sheet2

October 27, 2017

1 Exercise Sheet 2: Timing, Numpy, Plotting

The previous exercise sheet introduced several methods for classification: decision trees, nearest neighbors, and nearest means. Of those, the one that could learn from the data, and that also offered enough complexity to produce an accurate decision function was k-nearest neighbors. However, nearest neighbors can be slow when implemented in pure Python (i.e. with loops). This is especially the case when the number of data points or input dimensions is large.

In this exercise sheet, we will speed up nearest neighbors by utilizing `numpy` and `scipy` packages. Your task will be to replace list-based operations by vector-based operations between numpy arrays. The speed and correctness of the implementations will then be tested. In particular, performance graphs will be drawn using the library `matplotlib`.

1.1 Python Nearest Neighbor

The most basic element of computation of nearest neighbors is its distance function relating two arbitrary data points `x1` and `x2`. We assume that these points are iterable (i.e. we can use a loop over their dimensions). One way among others to compute the square Euclidean distance between two points is by computing the sum of the component-wise distances

```
In [1]: def pydistance(x1,x2):
            return sum([(x1d-x2d)**2 for x1d,x2d in zip(x1,x2)])
```

where the prefix "`py-`" of the function indicates that the latter makes use of `Python` instead of `numpy`. Once the distance matrix has been implemented, the nearest neighbor for a given unlabeled point `u` that we would like to classify is obtained by iterating over all points in the training set `(X,Y)`, selecting the point with smallest distance to `u`, and returning its corresponding label. Here `X` denotes the list of inputs in the training set and `Y` denotes the list of labels.

```
In [2]: def pynearest(u,X,Y,distance=pydistance):

            xbest = None
            ybest = None
            dbest = float('inf')

            for x,y in zip(X,Y):
                d = distance(u,x)
                if d < dbest:
                    ybest = y
```

```
                xbest = x
                dbest = d

        return ybest
```

Note that this function either uses function `pydistance` (given as default if the argument distance is not specified). Or one could specify as argument a more optimized function for distance compuation, for example, one that uses `numpy`. Finally, one might not be interested in classifying a single point, but many of them. The method below receives a collection of such unlabeled test points stored in the variable `U`. The function returns a list of predictions associated to each test point.

```
In [3]: def pybatch(U,X,Y,nearest=pynearest,distance=pydistance):
            return [nearest(u,X,Y,distance=distance) for u in U]
```

Again, such function uses by default the Python nearest neighbor search (with a specified distance function). However, we can also specified a more optimized nearest neighbor function, for example, based on `numpy`. Finally, one could consider an alternative function to `pybatch` that would use `numpy` from the beginning to the end. The implementation of such more optimized functions, and the testing of their correct behavior and higher performance will be the object of this exercise sheet.

1.2 Testing and correctness

As a starting point, the code below tests the output of the nearest neighbor algorithm for some toy dataset with fixed parameters. In particular, the function `data.toy(M,N,d)` generates a problem with `M` unlabeled test points stored in a matrix `U` of size `(M x d)`, then `N` labeled training points stored in a matrix `X` of size `(N x d)` and the output label is stored in a vector `Y` of size `N` composed of zeros and ones encoding the two possible classes. The variable `d` denotes the number of dimensions of each point. The toy dataset is pseudo-random, that is, for fixed parameters, it produce a random-looking dataset, but every time the method is called with the same parameters, the dataset is the same. The pseudo-randomness property will be useful to verify that each nearest neighbor implementation performs the same overall computation.

```
In [4]: import data
        U,X,Y = data.toy(20,100,50)
        print(pybatch(U,X,Y))
```

```
[1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0]
```

In particular, the output of this function will help us to verify that the more optimized `numpy`-based versions of nearest neighbor are still valid.

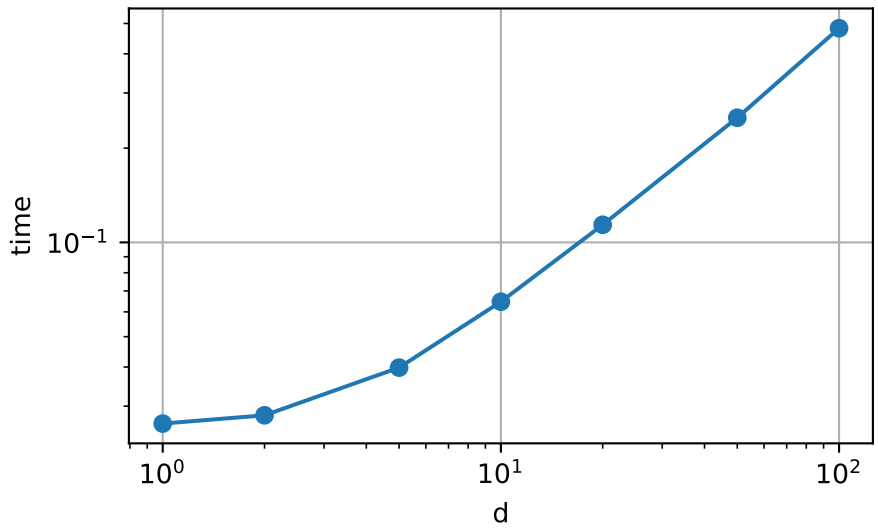
1.3 Plotting and performance

We now describe how to build a plot that relates a certain parameter of the dataset (e.g. the number of input dimensions `d` to the time required for the computation. We first initialize the basic plotting environment.

```
In [5]: import matplotlib
        from matplotlib import pyplot as plt
        %matplotlib inline
        from IPython.display import set_matplotlib_formats
        set_matplotlib_formats('pdf', 'png')
        plt.rcParams['savefig.dpi'] = 90
```

The command "`%matplotlib inline`" tells IPython notebook that the plots should be rendered inside the notebook. The following code plots the computation time of predicting 100 points from the test set using a training set of size 100, and where we vary the number of input dimensions.

```
In [6]: import time

        # Values for the number of dimensions d to test
        dlist = [1,2,5,10,20,50,100]

        # Measure the computation time for each choice of number of dimensions d
        tlist = []
        for d in dlist:
            U,X,Y = data.toy(100,100,d)
            a = time.clock()
            pybatch(U,X,Y)
            b = time.clock()
            tlist += [b-a]

        # Plot the results in a graph
        plt.figure(figsize=(5,3))
        plt.plot(dlist,tlist,'-o')
        plt.xscale('log');plt.yscale('log'); plt.xlabel('d'); plt.ylabel('time'); plt.grid(True)
```



The time on the vertical axis is in seconds. Note that the exact computation time depends on the speed of your computer. As expected, the computation time increases with the number of input dimensions. Unfortunately, for the small dataset considered here (`100` training and test points of `100` dimensions each), the algorithm already takes more than one second to execute. Thus, it is necessary for practical applications (e.g. the digit recognition task that we will consider at the end of this exercise sheet) to accelerate this nearest neighbor algorithm.

1.4 Accelerating the distance computation (25 P)

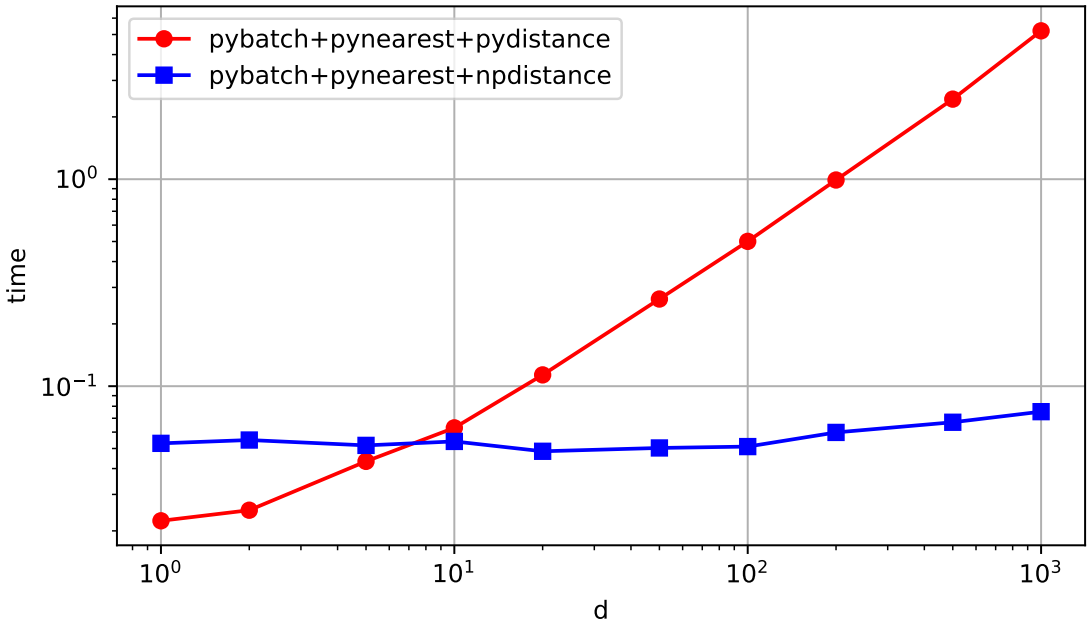
In this first exercise, we would like to accelerate the function that compute pairwise distances.

- Create a new function `npdistance(x1,x2)` with the same output as `pydistance(x1,x2)`, but that computes the squared Euclidean distance using `numpy` operations.
- Print its output for the same toy example with parameters `M=20`, `N=100`, `d=50` considered before (i.e. `data.toy(20,100,50)`). Verify that in both cases (i.e. using either `npdistance` or `pydistance` in the function `pybatch`) the output remains the same.
- Create a plot similar to the one above, but where the computation time required by both methods are shown in a superposed manner. Here, we fix `M=100`, `N=100`, and we let `d` vary from `1` to `1000`, taking the list of values `[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]`.

```
In [7]: import numpy
        def npdistance(x1,x2):
            return numpy.linalg.norm(numpy.array(x1)-numpy.array(x2))

        U,X,Y = data.toy(20,100,50)
        print((pybatch(U,X,Y,nearest=pynearest,distance=pydistance)) == (pybatch(U,X,Y,nearest=p
        print('pybatch+pynearest+pydistance',pybatch(U,X,Y,nearest=pynearest,distance=pydistance
        print('pybatch+pynearest+npdistance',pybatch(U,X,Y,nearest=pynearest,distance=npdistance

        # Values for the number of dimensions d to test
        dlist = [1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]

        # Measure the computation time for each choice of number of dimensions d
        tlist_py = []
        tlist_np = []
        for d in dlist:
            U,X,Y = data.toy(100,100,d)
            #Python
            a_py = time.clock()
            pybatch(U,X,Y,nearest=pynearest,distance=pydistance)
            b_py = time.clock()
            tlist_py += [b_py-a_py]
            #numpy
            a_np = time.clock()
            pybatch(U,X,Y,nearest=pynearest,distance=npdistance)
            b_np = time.clock()
            tlist_np += [b_np-a_np]
```

```
        # Plot the results in a graph
        plt.figure(figsize=(7,4))
        plt.plot(dlist,tlist_py,'-o',color='red',label='pybatch+pynearest+pydistance')
        plt.plot(dlist,tlist_np,'-s',color='blue',label='pybatch+pynearest+npdistance')
        plt.legend();plt.xscale('log');plt.yscale('log'); plt.xlabel('d'); plt.ylabel('time'); p
```

```
True
pybatch+pynearest+pydistance [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0]
pybatch+pynearest+npdistance [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0]
```



- Based on your results, explain what kind of speedup `numpy` provides, and in what regime do you expect the speedup to be the most important.

From the graph we can see that computational time of pure Python increases exponentially with d and using numpy the time is still almost the same even with increasing d. The speedup needs to be in the regime of distance function.

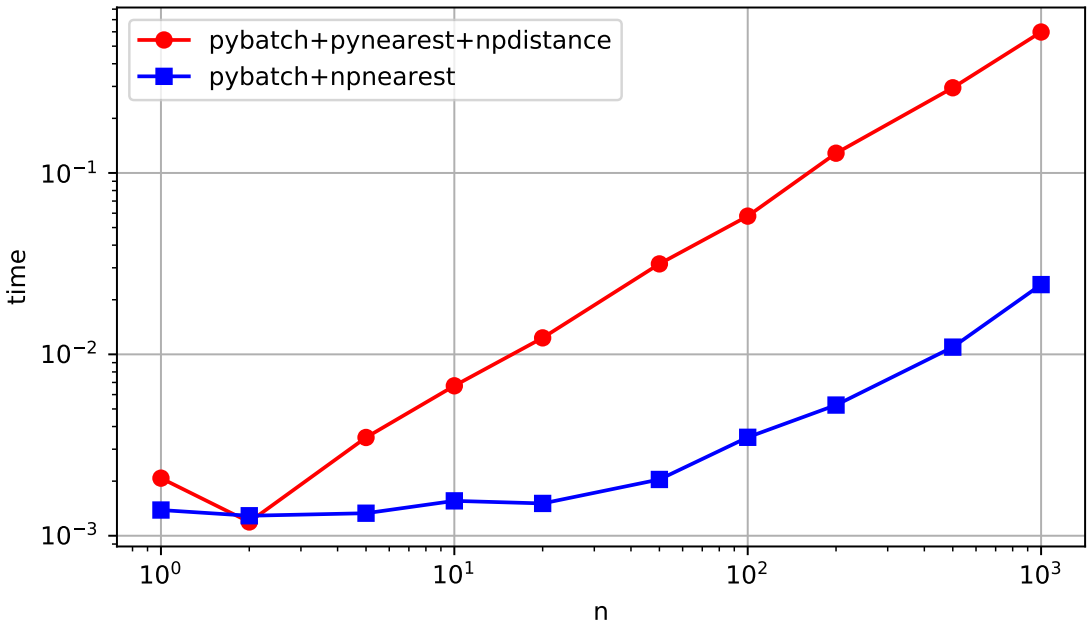
1.5 Accelerating the nearest neighbor search (25 P)

Motivated by the success of the `numpy` optimized distance computation, we would like further accelerate the code by performing nearest neighbor search directly in `numpy`.

- Create a new function `npnearest(u,X,Y)` as an alternative to the function `pynearest(u,X,Y,distance=npdistance)` that we have used in the previous exercise.

- Print its output for the same toy example as before (i.e. `data.toy(20,100,50)`). Verify that the output remains the same compared to the implementation of the previous exercise.
- Create a plot similar to the one above, where the new method is compared to the previous one. Here, we fix `M=100`, `d=100`, and we let `N` take different values `[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]`.

```
In [8]: U,X,Y = data.toy(20,100,50)
        nlist =[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]

        def npnearest(u,X,Y,distance=npdistance):
             return Y[numpy.argmin(numpy.linalg.norm(u-X,axis=1))]

        print((pybatch(U,X,Y,nearest=pynearest,distance=npdistance)) == (pybatch(U,X,Y,nearest=n
        print('pybatch+pynearest+npdistance',pybatch(U,X,Y,nearest=pynearest,distance=npdistance
        print('pybatch+npnearest           ',pybatch(U,X,Y,nearest=npnearest,distance=npdistance

        # Measure the computation time for each choice of number of dimensions d
        tlist_py = []
        tlist_np = []
        for n in nlist:
            U,X,Y = data.toy(100,n,100)
            #Python
            a_py = time.clock()
            pybatch(U,X,Y,pynearest,npdistance)
            b_py = time.clock()
            tlist_py += [b_py-a_py]
            #numpy
            a_np = time.clock()
            pybatch(U,X,Y,nearest=npnearest)
            b_np = time.clock()
            tlist_np += [b_np-a_np]

        # Plot the results in a graph
        plt.figure(figsize=(7,4))
        plt.plot(nlist,tlist_py,'-o',color='red',label='pybatch+pynearest+npdistance')
        plt.plot(nlist,tlist_np,'-s',color='blue',label='pybatch+npnearest')
        plt.legend();plt.xscale('log');plt.yscale('log'); plt.xlabel('n'); plt.ylabel('time'); p

True
pybatch+pynearest+npdistance [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0]
pybatch+npnearest            [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0]
```



- Based on your results, explain what kind of speedup this further optimization provides, and in what regime the speedup is the most significant.

We can see that nearest neighbour implemented purely in python(using npdistance) is little bit slower with higer N training points bit the speed up is not really significant and both grows more or less lineary. The speed up seems most significant for n higher than 100

1.6 Accelerating the processing of multiple test points (25 P)

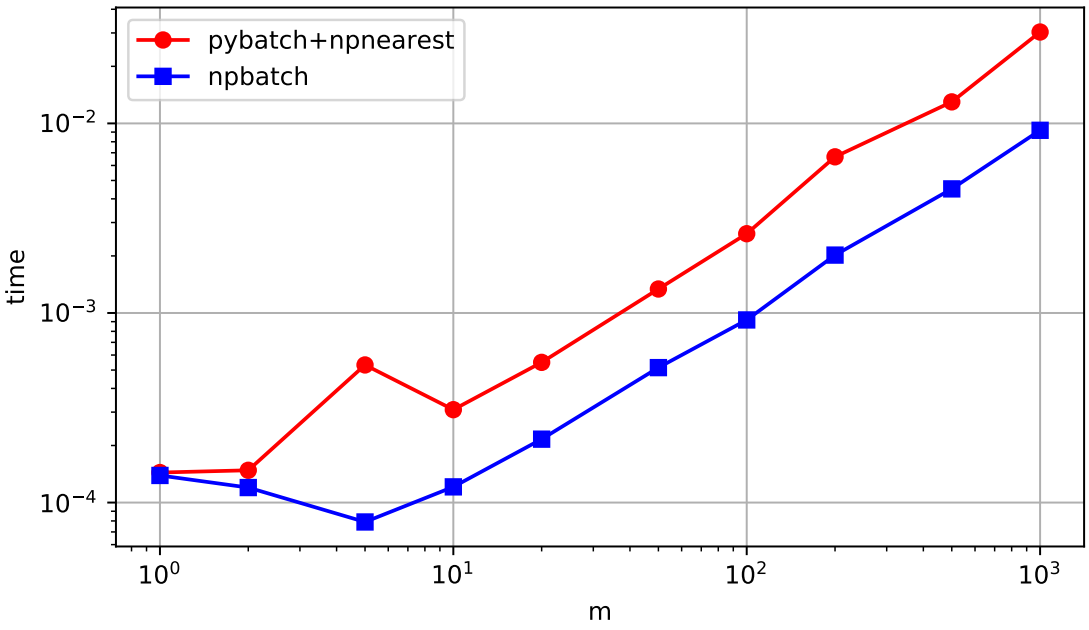
Not yet fully happy with the performance of the algorithm, we would like to further optimize it by avoiding performing a loop on the test points, and instead, classify them all at once.

- Create a new function `npbatch(U,X,Y)` as a replacement of the implementation `pybatch(U,X,Y,nearest=npnearest)` that we have built in the previous exercise.
- Print its output for the same dataset `data.toy(20,100,50)` and verify that the output remains the same as for the previous implementation.
- Create a plot comparing the computation time of the new implementation compared to the previous one. Here, we fix `N=100`, `d=100`, and we let `M` vary from `1` to `1000` with values `[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]`.

```
In [9]: import scipy
        from scipy import spatial

        U,X,Y = data.toy(20,100,50)
        def npbatch(U,X,Y):
```

```
        return Y[numpy.argmin(scipy.spatial.distance.cdist(U, X, 'euclidean'),axis=1)]

    print(pybatch(U,X,Y,nearest=npnearest) == npbatch(U,X,Y))
    print("pybatch+npnearest",pybatch(U,X,Y,nearest=npnearest))
    print("npbatch          ",npbatch(U,X,Y) )

    mlist=[1, 2, 5, 10, 20, 50, 100, 200, 500, 1000]
    # Measure the computation time for each choice of number of dimensions d
    tlist_py = []
    tlist_np = []
    for m in mlist:
        U,X,Y = data.toy(m,100,100)
        a = time.clock()
        pybatch(U,X,Y, npnearest)
        b = time.clock()
        tlist_py += [b-a]

        a = time.clock()
        npbatch(U,X,Y)
        b = time.clock()
        tlist_np += [b-a]

    # Plot the results in a graph
    plt.figure(figsize=(7,4))
    plt.plot(mlist,tlist_py,'-o',color='red',label='pybatch+npnearest')
    plt.plot(mlist,tlist_np,'-s',color='blue',label='npbatch')
    plt.legend();plt.xscale('log');plt.yscale('log'); plt.xlabel('m'); plt.ylabel('time'); p
```

```
[ True  True  True  True  True  True  True  True  True  True  True  True
  True  True  True  True  True  True  True  True]
pybatch+npnearest [1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0]
npbatch           [1 1 1 0 0 1 0 1 1 0 1 1 1 0 1 0 1 1 1 0]
```



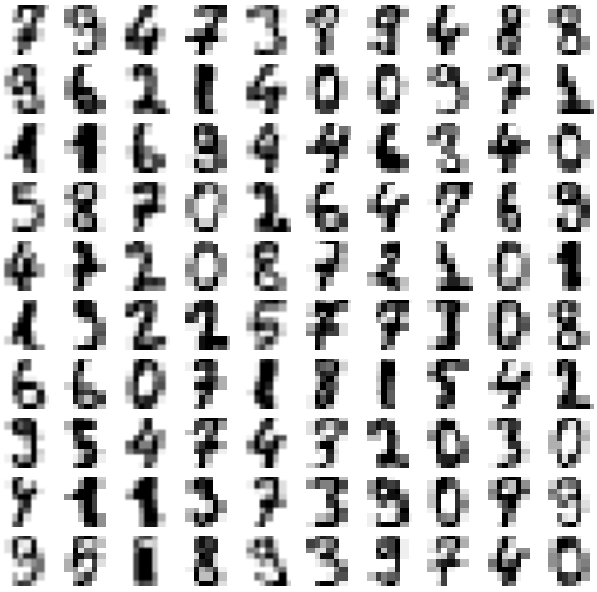
1.7 Application to real data (25 P)

Having now implemented an efficient K-nearest neighbor classifier, we can test it on real problems with many data points and dimensions. We consider a small handwritten digits recognition dataset, that can be directly obtained from the library `scikit-learn`. This dataset consists of handwritten digits of size `8 x 8` flattened into arrays of size `64`, with class between `0` and `9`. We use a function `data.digits()` to load the data and arrange data points in some predefined order.

```
In [10]: X,Y = data.digits()
```

Using the function `imshow` of `matplotlib` to visualize the first 100 digits of the dataset.

```
In [12]: for index, (image) in enumerate(X[:100]):
             plt.figure(1, figsize=(4,4))
             plt.subplot(10, 10, index + 1)
             plt.axis('off')
             plt.imshow(image.reshape(8,8), cmap=plt.cm.gray_r, interpolation='nearest')
         plt.show()
```



- Partition the data into a "training" set and "test" set. The first one contains the 1000 first digits of `X`, and the second one contains the remaining ones.
- Assume that you don't know the labels for the test data and classify the test data using your efficient nearest neighbor implementation.
- Print the predicted labels for the test set.

In [13]:
```
train_data=X[:1000]
test_data=X[1000:]

predicted = npbatch(test_data,train_data,Y[:1000])

print(predicted)
```

```
[0 7 3 5 9 4 7 2 5 6 1 2 7 0 0 6 2 2 4 4 3 4 0 2 7 9 1 4 4 4 9 4 7 7 3 1 4
 9 9 3 2 4 0 4 2 7 7 5 4 1 4 5 7 9 3 7 2 8 4 9 8 3 7 6 5 5 7 4 3 7 3 5 0 3
 5 0 0 7 0 5 9 3 3 4 7 9 4 8 6 4 0 0 8 2 9 4 6 4 9 0 0 3 1 6 5 1 0 1 9 2 2
 8 2 6 1 1 3 8 2 3 5 5 8 0 5 4 8 0 7 3 6 4 0 8 9 4 8 9 9 7 4 4 6 8 4 5 2 9
 9 4 0 5 8 5 2 2 7 6 4 8 3 0 7 6 5 6 1 0 9 3 5 6 3 6 3 3 0 0 1 4 1 1 9 3 8
 8 8 8 2 0 7 6 5 6 8 2 0 6 8 6 0 0 0 6 9 3 7 0 1 8 9 9 9 1 7 0 5 5 5 6 4 1
 4 8 6 6 8 3 1 0 5 2 2 6 8 4 2 1 0 4 6 9 9 6 1 7 2 3 4 0 5 5 7 4 8 1 1 7 8
 7 1 7 5 1 2 1 3 2 2 9 8 7 8 2 7 2 7 1 0 9 2 8 4 2 1 0 4 2 7 2 6 9 2 1 2 5
 4 7 1 6 3 4 4 7 0 0 9 9 9 9 9 9 1 1 9 5 7 3 8 4 8 6 6 3 1 8 6 8 4 3 6 2 3
 2 1 1 8 1 9 4 4 9 0 1 7 9 8 3 6 2 2 5 4 1 2 6 1 1 2 3 6 7 8 3 4 4 4 6 5 9
 4 6 6 1 3 2 6 5 7 9 4 7 6 8 5 6 1 9 1 9 7 5 7 8 8 5 1 0 7 2 6 9 1 0 7 3 2
```

```
 2 7 3 0 7 0 9 3 8 8 3 6 2 4 2 8 5 9 6 2 7 6 2 9 5 5 3 8 2 6 2 7 5 3 8 3 0
 3 0 4 0 2 8 5 1 0 7 7 3 2 7 6 8 8 3 7 4 9 5 0 4 1 1 2 1 3 6 5 0 5 1 2 6 1
 5 4 7 4 8 4 1 8 8 8 8 8 7 1 8 9 3 1 2 1 2 3 0 1 6 1 3 5 3 7 1 3 6 9 3 6 6
 2 3 2 0 0 1 9 6 5 9 5 9 4 4 2 5 4 9 0 0 6 2 5 7 2 1 7 0 3 0 6 2 1 6 9 9 5
 8 3 3 6 6 1 6 5 7 2 8 2 3 1 0 9 3 9 7 9 3 6 9 8 7 4 6 9 7 6 8 3 5 0 3 0 4
 4 3 1 4 2 6 5 6 1 7 5 6 6 3 2 9 2 3 1 1 1 2 3 3 2 0 9 4 9 8 3 2 0 0 9 5 1
 3 5 5 3 6 0 3 2 0 1 9 4 0 9 6 7 8 5 0 6 5 9 3 3 9 5 5 5 6 4 3 5 6 1 4 6 8
 7 1 8 6 5 2 4 6 1 1 8 2 5 7 0 6 4 3 0 9 7 0 6 5 7 7 7 5 5 4 3 5 0 3 2 5 9
 3 8 1 0 6 7 3 5 2 8 9 0 8 5 2 0 7 6 5 1 6 9 9 9 4 2 6 6 4 4 4 2 3 2 0 9 2
 5 3 9 3 3 6 0 0 2 3 9 5 5 6 6 4 7 0 0 9 4 7 9 7 5 4 1 0 8 1 1 6 9 2 9 7 8
 5 1 0 5 6 6 9 9 6 3 4 5 1 5 6 9 1 3 4 2]
```

Finally, in order to determine the accuracy of the classifier, we would like to compare the predictions with the ground truth (i.e. the true labels from the test data).

- Compute the fraction of the time on the test set where the predictions of the nearest neighbor algorithm and labels disagree.

```
In [14]: def accuracy():
             couter=0
             for x,y in zip(predicted,Y[1000:]):
                 if x != y:
                     couter += 1
             return round(couter/float(len(Y[1000:])),3)

         accuracy()
```

```
Out[14]: 0.009
```