nearest-neighbor

November 3, 2017

1 2.1 Exploring data for handwritten digits

In [1]:
```
from sklearn.datasets import load_digits
from sklearn import cross_validation
from sklearn.neighbors import KNeighborsClassifier

import numpy as np
import sklearn as skl
from matplotlib import pyplot as plt
%pylab inline
```

```
/Users/maxsimon/anaconda/lib/python3.6/site-packages/sklearn/cross_validation.py:44: Deprecati
  "This module will be removed in 0.20.", DeprecationWarning)


Populating the interactive namespace from numpy and matplotlib
```

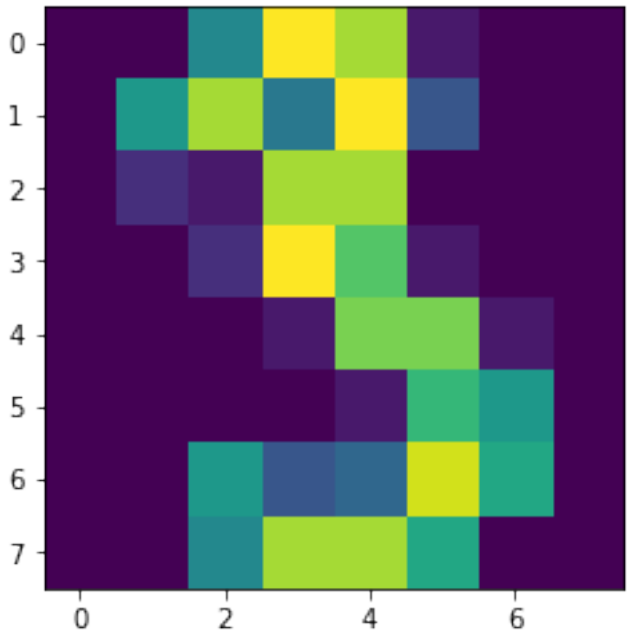
In [2]:
```
digits = load_digits()
print(digits.keys())
```

```
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

In [3]:
```
data = digits['data']
images = digits['images']
target = digits['target']
target_names = digits['target_names']
```

In [4]:
```
# show the first 3

ndx = np.argwhere(target == 3)[0][0]

print('First 3 at index {:d}'.format(ndx))
fig, ax = plt.subplots(1, 1)
ax.imshow(images[ndx])
plt.show()
```

```
First 3 at index 3
```




```
In [5]: # split data into training and test data
        X_all = data
        y_all = target

        X_train , X_test , y_train , y_test = cross_validation.train_test_split(digits.data, di
```

2 2.2 and 2.3 Distance functions

```
In [6]: def dist_loop(training, test):
            """
            This function calculates the distance matrix between training- and test data.
            """
            dist = np.empty((training.shape[0], test.shape[0]))
            for i in range(training.shape[0]):
                for j in range(test.shape[0]):
                    dist[i, j] = np.sqrt(np.sum(np.square(training[i] - test[j])))
            return dist

        def dist_vec(training, test):
            """
            This function calculates the distance matrix between training- and test data using
            (x-y)^2 = x^2 - 2*x*y + y^2
```

```
            """
            dist = np.sqrt(np.sum(training**2, axis=1) - 2*np.dot(test, training.T) + np.sum(te
            return dist.T
```

```
In [7]: # Measuring execution time for loop
        %timeit dist_loop(X_train, X_test)
        dist_from_loop = dist_loop(X_train, X_test)
```

```
1 loop, best of 3: 11.7 s per loop
```

```
In [8]: # measuring execution time for vectorisation
        %timeit dist_vec(X_train, X_test)
        dist_from_vec = dist_vec(X_train, X_test)
```

```
100 loops, best of 3: 13.7 ms per loop
```

```
In [9]: # check if both matrices are the same
        print(np.array_equal(dist_from_loop, dist_from_vec))
```

```
True
```

As you can see above, the vectorisation decreases the execution time much more (factor 1000) and the result is the same.

3 2.4 and 2.5 Nearest Neighbour classifier

```
In [10]: class NearestNeighbour:

             def __init__(self, k, dist_func, x_train, y_train):
                 # store some presets
                 self.k = k # k-value for 2.5
                 self.dist_func = dist_func # function to use for distance matrix calculation
                 # trainingsdata
                 self.x_train = x_train
                 self.y_train = y_train

             @staticmethod
             def filter_for_value(features, labels, values):
                 """
                 This function extracts from the features and labels only those, whose label i
                 """
                 return features[np.isin(labels, values)], labels[np.isin(labels, values)]

             def majorityVote(self, elements):
                 """
```

```
        Perform a simple majority vote on the array elements.
        """
        unique, counts = np.unique(elements, return_counts=True)
        return unique[np.argmax(counts)]

    def classify(self, x, label_values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]):
        """
        Classify the set of features x. For the classification only the labels in lab
        """
        # filter trainingsdata for the label_values
        reduced_x_train, reduced_y_train = NearestNeighbour.filter_for_value(self.x_t
        # calculate distance matrix
        distances = self.dist_func(reduced_x_train, x)

        # if only NearestNeighbour:
        # ndx = np.argmin(distances, axis=0); return reduced_y_train[ndx]

        # get the k indices with the smallest distance
        ndx = np.argsort(distances, axis=0)[:self.k, :]
        # create the decision array by making for each feature a majority vote
        decision = [self.majorityVote(reduced_y_train[ndx][:,i]) for i in range(ndx.s
        return np.array(decision)

    def test(self, x_test, y_test, label_values = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]):
        """
        Testing a dataset of features and labels. Allowed labels can be specified wit
        """
        # filter data for the label_values
        reduced_x_test, reduced_y_test = self.filter_for_value(x_test, y_test, label_
        # perform a classification
        classification = self.classify(reduced_x_test, label_values)
        # calculate the score
        score = np.count_nonzero(classification - reduced_y_test)/reduced_y_test.shap
        return score
```

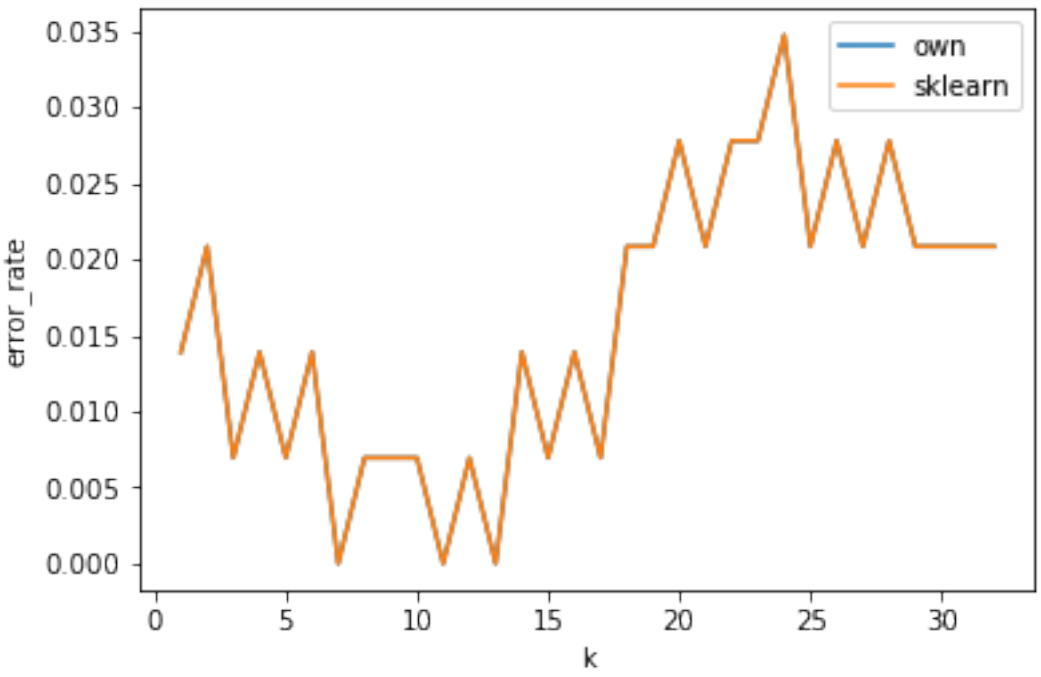
In [28]:
```
# decide whether 1 or 7
NN = NearestNeighbour(1, dist_vec, X_train, y_train)
print('score 1 - 7: ', NN.test(X_test, y_test, [1, 7]))
print('score 1 - 3: ', NN.test(X_test, y_test, [1, 3]))
```

```
score 1 - 7:  0.0
score 1 - 3:  0.0
```

In [26]:
```
# decide whether 3 or 9
NN = NearestNeighbour(1, dist_vec, X_train, y_train)
print('score 3 - 9: ', NN.test(X_test, y_test, [3, 9]))
```

```
score 3 - 9:  0.013888888888888888
```

```
In [27]: # try different k's
         ks = range(1, 33)
         errors = []
         errors_sklearn = []

         testing_labels = [3, 9] # allowed labels, original only [1, 7]

         for k in ks:
             NN = NearestNeighbour(k, dist_vec, X_train, y_train)
             errors.append(NN.test(X_test, y_test, testing_labels))
             # comparison to sklearn
             skNN = KNeighborsClassifier(n_neighbors=k)
             skNN.fit(*NearestNeighbour.filter_for_value(X_train, y_train, testing_labels))
             errors_sklearn.append(1 - skNN.score(*NearestNeighbour.filter_for_value(X_test, y_

         # present results
         fig, ax = plt.subplots(1, 1)
         ax.plot(ks, errors, label="own")
         ax.plot(ks, errors_sklearn, label="sklearn")
         ax.set_xlabel("k")
         ax.set_ylabel("error_rate")
         ax.legend()
         plt.show()
```



The NearestNeighbour-Algorithm can perfectly distinguish between 1 and 3 as well as between 1 and 7. Therefore I chose the pair 3 and 9. As you can see in the plot above, the consideration of up to 7 or 13 nearest neighbours lead to better results. Taking more neighbours into account, however, pushs the error rate again.

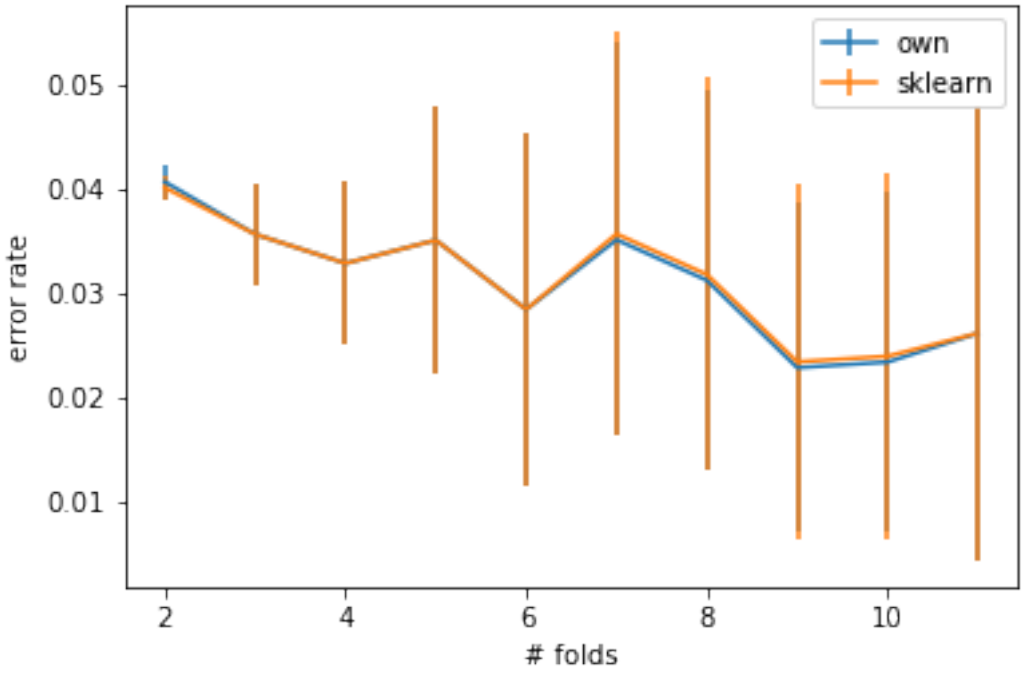
I compared the results to the KNeighborsClassifier from sklearn. As you can see in the plot, both implementations classify the test data the same way (orange line directly above the blue one).

4 3 Cross validation

```
In [37]: def get_folds(n, features, labels):
             """
             Shuffles the entries and put them into folds.
             """
             # create a list of indices
             indices = np.arange(labels.shape[0], dtype=int)
             # shuffle the indices
             np.random.shuffle(indices)

             # calculate the fold size
             fold_size = round(labels.shape[0]/n)

             # create folds, n-1 because last fold has to buffer the rounding
             folds = [[features[fold_size*i:fold_size*(i+1)], labels[fold_size*i:fold_size*(i+
             folds.append([features[fold_size*(n-1):], labels[fold_size*(n-1):]])

             return folds

         def create_train_test(folds, use_fold):
             """
             This function creates training and test data out of a fold array. The fold-index
             """
             training_features = np.array([]).reshape((0, folds[0][0].shape[1])) # reshape to
             training_labels = np.array([]) # array suitable for labels

             for j in range(len(folds)):
                 if j != use_fold:
                     training_features = numpy.append(training_features, folds[j][0], axis=0)
                     training_labels = numpy.append(training_labels, folds[j][1])

             return training_features, training_labels, folds[use_fold][0], folds[use_fold][1]
```

Now I test different foldsizes with both kNearestNeighbour implementations (my one and sklearn). Because the question refers to 2.4 I used $k = 1$ for both. The classification contains all digits.

In [39]:
```
ns = range(2, 12) # number of folds

# arrays to store the results
error_mean_own = []
error_std_own = []
error_mean_sklearn = []
error_std_sklearn = []

# loop over the ns to test
for n in ns:

    folds = get_folds(n, X_all, y_all) # get the folds

    # arrays to store results
    error_own = []
    error_sklearn = []

    for i in range(len(folds)): # loop over fold indices

        # split the data into training data and test data
        fold_x_train, fold_y_train, fold_x_test, fold_y_test = create_train_test(folds

        # create an instance of my implementation
        NN = NearestNeighbour(1, dist_vec, fold_x_train, fold_y_train) # set k = 1
        error_own.append(NN.test(fold_x_test, fold_y_test))
        # comparison to sklearn
        skNN = KNeighborsClassifier(n_neighbors=1) # set k = 1
        skNN.fit(fold_x_train, fold_y_train)
        error_sklearn.append(1 - skNN.score(fold_x_test, fold_y_test)) # score gives

    # calculate results
    error_mean_own.append(np.mean(error_own))
    error_std_own.append(np.std(error_own))
    error_mean_sklearn.append(np.mean(error_sklearn))
    error_std_sklearn.append(np.std(error_sklearn))


# plotting
fig, ax = plt.subplots(1, 1)

ax.errorbar(ns, error_mean_own, yerr=error_std_own, label="own")
ax.errorbar(ns, error_mean_sklearn, yerr=error_std_sklearn, label="sklearn")

ax.set_xlabel('# folds')
ax.set_ylabel('error rate')

ax.legend()
plt.show()
```



As you can see in the plot, both classifiers behave very similar. The error rate decreases for a larger number of folds. This is reasonable since the total amount of training data is growing with the number of folds. The standard deviation of the error rates gets also larger with the number of folds, which can be explained with the decreasing amount of test data.

In [ ]: