Assignment 4: PCA

Due: Thursday, October 24 at midnight

In this assignment you'll gain some hands-on experience with principal components analysis (PCA).

The assignment has three problems. The first problem investigates PCA and linear regression on a simple toy data set. The second two problems use the MNIST and Fashion MNIST data, and the database of faces that we began looking at during last week's lectures. In the second problem, you will study how different numbers of principal components represent the images visually. For third problem you will use logistic regression to predict the class label of images using the principal components representation of the images, and examine how the classification error changes with the number of principal components used.

For the second two problems, once you get your code to work on MNIST, it should be straightforward to just copy paste the code and then run it on Fashion MNIST and the face data.

Please submit your notebook and pdf (from html) following the usual instructions.

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import os, gzip
import warnings
```

Problem 1: Principal components and Least squares

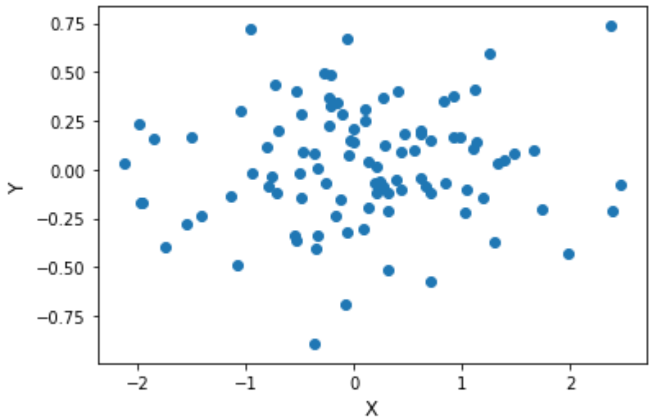
In least-squares regression one of the assumptions made is that the explanatory variable(s) are non-random and contain no measurement error. Therefore, the size of the residuals (vertical distances between each observed values of the response variable and the line) completely characterize the loss due to a given line. However, it is often the case that explanatory variables do have some randomness in them, in which case we may wish to characterize the loss with the orthogonal distances between data points and the line. This can be done with what is called Principal Component Regression, which you will have some time to use in this problem.

Part (a)

The cell below simulates two indepndent random variables, each from a Normal distribution with mean 0. It then rotates the data by an angle $\frac{\pi}{3}$. What is the slope and intercept of a horizontal line after it has been rotated about the origin by $\frac{\pi}{3}$ radians? Add a line with this slope and intercept to the plot generated in the following cell.

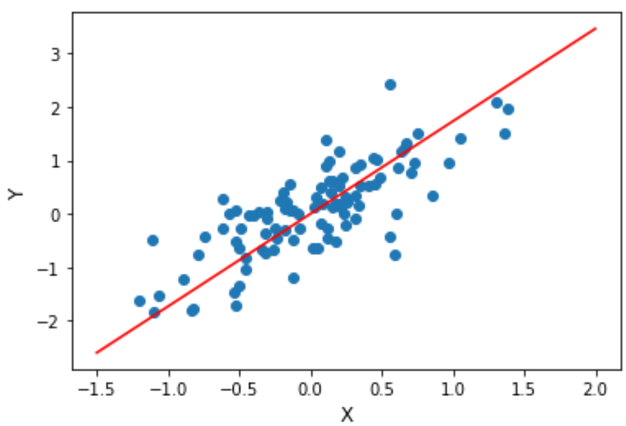
In [2]:
```
np.random.seed(10)
X = np.vstack((np.random.normal(0, 1, size=100), np.random.normal(0, 0.3, size=10
0))).T
```

In [3]:
```
plt.scatter(np.array(X[:,0]), np.array(X[:,1]))
plt.xlabel("X", fontsize=12)
plt.ylabel("Y", fontsize=12)
plt.show()
```



In [4]:
```
# rotate data
theta = np.pi/3
R = np.array([np.cos(theta), np.sin(theta), -np.sin(theta), np.cos(theta)]).reshap
e(2,2)
X = np.dot(X, R)
```

In [5]:
```
# plot data points
plt.scatter(np.array(X[:,0]), np.array(X[:,1]))
plt.xlabel("X", fontsize=12)
plt.ylabel("Y", fontsize=12)

# your code here
slope = np.tan((np.pi / 3))
line_x = np.linspace(-1.5, 2, 1000)
line_y = slope * line_x
plt.plot(line_x, line_y, "r")
plt.show()
```



$\pi/3$ radians is 60 degrees. tan(60) = 1.732 which is the slope of the gradient. Intercept is 0 because the line is rotated about the origin.

Part (b)

Use least-squares regression to fit a line (with a slope and intercept) to the data generated above. Create a plot that displays the data, the true line, and the least-squares regression line. Be sure to label the two lines with legends in your plot!

You could use `statsmodels.api.OLS` to fit the "ordinary least-squares" regression, or any other function of your choice.

In [6]:
```
from sklearn.linear_model import LinearRegression
```

In [7]:
```
X_train = np.array(X[:,0]).reshape(-1,1)
y_train = np.array(X[:,1])
```

In [8]:
```
assert(len(y_train) == len(X_train))
```

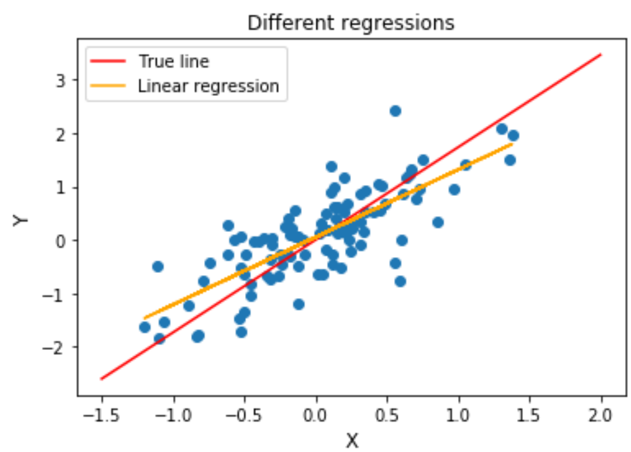
In [9]:
```
regr = LinearRegression()
regr.fit(X_train, y_train)
```

Out[9]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

In [10]:
```
regr.coef_
```

Out[10]:
```
array([1.26335002])
```

In [11]:
```
# plot data points
plt.scatter(np.array(X[:,0]), np.array(X[:,1]))
plt.xlabel("X", fontsize=12)
plt.ylabel("Y", fontsize=12)
plt.title("Different regressions")

# true line
slope = np.tan((np.pi / 3))
line_x = np.linspace(-1.5, 2, 1000)
line_y = slope * line_x
plt.plot(line_x, line_y, "r", label="True line")

# linear regression
plt.plot(X_train, regr.predict(X_train), "orange", label="Linear regression")

# show final plot
plt.legend()
plt.show()
```



Part (c)

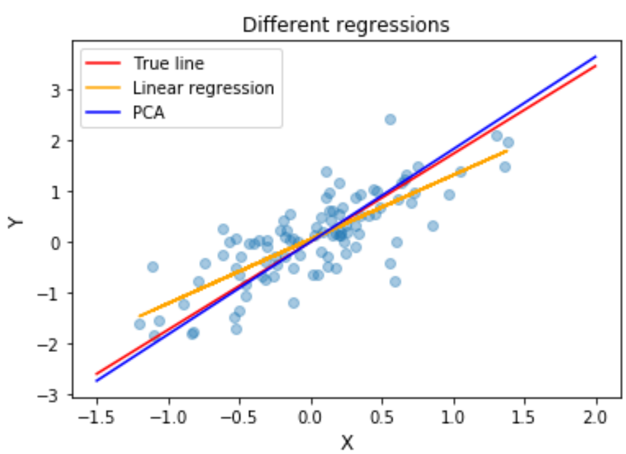
Now fit a line to the data by projecting onto the first principal component. What is the slope of the line created by the first principal component, and how does it relate to the true slope? Create a plot with all three lines, including those you constructed in parts (a) and (b).

In [12]:
```
from sklearn.decomposition import PCA
```

In [13]:
```
pca = PCA(n_components=1)
X_scaled = X - np.mean(X, 0)
pcs = pca.fit_transform(X_scaled) # the principal components
principal_vectors = pca.components_ # the principal vectors
```

In [14]:
```
pv1 = principal_vectors[0]
```

In [15]:
```
# plot data points
plt.scatter(np.array(X[:,0]), np.array(X[:,1]), alpha=0.4)
plt.xlabel("X", fontsize=12)
plt.ylabel("Y", fontsize=12)
plt.title("Different regressions")

# true line
slope = np.tan((np.pi / 3))
line_x = np.linspace(-1.5, 2, 1000)
line_y = slope * line_x
plt.plot(line_x, line_y, "r", label="True line")

# linear regression
plt.plot(X_train, regr.predict(X_train), "orange", label="Linear regression")

# plot projection of data onto first component
# pca_x = np.linspace()
# X_new = pca.inverse_transform(pcs)
pca_x = np.linspace(-1.5, 2, 1000)
pca_slope = pv1[1] / pv1[0]
pca_y = pca_slope * pca_x
plt.plot(pca_x, pca_y, "blue", label="PCA")

# show final plot
plt.legend()
plt.show()
```



- The PCA regression most closely models the true fit line (although the slope is slightly higher).
- The Linear regression has a considerably less steep (smaller) slope.

Part (d)

Explain why least-squares regression and principal components analysis give different fits to the data in part (c)? Can you say that one fit is better than the other?

- There was collinearity between X and Y in the initial, rotated dataset, which might explain poor fit of Linear Regression.
- PCA avoids the problems with multicollinearity because it produces a combination of variables/components that are uncorrelated/orthogonal.
- PCA fits better than Linear Regression.

Problems 2 and 3: MNIST and Fashion MNIST data

For the next two problems you will need the MNIST and Fashion MNIST data. You can download these data sets here:

MNIST [http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)](http://yann.lecun.com/exdb/mnist/)
FASHION-MNIST [https://github.com/zalandoresearch/fashion-mnist/tree/master/data/fashion (https://github.com/zalandoresearch/fashion-mnist/tree/master/data/fashion)](https://github.com/zalandoresearch/fashion-mnist/tree/master/data/fashion)

Download the following files:
train-images-idx3-ubyte.gz
train-labels-idx1-ubyte.gz
t10k-images-idx3-ubyte.gz
t10k-labels-idx1-ubyte.gz

To run the code, put the data in directories named `mnist` and `fashion-mnist` within the same directory as this notebook.

The following "helper functions" should be used to read in the MNIST and Fashion MNIST datasets.

In [16]:
```
import matplotlib.pyplot as plt
import numpy as np
import os, gzip

def load_data(dataset_name):
    # I CHANGED THIS BECAUSE I HAVE DATASETS IN A CENTRAL DIR
    # TO AVOID DUPLICATION
    data_dir = os.path.join("../../datasets/", dataset_name)

    def extract_data(filename, num_data, head_size, data_size):
        with gzip.open(filename) as bytestream:
            bytestream.read(head_size)
            buf = bytestream.read(data_size * num_data)
            data = np.frombuffer(buf, dtype=np.uint8).astype(np.float)
        return data

    data = extract_data(data_dir + '/train-images-idx3-ubyte.gz', 60000, 16, 28 *
28)
    trX = data.reshape((60000, 28, 28))

    data = extract_data(data_dir + '/train-labels-idx1-ubyte.gz', 60000, 8, 1)
    trY = data.reshape((60000))

    data = extract_data(data_dir + '/t10k-images-idx3-ubyte.gz', 10000, 16, 28 * 2
8)
    teX = data.reshape((10000, 28, 28))

    data = extract_data(data_dir + '/t10k-labels-idx1-ubyte.gz', 10000, 8, 1)
    teY = data.reshape((10000))

    trY = np.asarray(trY)
    teY = np.asarray(teY)

    X = np.concatenate((trX, teX), axis=0)
    y = np.concatenate((trY, teY), axis=0).astype(np.int)

    seed = 409
    np.random.seed(seed)
    np.random.shuffle(X)
    np.random.seed(seed)
    np.random.shuffle(y)
    return X / 255., y
```

Problem 2: PCA for Dimension Reduction

In this problem you will approximately reconstruct images by simplifying them to multiples of a few principal components.

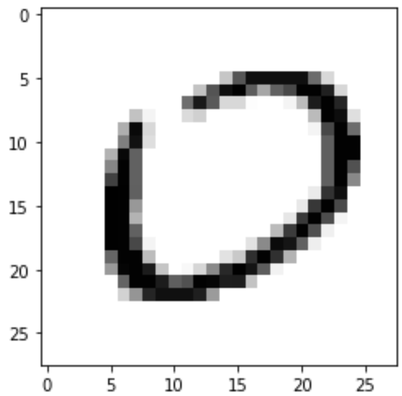
Note: When you display the images, use the color map `cmap=plt.cm.gray.reversed()` for MNIST and Fashion MNIST and use `cmap=plt.cm.gray` for the face data

Part (a)

Pick a random seed in the next cell to select a random image of a handwritten 0 from the MNIST data.

In [17]:
```
x, y = load_data('mnist')
x = x.reshape([70000, 28*28])
zeros = np.where(y==0)[0]
x = x[zeros,:] # 6903 * 784
y = y[zeros] # 6903 * 1 (all zeroes)
np.random.seed(13) # put your seed here
my_image = np.random.randint(0, len(y), size=1)

plt.imshow(x[my_image,:].reshape((28,28)), cmap=plt.cm.gray.reversed())
```

Out[17]: <matplotlib.image.AxesImage at 0x119b6a250>



In [18]:
```
x.shape
```

Out[18]: (6903, 784)

For $k = 0, 10, 20, \ldots, 100$, use k principal components for MNIST 0's to approximately reconstruct the image selected above. Display the reconstruction for each value of k. To display the set of images compactly, you may want to use subplot, as shown in the starter code for Problem 3(b) below.

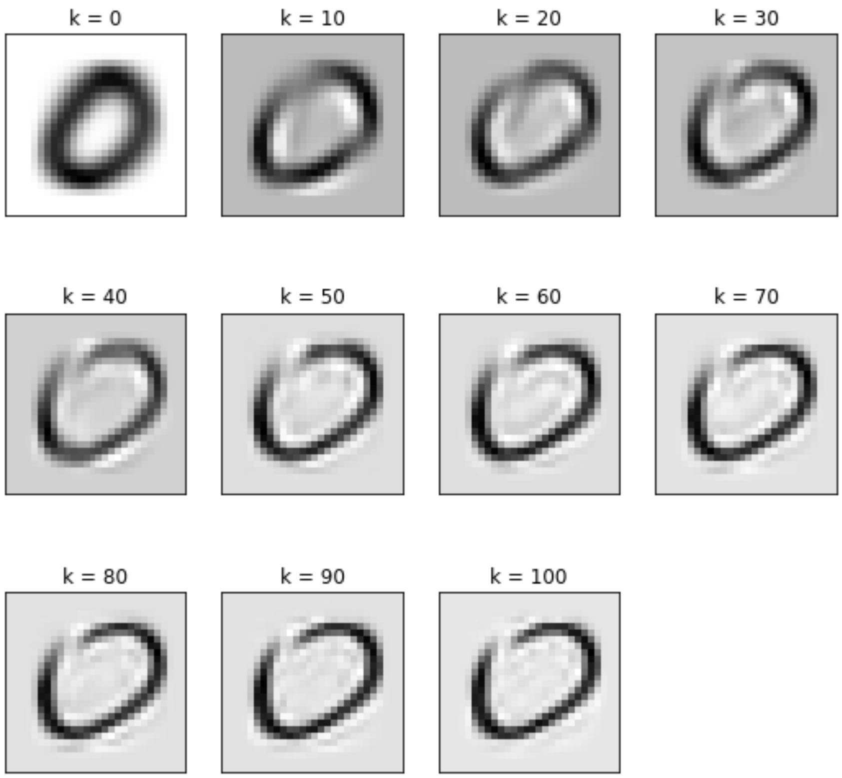
In [19]:
```
# plt.imshow(reconstructed[my_image].reshape((28,28)), cmap=plt.cm.gray.reversed
())
# plt.imshow(reconstructed[my_image, :].reshape((28,28)), cmap=plt.cm.gray.reverse
d())
# the two statements are equivalent. why?
```

In [20]:
```
def plot_images(_images, _titles, n_row=3, n_col=4):
    plt.figure(figsize=(1.8 * n_col, 2.4 * n_row))
    plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
    for i in range(n_row * n_col):
        if i < len(_images):
            plt.subplot(n_row, n_col, i + 1)
            plt.imshow(_images[i], cmap=plt.cm.gray.reversed())
            plt.title(f"k = {_titles[i]}")
            plt.xticks(())
            plt.yticks(())
```

In [21]:
```
k_values = np.arange(0, 110, 10)
reconstructed_samples = []
```

In [22]:
```
for k in k_values:
    _pca = PCA(n_components=k)
    _pcs = _pca.fit_transform(x)
    _reconstructed = _pca.inverse_transform(_pcs)
    _sample_img = _reconstructed[my_image].reshape((28,28))
    reconstructed_samples.append(_sample_img)
```

In [23]:
```
plot_images(reconstructed_samples, k_values)
```



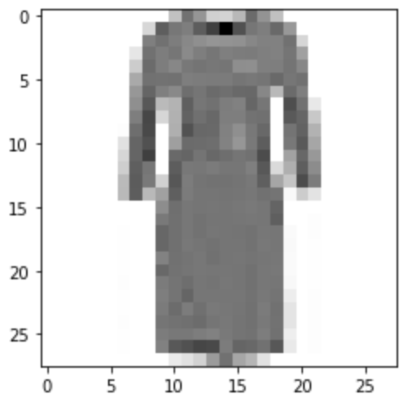
- k == 0 is just the average of all images, reconstructed to that particular image
- Using more components in the reconstruction captures more variance, which introduces noise

Part (b)

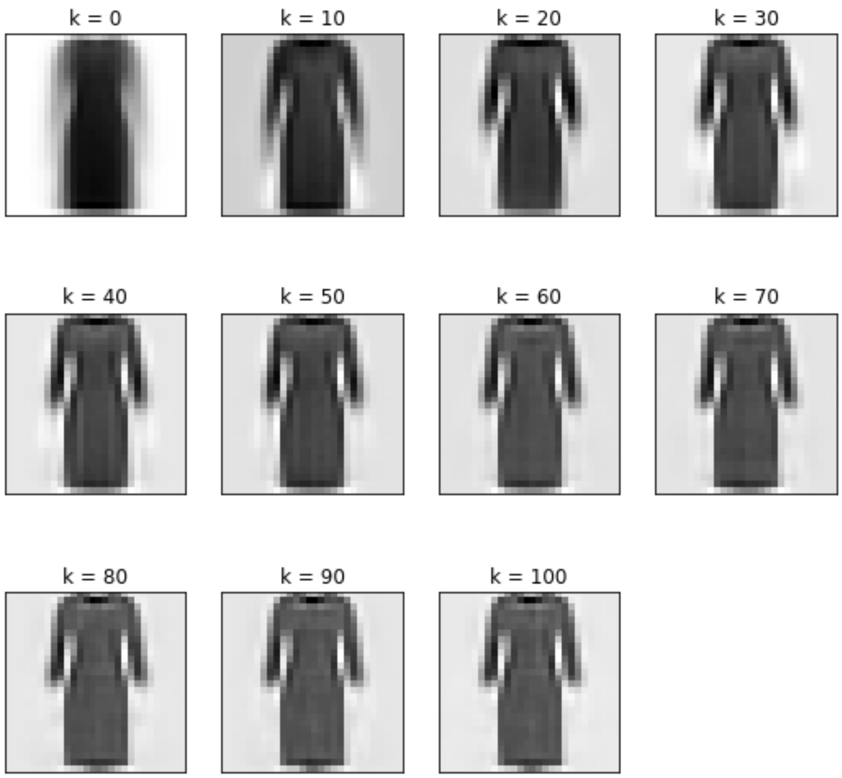
Repeat Part (a), but this time for the dresses in the Fashion-MNIST dataset.

In [24]:
```
x, y = load_data('fashion-mnist')
x = x.reshape([70000, 28*28])
zeros = np.where(y==3)[0]
x = x[zeros,:]
y = y[zeros]
np.random.seed(29) #put your seed here
my_image = np.random.randint(0, len(y), size=1)

plt.imshow(x[my_image,:].reshape((28,28)), cmap=plt.cm.gray.reversed())
```

Out[24]: <matplotlib.image.AxesImage at 0x1179c9b90>



In [25]:
```
k_values = np.arange(0, 110, 10)
reconstructed_samples = []
for k in k_values:
    _pca = PCA(n_components=k)
    _pcs = _pca.fit_transform(x)
    _reconstructed = _pca.inverse_transform(_pcs)
    _sample_img = _reconstructed[my_image].reshape((28,28))
    reconstructed_samples.append(_sample_img)
plot_images(reconstructed_samples, k_values)
```



- k == 0 is just the average of all images, reconstructed to that particular image
- Using more components in the reconstruction captures more variance, which introduces pixellation/noise

Part (c)

Do the same thing as in Parts (a) and (b), this time reconstructing an image of Gerhard Schroeder.

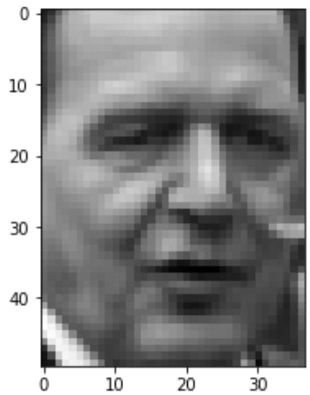
In [26]:
```
from sklearn.datasets import fetch_lfw_people
```

In [27]:
```
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
# X = lfw_people.data
# y = lfw_people.target
lfw_people.target_names
```

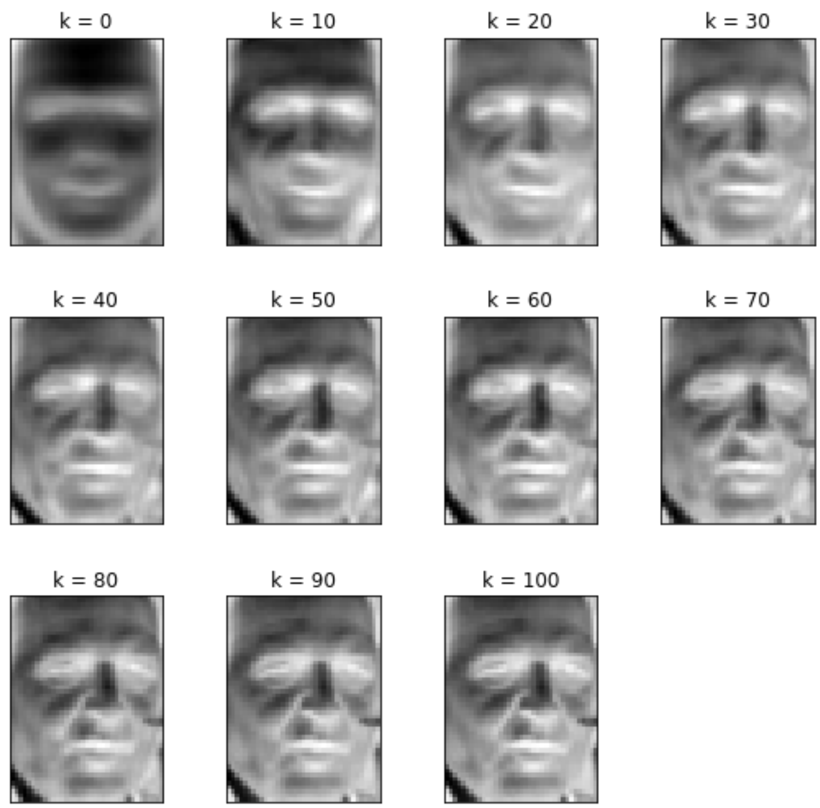
Out[27]:
```
array(['Ariel Sharon', 'Colin Powell', 'Donald Rumsfeld', 'George W Bush',
       'Gerhard Schroeder', 'Hugo Chavez', 'Tony Blair'], dtype='<U17')
```

In [28]:
```
x = lfw_people.data
y = lfw_people.target
zeros = np.where(y==4)[0]
x = x[zeros,:]
y = y[zeros]
np.random.seed(111) #put your seed here
my_image = np.random.randint(0, len(y), size=1)


plt.imshow(x[my_image,:].reshape((50,37)), cmap=plt.cm.gray)#, cmap=plt.cm.gray.re
versed())
```

Out[28]: <matplotlib.image.AxesImage at 0x11cd3d210>



In [29]:
```
k_values = np.arange(0, 110, 10)
reconstructed_samples = []
for k in k_values:
    _pca = PCA(n_components=k)
    _pcs = _pca.fit_transform(x)
    _reconstructed = _pca.inverse_transform(_pcs)
    _sample_img = _reconstructed[my_image].reshape((50,37))
    reconstructed_samples.append(_sample_img)
plot_images(reconstructed_samples, k_values)
```



- Here the differences between the reconstructions are less obvious, because the face is complex
- But notice a small line in the right edge and vertical center of the image, which becomes more prominent/amplified as the number of components increase

Problem 3: PCA for Classification

Part (a)

Load in the MNIST data with the labels as `y` and the images as `x` by running the next cell. Create a subset of the data by keeping only the images that have the label of either 4 or 9. Use Principal Components Analysis (PCA) to project the data onto the first two principal components, and create a plot of the projected data color-coded by the label. Does the plot make sense? Explain in a couple sentences.

create subsets

In [30]:
```
from sklearn.linear_model import LogisticRegression

x, y = load_data('mnist')
x = x.reshape([70000, 28*28]) # 70000 images
```

In [31]:
```
x.shape
```

Out[31]:
```
(70000, 784)
```

In [32]:
```
# create subset of data
mask = np.isin(y, [4,9])
```

In [33]:
```
# the number of rows with 4 and 9 as the label
y_subset = y[mask]
assert(len(y_subset) == 13782)
```

In [34]:
```
# apply mask to x to get the corresponding rows
x_subset = x[mask]
assert(len(x_subset) == 13782)
x_subset.shape
```

Out[34]:
```
(13782, 784)
```

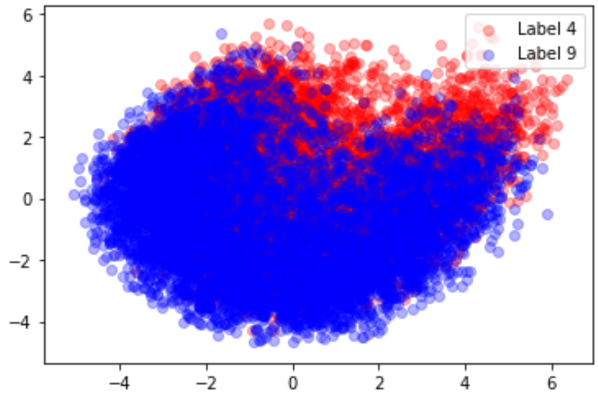
In [35]:
```
four_mask = np.isin(y_subset, 4)
nine_mask = np.isin(y_subset, 9)
```

use PCA

In [36]:
```
pca = PCA(n_components=2)
```

In [37]:
```
# do PCA
x_subset = x_subset - np.mean(x_subset) # center data
pcs = pca.fit_transform(x_subset) # project data onto first two components
pv = pca.components_ # the principal vectors / axes
```

In [38]:
```
# color coded plot
fig = plt.figure()
ax = fig.add_subplot(111)

labels = [4, 9]
labels_text = ["Label 4", "Label 9"]
colors = ['red', 'blue']
for i in np.arange(2):
    mask = y_subset==labels[i]
    ax.scatter(pcs[mask,0], pcs[mask,1], alpha=0.3, c=colors[i], label=labels_text
[i])

plt.legend(loc='upper right')
plt.show()
```



The plot makes sense because we see some separation between digits 4 and digits 9. However, there are some overlap in between, which indicates that 4's and 9's can be completely distinguished using the first two principal components.

Part (b)

Why not use more principal components! For $k = 2, 3, 4, \ldots, 15$, use PCA to project the data onto k principal components. For each k, use logistic regression to build a model to classify images as 4 or 9, and calculate the misclassification rate. Create a plot of misclassification rate as a function of k, the number of principal components used. Does the plot make sense? Explain.

In [39]:
```
from sklearn.metrics import accuracy_score
```

In [40]:
```
warnings.filterwarnings(action='once')
```

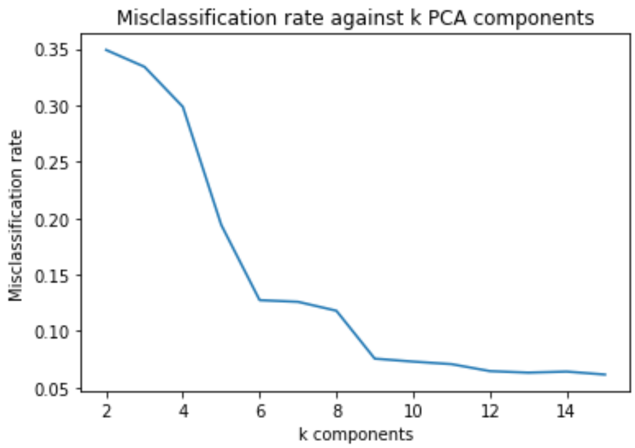
In [ ]:
```
misclass_rate = []
k_values = np.arange(2, 16, 1)
for k in k_values:
    # progress
    print(f"Working k = {k}...")

    # do PCA
    pca = PCA(n_components=k)
    pcs = pca.fit_transform(x_subset)
    pv = pca.components_

    # classify with logistic regression
    regr = LogisticRegression(random_state=42)
    regr.fit(pcs, y_subset)

    # LR is a linear model so it cannot overfit to every point
    # so i am not doing train test split
    y_pred = regr.predict(pcs)

    # add error
    misclass_rate.append(1 - accuracy_score(y_subset, y_pred))
```

In [42]:
```
misclass_rate
```

Out[42]:
```
[0.34907850819910025,
 0.3341314758380496,
 0.2987229719924539,
 0.1940937454650994,
 0.12741256711652882,
 0.12596139892613556,
 0.11805253228849222,
 0.07560586271948921,
 0.07299375997678126,
 0.07074444928167178,
 0.06457698447250038,
 0.06319837469162681,
 0.06414163401538242,
 0.061529531272674465]
```

In [43]:
```
plt.plot(k_values, misclass_rate)
plt.xlabel("k components")
plt.ylabel("Misclassification rate")
plt.title("Misclassification rate against k PCA components")
```

Out[43]:
```
Text(0.5, 1.0, 'Misclassification rate against k PCA components')
```



- The plot makes sense. As the number of PCA components increase, they explain more and more of the variance in the data
- As we near 100% variance explained, the Logistic Regression model is getting more overfit to the data, reducing misclassification rate.
- The biggest drop is from 4 to 6 components (from 0.35 to 0.13 approx).

Part (c)

Build a logistic regression model using 10 principal components. Create a list called `misclass` that lists the indices of all images that were misclassified with this model. Run the cell below to create a visualization of the first 16 of these images. Does it make sense that these would be hard to classify correctly?

In [44]:
```
pca = PCA(n_components=10)
pcs = pca.fit_transform(x_subset)

regr = LogisticRegression(random_state=42)
regr.fit(pcs, y_subset)
y_pred = regr.predict(pcs)
```

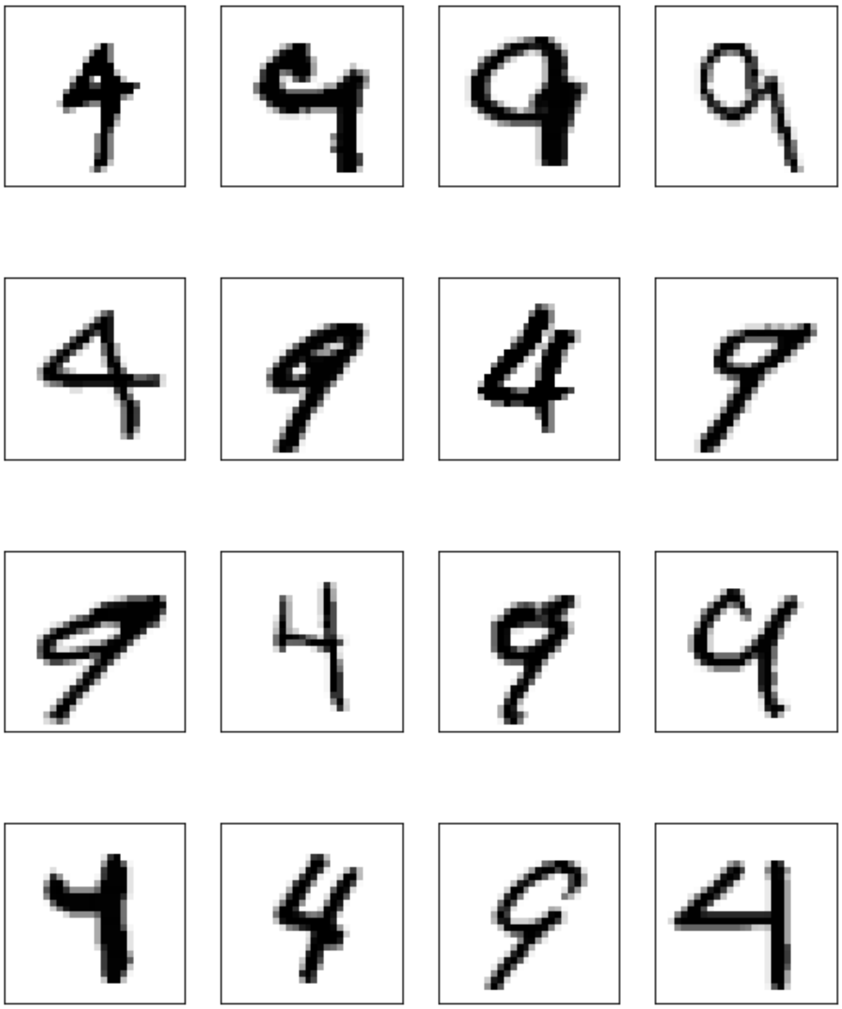
```
/Users/sarimabbas/Developer/virtualenv/py_general_venv/lib/python3.7/site-packag
es/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be c
hanged to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

In [45]:
```
indexes = np.where(y_pred != y_subset)[0]
first_sixteen = indexes[:16]
```

In [46]:
```
# Your Code Here, modify the line below
# misclass = np.zeros_like(y[keep])
misclass = indexes[:16]
```

In [47]:
```
# rebuild subset without the means
images = x[np.isin(y, [4,9])]
```

In [48]:
```
# The following code will display the images that are misclassified

plt.figure(figsize=(1.8 * 4, 2.4 * 4))
plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.imshow(images[misclass[i]].reshape((28, 28)), cmap=plt.cm.gray.reversed())
    plt.xticks(())
    plt.yticks(())
```



Many of these images look ambiguous, but some would be harder to classify correctly than others

- The 2nd image does not look like either a 9 or 4
- The 3rd image has a strange protrusion in the curve of the 9
- Many images have slants

The other images might similarly cause trouble for the classifier

Part (d)

Now use the Fashion-MNIST data and train logistic regression models to classify coats ($y = 4$) and handbags ($y = 8$). Again use $k = 2, 3, 4, \ldots, 15$ to project the data onto k principal components, and calculate the misclassification rate at each k. Create a plot of misclassification rate vs. k.

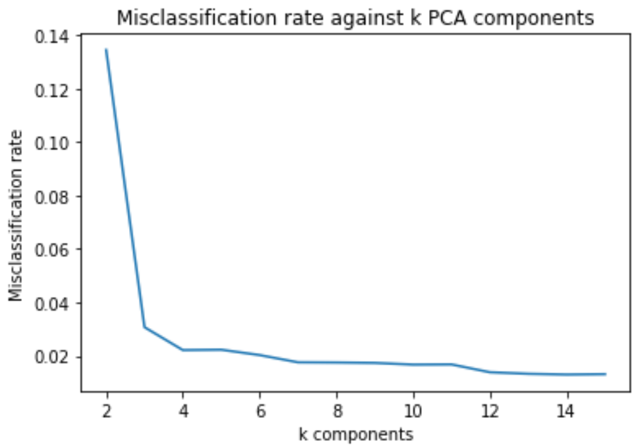
In [49]:
```
x, y = load_data('fashion-mnist')
fashion = x.reshape([70000, 28*28])
```

In [50]:
```
mask = np.isin(y, [4,8])
y_subset = y[mask]
x_subset = fashion[mask]
x_subset = x_subset - np.mean(x_subset)
```

In [ ]:
```
misclass_rate = []
k_values = np.arange(2, 16, 1)
for k in k_values:
    # progress
    print(f"Working k = {k}...")

    # do PCA
    pca = PCA(n_components=k)
    pcs = pca.fit_transform(x_subset)

    # classify with logistic regression
    regr = LogisticRegression(random_state=42)
    regr.fit(pcs, y_subset)

    # LR is a linear model so it cannot overfit to every point
    # so i am not doing train test split
    y_pred = regr.predict(pcs)

    # add error
    misclass_rate.append(1 - accuracy_score(y_subset, y_pred))
```

In [52]:
```
plt.plot(k_values, misclass_rate)
plt.xlabel("k components")
plt.ylabel("Misclassification rate")
plt.title("Misclassification rate against k PCA components")
```

Out[52]:
```
Text(0.5, 1.0, 'Misclassification rate against k PCA components')
```



- Misclassification rate drops much faster than with mnist dataset
- The biggest drop is from 2 to 3 components (from 0.14 to 0.03).
- In comparison, with MNIST the drop started and ended at a relatively higher error rate.

Part (e)

Follow the same procedure as in Parts (b) and (c) with the faces data instead of MNIST. This time, however, build a model that can classify whether an image is of George W. Bush ($y = 3$) someone else. Create a variable that takes the value 1 when the image is of George W. Bush and 0 when it is not, and use this to train each Logistic Regression model after projecting onto the first k principal components. Create a plot of misclassification rate as a function of k, letting k vary between 2 and 30. Then, show training examples that are misclassified for the $k = 30$ model.

In [53]:
```
from sklearn.datasets import fetch_lfw_people
lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)
x = lfw_people.data
y = lfw_people.target
```

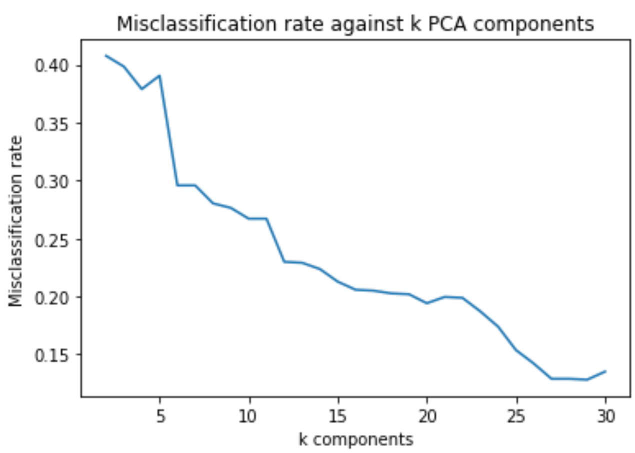
In [54]:
```
x_centered = x - np.mean(x)
```

In [55]:
```
z = np.array([1 if i == 3 else 0 for i in y])
```

In [ ]:
```
misclass_rate = []
k_values = np.arange(2, 31, 1)
for k in k_values:
    # progress
    print(f"Working k = {k}...")

    # do PCA
    pca = PCA(n_components=k)
    pcs = pca.fit_transform(x_centered)

    # classify with logistic regression
    regr = LogisticRegression(random_state=42)
    regr.fit(pcs, z)

    # LR is a linear model so it cannot overfit to every point
    # so i am not doing train test split
    y_pred = regr.predict(pcs)

    # add error
    misclass_rate.append(1 - accuracy_score(z, y_pred))
```

In [57]:
```
plt.plot(k_values, misclass_rate)
plt.xlabel("k components")
plt.ylabel("Misclassification rate")
plt.title("Misclassification rate against k PCA components")
```

Out[57]:
```
Text(0.5, 1.0, 'Misclassification rate against k PCA components')
```



- Here the drop in misclassification rate is more gradual than the other two plots

training examples for k=30 that were misclassified

In [58]:
```
pca = PCA(n_components=30)
pcs = pca.fit_transform(x_centered)

regr = LogisticRegression(random_state=42)
regr.fit(pcs, z)
y_pred = regr.predict(pcs)
```

```
/Users/sarimabbas/Developer/virtualenv/py_general_venv/lib/python3.7/site-packag
es/sklearn/linear_model/logistic.py:432: FutureWarning: Default solver will be c
hanged to 'lbfgs' in 0.22. Specify a solver to silence this warning.
  FutureWarning)
```

In [59]:
```
indexes = np.where(y_pred != z)[0]
misclass = indexes[0:16]
```

In [60]:
```
plt.figure(figsize=(1.8 * 4, 2.4 * 4))
plt.subplots_adjust(bottom=0, left=.01, right=.99, top=.90, hspace=.35)
for i in range(16):
    plt.subplot(4, 4, i + 1)
    plt.imshow(x[misclass[i]].reshape((50, 37)), cmap=plt.cm.gray.reversed())
    plt.xticks(())
    plt.yticks(())
```

- Perhaps these images were misclassified because the images were taken at an off-angle (e.g. profile), or the face is cut-off