In [1]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import itertools as it
from scipy.sparse import coo_matrix
%matplotlib inline

from lsq_code import remove_outlier, create_vandermonde, solve_linear_LS,
solve_linear_LS_gd, mnist_pairwise_LS

# Other possibly useful functions
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix
```

executed in 1.40s, finished 17:44:29 2019-11-18

Example 2.1

When $n = 1$, we can fit a degree-m polynomial by choosing $f_j(x) = x^{j-1}$ and $M = m + 1$. In this case, it follows that $A_{i,j} = x_i^{j-1}$ and the matrix A is called a Vandermonde matrix.

Write a function to create Vandermonde matrix **(5 pt)**

In [2]:

```
x = np.arange(1, 10)
create_vandermonde(x, 3)
```

executed in 10ms, finished 17:44:29 2019-11-18

Out[2]:

```
array([[  1.,   1.,   1.,   1.],
       [  1.,   2.,   4.,   8.],
       [  1.,   3.,   9.,  27.],
       [  1.,   4.,  16.,  64.],
       [  1.,   5.,  25., 125.],
       [  1.,   6.,  36., 216.],
       [  1.,   7.,  49., 343.],
       [  1.,   8.,  64., 512.],
       [  1.,   9.,  81., 729.]])
```

Exercise 2.2

Write a function to solve least-square problem via linear algebra **(5 pt)**

Implementation hint: check `numpy.linalg.lstsq`.

Using the setup in the previous example, try fitting the points $(1, 2), (2, 3), (3, 5), (4, 7), (5, 11), (6, 13)$ to a degree-2 polynomial.

Compute the minimum squared error. **(5 pt)**

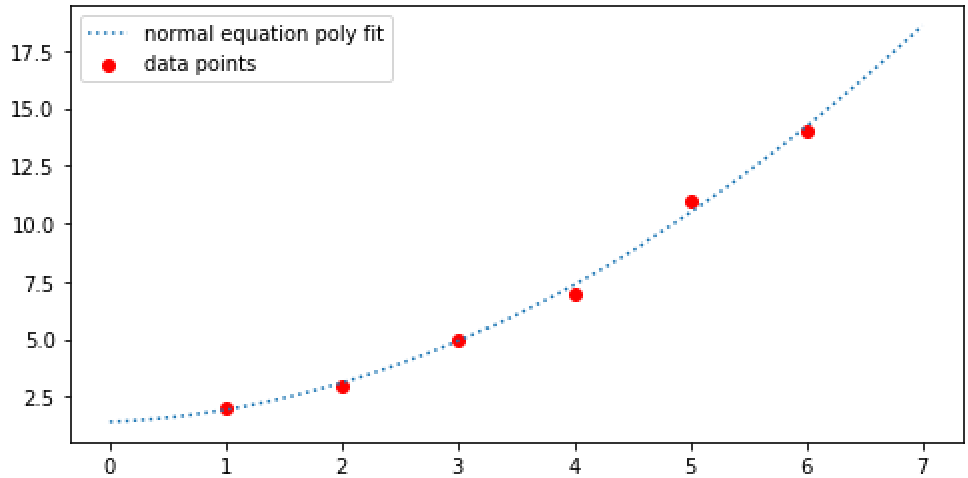
Plot this polynomial (for $x \in [0, 7]$) along with the data points to see the quality of fit. **(5 pt)**

In [3]:

```
x = np.array([1, 2, 3, 4, 5, 6])
y = np.array([2, 3, 5, 7, 11, 14])
m = 2

# Create Vandermonde matrix A
A = create_vandermonde(x,m)

# Use linear algebra to solve least-squares problem and minimize || y - A z ||^2
z_hat = solve_linear_LS(A, y)

# Compute the minimum square error
mse = np.linalg.norm(y-A.dot(z_hat.T))**2

# Generate x/y plot points for the fitted polynomial
xx = np.linspace(0, 7)
yy = create_vandermonde(xx,m).dot(z_hat.T).T

plt.figure(figsize=(8, 4))
plt.scatter(x, y, color='red', label='data points')
plt.plot(xx, yy, linestyle='dotted',label='normal equation poly fit')
plt.legend()

poly1_expr = ' + '.join(['{0:.4f} x^{1}'.format(v, i) for i, v in
enumerate(z_hat)][::-1])[:-4]
print('normal equation polynomial fit is {0}'.format(poly1_expr))
print('normal equation MSE is {0:.4f}'.format(mse))
```

executed in 228ms, finished 17:44:29 2019-11-18

```
normal equation polynomial fit is 0.3214 x^2 + 0.2071 x^1 + 1.4000
normal equation MSE is 0.4857
```



Exercise 2.3

Write a function to solve a least-squares problem via gradient descent. **(5 pt)**

Compute the minimum squared error. **(5 pt)**

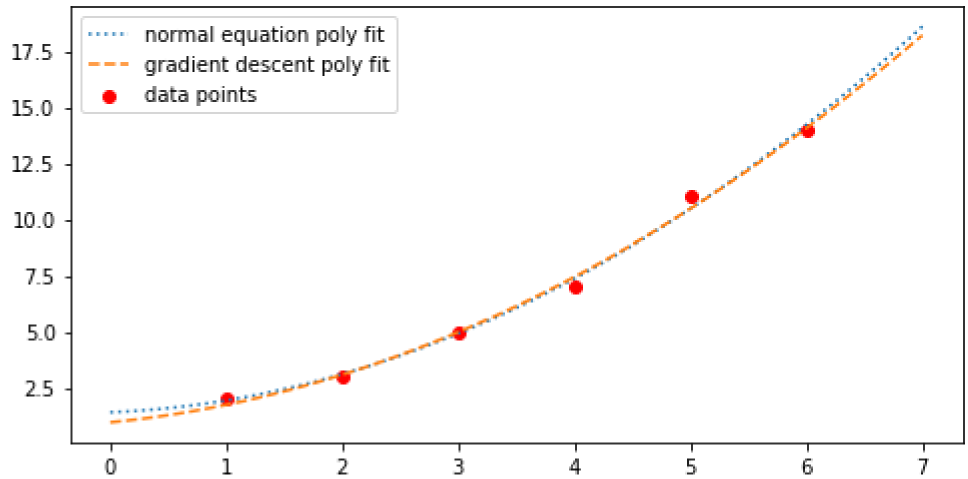
Plot the resulting polynomial (for $x \in [0, 7]$) along with previous polynomial and original data points to see the quality of fit. **(5 pt)**

In [4]:

```
# Use gradient descent to solve least-squares problem and minimize || y - A z2
||^2
z2_hat = solve_linear_LS_gd(A, y, 0.0002, 100000)

# Compute the minimum square error
print(y-A.dot(z2_hat.T))
mse2 = (np.linalg.norm(y-A.dot(z2_hat.T))**2)

# Generate y plot points for the gd fitted polynomial
yy2 = create_vandermonde(xx,m).dot(z2_hat.T).T

plt.figure(figsize=(8, 4))
plt.scatter(x, y, color='red', label='data points')
plt.plot(xx, yy, linestyle='dotted',label='normal equation poly fit')
plt.plot(xx, yy2, linestyle='dashed', label='gradient descent poly fit')
plt.legend()

poly2_expr = ' + '.join(['{0:.4f} x^{1}'.format(v, i) for i, v in
enumerate(z2_hat)][::-1])[:-4]
print('gradient descent polynomial fit is {0}'.format(poly2_expr))
print('gradient descent MSE is {0:.4f}'.format(mse2))
```

executed in 790ms, finished 17:44:30 2019-11-18

```
[ 0.26845223 -0.07394951  0.02198374 -0.44374802  0.52885521 -0.060206
57]
gradient descent polynomial fit is 0.2808 x^2 + 0.4999 x^1 + 0.9508
gradient descent MSE is 0.5582
```



MNIST

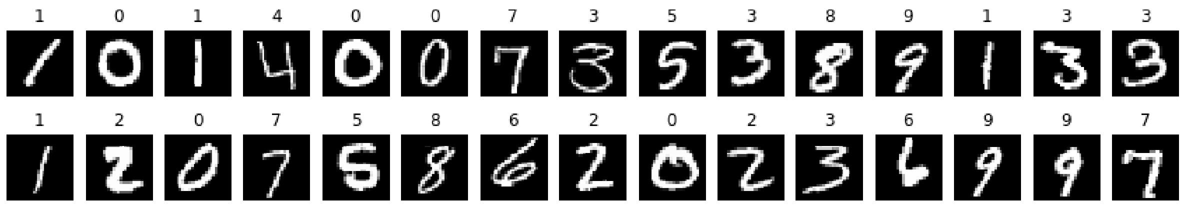
Read `mnist_train.csv` , create a dataframe with two columns, column `feature` contains all x and column `label` contains all y.

Plot the first 30 images.

In [5]:

```
# read mnist csv file to a dataframe
df = pd.read_csv('mnist_train.csv')
# append feature column by merging all pixel columns
df['feature'] = df.apply(lambda row: row.values[1:], axis=1)
# only keep feature and label column
df = df[['feature', 'label']]
# display first 5 rows of the dataframe
df.head()

# Plot the first 30 images
plt.figure(figsize=(15, 2.5))
for i, row in df.iloc[:30].iterrows():
    x, y = row['feature'], row['label']
    plt.subplot(2, 15, i + 1)
    plt.imshow(x.reshape(28, 28), cmap='gray')
    plt.axis('off')
    plt.title(y)
```

executed in 5.46s, finished 17:44:36 2019-11-18



Exercise 3.2

Write the function `extract_and_split` to extract the all samples labeled with digit n and randomly separate fraction of samples into training and testing groups. **(10 pt)**

Implementation hint: check `sklearn.model_selection.train_test_split` .

Pairwise experiment for applying least-square to classify digit a and digit b.

Follow the given steps in the template and implement the function for pairwise experiment **(15 pt)**

Possible implementation hint: check `sklearn.metrics.accuracy_score` , `sklearn.metrics.confusion_matrix`

In [6]:

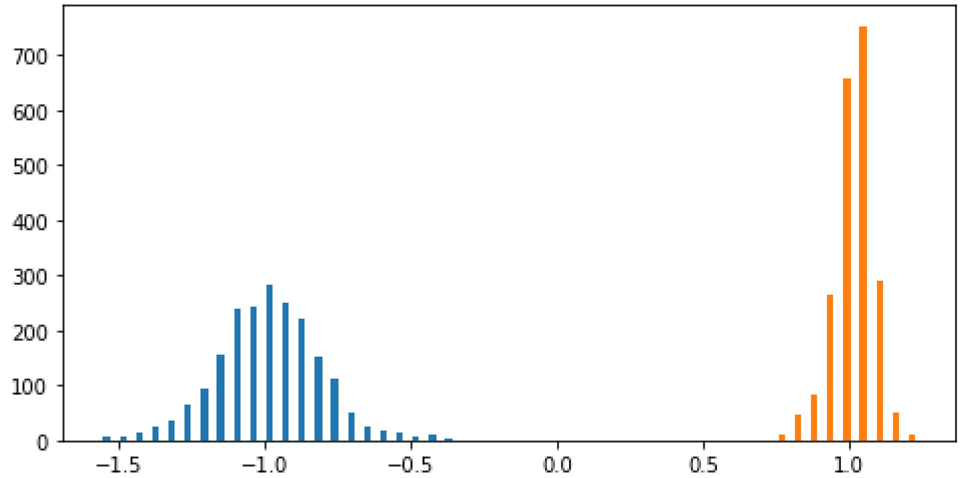
```
# Pairwise experiment for LSQ to classify between 0 and 1
mnist_pairwise_LS(df, 0, 1, verbose=True)
```

executed in 3.61s, finished 17:44:39 2019-11-18

```
Pairwise experiment, mapping 0 to -1, mapping 1 to 1
training error = 0.32%, testing error = 1.18%
Confusion matrix for the testing sets:
 [[2043   23]
 [  29 2313]]
Confusion matrix for the training sets:
 [[2062    4]
 [  10 2332]]
```

Out[6]:

```
array([0.00317604, 0.01179673])
```



Exercise 3.3

Repeat the above problem for all pairs of digits. For each pair of digits, report the classification error rates for the training and testing sets. The error rates can be formatted nicely into a triangular matrix. Put testing error in the lower triangle and training error in the upper triangle.

The code is given here in order demonstrate tqdm. Points awarded for reasonable values **(10 pt)**

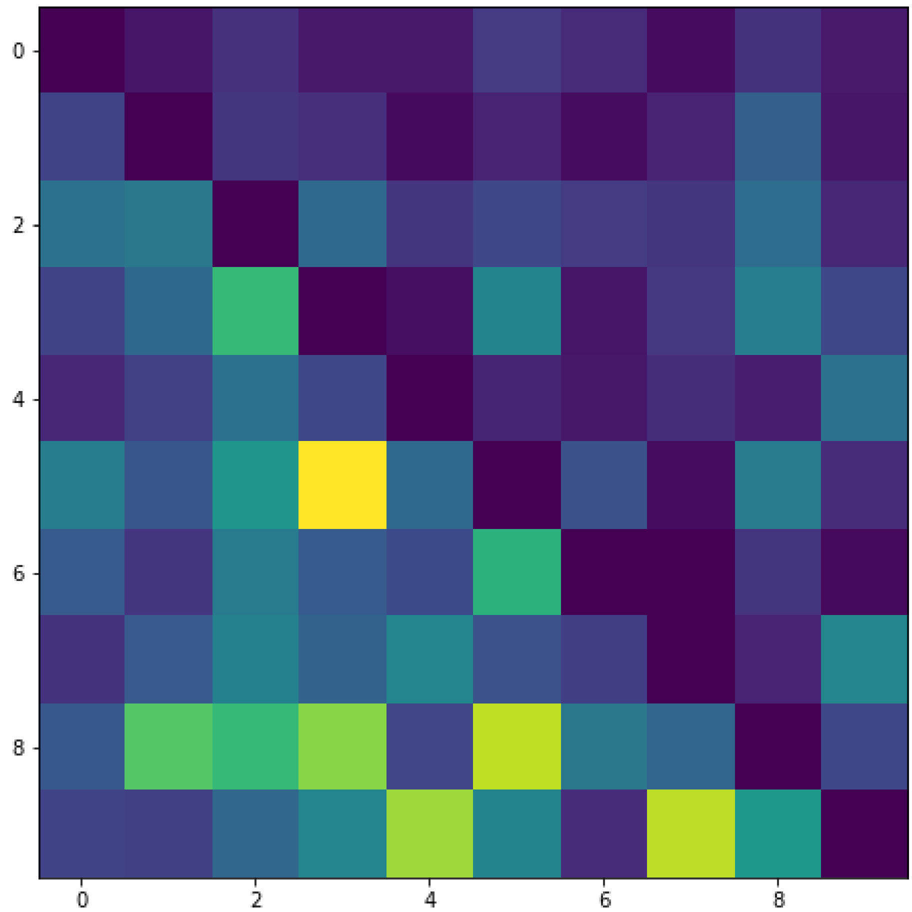
In [7]:

```
from tqdm import tqdm_notebook as tqdm
num_trial, err_matrix = 1, np.zeros((10, 10))
for a, b in tqdm(it.combinations(range(10), 2), total=45):
    err_tr, err_te = np.mean([mnist_pairwise_LS(df, a, b) for _ in range(num_trial)], axis=0)
    err_matrix[a, b], err_matrix[b, a] = err_tr, err_te

plt.figure(figsize=(8, 8))
plt.imshow(err_matrix)
print(np.round(err_matrix*100, 2))
```

executed in 2m 9s, finished 17:46:48 2019-11-18

100% 45/45 [02:08<00:00, 2.87s/it]

```
[[0.   0.32 0.82 0.4  0.39 0.98 0.7  0.16 0.83 0.41]
 [1.18 0.   0.88 0.8  0.14 0.57 0.18 0.55 1.74 0.34]
 [2.17 2.26 0.   1.95 0.87 1.23 0.99 0.89 2.04 0.67]
 [1.18 1.9  3.82 0.   0.24 2.58 0.33 0.96 2.45 1.24]
 [0.63 1.12 2.13 1.21 0.   0.58 0.37 0.76 0.44 2.15]
 [2.4  1.53 2.98 5.72 1.96 0.   1.41 0.2  2.37 0.73]
 [1.64 0.86 2.36 1.65 1.29 3.68 0.   0.   0.85 0.14]
 [0.84 1.61 2.52 1.83 2.62 1.46 1.05 0.   0.57 2.61]
 [1.56 4.21 3.83 4.71 1.2  5.17 2.27 1.87 0.   1.24]
 [1.18 1.08 1.89 2.6  4.89 2.58 0.72 5.15 3.05 0.  ]]
```



Exercise 3.4

But, what about a multi-class classifier for MNIST digits? For multi-class linear classification with d classes, one standard approach is to learn a linear mapping $f: \mathbb{R}^n \to \mathbb{R}^d$ where the "y"-value for the i-th class is chosen to be the standard basis vector $\underline{e}_i \in \mathbb{R}^d$. This is sometimes called one-hot encoding. Using the same A matrix as before and a matrix Y, defined by $Y_{i,j}$ if observation i in class j and $Y_{i,j} = 0$ otherwise, we can solve for the coefficient matrix $Z \in \mathbb{R}^d$ coefficients . Then, the classifier maps a vector $\underline{x}$ to class i if the i-th element of $Z^T \underline{x}$ is the largest element in the vector.

Follow the steps in the template and implement the multi-class classification experiment **(20 pt)**

In [8]:

```
X=df['feature']
y=df['label']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.5,
random_state=0)

# Randomly split into training/testing set
X_train=X_train.to_numpy()
X_test=X_test.to_numpy()
y_train=y_train.to_numpy()
y_test=y_test.to_numpy()

# Construct the training set
X_tr=np.zeros((X_train.shape[0],784))
for i in range(X_train.shape[0]):
    for j in range(784):
        X_tr[i,j]=X_train[i][j]

y_tr=np.zeros(y_train.shape[0])
for i in range(y_train.shape[0]):
    y_tr[i]=y_train[i]


# Construct the testing set
X_te=np.zeros((X_test.shape[0],784))
for i in range(X_test.shape[0]):
    for j in range(784):
        X_te[i,j]=X_test[i][j]

y_te=np.zeros(y_test.shape[0])
for i in range(y_test.shape[0]):
    y_te[i]=y_test[i]


# Apply one-hot encoding to training labels
Y = np.zeros((y_tr.shape[0],10))
for i in range(y_tr.shape[0]):
    for j in range(10):
        if (y_tr[i]==j):
            Y[i,j]=1

# Run least-square on training set
Z = solve_linear_LS(X_tr, Y)

# Compute estimation and misclassification on training set
Y_hat_tr=X_tr.dot(Z)
y_hat_tr = Y_hat_tr.argmax(axis=1)
misc_tr = accuracy_score(y_tr, y_hat_tr)
err_tr = 1 - misc_tr

# Compute estimation and misclassification on testing set
Y_hat_te=X_te.dot(Z)
y_hat_te = Y_hat_te.argmax(axis=1)
misc_te = accuracy_score(y_te, y_hat_te)
err_te = 1- misc_te

print('training error = {0:.2f}%, testing error = {1:.2f}%'.format(100 * err_tr,
100 * err_te))
# Compute confusion matrix
cm = np.zeros((10, 10), dtype=np.int64)
```

```python
for a in range(10):
    for b in range(10):
        cm[a, b] = ((y_te == a) & (y_hat_te == b)).sum()
print('Confusion matrix:\n {0}'.format(cm))
```

executed in 17.8s, finished 17:47:06 2019-11-18

```
training error = 13.85%, testing error = 15.73%
Confusion matrix:
 [[1963    5    8    5    9   14   32    1   26    1]
 [   1 2279   13    9    8    6   10    3   24    2]
 [  32   84 1693   65   47    8   88   34   72    9]
 [   9   51   66 1833   12   43   20   48   57   52]
 [   7   33   17    5 1755   26   20    8   31  124]
 [  53   27   11  206   30 1281   62   13  165   46]
 [  44   31   21    1   22   33 1907    0   18    0]
 [  24   58   34   23   52    5    2 1866   10  117]
 [  23  154   22   89   50   87   30    6 1476   54]
 [  33   17    5   39  146    7    1  162   25 1644]]
```

In [ ]: