data, learning_rate, batch_size,␣
 ↪print_every=100, plot_every=10):
    X_train, Y_train = training_data['X'], training_data['Y']
    X_dev, Y_dev = dev_data['X'], dev_data['Y']

    num_samples = 0
    print_loss_total = 0
    plot_loss_total = 0

    plot_losses = []
    plot_num_samples = []
    for idx, (X,Y) in enumerate(data_loader(X_train, Y_train,␣
 ↪batch_size=batch_size, shuffle=True),1):
        training_loss, grad_Ws, grad_bs = model.compute_gradients(X, Y)
        model.update(grad_Ws, grad_bs, learning_rate)
        num_samples += Y.shape[1]
        print_loss_total += training_loss
        plot_loss_total += training_loss
```

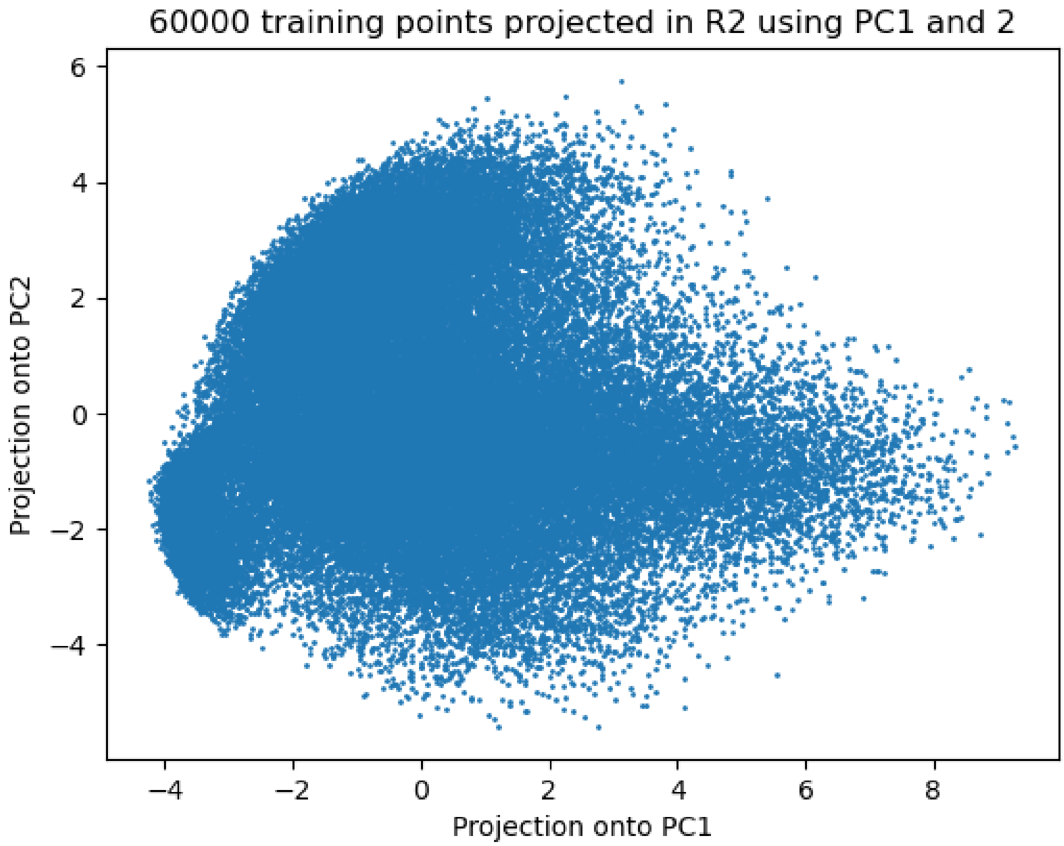
```
        if idx % print_every == 0:
            dev_acc = acc(model.predict(X_dev), Y_dev)
            print_loss_avg = print_loss_total/print_every
            print_loss_total = 0
            print("#Samples {: >5d}\tloss:{:.5f}\tdev_acc:{:.5f}".
↪format(num_samples, print_loss_avg, dev_acc))
        if idx % plot_every == 0:
            plot_loss_avg = plot_loss_total / plot_every
            plot_loss_total = 0
            plot_losses.append(plot_loss_avg)
            plot_num_samples.append(num_samples)

    return model, {"losses":plot_losses, "num_samples":plot_num_samples}
```

[55]:
```
if __name__ == "__main__":
    x_train, label_train = loadMNIST('data/train-images.idx3-ubyte', 'data/
↪train-labels.idx1-ubyte')
    x_test, label_test = loadMNIST('data/t10k-images.idx3-ubyte', 'data/
↪t10k-labels.idx1-ubyte')
    y_train = onehot(label_train)
    y_test = onehot(label_test)

    dr = PCA()
    #no need to scale data since the features (i.e. pixels) are on the same␣
↪scale
    #need to transpose x since pca requires (N x D)
    full_train = dr.fit_transform(x_train.T)
    #must transform x_test into same space/dimensions as x_train
    #note we cannot perform pca on x_train + x_test together since
    #we are not allowed to use information from x_test during training
    #also we do not perform pca on x_test separated as we would get
    #different principal components (i.e. a projection into a different space)
    #so we use the principal components from x_train
    full_test = dr.transform(x_test.T)

    #get cumulative variance explained
    variances = np.cumsum(dr.explained_variance_ratio_)
    bar(range(1,785), height=variances)
    title("Cumulative Variation Represented by Eigen Vectors")
    xlabel("# eigen vectors")
    ylabel("cumulative % variance")
    show()

    #plot projections onto different pairs of principle components
    components = [(1,2), (200, 201), (400, 401), (783,784)]
    for c1,c2 in components:
```

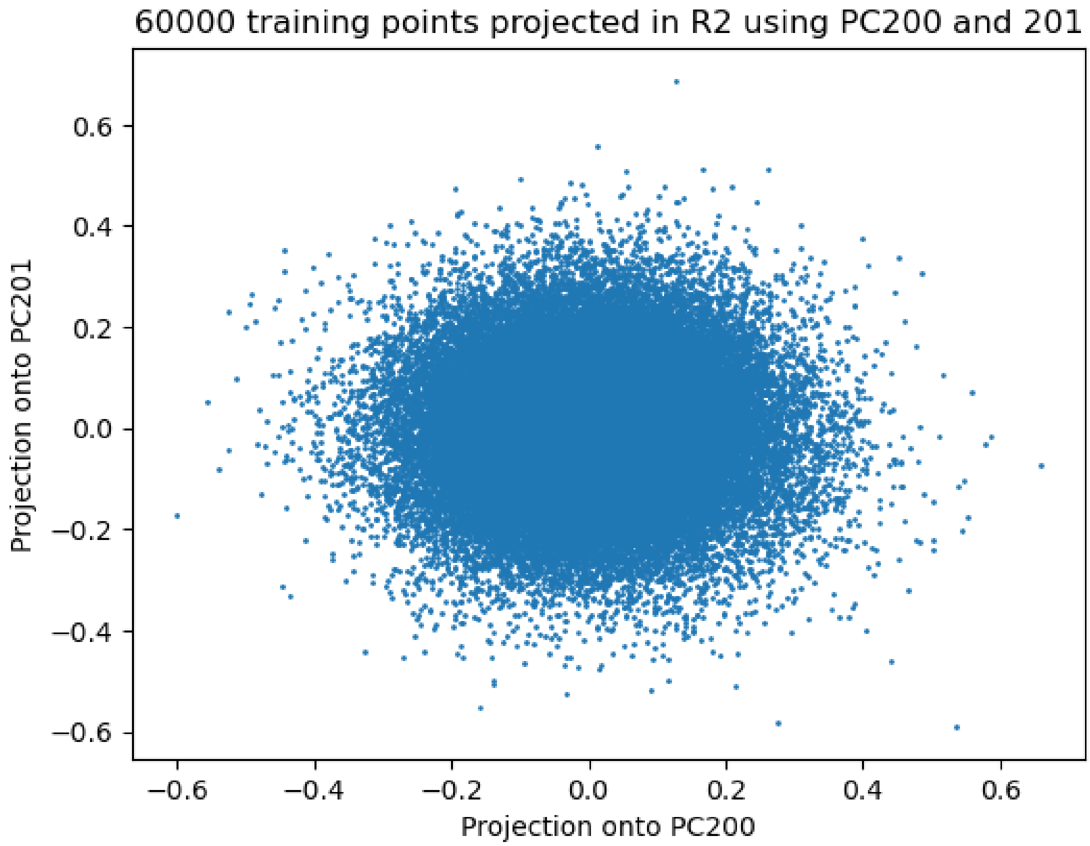
```
        scatter(full_train[:,c1-1], full_train[:,c2-1],s=1)
        title(f"{full_train.shape[0]} training points projected in R2 using␣
↪PC{c1} and {c2}")
        xlabel(f"Projection onto PC{c1}")
        ylabel(f"Projection onto PC{c2}")
        show()


    #what proportion of variance should the reduced data maintain
    proportions = [0.10, 0.25, 0.50, 0.80, 0.90, 0.99]
    dr_xtrain = []
    dr_xtest = []
    for p in proportions:
        #num of eigen vectors necessary to explain at least p of the variance
        evs = np.argmax(variances >= p) + 1
        #must take transpose since PCA requires different format of data than NN
        dr_xtrain.append(full_train[:, :evs].T)
        dr_xtest.append(full_test[:, :evs].T)

    lr = 1e-2
    max_epoch = 20
    batch_size = 128

    #track train + prediction times
    train_times = []
    test_times = []
    #track test accuracies
    accuracies = []
    for x_train,x_test in zip(dr_xtrain,dr_xtest):
        #y data can be left untouched, don't need to map into predictor space
        training_data = {"X":x_train, "Y":y_train}
        dev_data = {"X":x_test, "Y":y_test}

        model = NN(Relu(), SquaredLoss(), hidden_layers=[256, 256],␣
↪input_d=x_train.shape[0], output_d=10)
        model.print_model()

        # model, plot_dict = train_1pass(model, training_data, dev_data, lr,␣
↪batch_size)
        # save_plot(plot_dict["num_samples"], plot_dict["losses"])
        start_train = timer()
        model = train(model, training_data, dev_data, lr, batch_size, max_epoch)
        end_train = timer()

        start_test = timer()
        accuracy = acc(model.predict(x_test), y_test)
        end_test = timer()
```

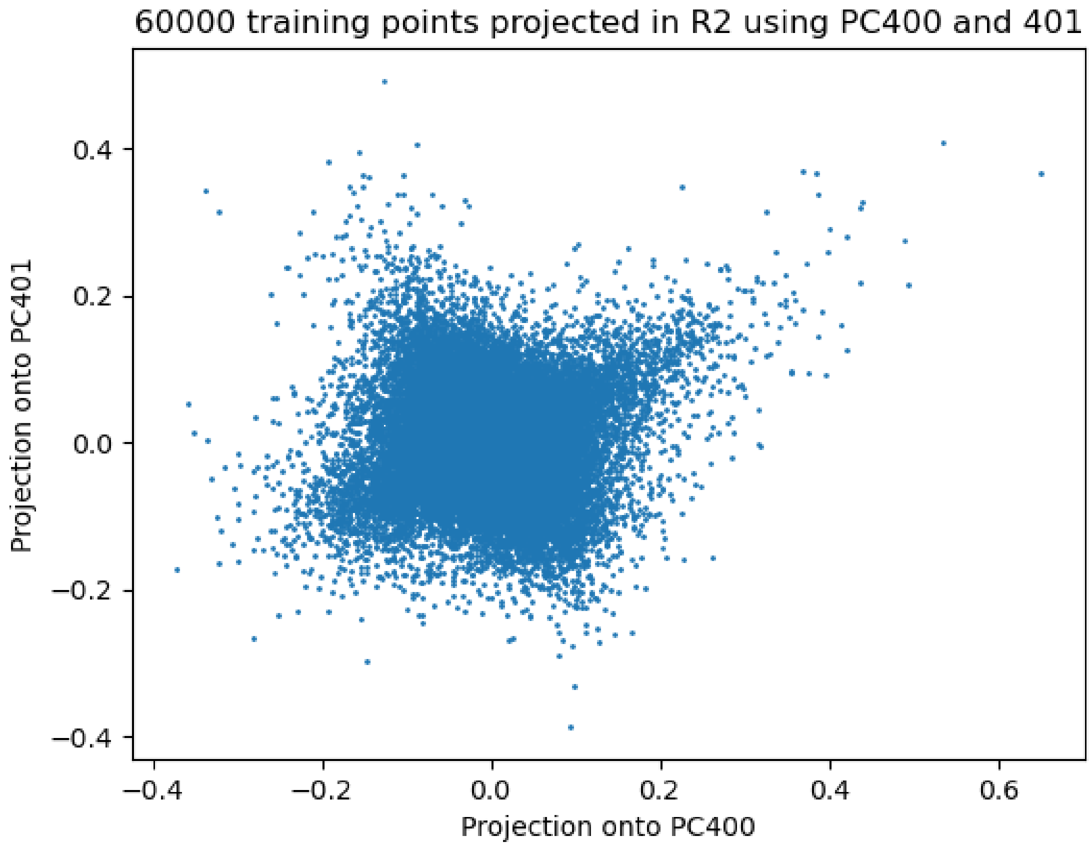
```
        train_times.append((end_train-start_train))
        test_times.append((end_test-start_test))
        accuracies.append(accuracy)
```

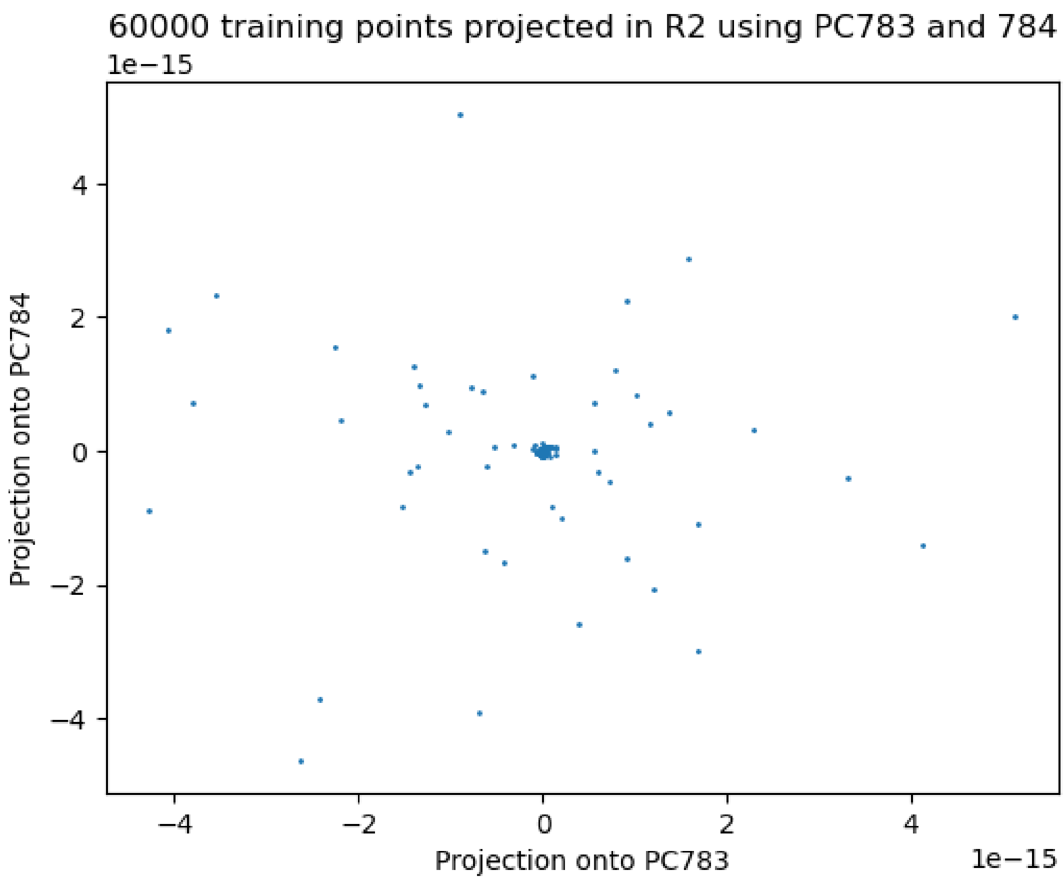




60000 training points projected in R2 using PC1 and 2
Projection onto PC2
Projection onto PC1
6
4
2
0
−2
−4
−4
−2
0
2
4
6
8


60000 training points projected in R2 using PC200 and 201
Projection onto PC201
Projection onto PC200
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6


60000 training points projected in R2 using PC400 and 401
Projection onto PC401
Projection onto PC400
0.4
0.2
0.0
−0.2
−0.4
−0.4
−0.2
0.0
0.2
0.4
0.6




```
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 2)      b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.40518    dev_acc:0.28230
Epoch   2/20    loss:0.39025    dev_acc:0.36500
Epoch   3/20    loss:0.37857    dev_acc:0.36240
Epoch   4/20    loss:0.34646    dev_acc:0.36240
Epoch   5/20    loss:0.34990    dev_acc:0.35980
Epoch   6/20    loss:0.36434    dev_acc:0.36000
Epoch   7/20    loss:0.37673    dev_acc:0.36330
Epoch   8/20    loss:0.36550    dev_acc:0.36310
Epoch   9/20    loss:0.38204    dev_acc:0.36320
Epoch  10/20    loss:0.38478    dev_acc:0.36730
Epoch  11/20    loss:0.35715    dev_acc:0.36590
Epoch  12/20    loss:0.35619    dev_acc:0.36600
Epoch  13/20    loss:0.36209    dev_acc:0.36910
Epoch  14/20    loss:0.37286    dev_acc:0.37020
```

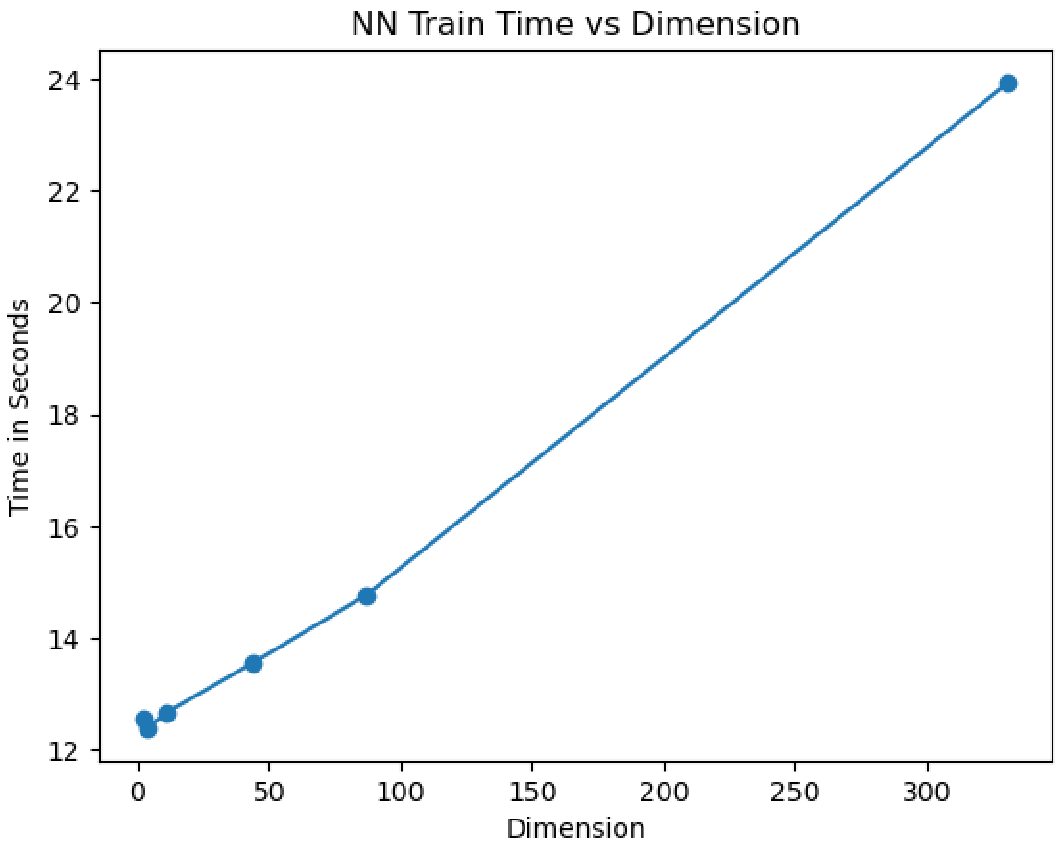
```
Epoch  15/20    loss:0.34633    dev_acc:0.37090
Epoch  16/20    loss:0.39030    dev_acc:0.37460
Epoch  17/20    loss:0.38398    dev_acc:0.38070
Epoch  18/20    loss:0.36882    dev_acc:0.38800
Epoch  19/20    loss:0.32137    dev_acc:0.38900
Epoch  20/20    loss:0.36206    dev_acc:0.39640
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 4)      b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.27992    dev_acc:0.56640
Epoch   2/20    loss:0.25188    dev_acc:0.60030
Epoch   3/20    loss:0.26181    dev_acc:0.60710
Epoch   4/20    loss:0.30816    dev_acc:0.60970
Epoch   5/20    loss:0.24993    dev_acc:0.60830
Epoch   6/20    loss:0.25189    dev_acc:0.61300
Epoch   7/20    loss:0.25538    dev_acc:0.61550
Epoch   8/20    loss:0.26183    dev_acc:0.61950
Epoch   9/20    loss:0.24325    dev_acc:0.62070
Epoch  10/20    loss:0.24099    dev_acc:0.62560
Epoch  11/20    loss:0.21912    dev_acc:0.62880
Epoch  12/20    loss:0.22752    dev_acc:0.63120
Epoch  13/20    loss:0.27346    dev_acc:0.63320
Epoch  14/20    loss:0.26013    dev_acc:0.63200
Epoch  15/20    loss:0.22241    dev_acc:0.62990
Epoch  16/20    loss:0.25441    dev_acc:0.63250
Epoch  17/20    loss:0.27335    dev_acc:0.63400
Epoch  18/20    loss:0.25775    dev_acc:0.63210
Epoch  19/20    loss:0.23246    dev_acc:0.63550
Epoch  20/20    loss:0.22184    dev_acc:0.63180
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 11)     b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.22560    dev_acc:0.78150
Epoch   2/20    loss:0.19151    dev_acc:0.82590
Epoch   3/20    loss:0.18807    dev_acc:0.84410
Epoch   4/20    loss:0.18349    dev_acc:0.85600
Epoch   5/20    loss:0.14607    dev_acc:0.86410
Epoch   6/20    loss:0.15925    dev_acc:0.87090
Epoch   7/20    loss:0.13038    dev_acc:0.87620
Epoch   8/20    loss:0.13943    dev_acc:0.87960
Epoch   9/20    loss:0.10670    dev_acc:0.88280
Epoch  10/20    loss:0.13092    dev_acc:0.88420
Epoch  11/20    loss:0.09497    dev_acc:0.88750
Epoch  12/20    loss:0.13118    dev_acc:0.88930
```

```
Epoch  13/20    loss:0.13066    dev_acc:0.89080
Epoch  14/20    loss:0.09913    dev_acc:0.89380
Epoch  15/20    loss:0.10485    dev_acc:0.89410
Epoch  16/20    loss:0.12371    dev_acc:0.89510
Epoch  17/20    loss:0.08048    dev_acc:0.89770
Epoch  18/20    loss:0.08336    dev_acc:0.89820
Epoch  19/20    loss:0.10241    dev_acc:0.90000
Epoch  20/20    loss:0.11362    dev_acc:0.90060
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 44)     b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.29112    dev_acc:0.80500
Epoch   2/20    loss:0.19906    dev_acc:0.87040
Epoch   3/20    loss:0.17111    dev_acc:0.89620
Epoch   4/20    loss:0.13935    dev_acc:0.90500
Epoch   5/20    loss:0.14651    dev_acc:0.91310
Epoch   6/20    loss:0.13677    dev_acc:0.91940
Epoch   7/20    loss:0.13632    dev_acc:0.92180
Epoch   8/20    loss:0.10831    dev_acc:0.92500
Epoch   9/20    loss:0.10881    dev_acc:0.92880
Epoch  10/20    loss:0.10649    dev_acc:0.93010
Epoch  11/20    loss:0.12163    dev_acc:0.93330
Epoch  12/20    loss:0.10654    dev_acc:0.93440
Epoch  13/20    loss:0.09359    dev_acc:0.93370
Epoch  14/20    loss:0.10720    dev_acc:0.93750
Epoch  15/20    loss:0.11755    dev_acc:0.93800
Epoch  16/20    loss:0.10768    dev_acc:0.93820
Epoch  17/20    loss:0.09176    dev_acc:0.94210
Epoch  18/20    loss:0.08523    dev_acc:0.94240
Epoch  19/20    loss:0.11427    dev_acc:0.94190
Epoch  20/20    loss:0.10601    dev_acc:0.94330
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 87)     b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.33521    dev_acc:0.78410
Epoch   2/20    loss:0.24325    dev_acc:0.85180
Epoch   3/20    loss:0.18809    dev_acc:0.87950
Epoch   4/20    loss:0.18775    dev_acc:0.89250
Epoch   5/20    loss:0.15293    dev_acc:0.90280
Epoch   6/20    loss:0.14781    dev_acc:0.90850
Epoch   7/20    loss:0.14043    dev_acc:0.91400
Epoch   8/20    loss:0.12751    dev_acc:0.91770
Epoch   9/20    loss:0.13242    dev_acc:0.92180
Epoch  10/20    loss:0.11663    dev_acc:0.92260
```

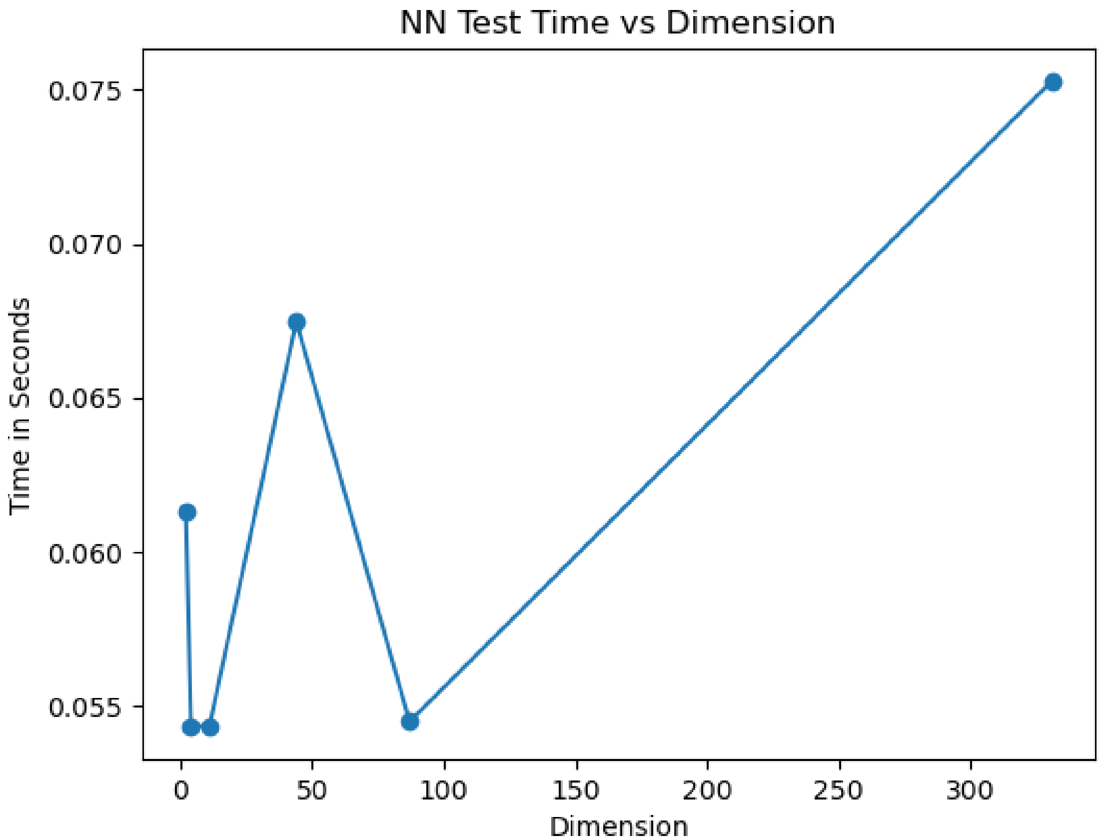
```
Epoch  11/20    loss:0.12600    dev_acc:0.92600
Epoch  12/20    loss:0.12722    dev_acc:0.92750
Epoch  13/20    loss:0.10065    dev_acc:0.92950
Epoch  14/20    loss:0.10828    dev_acc:0.93130
Epoch  15/20    loss:0.11343    dev_acc:0.93160
Epoch  16/20    loss:0.10297    dev_acc:0.93390
Epoch  17/20    loss:0.09474    dev_acc:0.93500
Epoch  18/20    loss:0.08703    dev_acc:0.93570
Epoch  19/20    loss:0.09442    dev_acc:0.93750
Epoch  20/20    loss:0.11349    dev_acc:0.93730
activation:Relu
loss function:SquaredLoss
Layer 1 w:(256, 331)    b:(256, 1)
Layer 2 w:(256, 256)    b:(256, 1)
Layer 3 w:(10, 256)     b:(10, 1)
Epoch   1/20    loss:0.30746    dev_acc:0.77380
Epoch   2/20    loss:0.23638    dev_acc:0.84390
Epoch   3/20    loss:0.20567    dev_acc:0.87370
Epoch   4/20    loss:0.18041    dev_acc:0.88940
Epoch   5/20    loss:0.14756    dev_acc:0.89770
Epoch   6/20    loss:0.13821    dev_acc:0.90500
Epoch   7/20    loss:0.16554    dev_acc:0.91130
Epoch   8/20    loss:0.12874    dev_acc:0.91420
Epoch   9/20    loss:0.13002    dev_acc:0.91890
Epoch  10/20    loss:0.12151    dev_acc:0.92210
Epoch  11/20    loss:0.12179    dev_acc:0.92430
Epoch  12/20    loss:0.13988    dev_acc:0.92660
Epoch  13/20    loss:0.15924    dev_acc:0.93020
Epoch  14/20    loss:0.12311    dev_acc:0.93200
Epoch  15/20    loss:0.12763    dev_acc:0.93380
Epoch  16/20    loss:0.11384    dev_acc:0.93500
Epoch  17/20    loss:0.08714    dev_acc:0.93570
Epoch  18/20    loss:0.10214    dev_acc:0.93710
Epoch  19/20    loss:0.09986    dev_acc:0.93830
Epoch  20/20    loss:0.09620    dev_acc:0.93930
```

[57]:

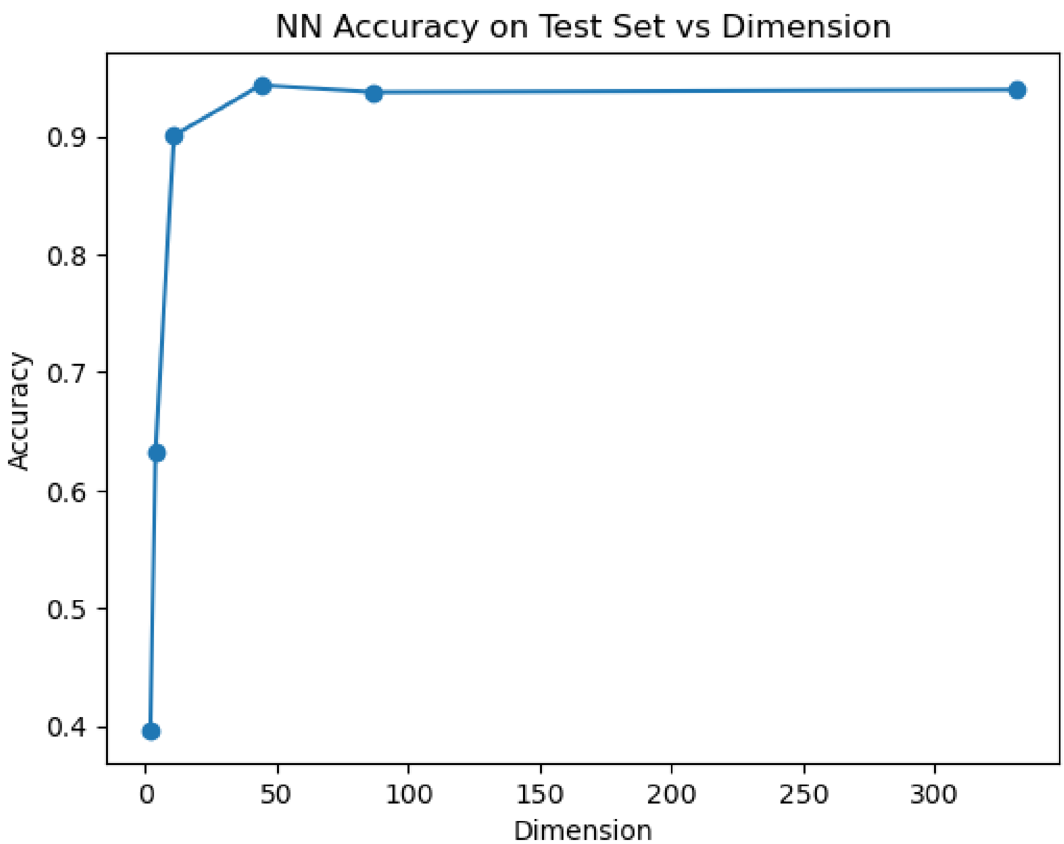
```
dimensions = [x_train.shape[0] for x_train in dr_xtrain]
plot(dimensions, train_times, marker='o')
title("NN Train Time vs Dimension")
xlabel("Dimension")
ylabel("Time in Seconds")
show()
```



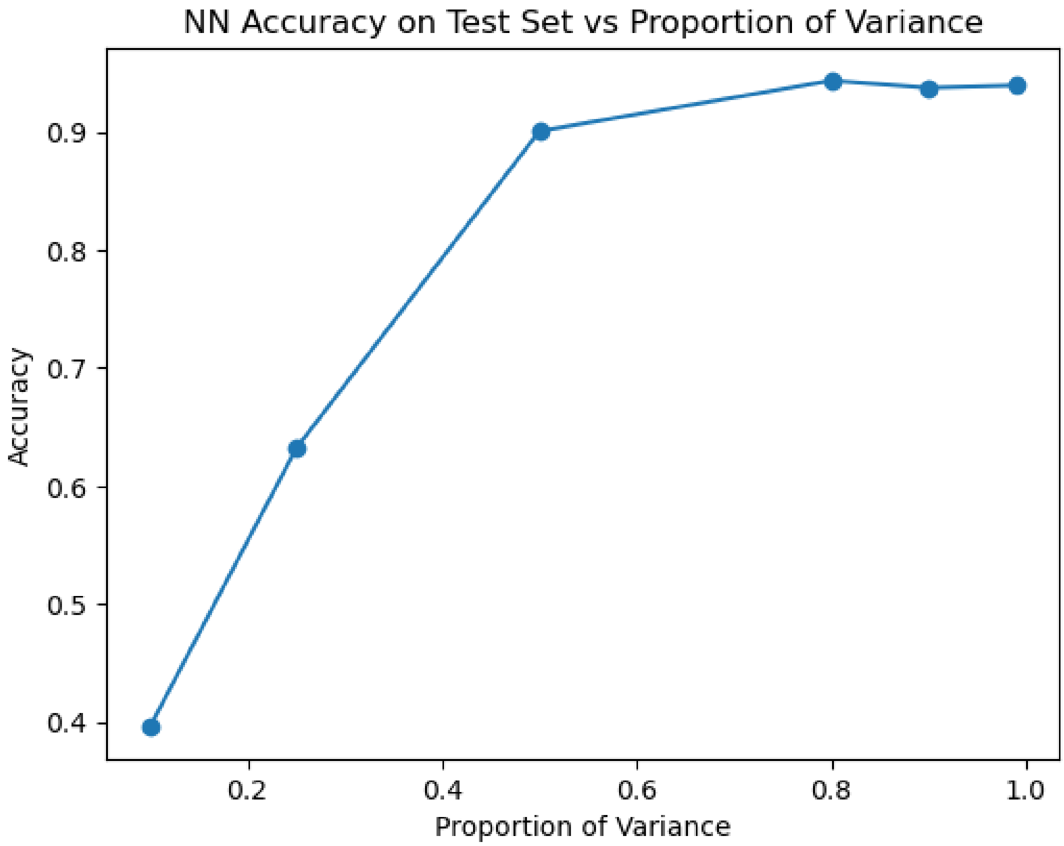
[58]:
```
plot(dimensions, test_times, marker='o')
title("NN Test Time vs Dimension")
xlabel("Dimension")
ylabel("Time in Seconds")
show()
```



[59]:
```
plot(dimensions, accuracies, marker='o')
title("NN Accuracy on Test Set vs Dimension")
xlabel("Dimension")
ylabel("Accuracy")
show()
```



[60]:
```
plot(proportions, accuracies, marker='o')
title("NN Accuracy on Test Set vs Proportion of Variance")
xlabel("Proportion of Variance")
ylabel("Accuracy")
show()
```



10 Qnn2.1 Explain what you did and what you found. Comment the code so that it is easy to follow. Support your results with plots and numbers. Provide the implementation so we can replicate your results.

I reduced the dimensionality of the mnist data set using the sklearn PCA library. I first ran the PCA on the training data to find all 784 principal components. Then I took the first principal components that made up 10%, 25%, 50%, 80%, 90%, and 99% of the variance of the training data. I used a NN with a relu activation function, 2 hidden layers, both with 256 hidden units (the same setting as in run_nn.py). I then provided the neural network the reduced training data as well as the reduced test data (the test data was projected/embedded into the same space/dimension using the training data's principle components) and computed the accuracies and times. Note that we could not use the test data's principle components since that would project it into a different space than the embedded training data, nor could we use the principle components of a combined train + test data set as that would be cheating since we would be using the test set to gain information (principle components).

The results show that lower dimensional data takes less time to train. This makes sense because it means that the first layer has less computations during forward propagation as your input layer matrix is smaller, and similarly during back propagation less gradients need to be calculated. This ends up shaving off a lot of time because training is an iterative process (you do forward and back propagation repeatedly over many batches), so saving a little time many times over accumulates. However, lower dimension data doesn't really improve test times which makes sense because making a prediction is a one time process and the only speed up you could get is from the original input layer which has a smaller weight matrix. However, this improvement is negligable due to modern hardware and it's why there is no apparent trend between dimension and prediction time.

You can also note, that as dimensionality increases so does prediction accuracy. This makes sense because a greater dimension implies that more of the original training data is preserved (i.e. more features to work with). You can also see in the related variance vs accuracy graph, that as more variance is preserved in the projected data, the accuracy improves. Hence we can conclude that since training time and dimension has a direct relationship, and since dimension and accuracy has a direct relationship: training time and accuracy have a direct relationship. So the trade off is, more dimensions costs more training time, but produces better accuracies. The numbers prove it as well as ~24 seconds of train time resulted in 94% accuracy, whereas ~12 seconds of train time resulted in 30% accuracy. However there is a catch. Clearly, test accuracy cannot keep linearly increasing as dimensions increases, otherwise you could get 100% accuracy on every data set (by just expanding # of features). So, even though test accuracy and dimensions are directly related, test accuracy levels off eventually. This also means, that even if you train the model longer, eventually the improvement in test accuracy will level off. This can be seen in the results as well where ~14, ~15, ~24 seconds of training time all resulted in a model with ~94% accuracy.

You can also see that for the mnist data set, you don't need a lot of dimensions (relative to the total 784) to get a good accuracy. This makes sense as some of the pixels like the ones on the border may have 0 impact on the actual number written. This might explain why ~50 dimensions got a slightly better test accuracy than ~100 or ~350 dimensions did (since there's less noise to look at). However, you do need enough dimensions to represent a majority of the data's variance (at least 50%) to get good accuracies (+90% on test set). Hence, we can see that proportion of variance is generally a better way to select the number of principal components, rather than picking an arbitrary fraction of the total number of features.