```
In [10]: # import numpy and pandas and matplotlib (as plt)
         import numpy as np
         import matplotlib.pyplot as plt
```

Purpose:

Classify MNSIT using SVCs and kernels. MNIST is a dataset of handwritten digits; the task is to classify the digits as 0-9. The images themselves are 28 x 28 pixels large.

Data Set-up ¶

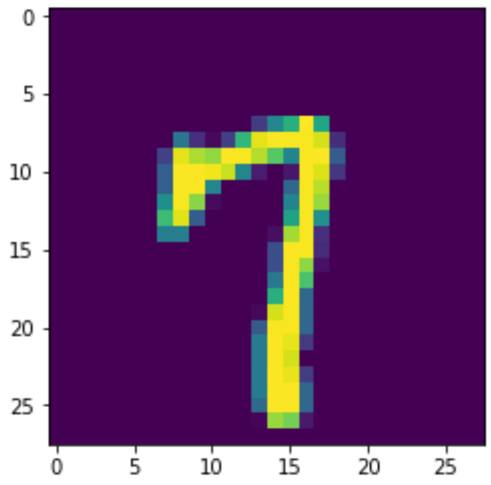
Import data and import train_test_split from sklearn.

```
In [11]: from sklearn.model_selection import train_test_split
         from sklearn.datasets import fetch_mldata

         # fetch "MNIST original"
         data = fetch_mldata("MNIST original")

         # determine X and y
         X = data.data
         y = data.target

         # print the shape of X and y

         print(X.shape,y.shape)
         # Use train_test_split. Keep test at 25%.

         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = .2
         5)


         # The SVM algorithm runs in O(n^2) time, where n is the number of traini
         ng points
         # To prevent the algorithm from taking forever, take only the first 1000
         0 data points
         # from the training set and the first 2000 data points from the test set

         X_train[:10000]
         X_test[:10000]
         y_train[:10000]
         y_test[:10000]

         (70000, 784) (70000,)

Out[11]: array([0., 9., 1., ..., 3., 5., 1.])
```

We will only use 1's and 7's for our classification problem. The following code block should filter out only the 1's and 7's:

In [21]:
```
# Make new test and train sets with only 1's and 7's

train_mask = ((y_train == 1).astype(int) + (y_train == 7).astype(int)).a
stype(bool)
test_mask = ((y_test == 1).astype(int) + (y_test == 7).astype(int)).asty
pe(bool)

X_test = X_test[test_mask, :]
y_test = y_test[test_mask]
X_train = X_train[train_mask, :]
y_train = y_train[train_mask]
```

In [20]:
```
# Use this to visualize the dataset
# Feel free to change the index
#                  v
plt.imshow(X_train[0].reshape(28,28))
plt.show()
```



Use PCA from sklearn

We will use Principal Component Analysis (PCA) to manipulate the data to make it more usable for SVC. The main idea of principal component analysis (PCA) is to reduce the dimensionality of a data set by projecting the data on to a space while still retaining as much variance in the data as possible.

In [26]:
```
# import PCA
from sklearn.decomposition import PCA
# There are a total of 28 * 28 features (one per pixel)
# Let's project this down to 2 features using pca (2 features so we can
 plot out the data in 2-d)
pca = PCA(n_components=2)

# Use pca to transform X_train, X_test
X_train_pca = pca.fit_transform(X_train, y_train)
X_test_pca = pca.fit_transform(X_test,y_test)

#print the shape of X_train_pca
print(np.shape(X_train_pca))
print(np.shape(X_test_pca))
```

```
(11362, 2)
(3808, 2)
```

What change do you notice between our old training data and our new one?

Answer: The new training data is MUCH smaller. This way, it is less computationally expensive.

SVC and Kernels

Now we will experiment with support vector classifiers and kernels. We will need LinearSVC, SVC, and accuracy_score.

SVMs are really interesting because they have something called the dual formulation, in which the computation is expressed as training point inner products. This means that data can be lifted into higher dimensions easily with this "kernel trick". Data that is not linearly separable in a lower dimension can be linearly separable in a higher dimension - which is why we conduct the transform. Let us experiment.

A transformation that lifts the data into a higher-dimensional space is called a kernel. A poly- nomial kernel expands the feature space by computing all the polynomial cross terms to a specific degree.

```
In [28]: # import SVC, LinearSVC, accuracy_score
         from sklearn.metrics import accuracy_score
         from sklearn.svm import LinearSVC, SVC
         # fit the LinearSVC on X_train_pca and y_train and then print train accu
         racy and test accuracy
         # CODE HERE
         lsvc = LinearSVC(loss="hinge")
         lsvc.fit(X_train_pca,y_train)

         print('train acc: ', accuracy_score(y_train,lsvc.predict(X_train_pca)))
         print('test acc: ', accuracy_score(y_test,lsvc.predict(X_test_pca)))

         # use SVC with an RBF kernel. Fit this model on X_train_pca and y_train
          and print accuracy metrics as before
         # CODE HERE
         svc = SVC()
         svc.fit(X_train_pca, y_train)
         print('train acc: ', accuracy_score(y_train,svc.predict(X_train_pca)))
         print('test acc: ', accuracy_score(y_train,svc.predict(X_train_pca)))
```

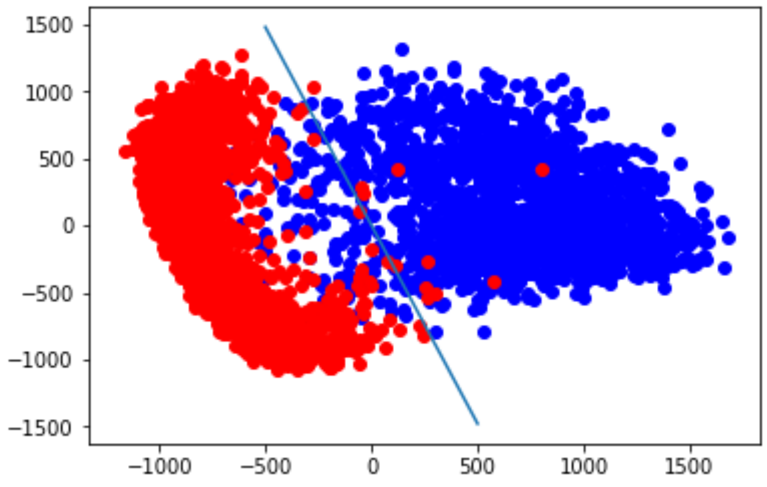
```
train acc:  0.9761485653934167
test acc:  0.9753151260504201
train acc:  1.0
test acc:  1.0
```

Visualize

Now plot out all the data points in the test set. Ones should be colored red and sevens should be colored blue. We have already provided the code to plot the decision boundary. The plot is a reault of using PCA on a 784 dimensional data.

Hint: `plt.scatter`

In [29]:
```
### YOUR CODE HERE ###


test7= X_test_pca[(y_test==7)]
lab7= y_test[(y_test==7)]
test1= X_test_pca[(y_test==1)]
lab1= y_test[(y_test==1)]
plt.scatter(test7[:,0],test7[:,1],color='blue')
plt.scatter(test1[:,0],test1[:,1],color='red')
# Code to plot the decision boundary of a linear svm

weights = lsvc.coef_[0]
x = np.linspace(-500,500,100)
y = x / weights[1] * -weights[0]
plt.plot(x,y)

plt.show()
```



Sort!

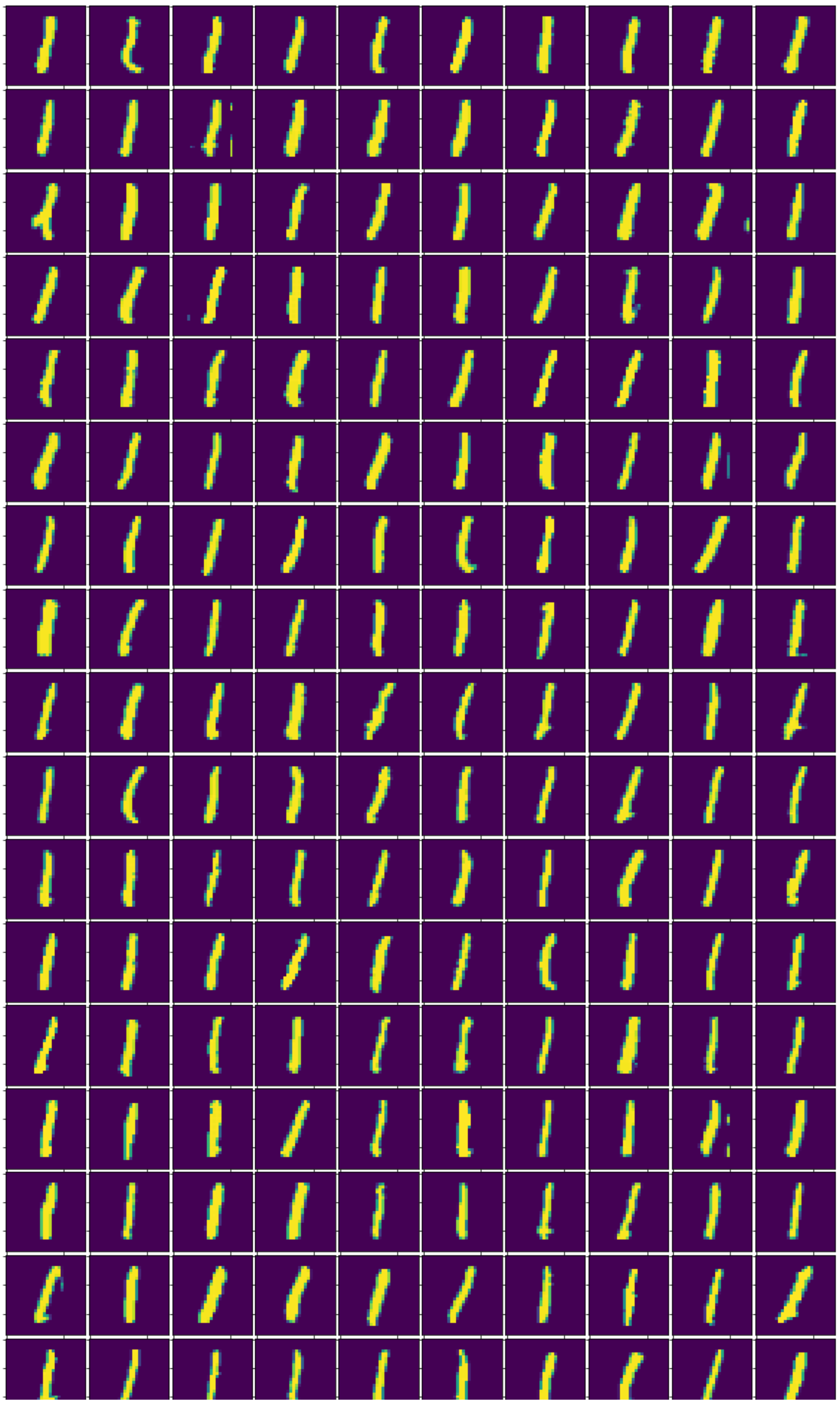
Now we're going to do a kind of hack. We've trained a linearSVM (SVC) on a binary classification problem. But what if we wanted something more regression-like? Say we wanted to score each datapoint on how "one-y" or how "seven-y" it looked. How would we do that? Check out the documentation (http://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html#sklearn.svm.LinearSVC).

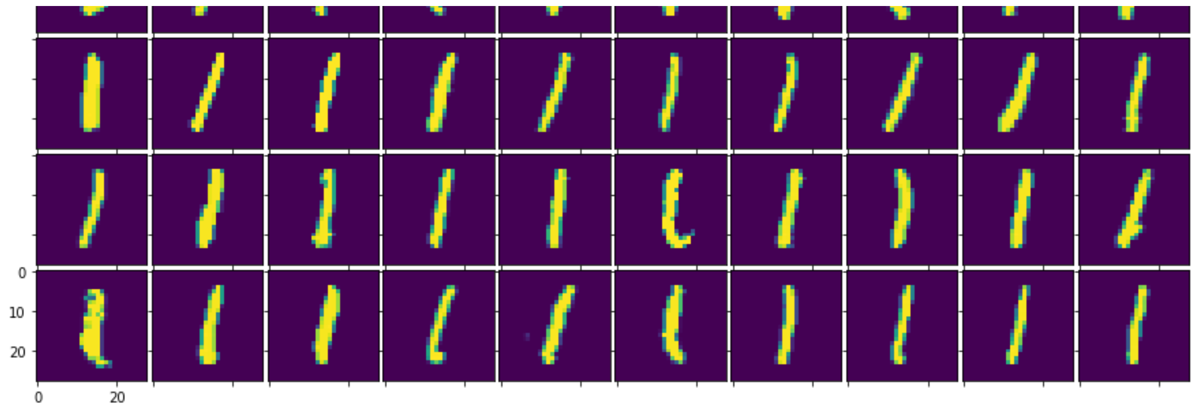
In the block below, create a list of scores for each datapoint in `X_test_pca`. Then sort `X_test` using the scores from `X_test_pca` (we're using `X_test` instead of `X_test_pca` because we want to plot the images). The block after contains code to plot out the sorted images. You should see 1's gradually turn in to 7's.

In [32]:
```
### YOUR CODE HERE ###

scores = lsvc.decision_function(X_test_pca)
sorted_X = [X_test[i] for i in np.argsort(scores)]
sorted_X = np.array(sorted_X)
```

Code to plot the images (this may take some time)

In [33]:
```
from mpl_toolkits.axes_grid1 import AxesGrid

def plot(x):
    plt.imshow(x.reshape(28,28))
    plt.show()

def plot_dataset(X):
    fig = plt.figure(1, (60, 60))

    fig.subplots_adjust(left=0.05, right=0.95)

    grid = AxesGrid(fig, 141,  # similar to subplot(141)
                    nrows_ncols=(20, 10),
                    axes_pad=0.05,
                    label_mode="1",
                    )

    for i in range(200):
        grid[i].imshow(X[i].reshape(28,28))

# We're assuming sorted_X has ~200 datapoints in it
# This may take a long time to run
plot_dataset(sorted_X)
```





Conclusions

1) What is a kernel and why is it important?

2) Can we kernelize all types of data? Why or why not?

3) What are some pros/cons of kernels? (look into runtime)

```
In [36]: """1. A kernel is a summation of functions that can be used to
         separate otherwise inseparable data. A kernel trick, for example,
         will add values to increase dimensionality and thus make it
         linearly separable."""

         """2. Virtually all types of data can be kernalized. """

         """3. Kernels take a long time to run. On very large datasets with many
          dimensions,
         it takes a long time to run and thus it can be expensive. """
```

```
Out[36]: '3. Kernels take a long time to run. On very large datasets with many d
         imensions,\nit takes a long time to run and thus it can be expensive. '
```