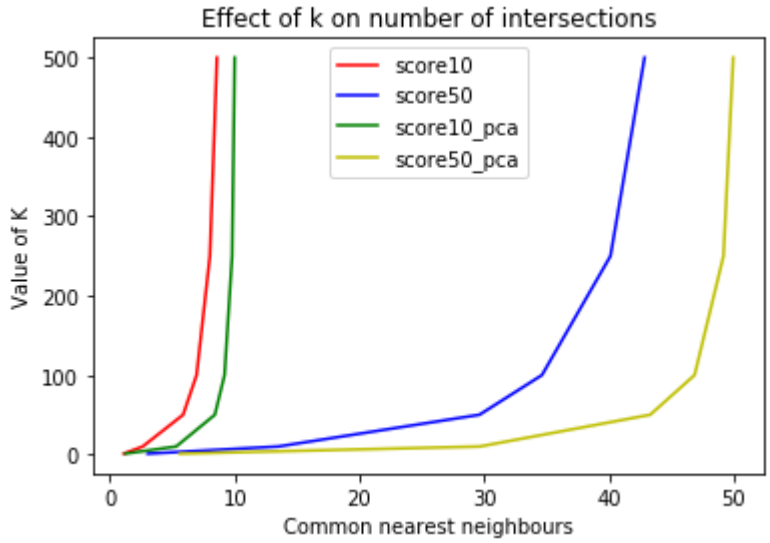
In [15]:
```
import pandas as pd
from sklearn.neighbors import NearestNeighbors
import numpy as np
from sklearn.decomposition import PCA
%pylab inline
import matplotlib.pyplot as plt

def convert(imgf, n):
    f = open(imgf, "rb")

    f.read(16)
    images = []

    for i in range(n):
        image=[]
        for j in range(28*28):
            image.append(ord(f.read(1)))
        images.append(image)
    df=pd.DataFrame(images)
    f.close()
    return df

def Neighbours(n,X):
    nbrs = NearestNeighbors(n_neighbors=n, algorithm='ball_tree',metric='eucli
dean').fit(X)
    distances, indices = nbrs.kneighbors(X)
    return distances, indices

def Intersection(array1, array2):
    Intersection = np.empty([ array1.shape[0]])
    for i in range(0, array1.shape[0]):
        Intersection[i] = len( set(array1[i]).intersection(array2[i]) )
    return Intersection

def DifferentK_random(k):
    R=np.random.randn(k, 784)
    R=pd.DataFrame(R)
    Xnew=R.dot(X)
    Xnew=pd.DataFrame(Xnew)
    distances10_new, indices10_new=Neighbours(10,Xnew.T)
    distances50_new, indices50_new=Neighbours(50,Xnew.T)
    intersect10=Intersection(indices10,indices10_new)
    intersect50=Intersection(indices50,indices50_new)
    score10=intersect10.sum()/n
    score50=intersect50.sum()/n
    return score10,score50

def DifferentK_PCA(k):
    pca = PCA(n_components=k)
    Xnew=pca.fit_transform(X.T, y=None)
    Xnew=pd.DataFrame(Xnew)
    distances10_new, indices10_new=Neighbours(10,Xnew)
    distances50_new, indices50_new=Neighbours(50,Xnew)
    intersect10=Intersection(indices10,indices10_new)
    intersect50=Intersection(indices50,indices50_new)
    score10=intersect10.sum()/n
```

```
    score50=intersect50.sum()/n
    return score10,score50

df=convert("t10k-images.idx3-ubyte", 10000)
df=(df.T)
n=2000
X=df.iloc[:,0:n]
distances10, indices10=Neighbours(10,X.T)
distances50, indices50=Neighbours(50,X.T)
k=[1,10,50,100,250,500]
score10=[]
score50=[]
score10_pca=[]
score50_pca=[]
for i in k:
    x,y=DifferentK_random(i)
    score10.append(x)
    score50.append(y)
for i in k:
    x,y=DifferentK_PCA(i)
    score10_pca.append(x)
    score50_pca.append(y)

plot(score10, k, 'r',label='score10')
plot(score50, k, 'b',label='score50')
plot(score10_pca, k, 'g',label='score10_pca')
plot(score50_pca, k, 'y',label='score50_pca')

xlabel('Common nearest neighbours')
ylabel('Value of K')
title('Effect of k on number of intersections')

legend(loc=0)
```

```
Populating the interactive namespace from numpy and matplotlib
```

Out[15]: <matplotlib.legend.Legend at 0x11b2bdd0cc0>



Q1 E)

As the value of k increases the error decreases which is exactly what lemma states (k=O(log(n)/$\varepsilon^2$ ) i.e. k is inversely proportional to the $\varepsilon^2$ hence as k increases relative error quadratically decreases. Higher value of K also means a greater number of random projections and a greater number of components, which is why it is logical to have less error.

Q1 F)

The nearest neighbor approximations generated via PCA are better than those generated via random projections because components in PCA accounts for maximum variance. If we take 784 PCA components then the total variance of the new data will remain the same, it is distributed in unequal fashion such that the 1st component accounts for maximum variance, second component for 2nd largest variance and so on.

Q2 A)

The number of free parameters is l*d + d,

l*d - from matrix B.

d – from vector $\varepsilon$.

B)

$$\Sigma = BB^T + D$$

Which can also be written as: $\Sigma = D + BIB^T$

Which is of the form: A + UCV

So, by Woodbury matrix inversion lemma:

$\Sigma^{-1} = \mathrm{D}^{-1} - \mathrm{D}^{-1}B(I + \mathrm{B}^T\mathrm{D}^{-1}B)^{-1}\mathrm{B}^T\mathrm{D}^{-1}$ ……. (known $\mathrm{I}^{-1} = I$)

D is a diagonal matrix hence its inverse is easy to compute and will only cost d operations.

The only inverse remaining to compute is $(I + \mathrm{B}^T\mathrm{D}^{-1}B)^{-1}$ which is the inverse of an l*l matrix.

C) Mahalanobis distance without storing $\Sigma^{-1}$.

Idea: To calculate column by column of $\Sigma^{-1}$ and directly compute column by column of $(\underset{x}{\rightarrow} - \underset{\mu}{\rightarrow})^T * (\Sigma^{-1})$ without storing $\Sigma^{-1}$ in memory.

Explaination:

Steps:

Store B - (d*l) and D(one dimensional array of diagonal elements) in the memory

1)Compute: $\mathrm{D}^{-1}B$ = M which is d*l and store it into the memory.

This also gives us the value of $B^TD^{-1} = M^T$ so we don't have to store $M^T$ separately if we have M.

2)Compute: $(I + B^T M)^{-1} = Q$ which is l*l and store it in the memory.

Now we have to compute: $(\mathrm{D}^{-1} - MQM^T)$

The only matrices which we need in the memory for this is D (one dimensional array of d elements) M (d*l) and Q(l*l).

3)

Compute MQ fully which is d*l and multiply it with a first column vector of with $M^T$ this will give you a column vector, subtract that from 1st column vector of D and result will be a column vector.

4)

$(\underset{x}{\rightarrow} - \underset{\mu}{\rightarrow})^T$*(result from step 3) = a single column vector

5) Repeat step 3 and 4 till you get an entire matrix $(\underset{x}{\rightarrow} - \underset{\mu}{\rightarrow})^T$*$(\Sigma^{-1})$ which is n*d. Store it in the memory.

6) Then multiply result of step 5 with $(\underset{x}{\rightarrow} - \underset{\mu}{\rightarrow})$