L5-spydi472-stysi607

March 9, 2020

1 Lab 5: Simple OOP and numpy

Student: spydi472 (Spyridon Dimitriadis)
Student: stysi607 (Stylianos Sidiropoulos)

2 2. Introduction

2.1 Object-oriented Programming

The point of Object-oriented Programming is to support encapsulation and the DRY (Don't Repeat Yourself) principle without things getting out of hand. Often, software architects (those high-level programmers who are responsible for how large systems are designed on a technical level) talk about Object-oriented design or Object-oriented analysis. The point of this is to identify the necessary *objects* in a system. An object in this sense is not exactly the same as a Python object but rather a somewhat higher level logical unit which can reasonably be thought of as an independent component within the system. These high level objects might then be further subdivided into smaller and smaller objects and at a some level the responsibility shifts from the system architect to the team or individual developer working on a specific component. Thus, Object-oriented thinking is necessary for anyone developing code which will be integrated with a larger system, for instance a data scientist implementing analytics tools.

2.2 OOP in Python

Python implements the Object-oriented paradigm to a somewhat larger degree than the Functional paradigm. However, there are features considered necessary for *strict* object-oriented programming missing from Python. Mainly, we are talking about data protection. Not in a software security sense, but in the sense of encapsulation. There is no simple way to strictly control access to member variables in Python. This does not affect this lab in any way but is worth remembering if one has worked in a language such as Java previously.

3 3. Simple instance tests in Python

Note: some of these questions will be extremely simple, and some might prove trickier. Don't expect that the answer needs to be hard.

[1]:

```python
class Person:
    def __init__(self, name):
        self.name = name
        self.age = 0            # Age should be non-negative.

    def get_age(self):
        """Return the Person's age, a non-negative number."""
        return self.age

    def return_five(self):
        """Return 5. Dummy function."""
        return 5

Jackal = Person

president = Person("Jeb")
psec = Jackal("CJ Cregg")
```

a) Change the age of the `president` to 65 (`psec` should be unaffected).

[2]:
```
president.age = 65
```

[Note: This mode of operation is sometimes considered poor OOP. We will remedy this later.]

b) How many `Person` instances are there? One, two or three?

[3]:
```
# two
print(f"{type(Jackal)}  {type(president)}  {type(psec)}")
```

```
<class 'type'>  <class '__main__.Person'>  <class '__main__.Person'>
```

c) Consider the following code snippets. What do you think that they will return, and why? Discuss amongst yourselves. After that, run the code and explain the output. You only need to write down your explanation of the output.

[4]:
```
"Jeb" is Person
```

[4]:
```
False
```

[5]:
```
president is Person  # we could run: 'type(president) is Person' to check the␣
 ↪object 'president'
```

[5]:
```
False
```

[6]:
```
# 'is' operator compares identities, so if we do 'Jackal is Person'  returns␣
 ↪True
# In the first "Jeb" is a string
# In the second president is an object
```

d) How would you go about checking whether or not the value bound to the name `president` is-a `Person`?

[7]:

```
isinstance(president, Person)
# type(president) is Person
```

[7]:

```
True
```

4 4. Subclasses

a) Create class `Employee`, a subclass of `Person` with data attributes (fields)

- `__work_days_accrued`
- `__daily_salary`.

These should be *the only* data attributes which you write in your class definition. In particular, you may not duplicate `name` and `age`.

There should be methods * `work` which increments the number of work days accrued. * `expected_payout` which just returns the expected payout for the employee based on the accrued work days and daily salary (but without doing any resets). * `payout` which returns the accrued salary and resets the number of work days accrued. The `payout` function may not perform any calculations itself.

[8]:

```
class Employee(Person):
    def __init__(self, name, daily_salary=15):
        super().__init__(name)
        #Person.__init__(self, name)
        self.work_days_accrued = 0
        self.daily_salary = daily_salary

    def work(self):
        self.work_days_accrued += 1

    def expected_payout(self):
        return self.work_days_accrued * self.daily_salary

    def payout(self):
        payment = self.expected_payout()
        self.work_days_accrued = 0
        return payment


# Ready-made tests.
print("--- Setting up test cases.")
cleaner = Employee(name = "Scruffy")  # Should have daily_salary 15.
josh = Employee(name = "Josh", daily_salary = 1000)
toby = Employee(name = "Toby", daily_salary = 9999)

josh.work()
josh.work()
toby.work()
toby.work()
toby.work()
cleaner.work()

print("--- Testing payout and expected_payout properties.")
assert cleaner.expected_payout() == 15, "default salary should be 15"
assert josh.expected_payout() == 1000*2
assert josh.payout() == 1000*2
assert josh.expected_payout() == 0, "salary should be reset afterwards"
assert toby.payout() == 9999*3, "toby and josh instances should be independent."
print("OK")

print("--- Testing non-data-accessing calls to superclass methods.")
assert josh.return_five() == 5, "Person.return_five should be accessible"
print("OK")

print("--- Testing data that should be set up by initialiser call.")
assert josh.get_age() == 0, "superclass method should be callable, values␣
 ↪should not be missing."
josh.age = 9
assert josh.get_age() == 9, "superclass method should be callable"
print("OK")
```

```
--- Setting up test cases.
--- Testing payout and expected_payout properties.
OK
--- Testing non-data-accessing calls to superclass methods.
OK
--- Testing data that should be set up by initialiser call.
OK
```

b) Which public data attributes (fields) does an Employee have? Can you access the age of an employee directly (without some transformation of the name)? The daily salary?

[9]:

```
print(dir(josh))

print(josh.age)  #access the age
print(josh.daily_salary)  #access the daily salary
```

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'age', 'daily_salary',
'expected_payout', 'get_age', 'name', 'payout', 'return_five', 'work',
'work_days_accrued']
9
1000
```

5 5. Introductory numpy

A lot of computations will likely end up using data stored `numpy` arrays. Therefore, it is a good idea to have a feeling for how they are used and manipulated. The following steps will provide some introduction, so that we can build upon this in future labs.

You may want to refer to the official numpy.org absolute beginners guide to numpy.

a) Import the module `numpy`, giving it the shorthand `np`.

```
[10]: import numpy as np
```

b) Create the 2×3 matrix $A = \begin{bmatrix} 1 & 2 & 3 \\ 0 & 0 & 1 \end{bmatrix}$

```
[11]: A = np.array([[1,2,3],[0,0,1]])
```

c) Create the vector $b = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$

```
[12]: b = np.array([1,2,3])
```

d) Perform the multiplication Ab. What is the result?

```
[13]: A @ b
```

```
[13]: array([14,  3])
```

e) Mathematically, what should the dimensions of $b^T b$ be? What should the dimensions of bb^T be?

```
[14]: """
      bTb should be a scalar (1x1 matrix)
      bbT should be a 3x3 matrix
      """
```

```
[14]: '\nbTb should be a scalar (1x1 matrix)\nbbT should be a 3x3 matrix\n'
```

f) Compute `b.T @ b`? `b @ b.T` in numpy. How would you go about calculating bb^T (if you wanted to actually store it as a matrix, which is rarely the most space- or time-efficient idea)?

```
[15]: print(b.T@b)
      print(b@b.T)

      b2 = np.array([[1],[2],[3]])
      print(b2@b2.T)
```

```
14
14
[[1 2 3]
 [2 4 6]
 [3 6 9]]
```

Note: as `a @ b` is read `a.dot(b)` this should perhaps not be entirely surprising. But it is non-obvious.

d) Try to solve the equation $Ax = b$ using the `solve` method. Does this make mathematical sense? What does numpy say? Read the error message.

[16]:
```
# np.linalg.solve(A, b)

##LinAlgError: Last 2 dimensions of the array must be square
```

[17]:
```
"""
This equation does not make mathimatical sense, A is 2x3 and b is 3x1..
We get a 'LinAlgError' which says 'Last 2 dimensions of the array must
be square'. It neeeds a square matrix for A

"""
```

[17]:
```
"\nThis equation does not make mathimatical sense, A is 2x3 and b is 3x1..\nWe
get a 'LinAlgError' which says 'Last 2 dimensions of the array must \nbe
square'. It neeeds a square matrix for A\n\n"
```

e) Try to solve the equation $A^T x = b$ using numpy. Does this make mathematical sense? What does numpy say?

[18]:
```
# np.linalg.solve(A.T, b)

##LinAlgError: Last 2 dimensions of the array must be square
```

[19]:
```
"""
This equation makes mathimatical sense because the dimensions match now.
But still the numpy rises the same error as above.
(mathimatical solution x=np.array([[1],[0]]))
"""
```

[19]:
```
'\nThis equation makes mathimatical sense because the dimensions match now.
\nBut still the numpy rises the same error as above. \n(mathimatical solution
x=np.array([[1],[0]]))\n'
```

f) One of the two tasks above make sense from a mathematical point of view. Find a best solution x in the least squares sense.

[20]:
```
x = np.linalg.solve(A@A.T, A@b)
x
```

```
[20]: array([1.0000000e+00, 4.6629367e-16])
```

g) Is the resulting Ax or $A^T x$ (depending on your choice above) close to b? What is the norm-2-distance between the vectors?

```
[21]: np.linalg.norm(A.T@x - b)
```

```
[21]: 2.482534153247273e-16
```

Just to get some practice, run the following to get some test data:

```
[22]: import sklearn
      import sklearn.datasets as ds
      houses = ds.fetch_california_housing()
```

h) How many rows does the dataset have? Columns? Find out using numpy. (First check what `houses` actually is)

```
[23]: print(type(houses))
      print(dir(houses))
      print(type(houses.data))

      print(f"The dataset has {houses.data.shape[0]} rows and {houses.data.shape[1]}␣
       ↪columns.")
```

```
<class 'sklearn.utils.Bunch'>
['DESCR', 'data', 'feature_names', 'target']
<class 'numpy.ndarray'>
The dataset has 20640 rows and 8 columns.
```

i) Get the third column (remember: indexing starts at zero, so column number 2) of the dataset by indexing/slicing.

```
[24]: houses.data[:,2]
```

```
[24]: array([6.98412698, 6.23813708, 8.28813559, ..., 5.20554273, 5.32951289,
             5.25471698])
```

j) Get the values of the third, fourth and hundredrth rows (that is, index 2, 3, 99).

```
[25]: houses.data[[2,3,99],]
```

```
[25]: array([[ 7.25740000e+00,  5.20000000e+01,  8.28813559e+00,
               1.07344633e+00,  4.96000000e+02,  2.80225989e+00,
               3.78500000e+01, -1.22240000e+02],
             [ 5.64310000e+00,  5.20000000e+01,  5.81735160e+00,
               1.07305936e+00,  5.58000000e+02,  2.54794521e+00,
               3.78500000e+01, -1.22250000e+02],
             [ 2.61040000e+00,  3.70000000e+01,  3.70714286e+00,
               1.10714286e+00,  1.83800000e+03,  1.87551020e+00,
               3.78200000e+01, -1.22260000e+02]])
```

k) Mathematical sanity check: without actually computing the rank of `houses.data`, can you provide a bound? Could it have 10^{15} linearly independent rows? 2000? Etc. Write down the tightest bound you can find, and explain why briefly.

[26]:
```
print(f"The rank of houses.data is less or equal to {np.min([houses.data.
 ↪shape[0],houses.data.shape[1]])}.")
print(f"It could have up to {houses.data.shape[0]}  linearly independent rows,␣
 ↪because they cannot be more than the actual number of rows.")
```

```
The rank of houses.data is less or equal to 8.
It could have up to 20640  linearly independent rows, because they cannot be
more than the actual number of rows.
```

l) Find out the actual rank of the matrix using numpy.

[27]:
```
print(f"The rank of 'houses.data' is {np.linalg.matrix_rank(houses.data)}.")
```

```
The rank of 'houses.data' is 8.
```

6 6. A simple classifier

Our goal in this section is to build a naïve subspace-projection based classifier which takes a numerical vector and produces a label. We will build this from the ground up (there are many libraries that perform this automatically, but may not use them in this case).

We will first practice our numpy (and linear algebra) skills to build the algorithm, and then encapsulate data and methods in a class. We will be taking a slightly different route than more general SVD methods for finding principal components, simply because this is not our task. This is worth remembering if you find material on the subject!

The main idea here is to pick out all the data corresponding to a certain set of vectors, and reduce this matrix to some rank $\leq k$ matrix which is a good approximation of the row space in some sense (here: the sense that the vectors correspond to maximal singular values). A naïve classification is then given by picking the subspace which is closest by orthogonal projection.

6.1 The dataset

First, we will import some data. Here we use the classic MNIST data set of handwritten digits. We can naturally partition the sets of vectors used for training and validation in many ways. Feel free to experiment with a nicer (possibly non-deterministic) version later. This is only to get us started, and to have a set of data with known properties (which is helpful for lab assistants).

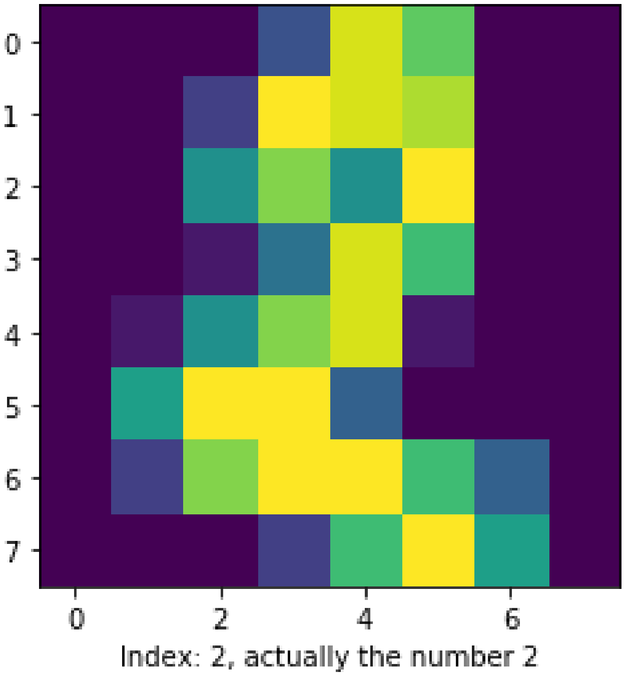
[28]:
```
import sklearn.datasets as ds
digits = ds.load_digits()
training_digits = digits.data[0:600, :]
training_labels = digits.target[0:600]
```

```
# The set of labels can be found in digits.target_names.
```

If we are curious, we can view the images from the dataset.

[29]:
```
%matplotlib inline
from matplotlib import pyplot as plt
# We show one of the digits.
d_index = 2
plt.imshow(training_digits[d_index].reshape(8,8))
plt.xlabel(f"Index: {d_index}, actually the number {training_labels[d_index]}")
# Note: digits.images[i] is the same as digits.data[i], but in 8x8 format (no need to reshape).
# We would however like to stress that the 64 pixel images we'll be working with are stored in vector format.
```

[29]:
```
Text(0.5, 0, 'Index: 2, actually the number 2')
```



6.2 Tasks

a) Get all rows with label 4. Store that in the matrix A_4. Similarly for A_9. This should be two single lines of code!

[30]:
```
a4 = training_digits[training_labels==4]     # (57, 64)
a9 = training_digits[training_labels==9]     # (59, 64)
```

b) Compute the thin SVD $A_4 = U_4\Sigma_4V_4^T$ and similarly for A_9. Thin here means that you do not necessarily get the full-rank square U, V^T matrices.

[31]:
```
u4, s4, vh4 = np.linalg.svd(a4)
u9, s9, vh9 = np.linalg.svd(a9)
```

Note Our data is stored by row (one row per digit). What we use the different matrices in the decomposition for is thus slightly different than if it was stored by column. Instead of picking the best k columns of U we pick the best k rows of V^T.

c) Let *sample*4 be the first row of A_4 and *sample*9 be the first row of A_9.

[32]:
```
sample4 = a4[0,:]  #(64,)
sample9 = a9[0,:]  #(64,)
```

d) Let R be the first 3 rows of V^T. Compute the orthogonal projection of *sample*4 onto the row space of R_4, and onto the row space of R_9

[33]:
```
r4 = vh4[0:3,:]     # (3, 64)
r9 = vh9[0:3,:]

coeffs_in_r4 = sample4 @ r4.T  # V is orthonormal i.e. unit length of columns
coeffs_in_r9 = sample4 @ r9.T

sample4_in_r4 = coeffs_in_r4 @ r4   # Linear combination of rows in R4
sample4_in_r9 = coeffs_in_r9 @ r9
```

[Mathematical aside: where did the Σ go? This is a very important matrix, and strongly affects both interpretations and (in general) row- and column spaces. *Assuming* that the rank is ≥ 3 the row spaces should be the same whether or not we scale the basis vectors by some σ_i (since $\sigma_i > 0$ for all $i = 1, 2, \ldots, r$). That is enough for our application, though the interpretations of the vectors might be different. The rank assumption is rather important, and we will make rather casually based on knowledge about the data set. Later, we'll include a check for this in our code.]

e) Which of the projections is closest to sample4 (in the sense that the difference (sample4_in_z**?**t - sample4) as the smallest 2-norm?

[34]:
```
print(np.linalg.norm(sample4 - sample4_in_r4))
print(np.linalg.norm(sample4 - sample4_in_r9))
```

```
19.75899698363074
43.52244724836977
```

f) Create a function `split_data(data, data_labels, labels)` which takes a matrix of data (such as `training_data`), the labels of all rows (such as `training_labels`) and a vector of the labels (here the numbers 0,. . . ,9) and returns a dictionary mapping a label to the matrix of all corresponding data.

Here the dictionary would for instance have the key 2, and the corresponding value be the matrix of all the images in `data` classified as the number 2.

[35]:
```
def split_data(data, data_labels, labels):
    dct = {}
    for label in labels:
        dct[str(label)] = data[data_labels==label,:]
    return dct
```

f) Create a function `subspaces(labelled_data, k = 3)` that takes a dictionary such as above, and returns a dictionary mapping every occurring label d to the respective Z_d matrix with at most k rows. Here the Z_d matrices is defined as above, the best norm-2-approximation. If a particular matrix has a rank $r < k$, we should only keep r rows (say, if there were many more *different* digit 9-rows in the training set, than digit 1:s)!

[36]:
```
def subspaces(labelled_data, k=3):
    dct_Z = {}
    for key_label, value_array  in labelled_data.items():
        U, S, Vt = np.linalg.svd(value_array)
        k_new = k
        if np.linalg.matrix_rank(Vt)<k:
            k_new = np.linalg.matrix_rank(Vt)

        dct_Z[key_label] = Vt[0:k_new,:]

    return dct_Z
```

g) Create a function `classification(spaces, x)` that takes a dictionary such as produced by `subspaces` above and a single vector and produces the label of the subspace with the smallest norm-2-distance. In the task above, this would have meant that classifying $sample4$ would have produced the label 4 rather than 9, since $sample4$ was closer to the subspace Z_4 than the subspace Z_9.

[37]:
```
def classification(spaces, x):
    norms = np.zeros(len(spaces.keys()))
    for key_label  in spaces.keys():
        coeffs_in_Z = x @ spaces[key_label].T
        x_in_Z = coeffs_in_Z @ spaces[key_label]
        norms[int(key_label)] = np.linalg.norm(x_in_Z-x)
    return np.argmin(norms)
```

h) Now pick the first 600 rows as training and the remaining (about 1100) as validation sets. What percentage does the classifier get right? What happens if you pick $k = 5, 10, 100, 1000$? Try it out. (The sharp-eyed student will notice that not all of these choices make sense from a mathematical perspective.)

[38]:
```
training_set = digits.data[0:600, :]
training_labels = digits.target[0:600]
validation_set = digits.data[600:, :]
```

```
validation_labels = digits.target[600:]

# rank(Z_i)=64. So k more than 64 does not make sense.

dct_data = split_data(training_set, training_labels, np.unique(training_labels))

for k in [5, 10, 100, 1000]:
    Z = subspaces(dct_data, k)
    predictions = [classification(Z, observation) for observation in␣
 ↪validation_set]
    hit_rate = np.mean(predictions==validation_labels)
    print(f"The hit rate for k={k} is {round(hit_rate*100, 4)} percent.")
```

```
The hit rate for k=5 is 92.8154 percent.
The hit rate for k=10 is 95.9064 percent.
The hit rate for k=100 is 54.2189 percent.
The hit rate for k=1000 is 54.2189 percent.
```

7 7. Encapsulating the classifier

Above we have a set of functions floating around in a global namespace. If we change some data and rerun some cells, we might get unexpected results. We instead want to encapsulate this in a single class.

a) Create a class `NaiveProjection`. It should have the following properties:

- An instance is created with `NaiveProjection(training_data, training_labels, all_labels, k)` where *k* is optional (and defaults to 3).

For instance, we should be able to run `dig_class = NaiveProjection(training_data = digits.data[0:600,:], training_labels = digits.target[0:600], all_labels = digits.target_names)`. * The class should have a method called `classification` which takes vector and returns the classification of that vector as defined above (minimum 2-norm distance to the subspace). * The class should have a method called `hitrate` which takes a matrix of row vectors, a vector of correct labels and returns the rate of successfully classified * All data about subspaces and labels should be stored within the instance. We may not use any data stored outside (though it is of course OK to call outside functions). We should be able to create two (or more) independent classifiers

```
dig15 = NaiveProjection(digits.data[0:600, :], digits.target[0:600], k = 15)
dig1 = NaiveProjection(digits.data[0:600, :], digits.target[0:600], k = 1)
```

which should be entirely independent.

[39]:
```
class NaiveProjection:
    def __init__(self, training_data, training_labels, all_labels, k=3):
        self.training_data = training_data
        self.training_labels = training_labels
        self.all_labels = all_labels
        self.k = k

    def split_data(self):
        dct = {}
        for label in self.all_labels:
            dct[str(label)] = self.training_data[self.training_labels==label]
        return dct

    def subspaces(self):
        dct_Z = {}
        labelled_data = self.split_data()
        for key_label, value_array  in labelled_data.items():
            U, S, Vt = np.linalg.svd(value_array)
            k_new = self.k
            if np.linalg.matrix_rank(Vt)<self.k:
                k_new = np.linalg.matrix_rank(Vt)

            dct_Z[key_label] = Vt[0:k_new,:]

        return dct_Z

    def classification(self, spaces, x):
        norms = np.zeros(len(spaces.keys()))
        for key_label  in spaces.keys():
            coeffs_in_Z = x @ spaces[key_label].T
            x_in_Z = coeffs_in_Z @ spaces[key_label]
            norms[int(key_label)] = np.linalg.norm(x_in_Z-x)
        return np.argmin(norms)


    def hitrate(self, validation_set, validation_labels):
        self.validation_set = validation_set
        self.validation_labels = validation_labels
        spaces = self.subspaces()
        predictions = [self.classification(spaces, observation) for observation␣
 ↪in validation_set]
        return round(np.mean(predictions==validation_labels), 4)
```

[40]:

```
# You can test here
dig1 = NaiveProjection(digits.data[0:600, :], digits.target[0:600], digits.
 ↪target_names, k = 1)
dig15 = NaiveProjection(digits.data[0:600, :], digits.target[0:600], digits.
 ↪target_names, k = 15)

print(f"Hit rate for k = 1: {dig1.hitrate(digits.data[600:], digits.target[600:
 ↪])}")
```

```
print(f"Hit rate for k = 15: {dig15.hitrate(digits.data[600:], digits.
 ↪target[600:])}")
```

```
Hit rate for k = 1: 0.848
Hit rate for k = 15: 0.9674
```

There are several noteworthy issues here. * We have taken a few mathematical shortcuts. * There is no error handling. * We have not discussed overlapping subspaces, sampling strategies or the like. * It might make a lot more sense to classify a matrix of vectors, rather than a single vector (to utilise the speed of computations).

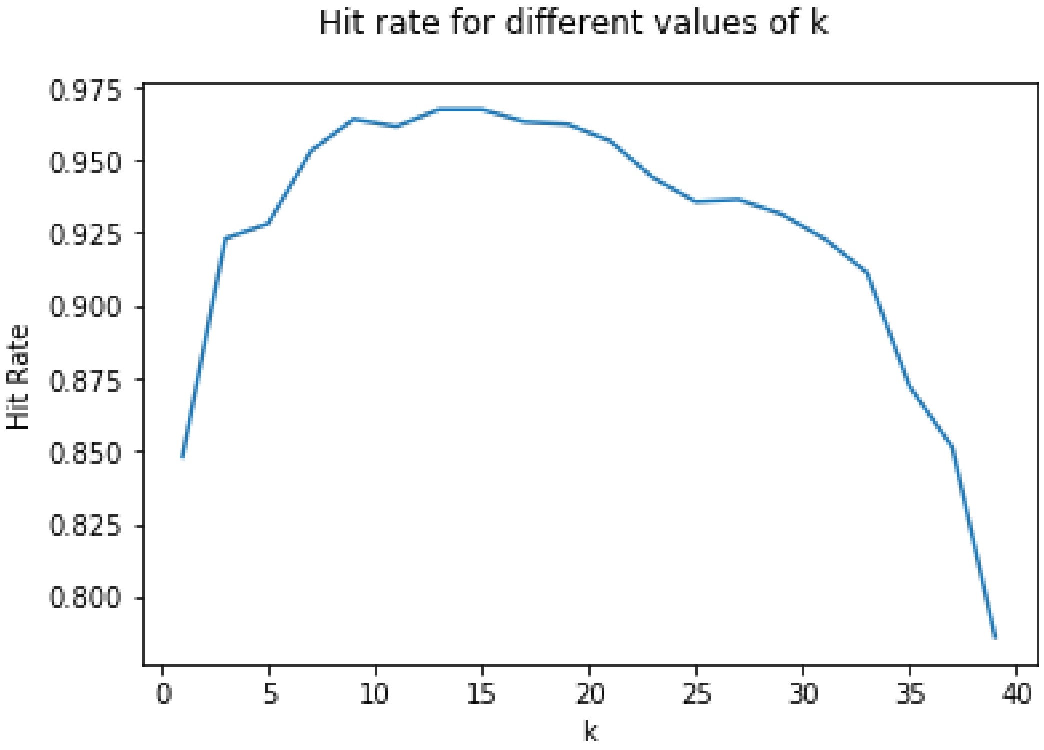
Most of this is left to an actual statistics course. The interested reader is referred to for instance Strang - Linear algebra and learning from data, or Golub & van Loan - Matrix computations.

b) Experiment with different values of k and training data/validation data sets. Plot the results using the library `matplotlib`.

[41]:
```
kvalues = np.arange(1,40,2)
hitrates = [NaiveProjection(digits.data[0:600, :], digits.target[0:600], digits.
 ↪target_names, k = kval).hitrate(digits.data[600:], digits.target[600:]) for␣
 ↪kval in kvalues]

fig = plt.figure()
plt.plot(kvalues, hitrates)
fig.suptitle('Hit rate for different values of k')
plt.xlabel('k')
plt.ylabel('Hit Rate')
```

[41]:
```
Text(0, 0.5, 'Hit Rate')
```



7.1 Acknowledgments

This lab in 732A74 is by Anders Märak Leffler (2019), with a major revision in 2020. The introductory text is by Johan Falkenjack (2018).