CSCI 3022 Homework

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below.

In [1]:
```
NAME = "Tyler Nevell"
COLLABORATORS = "Cody Hegwer, Stephen Kay, Kyle Staub"
```

If you referenced any web sites or solutions not of your own creation, list those references here:

- List any external references or resources here

In [2]:
```
%matplotlib inline
import numpy as np
import scipy as sp
import scipy.stats as stats
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import patsy
import sklearn
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
```

Problem 1 [30 Points]

Logistic regression uses the *logit* function to assign probabilities to predicted values. The parameters of the logit function are determined using maximum liklihood estimation. In certain cases, logistic classification fails to classify data we would consider "easy" to classify.

Create two samples of data, each of 3000 samples drawn from a $N(30, 10)$ distribution that is then limited to a lower range of 5 and upper range of 50 (*i.e.* using np.min/np.max). These samples represent the ages of a random population. Call the first sample `age18` and create a second set named `age0` that is `age18-18`. Then create a vector `is_adult` that is `1` for each `age18` entry that is ≥ 18 and 0 for all other entries.

In other words, both samples contain the same data, but `age0` is shifted by `18`. The `is_adult` vector encodes the same information for each dataset.

In [3]:
```
age18 = np.minimum(50, np.maximum(5, stats.norm(loc=30, scale=10).rvs(3000)))

is_adult = (age18 >= 18).astype("int")

#age18 = age18.reshape(-1,1)

print("age18:",age18)

age0 = age18 - 18
print("age0:",age0)
print("is_adult:",is_adult[:10])
```

```
age18: [21.56614686 20.36236731 47.78382213 ... 25.12786613 11.98611683
 32.1789676 ]
age0: [ 3.56614686  2.36236731 29.78382213 ...  7.12786613 -6.01388317
 14.1789676 ]
is_adult: [1 1 1 1 1 1 0 1 1 0]
```

Fit a logistic regression to the `age18` and `is_adult` dataset and print out the confusion matrix resulting from predicting the results for the `age18` dataset.

In [4]:
```
df = pd.DataFrame()
df['age18'] = age18
df['age0'] = age0
df['is_adult'] = pd.Series(is_adult, dtype = 'category')

y,X = patsy.dmatrices("C(is_adult, [[0],[1]]) ~ 0 + age18", data=df)

lr = sklearn.linear_model.LogisticRegression()
age18_lr = lr.fit(X.reshape(-1, 1), y.ravel())
a18_hat = age18_lr.predict(X.reshape(-1,1))

sklearn.metrics.confusion_matrix(y, a18_hat)
```

Out[4]:
```
array([[ 270,   44],
       [   0, 2686]], dtype=int64)
```

Fit a logistic regression to the `age0` and `is_adult` dataset and print out the confusion matrix resulting from predicting the results for the `age0` dataset.

In [29]:
```
lr2 = sklearn.linear_model.LogisticRegression()
y1,X1 = patsy.dmatrices("C(is_adult, [[0],[1]])~0+age0", data=df)
age0_lr = lr2.fit(X1.reshape(-1,1),y1.ravel())
a0_hat = age0_lr.predict(X1.reshape(-1,1))

sklearn.metrics.confusion_matrix(y1,a0_hat)
```

Out[29]:
```
array([[ 313,    1],
       [   0, 2686]], dtype=int64)
```

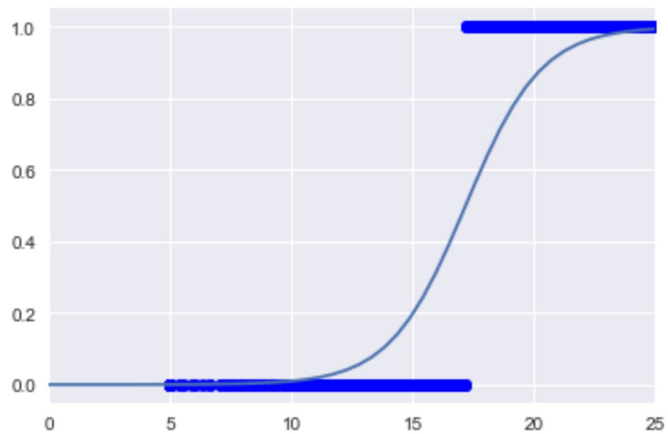
Examine the two confusion matricies, comment on their relationship to each other and their ability to predict the target data. Note that since the sample data is stochastic, you may want to run the code a few times to determine a common pattern between the confusion matricies.

Our models can predict the data in an accurate manner. But the most accurate model is for age0 which got no predictions wrong.

To help understand these results, you will prepare two plots of the `logit` plot. In the first plot, you should plot the logitistic function using the intercept and parameter from fitting the `age18` data. In the second plot, you should plot the logitstic function using $b_0 = 0$ and varying b_1 from 1 to 40 -- you only need 3 or 4 values in that range.

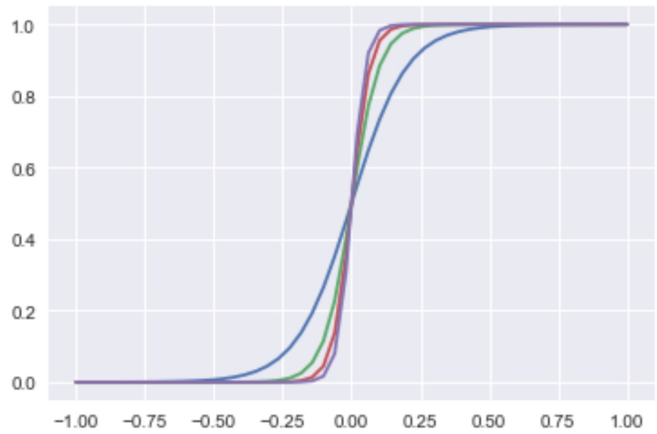
In [14]:
```
def logistic(model, x):
    z = np.exp(model.intercept_ + model.coef_[0] * x)
    return  z / (1 + z)

def logit(b0, b1, x):
    z = np.exp(b0 + b1 * x)
    return z / (1 + z)

exes = np.linspace(-1,25)
a18log = logistic(age18_lr, exes)

plt.plot(X, a18_hat, 'bo')
plt.plot(exes, a18log)
plt.xlim(0,25)
```

Out[14]: (0, 25)



In [15]:
```
b0 = 0
exes2 = np.linspace(-1,1)

plt.plot(exes2, logit(b0,10,exes2))
plt.plot(exes2, logit(b0,20,exes2))
plt.plot(exes2, logit(b0,30,exes2))
plt.plot(exes2, logit(b0,40,exes2))
```

Out[15]: `[<matplotlib.lines.Line2D at 0x25c2ce795c0>]`



Using those two plots and the two confusion matricies, explain the results from the logistic classification for `age18` and `age0`. Comment on the trend shown in the second graph of the ability of the logit to separate the values less than zero from those greater than zero; what value would likely result in a perfect separate? What happens when you re-run your logistic classification using 300,000 samples rather than 3,000?

We would prefer larger b coefficients.

Problem 2 - Surviving the Titantic [30 points]

In [16]:
```
ti = pd.read_csv('https://raw.githubusercontent.com/jorisvandenbossche/pandas-tutorial/master/data/titanic.csv'
)
print(ti.columns)
```

```
Index(['PassengerId', 'Survived', 'Pclass', 'Name', 'Sex', 'Age', 'SibSp',
       'Parch', 'Ticket', 'Fare', 'Cabin', 'Embarked'],
      dtype='object')
```

First, build a logistic classification model that maximizes the accuracy of predicting who survives the Titantic. You can use the Patsy tool to prepare the design matrix. The underlying survival rate is 0.40 and your model should achieve an accuracy of 0.80 or greater. You should print out your prediction accuracy.

In practice, you would split your data into training and testing data, but for this first set you should train and test on the full dataset.

In [17]:
```
ti2 = ti.drop(['Name', 'SibSp', 'Ticket', 'Fare', 'Embarked', 'Parch', 'Cabin'], axis = 1)
ti2 = ti2.dropna(axis=0, how='any')
```

In [30]:
```
lr3 = sklearn.linear_model.LogisticRegression()
yti,Xti = patsy.dmatrices("C(Survived,[[0],[1]]) ~ 0 + C(Sex,[[0],[1]]) * Age * C(Pclass, [[1],[2],[3]])",data=
ti2)
mSurv = lr3.fit(Xti, yti.ravel())

print('Accuracy: ', mSurv.score(Xti, yti))

surv_hat = mSurv.predict(Xti)
sklearn.metrics.confusion_matrix(yti, surv_hat)
```

```
Accuracy:  0.8025210084033614
```

Out[30]:
```
array([[387,  37],
       [104, 186]], dtype=int64)
```

Next, using your existing logistic design matrix, split the dataset into a train and test subset. The training set should use the even data elements and the testing set should use the odd. You can easily construct the even/odd sets using numpy indexing. (https://stackoverflow.com/questions/4988002/shortest-way-to-slice-even-odd-lines-from-a-python-array) We use even/odd rather than `sklearn's train_test_split` function because it produces predictable output letting us compare multiple homework solutions. In practice, you would do something more robust.

Re-run your regression model and report the prediction accuracy.

In [32]:
```
trainX, trainY = Xti[::2], yti[::2]
testX, testY = Xti[1::2], yti[1::2]

lr4 = sklearn.linear_model.LogisticRegression()
m2surv = lr4.fit(trainX, trainY.ravel())
print("Accuracy: ", m2surv.score(testX, testY))
```

```
Accuracy:  0.8291316526610645
```

Now, use the K-Nearest Neighbors classification method to predict the survival rate and print out the prediction accuracy. You should do this first using the full dataset. Vary the k to find the smallest $k \in \{2, 3, 4, 5\}$ that maximizes the accuracy. Remember that you may want to use a different model than for logistic regression.

In [36]:
```
k = 2
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors = k)
mNeigh = neigh.fit(Xti, yti.ravel())

knn_hat = neigh.predict(Xti)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, yti.ravel())

print(knn_cm)
print("K=2 accuracy: ", sklearn.metrics.accuracy_score(knn_hat, yti.ravel()))


k = 3
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors = k)
mNeigh = neigh.fit(Xti, yti.ravel())

knn_hat = neigh.predict(Xti)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, yti.ravel())

print(knn_cm)
print("K=3 accuracy: ", sklearn.metrics.accuracy_score(knn_hat, yti.ravel()))


k = 4
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors = k)
mNeigh = neigh.fit(Xti, yti.ravel())

knn_hat = neigh.predict(Xti)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, yti.ravel())

print(knn_cm)
print("K=4 accuracy: ", sklearn.metrics.accuracy_score(knn_hat, yti.ravel()))


k = 5
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors = k)
mNeigh = neigh.fit(Xti, yti.ravel())

knn_hat = neigh.predict(Xti)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, yti.ravel())

print(knn_cm)
print("K=5 accuracy: ", sklearn.metrics.accuracy_score(knn_hat, yti.ravel()))
```

```
[[412  80]
 [ 12 210]]
K=2 accuracy:  0.8711484593837535
[[391  60]
 [ 33 230]]
K=3 accuracy:  0.8697478991596639
[[403  83]
 [ 21 207]]
K=4 accuracy:  0.8543417366946778
[[388  72]
 [ 36 218]]
K=5 accuracy:  0.8487394957983193
```

Now, use the K-Nearest Neighbors classification method to predict the survival rate and print out the prediction accuracy. You should now do this **using the test & train sets**. Vary the k to find the smallest $k \in \{1, 2, 3, 4, 5\}$ that maximizes the accuracy.

In [63]:
```
k = 1
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors= k)
mNeigh = neigh.fit(trainX, trainY.ravel())

knn_hat = neigh.predict(testX)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, testY.ravel())

print(knn_cm)
print("Accuracy for k = 1: ", sklearn.metrics.accuracy_score(knn_hat, testY.ravel()))

k = 2
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors= k)
mNeigh = neigh.fit(trainX, trainY.ravel())

knn_hat = neigh.predict(testX)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, testY.ravel())

print(knn_cm)
print("Accuracy for k = 2: ", sklearn.metrics.accuracy_score(knn_hat, testY.ravel()))

k = 3
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors= k)
mNeigh = neigh.fit(trainX, trainY.ravel())

knn_hat = neigh.predict(testX)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, testY.ravel())

print(knn_cm)
print("Accuracy for k = 3: ", sklearn.metrics.accuracy_score(knn_hat, testY.ravel()))

k = 4
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors= k)
mNeigh = neigh.fit(trainX, trainY.ravel())

knn_hat = neigh.predict(testX)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, testY.ravel())

print(knn_cm)
print("Accuracy for k =4 : ", sklearn.metrics.accuracy_score(knn_hat, testY.ravel()))

k = 5
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors= k)
mNeigh = neigh.fit(trainX, trainY.ravel())

knn_hat = neigh.predict(testX)
knn_cm = sklearn.metrics.confusion_matrix(knn_hat, testY.ravel())
```

```
[[169  48]
 [ 43  97]]
Accuracy for k = 1:  0.7450980392156863
[[202  67]
 [ 10  78]]
Accuracy for k = 2:  0.7843137254901961
[[184  42]
 [ 28 103]]
Accuracy for k = 3:  0.803921568627451
[[196  59]
 [ 16  86]]
Accuracy for k =4 :  0.7899159663865546
[[184  39]
 [ 28 106]]
Accuracy for k = 5:  0.8123249299719888
```

Skim through this paper that analyzes the survival information concerning the titantic (https://espace.library.uq.edu.au/data/UQ_152940/HallSSM2261986.pdf?Expires=1522708604&Signature=ETHPpxGdlnk5iXlzs8THm8EQgpdAtWAo-YwCpOvLLFZrJZi6MRKAmEbhEO~tFsJpbTWRR-pFNGdoY54-j~cZ4iWFbarSyUYVxVc5rYYER8iRvkHe0GTF8xoKO43DlKE5gTKDSB-0VCRs1E55-MXMbmW1puzip9qFZtwp7hc3qxFNNcORI83k1zt6MSRB2BQXXpn5nc~9Fjrjkfrv~t8cwCNfIKNLGRze9s5L98-PivrTBcZx6UVgmCMAVuHc9uAUuFTdqfmmd2wWs8pXec6e585TFM5A-jnSGIM~AkbZGAg7xulMez16extI6P5A6uzxLcziBVvWeWymLFQAxRirhw__&Key-Pair-Id=APKAJKNBJ4MJBJNC6NLQ). That paper uses z-test comparisons (because n is "large") to compare survival rates between different groups and uses the historical Mersey investigation to attribute reasons behind the differences.

Assuming that `sklearn` could easily produce effects tables such as Table 4.3 in ISLR, describe the benefit of the logistic classification technique compared to the KNN technique for a researcher such as Wayne Hall. Then, describe the benefit of KNN for other applications. Use the terminology of ISLR 2.1.1 ("Why estimate f?") in your discussion. Assuming similar accuracy performance, which would you use if you were trying to suggest what movie to watch rather than who survived the titantic?

If you would like to compute an effects table, you can use the `StatsModels` logit (http://www.statsmodels.org/dev/generated/statsmodels.discrete.discrete_model.Logit.html) library, but I found that it depended on an older version of `scipy.stats` and it's hard to make it work, although you can make it dump out the `pvalues` table.

Image Recognition using Classification

In this problem, you're going to use classification methods such as Logistic, LDA and KNN to classify handwritten numbers. This problem was originally posed for the US Post Office and one of the data sets was assembled by the National Institute for Standards and Technology (NIST). The problem is now a standard problem in machine learning. We'll see that we can get ~92% accuracy for the full problem using Logistic or LDA and about 98% accuracy using KNN.

This tutorial (https://www.tensorflow.org/versions/r1.1/get_started/mnist/beginners) shows how to use Google's TensorFlow software to tackle the same problem using (multinomial) logistic regression, which is the same mathematics as we're using rendered into a different software package. That solution also gets ~92% accuracy. You can also use TensorFlow to implement a "deep learning" solution (https://www.tensorflow.org/versions/r1.1/get_started/mnist/pros) that achieves ~99% accuracy.

Using a Small Dataset

We're going to focus on the trade-off of training time vs. prediction time and the accuracy achieved by different classification methods. We will first use a dataset that uses 1797 small 8x8 (64 pixel) images. In the code below, we load the dataset using an `sklearn` interface.

In this section, your goal will be to understand how *multinomial* classification functions and how you can use outputs of (some) classification tools to understand how certain or confidence you should be in a classification result.

In [21]:

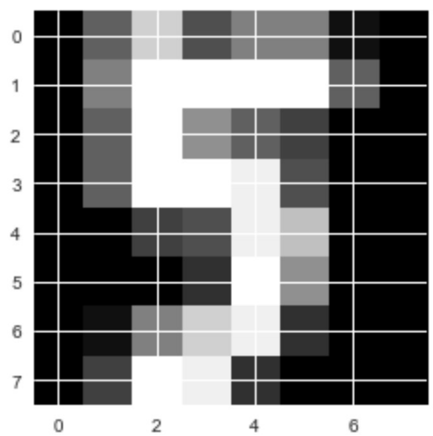
```
%matplotlib inline
from sklearn.datasets import load_digits
digits = load_digits()
print("Image Data Shape" , digits.data.shape)
print("Label Data Shape", digits.target.shape)
```

```
Image Data Shape (1797, 64)
Label Data Shape (1797,)
```

Each image is encoded as values between 0 and 16. When training the data, we view the image as a 64-element set of features or factors. We can view the image by arranging it an 8x8 array and using pyplot's `imshow` (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.imshow.html) routine, as we do below:

In [22]:
```
k=33
plt.imshow(digits.data[k].reshape(8,8), cmap=plt.cm.gray)
print('Digit:', digits.target[k])
print('Image:\n', digits.data[k].reshape(8,8))
```

```
Digit: 5
Image:
 [[ 0.  6. 13.  5.  8.  8.  1.  0.]
 [ 0.  8. 16. 16. 16. 16.  6.  0.]
 [ 0.  6. 16.  9.  6.  4.  0.  0.]
 [ 0.  6. 16. 16. 15.  5.  0.  0.]
 [ 0.  0.  4.  5. 15. 12.  0.  0.]
 [ 0.  0.  0.  3. 16.  9.  0.  0.]
 [ 0.  1.  8. 13. 15.  3.  0.  0.]
 [ 0.  4. 16. 15.  3.  0.  0.  0.]]
```



Now, divide the `digits` dataset into a train/test split using even/odd images as before. Again, we do this to allow precise comparison o the results betwen solutions and students.

In [23]:
```
dX_train, dX_test, dy_train, dy_test = digits.data[0::2], digits.data[1::2],digits.target[0::2], digits.target[
1::2]
print(dX_train.shape)
```

```
(899, 64)
```

Small Digits using Logistic

Use a logistic classifier fit with the training data to then predict the test data. Report the accuracy score and the confusion matrix.

In [38]:
```
lrnumz = sklearn.linear_model.LogisticRegression()
numFit = lrnumz.fit(dX_train, dy_train)
num_hat = numFit.predict(dX_test) #predictor

num_cm = sklearn.metrics.confusion_matrix(dy_test,num_hat)
print(num_cm)
for i in range(len(num_cm)):
    for j in range(len(num_cm)):
        if i != j and num_cm[i][j] > 1:
            print("{", i ,",", j , "}")
```

```
[[86  0  0  0  2  0  0  0  0  0]
 [ 0 87  0  1  0  0  0  0  1  0]
 [ 0  1 90  0  0  0  0  0  0  0]
 [ 2  0  0 86  0  3  0  1  1  0]
 [ 0  1  0  0 86  0  0  0  0  1]
 [ 1  1  0  0  0 87  1  0  0  1]
 [ 0  2  0  0  0  0 88  0  0  0]
 [ 0  0  0  0  3  0  0 88  0  0]
 [ 0  7  1  2  1  0  0  0 74  1]
 [ 0  1  2  3  4  1  0  3  5 72]]
{ 0 , 4 }
{ 3 , 0 }
{ 3 , 5 }
{ 6 , 1 }
{ 7 , 4 }
{ 8 , 1 }
{ 8 , 3 }
{ 9 , 2 }
{ 9 , 3 }
{ 9 , 4 }
{ 9 , 7 }
{ 9 , 8 }
```

Based on the test data, which pairs of digits are confused more than once? In other words, if you examine the first column, you see 2 predictions where a '0' is misclassified as a '4'; you would report this as {0,4}. Construct similar sets of confused digits for all entries confused more than 1 time. Comment on the any expected and suprising outcomes.

9 is the most difficult number for our machine to read

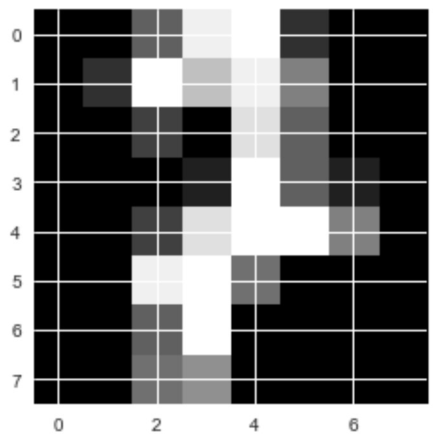
The digit classification problem involves a *multinomial*, or more than two levels in the outcome. By default, the `LogisticRegression` method uses a series of binomial logistic regression fits to the different outcomes of the multinomial. The `predict_proba` routine in `LogisticRegression` (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) returns the probability of the fit to each individual possible outcome (e.g. the digits '0' through '9'). The predicted outcome (i.e. the result of `predict`) is then the outcome with the largest predicted outcome.

For the two examples where the predicted digit is '4' but the actual digit is '0', plot the images corresponding to those digits and print out the results of `predict_proba` for those targets. In my solution to this, produced a vector of True/False values using element-wise comparisons and then used `np.nonzeros` (https://docs.scipy.org/doc/numpy/reference/generated/numpy.nonzero.html) to extract the indicies in the test data of the "true" values (corresponding to the samples that matched '4' in my prediction but whose actual target was '0'). you may also want to use `np.round` to round up the `predict_proba` results to 3-4 digits, making it eaiser to read.

In [50]:
```
empty = np.empty(len(dX_test))

for i in range(len(empty)):
    if num_hat[i] == 4 and dy_test[i] == 0:
        empty[i] = 1
    else:
        empty[i] = 0

indices = np.nonzero(empty)
pred_prob = numFit.predict_proba(dX_test)

print('Probability that this is a four:', np.max(pred_prob[indices[0][0]]))
plt.imshow(digits.data[indices[0][0]].reshape(8,8), cmap=plt.cm.gray)
```

```
Probability that this is a four: 0.5913664830017584
```

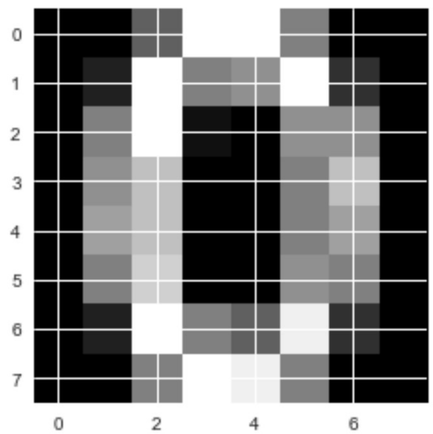
Out[50]:
```
<matplotlib.image.AxesImage at 0x25c2d3f8ef0>
```



In [44]:
```
print('Probability that this is a four:', np.max(pred_prob[indices[0][1]]))
plt.imshow(digits.data[indices[0][1]].reshape(8,8), cmap=plt.cm.gray)
```

```
Probability that this is a four: 0.7670205700331016
```

Out[44]:
```
<matplotlib.image.AxesImage at 0x25c2d380080>
```



Using the values from `predict_prob` are both mis-classified '0' values equally likely to have been classified as a '4'? Do the probabilities of the predicted outcomes comport with your visual interpretation of the digits?

The first digit is not a number, so it's unrealistic to expect the computer to read it as one.

Small Digits using KNN

Now, using the K-Nearest Neighbors method to fit and predict the test and train data. Select $k \in \{1, 2, 3, 4, 5\}$ that achieves the highest accuracy. Print the accuracy score and confusion matrix for the k with the highest accuracy.

In [42]:
```
k = np.linspace(1,5,5)

for each in k:

    neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors = each.astype('int'))
    sd_nmodel = neigh.fit(dX_train, dy_train.ravel())
    knn_hat = sd_nmodel.predict(dX_test)
    knn_cm = sklearn.metrics.confusion_matrix(knn_hat, dy_test)
    print('KNN(', each ,') confusion matrix is = \n', knn_cm)
    print('\nScore = ', sklearn.metrics.accuracy_score(knn_hat, dy_test.ravel()))
    print('\n Pairs of Digits Misclassified More Than Once:\n')
    for i in range(len(num_cm)):
        for j in range(len(num_cm)):
            if i != j and knn_cm[i][j] > 1:
                print("{", i ,",", j , "}")
    print('\n ################################### \n')
```

```
KNN( 1.0 ) confusion matrix is =
 [[88  0  0  0  0  0  0  0  0  0]
 [ 0 89  0  0  1  0  0  0  2  1]
 [ 0  0 91  1  0  0  0  0  0  0]
 [ 0  0  0 92  0  0  0  0  0  0]
 [ 0  0  0  0 87  0  0  0  0  1]
 [ 0  0  0  0  0 88  0  0  0  0]
 [ 0  0  0  0  0  1 90  0  1  0]
 [ 0  0  0  0  0  0  0 91  1  0]
 [ 0  0  0  0  0  0  0  0 82  1]
 [ 0  0  0  0  0  2  0  0  0 88]]

Score =  0.9866369710467706

 Pairs of Digits Misclassified More Than Once:

{ 1 , 8 }
{ 9 , 5 }

 ##################################

KNN( 2.0 ) confusion matrix is =
 [[88  0  0  0  0  0  0  0  0  0]
 [ 0 89  0  0  2  0  0  0  5  1]
 [ 0  0 91  2  0  0  0  0  0  0]
 [ 0  0  0 91  0  0  0  0  0  0]
 [ 0  0  0  0 86  0  0  0  0  1]
 [ 0  0  0  0  0 88  0  0  0  1]
 [ 0  0  0  0  0  1 90  0  1  0]
 [ 0  0  0  0  0  0  0 91  1  0]
 [ 0  0  0  0  0  0  0  0 79  3]
 [ 0  0  0  0  0  2  0  0  0 85]]

Score =  0.977728285077951

 Pairs of Digits Misclassified More Than Once:

{ 1 , 4 }
{ 1 , 8 }
{ 2 , 3 }
{ 8 , 9 }
{ 9 , 5 }

 ##################################

KNN( 3.0 ) confusion matrix is =
 [[88  0  0  0  0  0  0  0  0  0]
```

As before, based on the test data, which pairs of digits are confused more than once? Comment on the any expected and suprising outcomes.

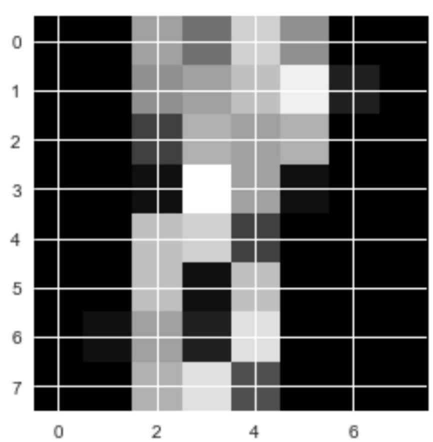
9 and 1 are the two that are confused with each other the most. Nothing is surprising to me. Most of these are exescusable.

Selecting one pair of confused digits, print out the image and the probability estimates (`predict_proba`).

In [52]:
```
neigh = sklearn.neighbors.KNeighborsClassifier(n_neighbors = 1)
sd_nmodel = neigh.fit(dX_train, dy_train.ravel())
knn_hat = sd_nmodel.predict(dX_test)


for i in range(len(empty)):
    if num_hat[i] == 5 and dy_test[i] == 9:
        empty[i] = 1
    else:
        empty[i] = 0

indices = np.nonzero(empty)
pred_prob = sd_nmodel.predict_proba(dX_test)

print('Probability that this is a five:', np.max(pred_prob[indices[0][0]]))
plt.imshow(digits.data[indices[0][0]].reshape(8,8), cmap=plt.cm.gray)
```

```
Probability that this is a five: 1.0
```

Out[52]:
```
<matplotlib.image.AxesImage at 0x25c3191b198>
```



Comment on differences in the results of `predict_proba` between the logistic and KNN classifiers. Would the results be similar for different values of k in the KNN sarch?

Even though this number is not a 5, my computer is 100% confident it is. This would result in serious errors in the real world.

Using the larger MNIST data

We will now use the MNIST dataset, which is the same used in the TensorFlow tutorial. This dataset is large, and contiains 70,000 images each of which are 28x28 pixels.

In this section, your goal will be to understand the performance of difference classification tools and their impact on usability in an application.

We first load the dataset. This may take a while the first time because the data has to be downloaded.

In [24]:
```
import sklearn.datasets
mnist = sklearn.datasets.fetch_mldata('MNIST original')
print("Image Data Shape" , mnist.data.shape)
print("Label Data Shape", mnist.target.shape)
```

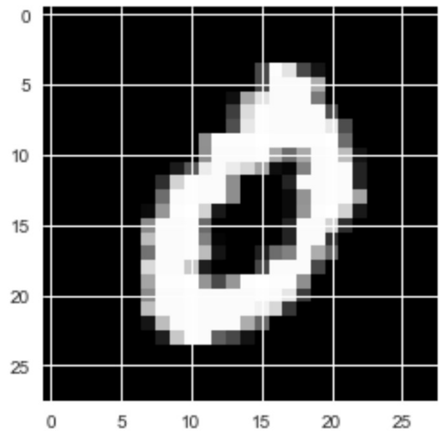
```
Image Data Shape (70000, 784)
Label Data Shape (70000,)
```

As before, the dataset has a `data` array of 784 features or factors that can be reorganized into an image. There is also a `target` value indicating the correct digit.

In [25]:

```
k=3
plt.imshow(np.reshape(mnist.data[k], (28,28)), cmap=plt.cm.gray, label='Digit:' + str(mnist.target[k]))
print('values:', mnist.data[k].reshape(28,28))
```

```
values: [[  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  73 253 227
   73  21   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0  73 251 251
  251 174   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  16 166 228 251 251
  251 122   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  62 220 253 251 251
  251 251  79   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0   0   0  79 231 253 251 251
  251 251 232  77   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 145 253 253 253 255 253 253
  253 253 255 108   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0   0   0 144 251 251 251 253 168 107
  169 251 253 189  20   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0   0  27  89 236 251 235 215 164  15   6
  129 251 253 251  35   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  47 211 253 251 251 142   0   0   0  37
  251 251 253 251  35   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0 109 251 253 251 251 142   0   0   0  11
  148 251 253 251 164   0   0   0   0   0]
 [  0   0   0   0   0   0   0  11 150 253 255 211  25   0   0   0   0  11
  150 253 255 211  25   0   0   0   0   0]
 [  0   0   0   0   0   0   0 140 251 251 253 107   0   0   0   0   0  37
  251 251 211  46   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 190 251 251 253 128   5   0   0   0   0  37
  251 251  51   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 115 251 251 253 188  20   0   0  32 109 129
  251 173 103   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 217 251 251 201  30   0   0   0  73 251 251
  251  71   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 166 253 253 255 149  73 150 253 255 253 253
  143   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 140 251 251 253 251 251 251 251 253 251 230
   61   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0 190 251 251 253 251 251 251 251 242 215  55
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0  21 189 251 253 251 251 251 173 103   0   0
    0   0   0   0   0   0   0   0   0   0]
 [  0   0   0   0   0   0   0   0  31 200 253 251  96  71  20   0   0   0
```



Again, split your data into an even/odd train/test dataset using numpy indexing. You should name the data something different than your smalled 'digits' data.

In [26]:
```
mX_train, mX_test, my_train, my_test = mnist.data[0::2], mnist.data[1::2],mnist.target[0::2], mnist.target[1::2
]
print(mX_train.shape)
```

```
(35000, 784)
```

Now, train a logistic regression model on the MNIST training data. You should prefix your fit function call using the %time "magic" command (http://ipython.readthedocs.io/en/stable/interactive/magics.html) to measure how long the fitting process takes.

This will take a long time for the default method we've been using to run logistic classification problems (like more than 30 minutes), in part because the default method fits n binomial classification problems to determine the multinomial model. If you start using the standard solver (`liblinear`) and decide it's too slow, use the Kernel -> Interrupt menu to stop the evaluation.

Logsitic regression uses *maximum liklihood estimation* to determine the most likely outcome. There are numerous *solvers* (see the LogisticRegression manual (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html)) that can be used and some of them are more appropriate for large multinomial problems because they fit the data to all the outcomes in one go. Find one that doesn't take forever (some should take ~15 seconds) and fit your model to the training data.

In [53]:
```
#%time mlr = sklearn.linear_model.Stuff()

lrnumz2 = sklearn.linear_model.LogisticRegression(solver = 'lbfgs', multi_class = 'multinomial')
%time numFit2 = lrnumz2.fit(mX_train, my_train)
num_hat2 = numFit2.predict(mX_test)
```

```
Wall time: 5.03 s
```

Now, compute the predictions and `predict_proba` for the test dataset and use %time to determine how long the predictions take. Report the accuracy score and the confusion matrix.

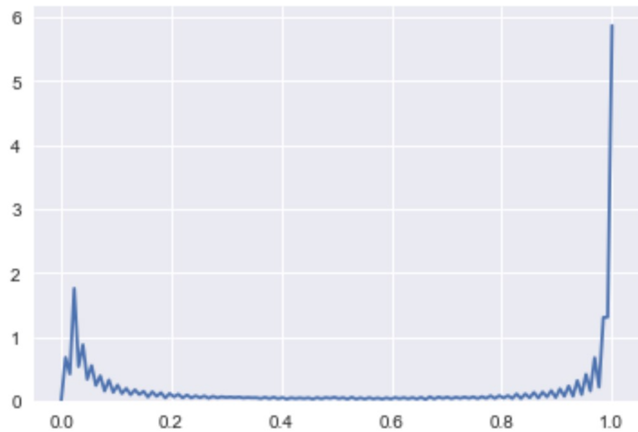
In [54]:
```
%time pred_prob = numFit2.predict_proba(mX_test)
print('Score is ', numFit2.score(mnist.data, mnist.target))

num_cm2 = sklearn.metrics.confusion_matrix(num_hat2,my_test)
print('CM: \n', num_cm2)
```

```
Wall time: 108 ms
Score is  0.9265
CM:
 [[3313    0   13   17   10   30   23   11   20   23]
 [   0 3826   25   13   15   16    6   15   56   11]
 [  15   21 3120   88   26   28   39   41   58    9]
 [   6   12   66 3149    5  128    2   27   85   43]
 [   8    6   32    7 3145   35   35   27   17  105]
 [  31   14   27  139    4 2693   36    6  114   24]
 [  25    2   49    7   35   51 3272    1   32    0]
 [  13    8   44   37   21   12    2 3367   12  114]
 [  36   40  111   76   38  131   18   11 2974   41]
 [   4    9    8   38  113   32    5  141   44 3110]]
```

Now, compute the probability scores for each outcome class using `predict_proba` and plot either a histogram or KDE plot of their values. You can use the output of `predict_proba` and then use `ravel()` to turn it into single flat array suitable for feeding to `plt.hist` or `sns.kdeplot`.

In [55]:
```
sns.kdeplot(pred_prob.ravel())
```

Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x25c2af5a198>



Linear Discriminant Analysis is supposed to be superior for multinomial classification. Run the same classification problem using LDA and time the fitting proccess.

In [56]:
```
numlda = LinearDiscriminantAnalysis()

%time num_lda_model = numlda.fit(mX_train, my_train)
num_lda_hat = num_lda_model.predict(mX_test)
```

```
C:\ProgramData\Anaconda3\lib\site-packages\sklearn\discriminant_analysis.py:442: UserWarning: The priors do n
ot sum to 1. Renormalizing
  UserWarning)

Wall time: 2.42 s

C:\ProgramData\Anaconda3\lib\site-packages\sklearn\discriminant_analysis.py:388: UserWarning: Variables are c
ollinear.
  warnings.warn("Variables are collinear.")
```

Predict the outcomes and report the accuracy score and confusion matrix. Time how long it takes to run the prediction using `%time`.

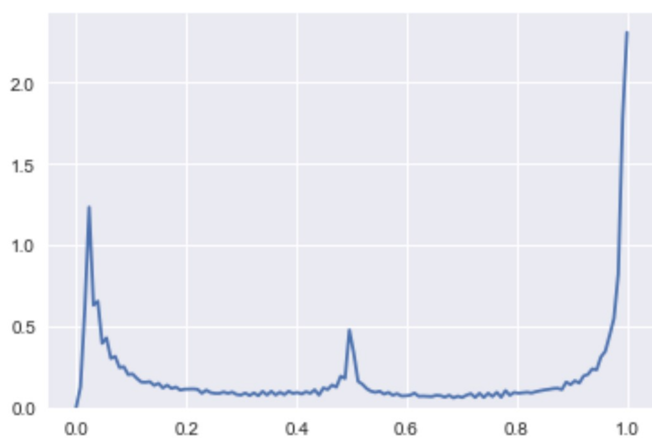
In [60]:
```
print('Score is ', sklearn.metrics.accuracy_score(num_lda_hat, my_test))
print('Confusion matrix:\n', sklearn.metrics.confusion_matrix(num_lda_hat,my_test))
```

```
Score is  0.8623428571428572
Confusion matrix:
 [[3248    0   43   10    6   35   42   15   17   23]
 [   2 3754  108   50   28   42   39   91  196   19]
 [  16   23 2821   95   29   21   34   35   32    8]
 [  19    7  103 2998    3  179    3   25  125   60]
 [  11   10   77   14 3049   37   57  114   47  204]
 [  61   22   21  136   21 2537   84   13  170   11]
 [  31    7  114   14   15   64 3118    1   23    0]
 [   4    6   22   57    1   24    0 3040   10  174]
 [  52  103  165  109   37  134   57   15 2680   44]
 [   7    6   21   88  223   83    4  298  112 2937]]
```

Print the distribution of outcome probabilities from `predict_proba` using a histogram or KDE.

In [61]:
```
pred_prob2 = num_lda_model.predict_proba(mX_test)
sns.kdeplot(pred_prob2.ravel())
```

Out[61]: <matplotlib.axes._subplots.AxesSubplot at 0x25c2cd0c668>



Compare the distribution of probability of prediction values for Logistic and LDA classification. Comment on the differences and/or similarities of the range of values from `predict_proba` returned by each method.

Logistic Regression measures the handwritten numbers much better than LDA.

Lastly, we're going to do the same steps using the KNN algorithm. You should use $k = 1$ for the KNN method and record the fitting time.

In [62]:
```
good = sklearn.neighbors.KNeighborsClassifier(n_neighbors = 2, n_jobs=-1, algorithm = 'kd_tree')
mGood = good.fit(mX_train, my_train)
k = 3
%time knn_hat = mGood.predict(mX_test[::k])

print('Score is ', sklearn.metrics.accuracy_score(knn_hat, my_test[::k]))
print('Confusion matrix:\n', sklearn.metrics.confusion_matrix(knn_hat, my_test[::k]))
```

```
Wall time: 1min 20s
Score is  0.9595440130281992
Confusion matrix:
 [[1147    1   17    0    1    7    9    1    5    5]
 [   0 1303   13    2   10    5    3   18   10    3]
 [   0    3 1118    7    1    0    1    9   17    4]
 [   0    1    5 1164    0   31    1    3   31   12]
 [   2    2    0    0 1113    6    5    8   10   39]
 [   1    0    0   12    0  996    7    0   31    2]
 [   0    0    1    0    0    3 1120    0    9    0]
 [   0    3    8    2    2    1    0 1170    6   38]
 [   1    0    3    1    1    2    0    0 1012    4]
 [   0    0    0    2   10    0    0    7    7 1052]]
```

Now run the prediction using your KNN model. Note that this will take a long time (40 minutes?). If specify `n_jobs=-1` when you create your `KNeighborsClassifier`, then predictions will use all the cores on your computer. For example, that chnaged my 40 minute run time for the full dataset to 5 minutes.

You should first run the prediction on a small test set (e.g. perhaps every 40th sample) to make certain you're doing it right. The digits of the same outcome are usually bunched together and if you just e.g.select the first 1000 items, you'll find they only belong to one output class. Once you have your code working, run it for the full dataset.

In [ ]:
```
# your code here
```

Comparison

Now, compare the three methods. For each method, describe the accuracy achieved, the fitting time and the prediction time. For the Logistic and LDA model, describe how the distribution of the outcome probabilities may affect the accuracy score. Assume you're trying to apply the digit classification problem in the post-office. Which method would you use? Given that the accuracy isn't 100%, what outputs of the models could you use to improve mail sorting?

Best model was KNN with 95% accuracy. We could use the most accurate model and input more tests into it in order to icnrease it's learning and training.