[CS156] Assignment 1

Akmarzhan Abylay

September 2020

1 Assignment 1

1.1 1. Moore's Law

1.1.1 Preparing the data

We will first import the data from both files and merge the dataframes based on the `textID` column.

[1]:
```
#required libraries
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import datetime


#ignoring the warnings
import sys
import warnings

if not sys.warnoptions:
    warnings.simplefilter("ignore")
```

[2]:
```
#loading the files into pandas dataframes
benchmarks = pd.read_csv('benchmarks.txt', sep=",")
summary = pd.read_csv("summaries.txt", sep=",",  encoding= 'unicode_escape',␣
 ↪low_memory=False)
```

[3]:
```
#merging the datasets based on the "testID' column
dataset = benchmarks.merge(summary, on='testID', how='inner', suffixes=('_1',␣
 ↪'_2'))
```

1.1.2 1.1 Extract the date and base speed for a benchmark of your choice.

I chose a benchmark called "126.gcc". I found the dates using two methods: extraction from the textID column, shown below (commented), and from the "summaries.txt" file, which explicitly gives out the date. I ended up using the second method because it seemed to give a more accurate representation of the dates.

[4]:
```
#extraction from the "textID" column

# import datetime
# date=[]
# base=[]
# for i in range(136995):
#     if dataset["benchName"][i]=="126.gcc":
#         if testID[i][0]!="p":
#             new_t=(testID[i][dataset["testID"][i].find('-') + 1:])
#             date.append(new_t[:new_t.find('-')])
#             base.append(dataset["base"][i])

# for i in range(len(date)):
#     if len(date[i])<7:
#         date[i]='19'+date[i]

# for i in range(len(date)):
#     if len(date[i])<7:
#         date[i]=dt.datetime.strptime(date[i], "%y%m%d")
#     else:
#         date[i]=dt.datetime.strptime(date[i], "%Y%m%d")

# time=[]
# for i in date:
#     time.append(i)

# data=pd.DataFrame()
# data['time']=time
# data['speed']=base
# data=data.sort_values(by='time')
```

[5]:
```
#isolating the entries for my specific benchmark and only leaving the date/speed␣
 ↪columns
data = dataset.loc[dataset['benchName'] == '126.gcc']
data.drop(data.columns.difference(['hwAvail','base_1']), 1, inplace=True)
```

I picked the entries for my benchmark and dropped all columns except the needed date/base speed columns. I used a base speed from the "benchmarks.txt" since the "summaries.txt" isn't as rich when it comes to the range of base speeds. Below you can see a snippet of the "data" data frame, which contains both the date and the corresponding base speed. It has 525 entries.

[6]:
```
data.head()
```

[6]:

	base_1	hwAvail
12	18.3	Jan-1999
20	18.1	Nov-1998
38	15.8	Dec-1998

```
126     21.1  Feb-1999
134     16.7  Dec-1998
```

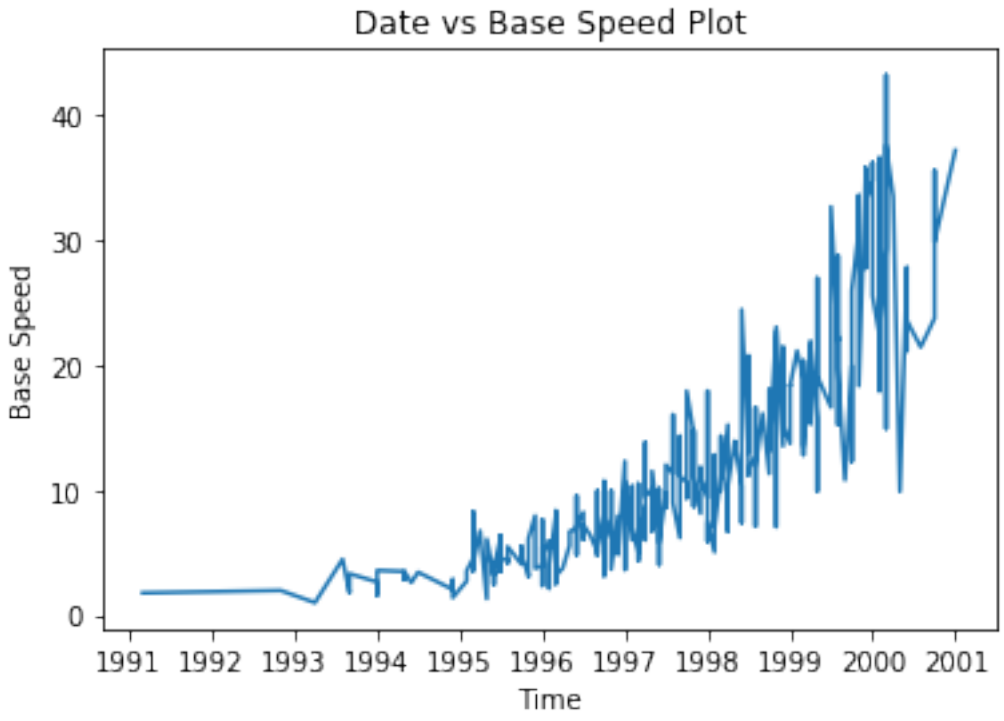
[7]:
```
data['hwAvail'] = pd.to_datetime(data['hwAvail'], format="%b-%Y")
data.sort_values(by="hwAvail", inplace=True)
```

1.1.3 1.2. Plot the data in a semi-log plot.

I plotted the data I just extracted in a basic manner and in a semilogy plot where the log is scaling on the base speed (y-axis).

[8]:
```
plt.plot(data['hwAvail'], data['base_1'])
plt.xlabel("Time")
plt.ylabel("Base Speed")
plt.title("Date vs Base Speed Plot")
```

[8]:
```
Text(0.5, 1.0, 'Date vs Base Speed Plot')
```

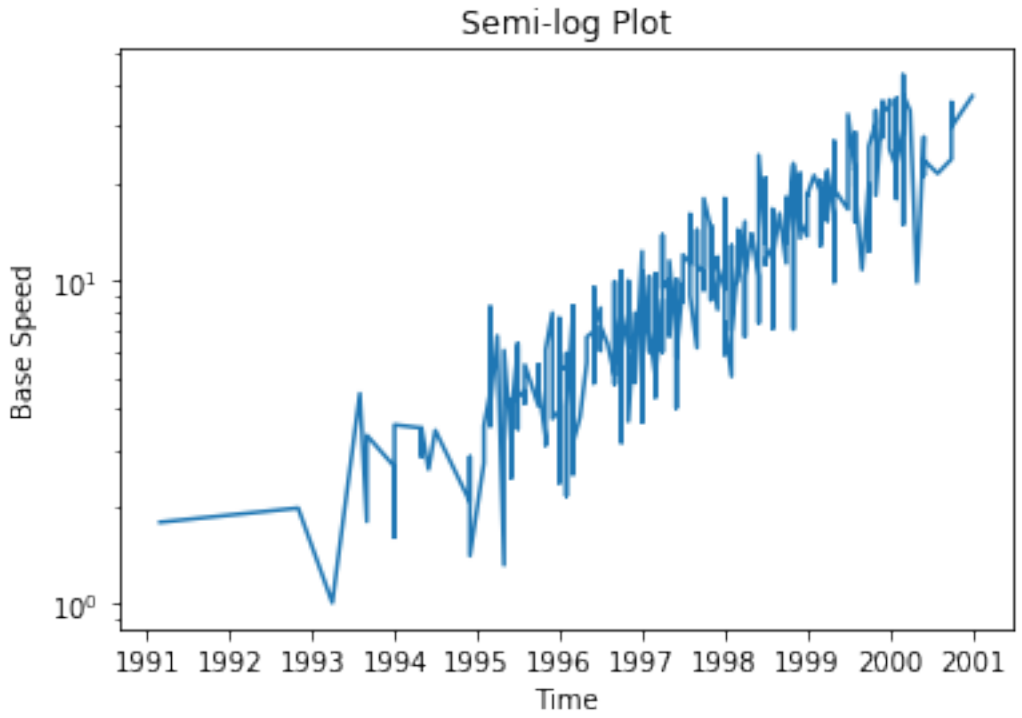


[9]:
```
#PLOT

import matplotlib
plt.semilogy(data['hwAvail'], data['base_1'])
plt.xlabel("Time")
```

```
plt.ylabel("Base Speed")
plt.title("Semi-log Plot")
```

[9]:
```
Text(0.5, 1.0, 'Semi-log Plot')
```



As we can see from the plot, it makes up almost a line which means that our base speed is scaling exponentially as a function of time.

1.1.4 1.3 Now train a linear model to fit your plot.

Below is a linear regression model. I needed to transform the dates into ordinal numbers so that the regression could actually work.

[10]:
```
#creating a new column with ordinal representations of the dates
import datetime as dt
data['ordinal'] = pd.to_datetime(data['hwAvail'])
data['ordinal']=data['ordinal'].map(dt.datetime.toordinal)
```

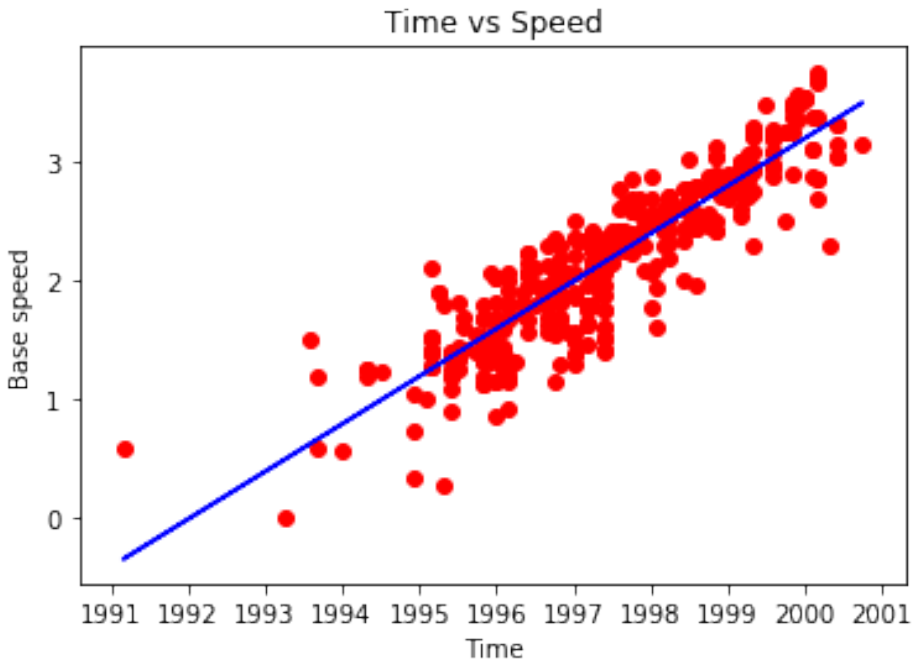
[11]:
```
#setting the independent and dependent variables
X = data['ordinal'].values.reshape(-1, 1)
#using the log of the dependent variable - base speed, since we are using a␣
 ↪linear model
#to be able to describe the relationship that is not linear (exponential)
y = np.log(data['base_1'].values)
```

```
#splitting the data into train/test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,␣
 ↪random_state = 0)
```

[12]:
```
#linear regression itself
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(X_train, y_train)
```

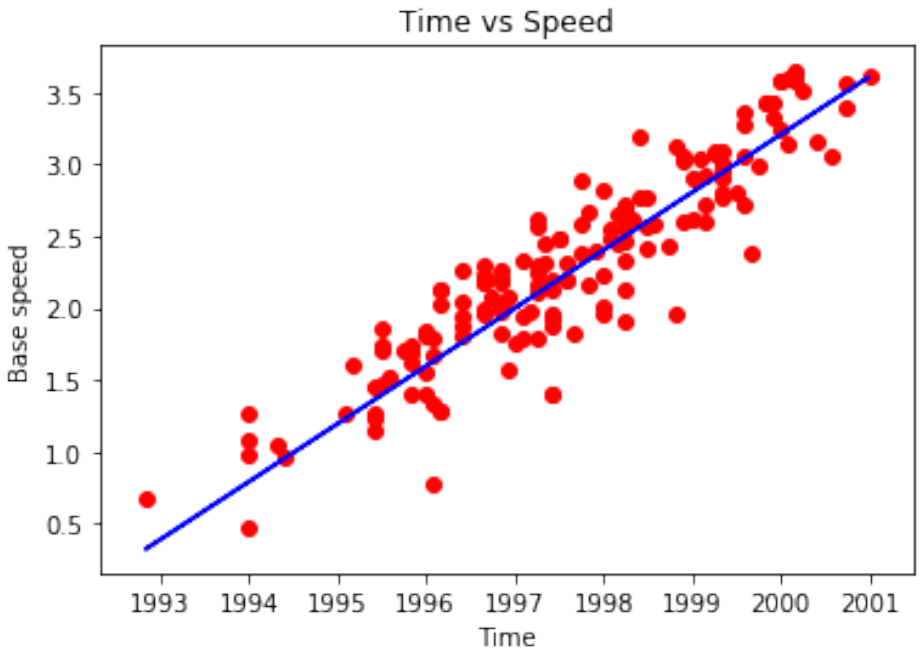
[12]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)
```

[13]:
```
#plotting the points and the regression line for the training data
X_tr = [dt.datetime.fromordinal(x) for x in X_train]
plt.scatter(X_tr, y_train, color = 'red')
plt.plot(X_tr, lin_reg.predict(X_train), color = 'blue')
plt.title('Time vs Speed')
plt.xlabel('Time')
plt.ylabel('Base speed')
plt.show()
```



[14]:
```
#plotting the points and the regression line for the testing data
X_te = [dt.datetime.fromordinal(x) for x in X_test]
plt.scatter(X_te, y_test, color = 'red')
```

```
plt.plot(X_te, lin_reg.predict(X_test), color = 'blue')
plt.title('Time vs Speed')
plt.xlabel('Time')
plt.ylabel('Base speed')
plt.show()
```



You can see the regression line below for both the train and test sets. We can see that the line fits very nicely, which supports the exponential relationship assumption between the date and base speed.

[15]:
```
from sklearn.metrics import r2_score
print("The intercept is: ", lin_reg.intercept_)
print("The slope is: ", lin_reg.coef_[0])
print("The R-squared is: ", r2_score(y_test, lin_reg.predict(X_test)))
```

```
The intercept is:  -801.214934558308
The slope is:  0.0011017655089164532
The R-squared is:  0.827846992874206
```

I printed the intercept, slope, and the R-squared value for this model. There is a positive correlation between the date and the base speed log. The R-squared is pretty high (around 0.83), which means that around 83% of the variation in the log of base speed is explained by the date/time in this regression model.

1.1.5 1.4 How well is Moore's Law holding up?

Moore's Law states that every two years, the number of transistors we could integrate doubles. Thus, the speed increases exponentially too (although, additionally, there might be other factors

that contribute to the speed). We made it easier to observe the relationships by transforming the base speed into a log so that the linear regression could explain the relationships better. As we saw from the regression model that we have built and the semilogy plots, Moore's Law seems to hold, as there is a strong linear relationship between the date and the log of base speed (i.e., the exponential relationship between the date/time and the base speed). The other 17% of the base speed variation could be possibly explained through the type of computer and its purpose because budget computers usually stay at their relative speed while faster computers made for, for example, scientific purposes, develop faster.

1.2 2. MNIST Digits

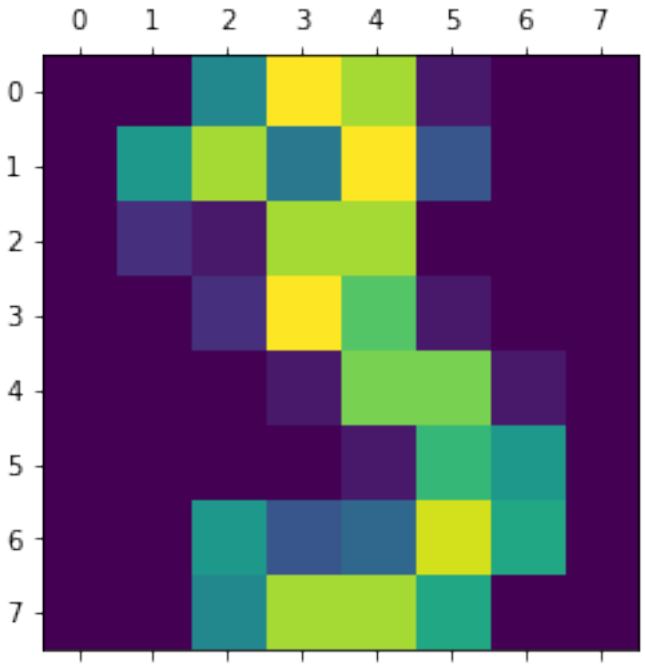
1.2.1 2.1 Load the MNIST digits.

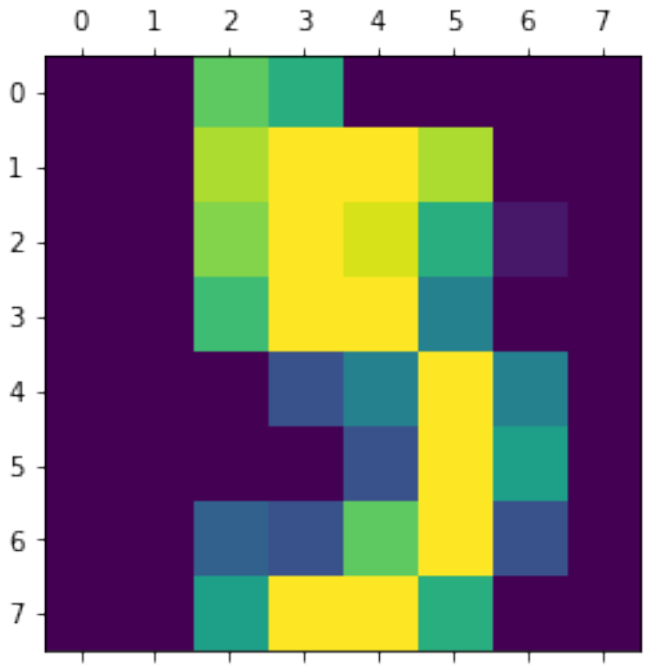
[16]:
```
from sklearn.datasets import load_digits
digits = load_digits()
```

1.2.2 2.2 Plot some of the examples.

[17]:
```
import matplotlib.pyplot as plt

plt.matshow(digits.images[3])
plt.matshow(digits.images[5])
plt.show()
```





1.2.3 2.3 Choose two digit classes (e.g. 7s and 3s), and train a k-nearest neighbor classifier.

I chose digits 3 and 5 and trained a model.

[18]:
```
X = [] #for the flattened data matrix based on which we will try to classify
y = [] #classification target (digit) or the dependent variable

for i in range(len(digits.target)): #getting the needed data
    if digits.target[i]==3 or digits.target[i]==5:
        X.append(digits.data[i])
        y.append(digits.target[i])
```

[19]:
```
#splitting the data
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25,␣
 ↪random_state = 0)
```

[20]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

#StandardScaler: reducing bias - standardize features by removing the mean and␣
 ↪scaling to unit variance
```

```
pipe = Pipeline([
        ('sc', StandardScaler()),
        ('knn', KNeighborsClassifier())])

#parameters we want to tune
params = {'knn__n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]}

#grid search
search = GridSearchCV(estimator=pipe,
                      param_grid=params,
                      cv=10,
                      return_train_score=True)
search.fit(X_train, y_train)
```

[20]:
```
GridSearchCV(cv=10, error_score=nan,
             estimator=Pipeline(memory=None,
                                steps=[('sc',
                                        StandardScaler(copy=True,
                                                       with_mean=True,
                                                       with_std=True)),
                                       ('knn',
                                        KNeighborsClassifier(algorithm='auto',
                                                             leaf_size=30,
                                                             metric='minkowski',
                                                             metric_params=None,
                                                             n_jobs=None,
                                                             n_neighbors=5, p=2,
weights='uniform'))],
                                verbose=False),
             iid='deprecated', n_jobs=None,
             param_grid={'knn__n_neighbors': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                                              11]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring=None, verbose=0)
```

[21]:
```
best_accuracy = search.best_score_
best_parameters = search.best_params_

print("Best Accuracy: {:.2f} %".format(best_accuracy*100))
print("Best Parameters:", best_parameters)
```

```
Best Accuracy: 100.00 %
Best Parameters: {'knn__n_neighbors': 3}
```

I didn't know what number of k would give us the best result, so I ran a quick Grid Search using different values, and it turned out that in the best-case scenario, k=3 gives us the best accuracy of 100%, which is very good.

1.2.4 2.4 Report your error rates on a held out part of the data.

[22]:
```
classifier = KNeighborsClassifier(n_neighbors=3)
classifier.fit(X_train, y_train)

from sklearn.model_selection import cross_val_score
accuracies = cross_val_score(estimator = classifier, X = X, y = y, cv = 10)
print("Accuracy: {:.2f} %".format(accuracies.mean()*100))
print("Standard Deviation: {:.2f} %".format(accuracies.std()*100))
```

```
Accuracy: 99.18 %
Standard Deviation: 1.25 %
```

[23]:
```
y_pred = classifier.predict(X_test)
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_test, y_pred)
print(cm)
accuracy_score(y_test, y_pred)
```

```
[[44  0]
 [ 0 48]]
```

[23]:
```
1.0
```

I used cross-validation to assess the test set error rate better. I also used a 10-fold CV method to evaluate the accuracy rate of the model when k=3. In 10-fold cross-validation, we divide our dataset into 10 folds, choose 9 as the training data, 1 as the test data, and repeat this 10 times for different configurations. We obtained an accuracy rate of 99.18%, which is pretty good, given the fact that we have a small dataset (365 entries). However, if we didn't use CV and just used the train/test data that we split initially, we would have a 100% accuracy rate since the model classified all of the digits correctly, as seen below. Anyway, CV is a better representation of the test set error as the error rate wouldn't depend on the example, and the results would generalize to an independent data set.

1.2.5 2.5 Test your model on the full dataset.

I loaded the data and isolated the entries which specifically have the digits 3 and 5 to test the model on a smaller subset first.

[24]:
```
#packages for importing the data
from mlxtend.data import loadlocal_mnist
import platform
```

[25]:
```
X_train1, y_train1 = loadlocal_mnist(
            images_path='train-images-idx3-ubyte',
            labels_path='train-labels-idx1-ubyte')
```

[26]:
```
X_test1, y_test1 = loadlocal_mnist(
            images_path='t10k-images-idx3-ubyte',
            labels_path='t10k-labels-idx1-ubyte')
```

[27]:
```
#isolating the digits 3 and 5
X_tr = []
y_tr = []
X_te = []
y_te = []

for i in range(len(y_train1)):
    if y_train1[i]==3 or y_train1[i]==5:
        X_tr.append(X_train1[i])
        y_tr.append(y_train1[i])

for i in range(len(y_test1)):
    if y_test1[i]==3 or y_test1[i]==5:
        X_te.append(X_test1[i])
        y_te.append(y_test1[i])
```

[28]:
```
classifier.fit(X_tr, y_tr)
```

[28]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

[29]:
```
y_pred = classifier.predict(X_te)
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_pred, y_te)
print(cm)
accuracy_score(y_te, y_pred)
```

```
[[994  14]
 [ 16 878]]
```

[29]:
```
0.9842271293375394
```

As we see from the results above, we got an accuracy rate of 98.42%, which is very good since the classifier misclassified only a small portion of the entries (30 out of 1902). Next, I decided to test on the full dataset with all digits without explicitly focusing on 3 and 5.

[30]:
```
classifier.fit(X_train1, y_train1)
```

[30]:
```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=3, p=2,
                     weights='uniform')
```

[31]:
```
y_pred = classifier.predict(X_test1)
from sklearn.metrics import confusion_matrix, accuracy_score
cm = confusion_matrix(y_pred, y_test1)
print(cm)
accuracy_score(y_test1, y_pred)
```

```
[[ 974    0   10    0    1    6    5    0    8    4]
 [   1 1133    9    2    6    1    3   21    2    5]
 [   1    2  996    4    0    0    0    5    4    2]
 [   0    0    2  976    0   11    0    0   16    8]
 [   0    0    0    1  950    2    3    1    8    9]
 [   1    0    0   13    0  859    3    0   11    2]
 [   2    0    0    1    4    5  944    0    3    1]
 [   1    0   13    7    2    1    0  991    4    8]
 [   0    0    2    3    0    3    0    0  914    2]
 [   0    0    0    3   19    4    0   10    4  968]]
```

[31]: 0.9705

We got an accuracy rate of 97.05%, which is once again good enough. However, whether the accuracy rate is good or not depends on the purpose of the classification and the needs of the person who is trying to classify objects. In my case, 97% seems like a pretty high accuracy, and maybe other types of classification algorithms would do better in this case.