Machine Learning & Pattern Recognition (MLPR)

Assignment 1

Georgios Pligoropoulos - s1687568

October 2016 - 1st Semester

In [1]:

```
from scipy import io
import numpy as np
from matplotlib import pyplot as plt
from sklearn.cross_validation import train_test_split

#or notebook
%matplotlib inline
```

```
/home/student/anaconda2/envs/mlpr/lib/python2.7/site-packages/sklear
n/cross_validation.py:44: DeprecationWarning: This module was deprec
ated in version 0.18 in favor of the model_selection module into whi
ch all the refactored classes and functions are moved. Also note tha
t the interface of the new CV iterators are different from that of t
his module. This module will be removed in 0.20.
  "This module will be removed in 0.20.", DeprecationWarning)
```

In [2]:

```
import sys
mlprDir = '/home/student/Dropbox/MSc_Artificial_Intelligence/mlpr_Machine_Learni
ng_Pattern_Recognition/mlpr'
sys.path.append(mlprDir)
```

In [3]:

```
import ploty
```

In [4]:

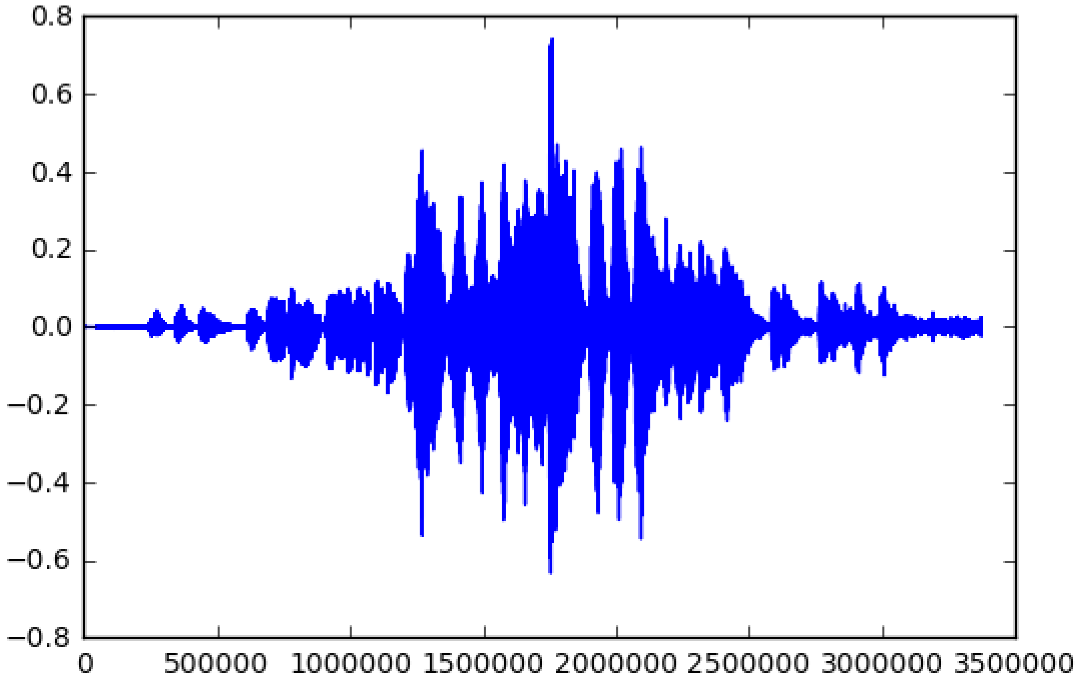
```
randomSeed = 0
```

In [5]:

```
amps = np.load('fifth.npy')
dataLen = len(amps)
```

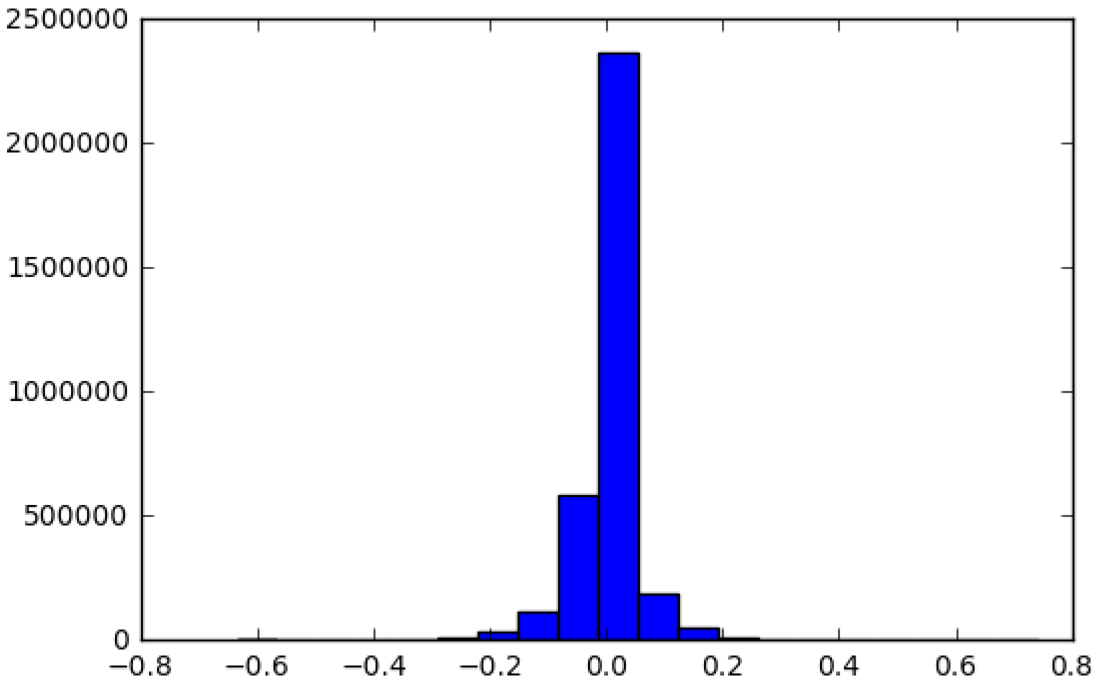
In [6]:

```
t = np.arange(0, dataLen)
ploty.plotNsave(t, amps, 'figs/amplitudes.png')
```



In [7]:

```
plt.figure()
plt.hist(amps, bins=20)
plt.show()
```



In [8]:

```
columnCount = 21
howManyToBeDiscarded = dataLen % columnCount

C = dataLen - howManyToBeDiscarded

ampsTrimmed = amps[:C]

ampsMatrix = ampsTrimmed.reshape((C / columnCount, columnCount))

randomState = np.random.RandomState(seed=randomSeed)
# np.random.shuffle(ampsMatrix)   #this is not functional, affects the array dir
ectly
randomState.shuffle(ampsMatrix)

dataTrain, dataRest = train_test_split(ampsMatrix, train_size=0.7,
test_size=0.3, random_state=randomSeed)

dataValidation, dataTest = train_test_split(dataRest, train_size=0.5, test_size=
 random_state=randomSeed)
```

In [9]:

```
def makeLastColumnOutput(matrix):
    return matrix[:, :matrix.shape[1] - 1], matrix[:, matrix.shape[1] - 1]
```

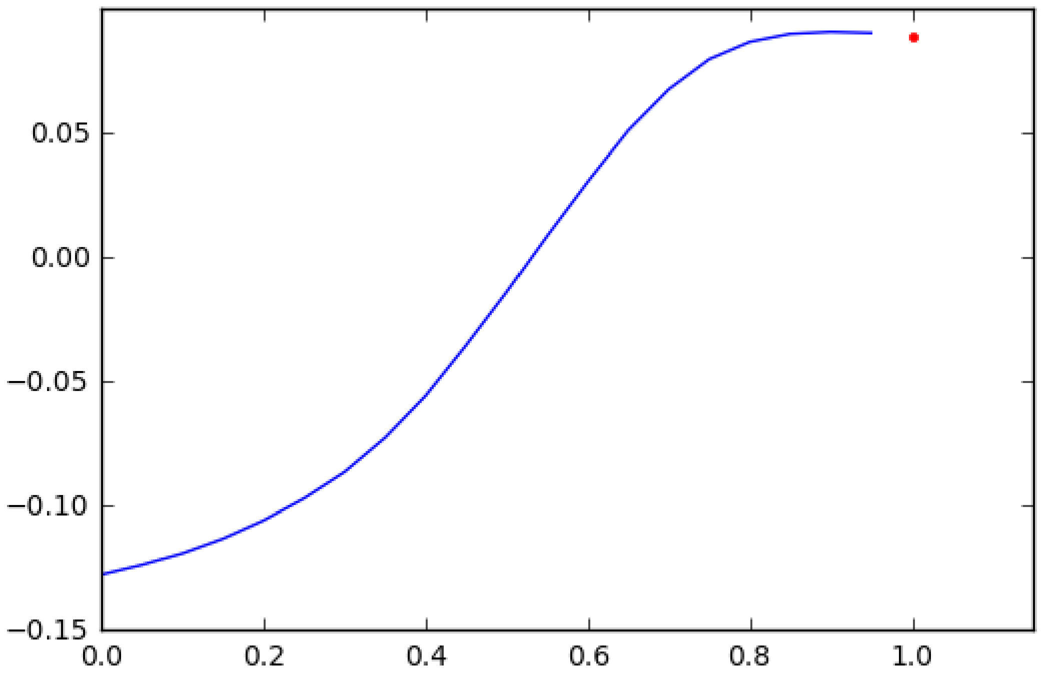
In [10]:

```
XshufTrain, yShufTrain = makeLastColumnOutput(dataTrain)
print "train shapes"
print XshufTrain.shape
print yShufTrain.shape

XshufVal, yShufVal = makeLastColumnOutput(dataValidation)
print "validation shapes"
print XshufVal.shape
print yShufVal.shape

XshufTest, yShufTest = makeLastColumnOutput(dataTest)
print "test shapes"
print XshufTest.shape
print yShufTest.shape
```

```
train shapes
(112377, 20)
(112377,)
validation shapes
(24081, 20)
(24081,)
test shapes
(24081, 20)
(24081,)
```

In [11]:

```
def plotInputOutput(t, input, output, index=0):
    fig = plt.figure()
    step_t = t[len(t) - 1] - t[len(t) - 2]
    next_t = t[len(t) - 1] + step_t
    plt.xlim([0, next_t + 3 * step_t])
    plt.hold(True)
    plt.plot(t, input[index], 'b-')
    plt.plot([next_t], output[index], 'r.')
    plt.hold(False)
    plt.show()

t = np.arange(0, 20) / 20.0

plotInputOutput(t, XshufTrain, yShufTrain)
```



In [12]:

```
def coefficientOfDetermination(realTargets, predictions):
    matrix = np.corrcoef(realTargets, predictions)
    cc = matrix[1,0]
    assert round(matrix[0,1],12) == round(matrix[1,0],12)
    return cc**2
```

In [13]:

```
# w, residuals, rank, s = np.linalg.lstsq(XshufTrain, yShufTrain)
# print "coefficient of determination seems very good for the training data"
# print coefficientOfDetermination(realTargets=yShufTrain, predictions=XshufTrai
n.dot(w))
# print "let's try for the validation set instead"
# print coefficientOfDetermination(realTargets=yShufVal, predictions=XshufVal.do
t(w))
# print "and the testing set also"
# print coefficientOfDetermination(realTargets=yShufTest, predictions=XshufTest.
dot(w))
# print "coefficient of determination is good after all"
```

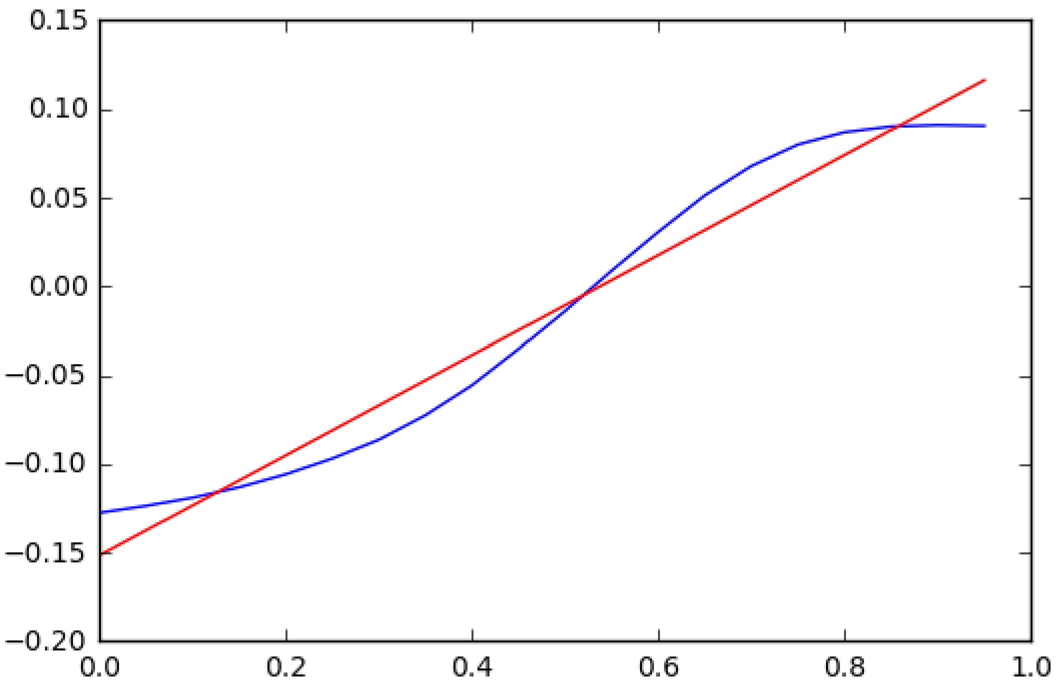
In [14]:

```
sampleInput = XshufTrain[0]
sampleOutput = yShufTrain[0]

#print .shape
def getWeightsFromCurRow(feature, output):
    X = np.hstack( (np.ones(len(feature))[np.newaxis].T, feature[np.newaxis].T)
)
    w, residuals, rank, s = np.linalg.lstsq(X, output)
    return w

w = getWeightsFromCurRow(t, sampleInput)
print w
```

```
[-0.15148097  0.2818397 ]
```

In [15]:

```
y = w[0] + t*w[1]

plt.figure()
plt.hold(True)
plt.plot(t, sampleInput, 'b-')
plt.plot(t, y, 'r-')
plt.hold(False)
plt.show()
```



In [16]:

```
def getPredictionForNextAmp(feature, output):
    w = getWeightsFromCurRow(feature, output)
    return w[0] + 1*w[1]
```

In [17]:

```
print getPredictionForNextAmp(t, sampleInput)
print sampleOutput
```

```
0.1303587261
0.0888061523438
```

In [18]:

```
#now let's do it for all sample
predictions = []
for i in range(len(XshufTrain)):
    predictions.append(
        getPredictionForNextAmp(t, XshufTrain[i])
    )
```

In [19]:

```
assert len(predictions) == len(yShufTrain)
print len(predictions)
```

```
112377
```

In [20]:

```
print "coeff of determination when using all 20 values"
coefficientOfDetermination(realTargets=yShufTrain, predictions=predictions)
```

```
coeff of determination when using all 20 values
```

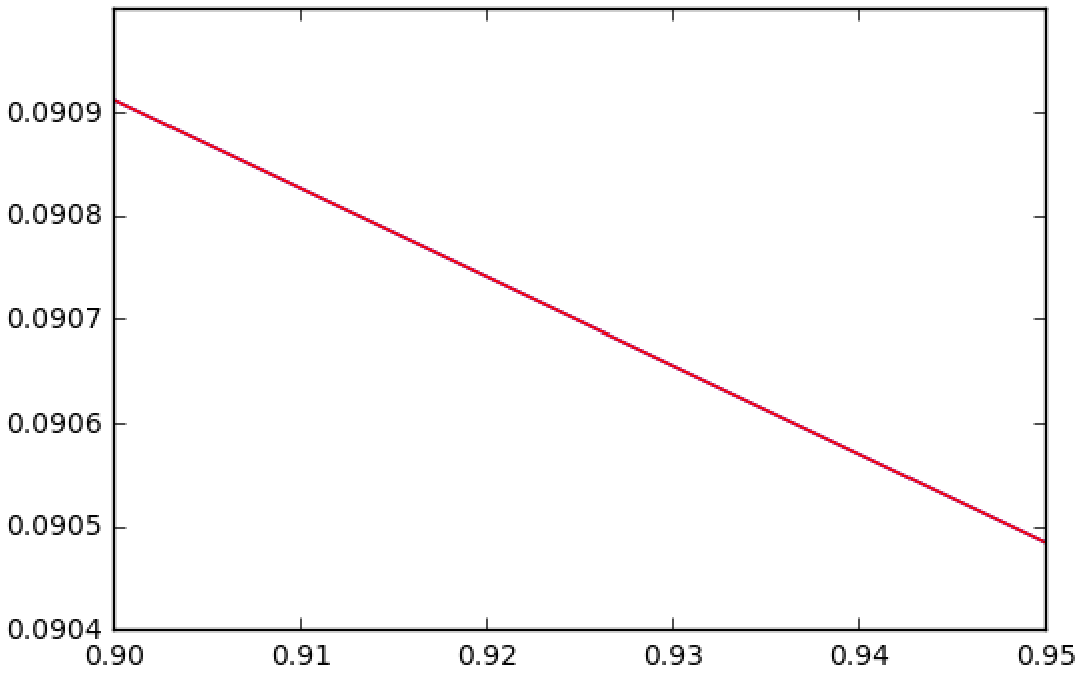
Out[20]:

```
0.76488519663577159
```

In [21]:

```
#Now let's try and use only the last two values as suggested
shortTime = t[len(t)-2:len(t)]
lastTwoAmps = XshufTrain[:, len(t)-2:len(t)]
print lastTwoAmps.shape
print lastTwoAmps.shape

w = getWeightsFromCurRow(shortTime, lastTwoAmps[0])

line = w[0] + shortTime*w[1]

plt.figure()
plt.hold(True)
plt.plot(shortTime, lastTwoAmps[0], 'b-')
plt.plot(shortTime, line, 'r-')
plt.hold(False)
plt.show()
```

```
(112377, 2)
(112377, 2)
```



Obviously when looking only at two points then these two samples form a line and the best fitted line is that same line! That's why here the blue line (actual data) and the red line (fitted line) fall on top of each other and you will be able to see only one of them at the plot above

```
In [22]:
```

```
#let's make our predictions based only on the last two points in time for every
 sample

predictions = []
for i in range(len(lastTwoAmps)):
    predictions.append(
        getPredictionForNextAmp(shortTime, lastTwoAmps[i])
    )

len(predictions)
```

```
Out[22]:

112377
```

```
In [23]:
```

```
print "coeff of determination when using only the last 2 values"
coefficientOfDetermination(realTargets=yShufTrain, predictions=predictions)
```

```
coeff of determination when using only the last 2 values

Out[23]:

0.98571503302754004
```

2.b explanation

The problem here when trying to predict the next amplitude using only as input the time is that there is no real correlation between the input which is time, which we picked arbitrarily to be between [0, 1), and the output which is the amplitude.

The correlation is between amplitudes that are close to the time axis. We are trying to guess the next amplitude based on the previous ones. The issue here is that the amplitudes change with very high frequency and our sampling frequency is low. We do not want to increase the sampling frequency because this would defeat the purpose of compression, we would actually make the audio file bigger!
So when amplitudes change with high frequency you could have a large variation of them within the short period of time, of 20 samples, that we are trying to make the predictions.
Linear regression is very sensitive to these amplitudes which are away from the desired ones.

Comparing the coefficient of determination for the 20 values and the 2 values case we see that for 2 values the performace is very superior. We are much closer to the next value and therefore better able to determine it.
The cos here is that we would need higher computational power to guess amplitudes for every two samples!

2.c

```
In [24]:
```

```
X = np.hstack((
        np.ones(len(t))[np.newaxis].T,
        t[np.newaxis].T,
        t[np.newaxis].T**2,
        t[np.newaxis].T**3,
        t[np.newaxis].T**4,
    ))

X.shape
```

```
Out[24]:

(20, 5)
```

First of all to answer in theory why we might want to use a longer context than for the straight line model is because we have too many unknowns and we would not be able to solve for this equation if we had only two points. Because now we have w0, w1, w2, w3, w4, five unknowns. Therefore we need at least five equations to solve this linear algebra problem

```
In [25]:
```

```
#this case works fine
sample = XshufTrain[0]
w, residuals, rank, s = np.linalg.lstsq(X, sample)
print w
```

```
[-0.12507053  0.00214574  0.34093803  0.55871645 -0.7102236 ]
```

```
In [26]:
```

```
#Let's try the case for only the two last parts of the X and see if this is goin
g to work
shortSample = XshufTrain[0][XshufTrain.shape[1]-2:]
print shortSample.shape
shortX = X[len(X)-2:,:]
print shortX.shape

np.linalg.lstsq(shortX, shortSample) #TODO find out why this does not crash as i
t was expected
#http://stats.stackexchange.com/questions/240573/how-numpy-solves-least-squares-
with-insufficient-data
```

```
(2,)
(2, 5)
```

```
Out[26]:

(array([ 0.06177781,  0.03656294,  0.0148214 , -0.00384172, -0.01977
947]),
 array([], dtype=float64),
 2,
 array([ 2.74462275,  0.10282372]))
```

In [27]:

```
def getPredictionForNextAmp(matrix, output):
    w, residuals, rank, s = np.linalg.lstsq(matrix, output)
    time = 1
    return w[0] + time*w[1] + (time**2)*w[2] + (time**3)*w[3] + (time**4)*w[4]
```

In [28]:

```
getPredictionForNextAmp(X, sample)
```

Out[28]:

```
0.06650609743730429
```

In [29]:

```
predictions = []
for i in range(len(lastTwoAmps)):
    predictions.append(
        getPredictionForNextAmp(X, XshufTrain[i])
    )

len(predictions)
```

Out[29]:

```
112377
```

In [30]:

```
print "coeff of determination when using all 20 values on a polynomial of 1 + t
 + t^2 + t^3 + t^4"
coefficientOfDetermination(realTargets=yShufTrain, predictions=predictions)
```

```
coeff of determination when using all 20 values on a polynomial of 1
 + t + t^2 + t^3 + t^4
```

Out[30]:

```
0.88557028683778172
```

From the above plots and only judging by eye I believe that since the curves have two inflection points then it seems that a polynomial of t^3 would be the best fit.

3.a

We have that f(t=1) = w^T * φ(t=1)

We also have that w = (Φ^T *Φ)^-1* Φ^T*x

And we want to derive f(t=1) = v^T * x

φ(t = 1) = [1^0 1^1 1^2 1^3 ...] = [1 1 1 1 ....]

f(t=1) is a single element 1x1 and it is valid to say f(t=1)^T = f(t=1)

So f(t=1)^T = ((w^T) * φ(t=1))^T =>

f(t=1) = φ(t=1)^T *(w^T)^T = φ(t=1)^T* w = φ(t=1)^T *(Φ^T* Φ)^-1 *Φ^T*x = (((Φ^T *Φ)^-1* Φ^T)^T *φ(t=1))^T* x

=> f(t=1) = (Φ*(Φ^T* Φ)^-T *φ(t=1))^T* x =>

v = Φ*(Φ^T* Φ)^-T * φ(t=1)

But let's verify that the theory above is correct by doing some calculations to see if we derive to the same result

In [31]:

```
H = np.hstack((
        np.ones(len(t))[np.newaxis].T,
        t[np.newaxis].T,
        t[np.newaxis].T**2,
        t[np.newaxis].T**3,
        t[np.newaxis].T**4,
    ))

H.shape
```

Out[31]:

```
(20, 5)
```

In [32]:

```
v = H.dot(np.linalg.inv(H.T.dot(H)).T)

v.shape
```

Out[32]:

```
(20, 5)
```

In [33]:

```
x = XshufTrain[0]

x.shape
```

Out[33]:

```
(20,)
```

In [34]:

```
w = v.T.dot(x)
f_tEqualsOne = w.dot(np.ones(len(w)))
print f_tEqualsOne
```

```
0.0665060974369
```

In [35]:

```
def getPredictionForNextAmp(matrix, output):
    w, residuals, rank, s = np.linalg.lstsq(matrix, output)
    time = 1
    return w[0] + time*w[1] + (time**2)*w[2] + (time**3)*w[3] + (time**4)*w[4]
```

In [36]:

```
computedPrediction = getPredictionForNextAmp(H, x)

computedPrediction
```

Out[36]:

```
0.06650609743730429
```

In [37]:

```
assert round(f_tEqualsOne,12) == round(computedPrediction,12)
```

3.b

In [38]:

```
def getTimeMatrix(C, K):
    """C are the time steps and K is the length of the polynomial of (K-1) orde
r"""
    assert C >= 1
    assert K >= 0

    ones = np.ones(C)[np.newaxis].T

    if K == 0:
        return ones
    else:
        t = (np.arange(0, C) / float(C))[np.newaxis].T
        #matrix = np.hstack((ones, t))
        matrix = ones
        for k in range(1,K):
            matrix = np.hstack((matrix, t**k))
        return matrix
```

In [39]:

```
getTimeMatrix(4, 5)
```

Out[39]:

```
array([[ 1.        ,  0.        ,  0.        ,  0.        ,  0.
   ],
       [ 1.        ,  0.25      ,  0.0625    ,  0.015625  ,  0.00390
625],
       [ 1.        ,  0.5       ,  0.25      ,  0.125     ,  0.0625
   ],
       [ 1.        ,  0.75      ,  0.5625    ,  0.421875  ,  0.31640
625]])
```

In [40]:

```
def get_v(H):
    return H.dot(np.linalg.inv(H.T.dot(H)).T).dot(np.ones(H.shape[1])[np.newaxis
T)
```

In [41]:

```
get_v(getTimeMatrix(10, 5)).shape
```

Out[41]:

```
(10, 1)
```

In [42]:

```
x = XshufTrain[0]
get_v(getTimeMatrix(len(x), 5)).T.dot(x)
```

Out[42]:

```
array([ 0.0665061])
```

3.c

In [43]:

```
def getPredictions(K, X):
    C = len(X) if len(X.shape) == 1 else X.shape[0]
    return get_v(getTimeMatrix(C, K)).T.dot(X)[0]
    #return get_v(getTimeMatrix(C, K)).dot(X)[0]
```

In [44]:

```
getPredictions(5, XshufTrain[0])
```

Out[44]:

```
0.066506097436917599
```

In [45]:

```
getPredictions(0, XshufTrain.T).shape
```

Out[45]:

```
(112377,)
```

In [46]:

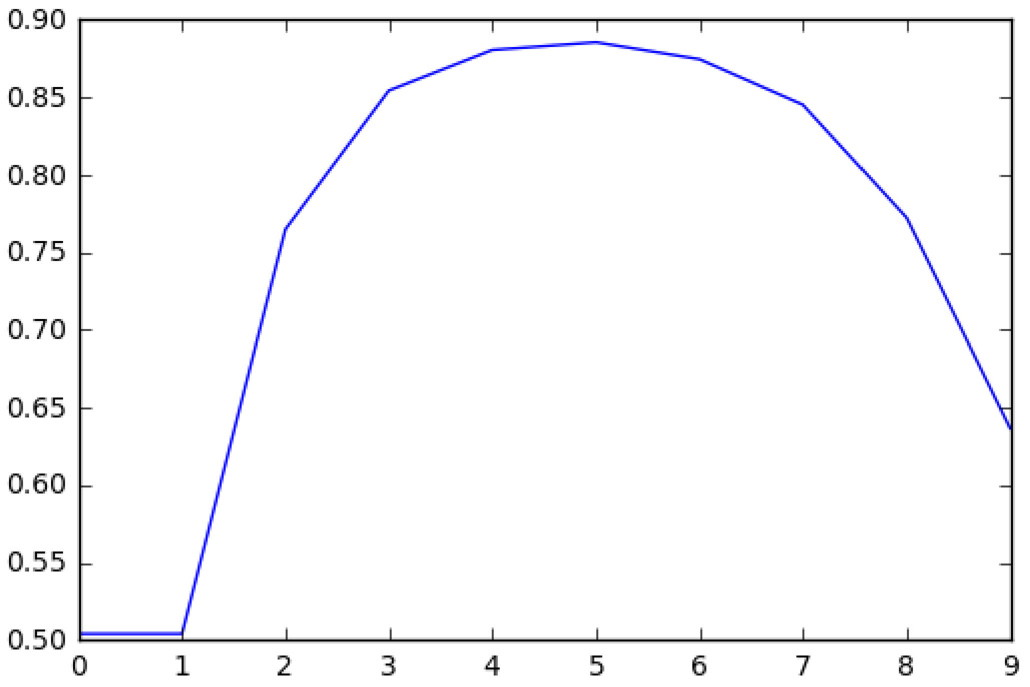
```
#Let's try to see what happens with different values of K for the validation set
scoreDict = dict()
for k in range(10):
    curScore = coefficientOfDetermination(realTargets=yShufTrain, predictions=ge
tPredictions(k, XshufTrain.T))
    #print curScore
    scoreDict[k] = curScore

scoreDict
```

Out[46]:

```
{0: 0.5042779078778995,
 1: 0.5042779078778995,
 2: 0.76488519663577159,
 3: 0.85438140231560333,
 4: 0.88058676171890993,
 5: 0.88557028683662542,
 6: 0.87464261043181557,
 7: 0.84527227367471103,
 8: 0.77232268004229709,
 9: 0.63685611775133788}
```

In [47]:

```
plt.figure()
plt.plot(scoreDict.keys(), scoreDict.values(), 'b-')
plt.show()
```



The above analysis results that for K=5 we have the best prediction

let's do the above analysis for the validation set

first we need to get the vector **v** from the training set

In [48]:

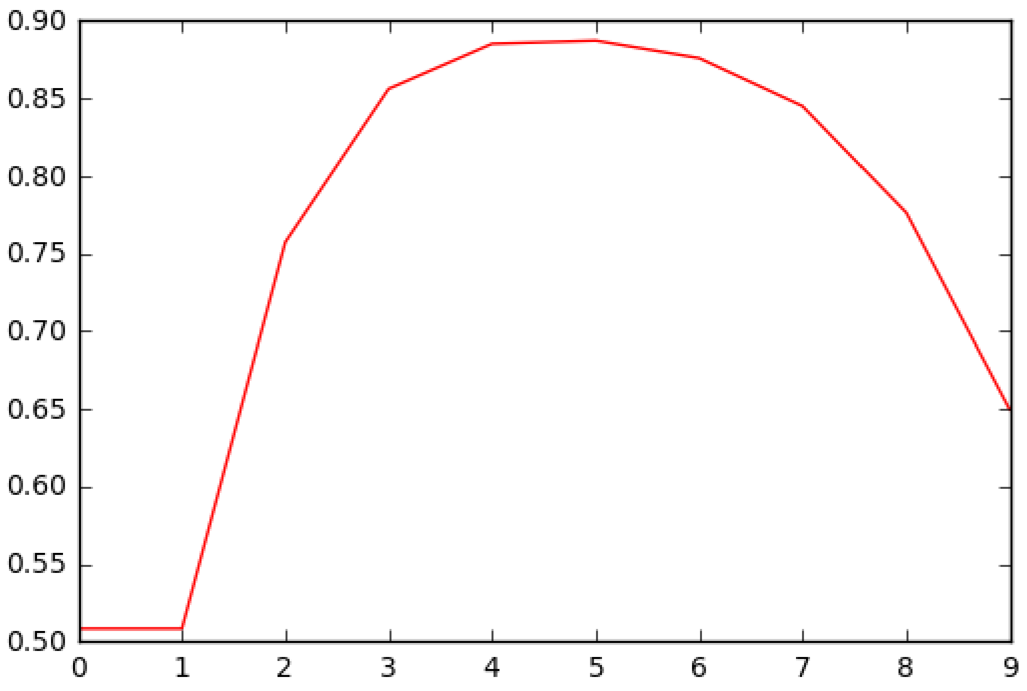
```
scoreDict = dict()
for k in range(10):
    C = XshufTrain.shape[1]
    v = get_v(getTimeMatrix(C, k))
    preds = v.T.dot(XshufVal.T)[0]
    scoreDict[k] = coefficientOfDetermination(realTargets=yShufVal,
predictions=preds)

scoreDict
```

Out[48]:

```
{0: 0.50852237074290396,
 1: 0.50852237074290396,
 2: 0.75736073417723393,
 3: 0.85620687397486916,
 4: 0.88523259986407998,
 5: 0.88730227578787735,
 6: 0.8760135529881542,
 7: 0.84511572303541338,
 8: 0.77642882797185575,
 9: 0.64980997359583026}
```

In [49]:

```
plt.figure()
plt.plot(scoreDict.keys(), scoreDict.values(), 'r-')
plt.show()
```



The analysis for validation set happens to result the same thing. For 20 samples the polynomial of K-1=5-1=4th order is the best

In [50]:

```
def getRSS(trueTargets, predictions):
    assert len(trueTargets.shape) == 1, "we want true targets to be a vector"
    assert len(predictions.shape) == 1, "we want predictions to be a vector"
    length = len(trueTargets)
    assert length == len(predictions)
    return np.sum((trueTargets - predictions)**2) / float(length)
```

In [51]:

```
#Let's do the same but now using RSS
trainScores = dict()
valScores = dict()
C = XshufTrain.shape[1]
for k in range(10):
    v = get_v(getTimeMatrix(C, k))
    trainPreds = v.T.dot(XshufTrain.T)[0]
    valPreds = v.T.dot(XshufVal.T)[0]
    trainScores[k] = getRSS(trueTargets=yShufTrain, predictions=trainPreds)
    valScores[k] = getRSS(trueTargets=yShufVal, predictions=valPreds)

np.hstack((
        np.arange(k+1)[np.newaxis].T,
        np.array(trainScores.values())[np.newaxis].T,
        np.array(valScores.values())[np.newaxis].T,
    ))
```

Out[51]:

```
array([[  0.00000000e+00,   1.37236907e-03,   1.39226820e-03],
       [  1.00000000e+00,   1.37236907e-03,   1.39226820e-03],
       [  2.00000000e+00,   7.91637800e-04,   8.35961926e-04],
       [  3.00000000e+00,   5.30891037e-04,   5.26368483e-04],
       [  4.00000000e+00,   4.02128860e-04,   3.87018052e-04],
       [  5.00000000e+00,   3.53808358e-04,   3.61046807e-04],
       [  6.00000000e+00,   3.77158008e-04,   3.85897071e-04],
       [  7.00000000e+00,   4.84179853e-04,   4.84852523e-04],
       [  8.00000000e+00,   7.90229892e-04,   7.84813201e-04],
       [  9.00000000e+00,   1.55556858e-03,   1.51230063e-03]])
```

In [52]:

```
print "smallest RMSE for training was for k:"
trainMin = np.argmin(trainScores.values())
print trainMin
print "with value"
print trainScores[trainMin]
print
print "smallest RMSE for validation was for k:"
valMin = np.argmin(valScores.values())
print valMin
print "with value"
print valScores[valMin]
```

```
smallest RMSE for training was for k:
5
with value
0.00035380835799

smallest RMSE for validation was for k:
5
with value
0.000361046807091
```

4.a

In [53]:

```
#first we need to get the V
def getScoresForPolynomials(C, X, y):
    scoreDict = dict()
    for k in range(10):
        v = get_v(getTimeMatrix(C, k))
        preds = v.T.dot(X.T)[0]
        scoreDict[k] = getRSS(trueTargets=y, predictions=preds)

    return scoreDict
```

In [54]:

```
getScoresForPolynomials(C = XshufTrain.shape[1], X = XshufTrain,y = yShufTrain)
```

Out[54]:

```
{0: 0.0013723690705458166,
 1: 0.0013723690705458166,
 2: 0.0007916377998583951,
 3: 0.00053089103667260692,
 4: 0.00040212886044472797,
 5: 0.00035380835798965316,
 6: 0.00037715800815934001,
 7: 0.00048417985305404808,
 8: 0.00079022989203204332,
 9: 0.0015555685767366851}
```

In [55]:

```
def getClosedFormSolution(H, y):
    return np.linalg.inv(H.T.dot(H)).dot(H.T).dot(y)
```

In [56]:

```
C = 20
X = XshufTrain
y = yShufTrain

v = getClosedFormSolution(X, y)[np.newaxis].T
print v.shape

#for c in range(1,C+1):
#    curX = Xshuf
preds = v.T.dot(X.T)
preds.shape
```

```
(20, 1)
```

Out[56]:

```
(1, 112377)
```

In [57]:

```
def calcRSS(X, y):
    v = getClosedFormSolution(X, y)[np.newaxis].T
    preds = v.T.dot(X.T)[0]
    return getRSS(trueTargets=y, predictions=preds)
```

In [58]:

```
def getLastColumnsOfMatrix(C, X):
    return X[:, X.shape[1]-C:]
```

In [59]:

```
def calcRSSforAssignment(C, X, y):
    curX = getLastColumnsOfMatrix(C, X)
    return calcRSS(curX, y)
```

In [60]:

```
#for c in range(1, C+1):
calcRSSforAssignment(3, XshufTrain, yShufTrain)
```

Out[60]:

```
3.0570826984733332e-05
```

```
In [61]:
```

```
C=20
trainScores = dict()
valScores = dict()
for c in range(1, C+1):
    trainX = getLastColumnsOfMatrix(c, XshufTrain)
    valX = getLastColumnsOfMatrix(c, XshufVal)

    v = getClosedFormSolution(trainX, yShufTrain)[np.newaxis].T

    trainPreds = v.T.dot(trainX.T)[0]
    valPreds = v.T.dot(valX.T)[0]

    trainScores[c] = getRSS(trueTargets=yShufTrain, predictions=trainPreds)
    valScores[c] = getRSS(trueTargets=yShufVal, predictions=valPreds)

np.hstack((
        np.arange(1,C+1)[np.newaxis].T,
        np.array(trainScores.values())[np.newaxis].T,
        np.array(valScores.values())[np.newaxis].T,
    ))
```

```
Out[61]:
```

```
array([[  1.00000000e+00,   6.66787094e-05,   6.76561309e-05],
       [  2.00000000e+00,   3.29619428e-05,   3.26775886e-05],
       [  3.00000000e+00,   3.05708270e-05,   3.00241458e-05],
       [  4.00000000e+00,   2.84490540e-05,   2.76139871e-05],
       [  5.00000000e+00,   2.72817907e-05,   2.65276043e-05],
       [  6.00000000e+00,   2.57852710e-05,   2.56089750e-05],
       [  7.00000000e+00,   2.36135167e-05,   2.37171740e-05],
       [  8.00000000e+00,   2.31610776e-05,   2.28621332e-05],
       [  9.00000000e+00,   2.27563853e-05,   2.21295569e-05],
       [  1.00000000e+01,   2.23604814e-05,   2.15541984e-05],
       [  1.10000000e+01,   2.21697357e-05,   2.09979899e-05],
       [  1.20000000e+01,   2.18841327e-05,   2.03629976e-05],
       [  1.30000000e+01,   2.15710924e-05,   2.02063206e-05],
       [  1.40000000e+01,   2.14608583e-05,   2.02012370e-05],
       [  1.50000000e+01,   2.12595790e-05,   2.00954098e-05],
       [  1.60000000e+01,   2.10831975e-05,   2.01309012e-05],
       [  1.70000000e+01,   2.08470341e-05,   1.99499499e-05],
       [  1.80000000e+01,   2.07921461e-05,   1.98938105e-05],
       [  1.90000000e+01,   2.07441795e-05,   1.98371707e-05],
       [  2.00000000e+01,   2.07106605e-05,   1.98582280e-05]])
```

In [62]:

```
print "smallest RMSE for training was for C:"
trainMin = np.argmin(trainScores.values())
print trainScores.keys()[trainMin]
print "with value"
print trainScores.values()[trainMin]
print
print "smallest RMSE for validation was for C:"
valMin = np.argmin(valScores.values())
print valScores.keys()[valMin]
print "with value"
print valScores.values()[valMin]
```

```
smallest RMSE for training was for C:
20
with value
2.07106605182e-05

smallest RMSE for validation was for C:
19
with value
1.98371706823e-05
```

4.b

From the analysis above we can clearly see that using the closed form solution yields to a much better model for the current data we have at hand.

For validation it peaks at 19 previous values.

Here we are not using at all the time as an input to predict values at output.
We rather take advantage of the amplitudes from previous steps and we are using them to predict the next step. This seems to work much better according to the RMSE metric

In [64]:

```
bestRMSEq3 = 0.000361046807091
print "The best RMSE for question 3 is: "
print bestRMSEq3
print
bestRMSEq4 = 1.98371706823e-05
print "The best RMSE for question 4 is: "
print bestRMSEq4
print
print (1/bestRMSEq4) / (1/bestRMSEq3)
```

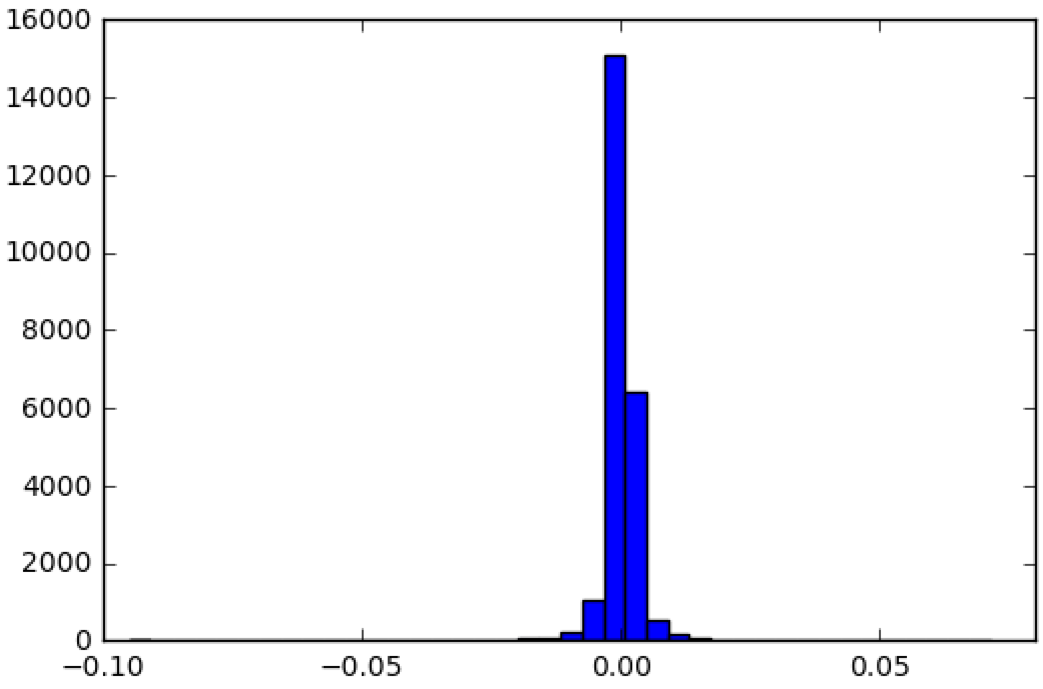
```
The best RMSE for question 3 is:
0.000361046807091

The best RMSE for question 4 is:
1.98371706823e-05

18.2005192612
```

It seems that our method at question 4 yields an 18 times better RMSE

4.c

In [80]:

```
plt.figure()
plt.title('Histogram of residuals on the validation data for best model')
plt.hist(np.log(yShufVal - valPreds + 1), bins=40)
plt.show()
```



We see that the histogram of the residuals is similar to the histogram of the amplitudes.
There are many amplitudes which are near zero that's why the histogram peaks over there.
Similarly for a good model most of the residuals are near to zero, meaning that our predictions are good, that's why this histogram peaks again near zero.

5.

Perhaps instead of predicting the amplitudes as amplitudes we could transform the input data by using a sin (sinusoid). Perhaps there are some periodicity of the amplitudes that could help us derive better predictions.

In [ ]:

```

```