NumPy / SciPy Recipes for Data Science: Kernel Least Squares Optimization (1)

Technical Report · March 2015

DOI: 10.13140/RG.2.1.4299.9842

1 author:

Christian Bauckhage
University of Bonn
272 PUBLICATIONS **3,920** CITATIONS

SEE PROFILE

NumPy / SciPy Recipes for Data Science: Kernel Least Squares Optimization (1)

Christian Bauckhage
B-IT, University of Bonn, Germany
Fraunhofer IAIS, Sankt Augustin, Germany

***Abstract*—In this note, we show that least squares optimization is amenable to the kernel trick. This provides great flexibility in model fitting and we consider examples that illustrate this. In particular, we will discuss how kernel functions can implicitly introduce non-linearity into the least squares method.**

I. Introduction

Continuing our study of least squares, this note will show how to invoke the *kernel trick*. This trick has become a staple of data science because it allows for using linear techniques to tackle nonlinear problems [1]. Invoking the kernel trick, we thus derive an alternative least squares solution that is of considerable practical interest for it offers a lot of flexibility in model fitting. Alas, there is no free lunch! Since the increased flexibility comes at the cost of increased computational efforts, we will elaborate on pros and cons of kernel least squares[1].

We begin with a brief reminder of what we discussed earlier, then summarize the theory behind kernel least squares, and finally present *NumPy* implementations. As always, we assume our readers to be familiar with linear algebra and to have basic knowledge of *NumPy*, *SciPy*, and *Matplotlib* [4], [5].

II. Brief Recap

In preparation for the main topic of this note, we briefly summarize the basic ideas worked out in [6]–[8].

We consider the following problem setup: given a vector $\boldsymbol{x} = [x_1, \ldots, x_{m-1}]$ of real valued input variables, we want to predict a real valued output variable y. Arguably the simplest reasonable ansatz is to approximate the output as a linear combination of the inputs, namely

$$y(\boldsymbol{x}) \approx \sum_{j=1}^{m-1} x_j \, w_j + w_0. \tag{1}$$

In practice, the x_j may signify various kinds of information; common examples are that

- x_j represents a certain quantitative input
- x_j represents a transformation of a quantitative input such as, for instance, $x_j = \log(z_j)$ or $x_j = \sqrt{z_j}$
- x_j represents a polynomial over quantitative inputs, for instance, $x_j = x_1^j$ or $x_4 = x_1^3 \cdot x_2 \cdot x_3$.

Therefore, even though the linear model in (1) is conceptually simple, it can account for fairly complex dependencies between input and output

[1]Readers unfamiliar with the kernel trick and the underlying theory might want to consult our introductory lecture notes [2], [3].

We also note that, if we extend the input- and the coefficient vector as follows

$$\boldsymbol{x} \leftarrow [\boldsymbol{x}, 1] \quad \text{and} \quad \boldsymbol{w} \leftarrow [\boldsymbol{w}, w_0], \tag{2}$$

we may cast (1) as an inner product between vectors in $\mathbb{R}^m$

$$y(\boldsymbol{x}) \approx \boldsymbol{x}^T \boldsymbol{w}. \tag{3}$$

In practice, we are usually given training data $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$ and have to determine suitable model parameters $\boldsymbol{w}$ therefrom. Introducing an $n \times m$ data matrix $\boldsymbol{X}$ and an n-dimensional target vector $\boldsymbol{y}$, we can cast the function we aim to minimize for this purpose as

$$L(\boldsymbol{w}, \lambda) = \big\|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\big\|^2 + \lambda \big\|\boldsymbol{w}\big\|^2 \tag{4}$$

where $\lambda \geq 0$ is a Lagrange parameter. The solution to our problem, i.e. the minimizer of (4), then amounts to

$$\boldsymbol{w} = \big(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_m\big)^{-1} \boldsymbol{X}^T \boldsymbol{y} \tag{5}$$

where $\boldsymbol{I}_m$ denotes the $m \times m$ identity matrix.

Finally, we recall that the objective function in (4) is called the *primal form* of the least squares problem and that the vector $\boldsymbol{w} \in \mathbb{R}^m$ in (5) is called the *primal vector*.

III. Theory

In this section, we derive an alternative solution of the least squares problem, show that it allows for invoking the kernel trick, and examine the effect of different kernels on a simple regression problem.

To begin with, we perform a series of simple manipulations of the solution in (5). Note that we are aware that the following may look like algebraic mumbo-jumbo. But bear with us! The result we derive is of significance.

Multiplying both sides of (5) by $\big(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_m\big)$ from the left, we immediately obtain

$$\big(\boldsymbol{X}^T\boldsymbol{X} + \lambda \boldsymbol{I}_m\big)\, \boldsymbol{w} = \boldsymbol{X}^T \boldsymbol{y}. \tag{6}$$

Expanding the product on the left hand side then yields

$$\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w} + \lambda \boldsymbol{w} = \boldsymbol{X}^T \boldsymbol{y} \tag{7}$$

so that

$$\lambda \boldsymbol{w} = \boldsymbol{X}^T\big(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\big) \tag{8}$$

which, upon division by λ, gives

$$\boldsymbol{w} = \frac{1}{\lambda}\boldsymbol{X}^T\big(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\big). \tag{9}$$

Yet, this result seems rather useless. After all, why would we be interested in an expression where $\boldsymbol{w}$ occurs on both sides of the equation? However, we may brutally eliminate $\boldsymbol{w}$ from the right hand side simply by substituting

$$\boldsymbol{a} = \frac{1}{\lambda}\big(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}\big) \tag{10}$$

so that we obtain

$$\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{a}. \tag{11}$$

Now, the result is indeed interesting for (11) more clearly indicates than (5) that the parameter vector $\boldsymbol{w}$ lies in the linear span of the data vectors $\boldsymbol{x}_i$. In other words, the solution to the (regularized) least squares problem is a linear combination of the $\boldsymbol{x}_i$.

The vector $\boldsymbol{a}$ we introduced in (10) is called the *dual vector*. Note that, while the primal vector $\boldsymbol{w} \in \mathbb{R}^m$, the dual vector $\boldsymbol{a} \in \mathbb{R}^n$ where m and n are the dimension and number of data, respectively.

Next, things will get really interesting. Rearranging (10) and plugging in (11), we find

$$\begin{aligned}\lambda\boldsymbol{a} &= \boldsymbol{y} - \boldsymbol{X}\boldsymbol{w} &(12)\\ &= \boldsymbol{y} - \boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a} &(13)\end{aligned}$$

so that

$$\boldsymbol{y} = \big(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I}_n\big)\boldsymbol{a} \tag{14}$$

and vice versa

$$\boldsymbol{a} = \big(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I}_n\big)^{-1}\boldsymbol{y}. \tag{15}$$

where $\boldsymbol{I}_n$ is the $n \times n$ identity matrix.

The result in (15) is truly of significance because we realize that it is the solution to the so called *dual problem*

$$L(\boldsymbol{a}, \lambda) = \big\|\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a} - \boldsymbol{y}\big\|^2 + \lambda\big\|\boldsymbol{X}^T\boldsymbol{a}\big\|^2 \tag{16}$$

which we obtain from plugging $\boldsymbol{w} = \boldsymbol{X}^T\boldsymbol{a}$ into (4).

Let us verify this claim. Expanding (16), i.e. writing it in terms of a more clearly recognizable quadratic form in $\boldsymbol{a}$, we have

$$\begin{aligned}L(\boldsymbol{a}, \lambda) = \boldsymbol{a}^T\big(\boldsymbol{X}\boldsymbol{X}^T\big)^2\boldsymbol{a} - 2\boldsymbol{a}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{y}\\ + \boldsymbol{y}^T\boldsymbol{y} + \lambda\boldsymbol{a}^T\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a}\end{aligned}$$

and from

$$\frac{\partial L}{\partial \boldsymbol{a}} = 2\big(\boldsymbol{X}\boldsymbol{X}^T\big)^2\boldsymbol{a} - 2\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{y} + 2\lambda\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{a} \stackrel{!}{=} \boldsymbol{0},$$

we find

$$\big(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I}_n\big)\boldsymbol{a} = \boldsymbol{y}$$

as in (14).

Summarizing our derivation up to this point, we now have *two equivalent solutions* that both minimize the Lagrangian in (4) with respect to $\boldsymbol{w}$, namely

$$\begin{aligned}\boldsymbol{w} &= \big(\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I}_m\big)^{-1}\boldsymbol{X}^T\boldsymbol{y} &(17)\\ &= \boldsymbol{X}^T\big(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I}_n\big)^{-1}\boldsymbol{y}. &(18)\end{aligned}$$

Looking at these results, we emphatically emphasize the following:

- the expression in (17) requires inverting an $m \times m$ matrix
- the expression in (18) requires inverting an $n \times n$ matrix

Having said this, we recall our discussion in [8] where we saw that growing dimensionality is "detrimental" for the run time of least squares approaches. At the same time, all our practical examples so far [6]–[8] considered scenarios where $n \gg m$, i.e. where the number of data points clearly exceeded their dimensionality. In these cases our new solution in (18) will therefore be more demanding with respect to computation time than the one in (17). At the first sight, it therefore appears to be of limited practical interest.

A. Invoking the Kernel Trick

Yet, the dual solution is actually very appealing. This is because the matrix $\boldsymbol{X}\boldsymbol{X}^T$ in (18) is a Gram matrix whose elements correspond to inner products between training data vectors. In particular, we have

$$\big(\boldsymbol{X}\boldsymbol{X}^T\big)_{ij} = \boldsymbol{x}_i^T\boldsymbol{x}_j \tag{19}$$

which suggests that we may invoke the kernel trick.

To justify this claim, we recall that, in order to kernelize an algorithm, there are two necessary prerequisites:

1) any occurrences of the training data $\boldsymbol{x}_i$ must be in terms of inner products
2) the input variable $\boldsymbol{x}$, too, has to enter the computation in form of inner products.

For the training data $\boldsymbol{x}_i$ we just established the first prerequisite. For the input variable $\boldsymbol{x}$ we can verify the second one as well. If we resort to the dual solution for $\boldsymbol{w}$, we observe that our linear model in (3) becomes

$$y(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{w} = \boldsymbol{x}^T\boldsymbol{X}^T\big(\boldsymbol{X}\boldsymbol{X}^T + \lambda\boldsymbol{I}_n\big)^{-1}\boldsymbol{y} \tag{20}$$

where

$$\boldsymbol{x}^T\boldsymbol{X}^T = [\boldsymbol{x}^T\boldsymbol{x}_1 \ldots \boldsymbol{x}^T\boldsymbol{x}_n]. \tag{21}$$

The dual version of least squares regression therefore indeed allows for invoking the kernel trick. If we replace every inner product in (20) by a kernel evaluation, we obtain the following prediction for a new input $\boldsymbol{x}$

$$y(\boldsymbol{x}) = \boldsymbol{k}(\boldsymbol{x})^T\big(\boldsymbol{K} + \lambda\boldsymbol{I}_n\big)^{-1}\boldsymbol{y}. \tag{22}$$

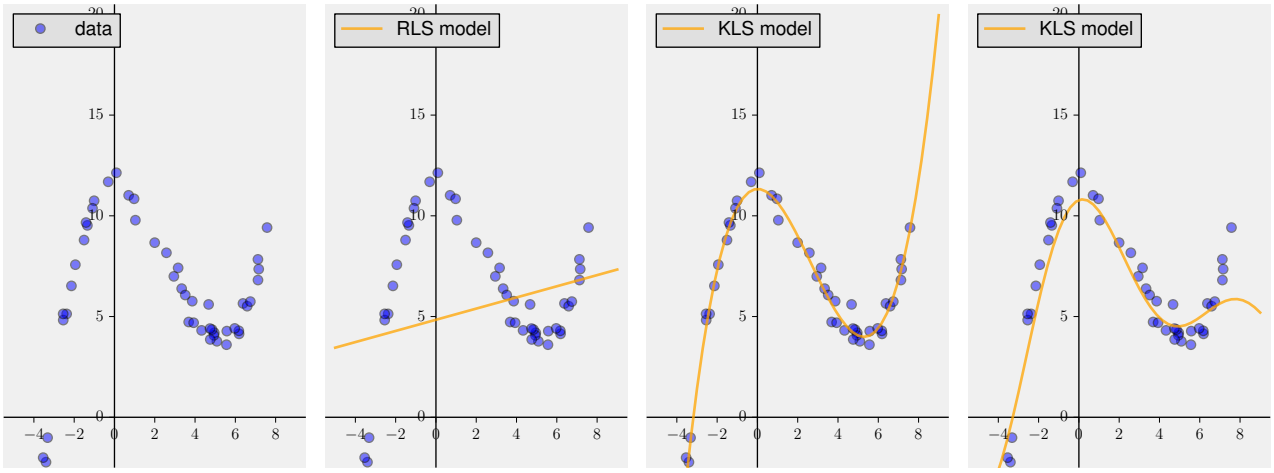
Obviously, the $n \times n$ kernel matrix $\boldsymbol{K}$ replaces the Gram matrix $\boldsymbol{X}\boldsymbol{X}^T$ of the training data. Each of its entries

$$K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) \tag{23}$$

results from evaluating a kernel function $k(\cdot, \cdot)$ on a corresponding pair of training data vectors.

The n-dimensional vector $\boldsymbol{k}(\boldsymbol{x})^T$ replaces $\boldsymbol{x}^T\boldsymbol{X}^T$ and its elements are given by

$$k_i(\boldsymbol{x}) = k(\boldsymbol{x}_i, \boldsymbol{x}). \tag{24}$$



(a) 2D data points (b) regularized linear regression (c) polynomial kernel regression (d) Gaussian kernel regression

Fig. 1: Examples of linear ridge regression and kernel ridge regression on a sample of two-dimensional data points (x_i, y_i) which were generated using $y_i = 0.1 \cdot x_i^3 - 0.8 \cdot x_i^2 + 11.5 + \epsilon$. While a linear model is unable to properly characterize this data, a third degree polynomial kernel automatically uncovers the cubic relation between input and output; a Gaussian kernel with $\sigma = 2.5$, on the other hand, produces an overly smooth model which at the boundaries of the sampling interval notably deviates from the training data.

B. Kernel Functions

Non-linear kernel functions allow us to implicitly introduce non-linearity into the least squares framework. That is, we are able to apply least squares techniques to non-linear data without having to specify a non-linear model.

Figure 1 compares ordinary least squares and kernelized least squares on a 2D data sample where there is no apparent linear relation between the input- and the output variable. For the kernelized least squares solution in Fig. 1c, we considered a third degree *polynomial kernel*

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \left(\boldsymbol{x}_i^T \boldsymbol{x}_j + 1\right)^3 \tag{25}$$

and, for the solution in Fig. 1d, we applied a *Gaussian kernel*

$$k(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\frac{1}{2\sigma^2}\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2}. \tag{26}$$

Apparently, the polynomial kernel automatically uncovered the underlying cubic relationship between x and y. On the other hand, the Gaussian kernel gave an overly smooth result which, especially at the boundaries of the training domain, notably deviates from the given data. Already this didactic example thus illustrates the need to know what kind of kernel to use. We will discuss possible solutions in more detail in a subsequent note. For the remainder of this note, we will focus on how to implement kernel least squares models in *Python*.

Listing 1: data generation for uni-variate regression

```
def create_data_2D(n, w, xmin=-4, xmax=8):
    x = rnd.random(n) * (xmax-xmin) + xmin
    X = np.vander(x, len(w))
    y = np.dot(X, w) + rnd.randn(n) * 0.5
    return x, y
```

IV. Practice

In this section, we discuss *Python* recipes for kernel least squares. Readers who would like to experiment with our examples should import the following *NumPy*, *SciPy*, and *Matplotlib* modules

```
# import numpy / numpy modules
import numpy as np
import numpy.linalg as la
import numpy.random as rnd
# import scipy modules
import scipy.spatial as spt
import scipy.sparse.linalg as sla
# import matplotlib (pyplot)
import matplotlib.pyplot as plt
```

Before we begin, we must point out a caveat: our recipes in this note mainly serve didactic purposes; they are not necessarily optimized for speed but intended to expose the underlying math. Of course we can devise more efficient code for kernel least squares but since it would be less readable, we postpone corresponding recipes to later.

To create a didactic 2D data sample $\{(x_i, y_i)\}_{i=1}^n$ such as in Fig. 1a, we sample n real numbers $x_{\min} \leq x_i \leq x_{\max}$, compute corresponding values y_i using a polynomial model, and then distort them to make the task of model fitting more interesting. Here, we accomplish this using *create_data_2D* in Listing 1. We already discussed this recipe in [7] and therefore do not elaborate on it again; suffice it to say that a plot of training data as in Fig. 1a can be generated as follows:

```
n = 50
wTrain = np.array([0.1, -0.8, 0.0, 11.5])
xTrain, yTrain = create_data_2D(n, wTrain)
plt.scatter(xTrain, yTrain)
plt.show()
```

We include the result in Fig. 1b in order to point out crucial differences between (regularized) least squares and kernel least squares. Regarding the former, we saw in [8] that, once 2D training data is available, we can compute the data matrix

```
XTrain = np.vander(xTrain, 2)
```

and then solve for the optimal parameter vector using the *SciPy* function *lsmr*

```
w = sla.lsmr(XTrain, yTrain, damp=1.)
```

where `damp` indicates the Lagrange multiplier in (4). Once `w` is available, we can create and plot N test data using

```
N = 2 * n
xTest = np.linspace(-4, 8, N)
XTest = np.vander(xTest, 2)
yTest = np.dot(XTest, w)
plt.scatter(xTest, yTest)
plt.show()
```

So far, this is nothing new but basically repeats our earlier considerations as to how to compute (3) once the parameters in $\boldsymbol{w}$ have been determined from training data.

We also note that we deliberately naïvely tried to fit a linear model to obviously non-linear data and, accordingly, got a result of questionable use. We further note that the non-linear least squares approach we discussed in [7] would have likely produced a result more fitting to our current example but would have required us to consider more involved input data. In other words, the approach in [7] would have required us to *explicitly* map our input data $\boldsymbol{x} = [x, 1]$ to, say, $\tilde{\boldsymbol{x}} = [x^3, x^2, x, 1]$.

However, the whole purpose of our present discussion is to demonstrate that, instead of using explicit transformations, we can achieve more fitting results by means of the *implicit* transformations realized by kernel functions. Next, we therefore consider *NumPy* and *SciPy* solutions to the problem of computing (22).

Assuming that training data `xTrain`, `yTrain`, and `XTrain` have been created as above, we first look at kernel least squares with polynomial kernels.

Given `XTrain` and a polynomial degree, say `p=3`, we apply *polyKernelMat* in Listing 2 to compute an $n \times n$ kernel matrix

```
K = polyKernelMat(XTrain, p)
```

where *polyKernelMat* is a straightforward and properly vectorized *NumPy* implementation of the polynomial kernel function in (25). Once the kernel matrix $\boldsymbol{K}$ is available, we compute $\big(\boldsymbol{K} + \lambda \boldsymbol{I}_n\big)^{-1}$ for instance with $\lambda = 1$

```
KI = la.inv(K + 1. * np.identity(n))
```

as well as the matrix-vector product $\big(\boldsymbol{K} + \lambda \boldsymbol{I}_n\big)^{-1}\boldsymbol{y}$

```
KIy = np.dot(KI, y)
```

Now, we have reached the point from which on kernel least squares fundamentally differs from the approaches we studied earlier! Following the latter, we might actually be tempted to compute

$$\boldsymbol{w} = \boldsymbol{X}^T\big(\boldsymbol{K} + \lambda \boldsymbol{I}_n\big)^{-1}\boldsymbol{y} \tag{27}$$

Listing 2: functions for polynomial kernels

```
def polyKernelMat(X, p):
    return (np.dot(X, X.T) + 1.)**p

def polyKernelVec(x, X, p):
    return (np.dot(x, X.T) + 1.)**p
```

as in (18) and then, given an input $\boldsymbol{x}$, compute the corresponding output according to

$$y(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{w} = \boldsymbol{x}^T\boldsymbol{X}^T\big(\boldsymbol{K} + \lambda \boldsymbol{I}_n\big)^{-1}\boldsymbol{y}. \tag{28}$$

But this would not make sense! The expression in (28) realizes neither ordinary- nor kernel least squares but crudely mixes both approaches. Although it contains a kernel matrix, it is not yet properly kernelized because it still involves an inner product in $\boldsymbol{x}$.

Put in rather drastic terms, we must never ever even think of using (28) but have to resort to (22) which we repeat here for emphasis:

$$y(\boldsymbol{x}) = \boldsymbol{k}(\boldsymbol{x})^T\big(\boldsymbol{K} + \lambda \boldsymbol{I}_n\big)^{-1}\boldsymbol{y}. \tag{29}$$

Invoking the kernel trick thus impacts the way we process test data: in order to be able to map an input $\boldsymbol{x}$ to an output y, we first need to compute a "kernel" vector $\boldsymbol{k}(\boldsymbol{x})$.

For our 2D example, we therefore proceed as follows. We compute N input test data using

```
xTest = np.linspace(-4, 8, N)
XTest = np.vander(xTest, 2)
```

then initialize a vector of N zeros to store our output test data

```
yTest = np.zeros(N)
```

and finally execute the following *for* loop to fill `yTest` with appropriate values

```
for i in range(N):
    k = polyKernelVec(XTest[i,:], XTrain, p)
    yTest[i] = np.dot(k, KIy)
```

Note our use of function *polyKernelVec* in Listing 2 to compute $\boldsymbol{k}(\boldsymbol{x})$. In particular, we call it with the i-th row of the $N \times 2$ test data matrix `XTest`; the second argument is the $n \times 2$ training data matrix `XTrain`; the third argument again indicates the polynomial degree of `p=3` in our case.

Apparently, this procedure is more involved than the least squares algorithms we studied earlier. Yet, once `xTest` and `yTest` have been properly computed, we may again plot the result using

```
plt.plot(xTest, yTest, '-')
plt.show()
```

and observe an output similar to Fig. 1c.

Next we discuss a solution for kernel least squares with Gaussian kernels. Again assuming that training data `xTrain`, `yTrain`, and `XTrain` are available, we compute an $n \times n$ kernel matrix

```
K = gaussKernelMat(XTrain, s)
```

Listing 3: functions for Gaussian kernels

```
def squaredEDM(X):
    V = spt.distance.pdist(X, 'sqeuclidean')
    D = spt.distance.squareform(V)
    return D

def gaussKernelMat(X, s):
    D = squaredEDM(X)
    K = np.exp(-0.5/s**2 * D)
    return K

def gaussKernelVec(x, X, s):
    d = np.sum((X-x)**2., axis=1)
    k = np.exp(-0.5/s**2. * d)
    return k
```

using the function *gaussKernelMat* in Listing 3.

Its arguments are the training data matrix `XTrain` and a variance parameter `s`. When called, *gaussKernelMat* first computes a squared Euclidean distance matrix $\boldsymbol{D}$ where $D_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2$ for training data vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. In order to compute this matrix, we apply the function *squaredEDM* in Listing 3 which we discussed in [9]. Once $\boldsymbol{D}$ is available, *gaussKernelMat* computes and returns a matrix whose entries are determined by the Gaussian kernel in (26).

We then proceed as above and compute

```
KI = la.inv(K + 1. * np.identity(n))
KIy = np.dot(KI, y)
```

then create test data

```
xTest = np.linspace(-4, 8, N)
XTest = np.vander(xTest, 2)
yTest = np.zeros(N)
```

and finally execute the following *for* loop to fill `yTest` with appropriate values

```
for i in range(N):
    k = gaussKernelVec(XTest[i,:], XTrain, s)
    yTest[i] = np.dot(k, KIy)
```

Here, we obviously use *gaussKernelVec* in Listing 3 to compute $\boldsymbol{k}(\boldsymbol{x})$. This recipe provides a fairly efficient way of computing a distance vector $\boldsymbol{d}$ where $d_i = \|\boldsymbol{x} - \boldsymbol{x}_i\|^2$. This vector is then used to produce a vector whose entries are determined by the kernel in (26).

Once `xTest` and `yTest` have been computed, we can use

```
plt.plot(xTest, yTest, '-')
plt.show()
```

to produce a plot similar to Fig. 1d. However, we point out that, this time, the result will strongly depend on our choice of `s` and we encourage the reader to experiment with different choices to verify this.

V. Discussion

The above recipes illustrate how to implement kernel least squares using *NumPy* and *SciPy*. Other, more properly vectorized solutions are possible but we ignored those because our main goal in this note was to comprehensively convey how to put the underlying math into action.

We saw that the implementations of the kernel least squares are more involved than the solutions in [6]–[8]. But we also saw that invoking the kernel trick can produce non-linear models even if the input is linear. That is, in our practical examples we dealt with input of the form $[x, 1]$ rather than $[x^3, x^2, x, 1]$ as in [7] but still obtained result that modeled the behavior of data points sampled from a cubic polynomial.

In other words, rather than having to worry about explicit transformations of input variables, we can resort to the kernel trick which implicitly transforms the data. Nevertheless, our practical examples also showed that choosing a proper kernel is important. The problem as to how this can be done is fundamental and we will not discuss it further at this point.

Finally, the flexibility offered by kernel least squares comes at a price. While ordinary least squares solves an optimization problem in the order of the data dimensionality m, kernel least squares has to solve an optimization problem in the order of the number of data points n. In many modern practical settings (BIG DATA), we have $n \gg m$. All this goes to say that the kernel trick should not be invoked naïvely. Analysts need to know what they are doing and which solutions are most suitable to the problems they are dealing with.

VI. Notes and References

Often, the technique discussed in this note is also referred to as *kernel ridge regression* [10], [11]. Examples of practical applications can be found in areas such as computer vision or signal processing [12]–[14].

References

[1] J. Shaw-Taylor and N. Cristianini, *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[2] C. Bauckhage, "Lecture Notes on the Kernel Trick (I)," researchgate.net, Mar. 2015, https://dx.doi.org/10.13140/2.1.4524.8806.

[3] ——, "Lecture Notes on the Kernel Trick (III)," researchgate.net, Mar. 2015, https://dx.doi.org/10.13140/RG.2.1.2471.6322.

[4] T. Oliphant, "Python for Scientific Computing," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[5] J. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, 2007.

[6] C. Bauckhage, "NumPy / SciPy Recipes for Data Science: Ordinary Least Squares Optimization," researchgate.net, Mar. 2015, https://dx.doi.org/10.13140/2.1.3370.3209/1.

[7] ——, "NumPy / SciPy Recipes for Data Science: Non-Linear Least Squares Optimization," researchgate.net, Mar. 2015, https://dx.doi.org/10.13140/RG.2.1.1431.2484.

[8] ——, "NumPy / SciPy Recipes for Data Science: Regularized Least Squares Optimization," researchgate.net, Mar. 2015, https://dx.doi.org/10.13140/RG.2.1.2565.3286.

[9] ——, "NumPy / SciPy Recipes for Data Science: Squared Euclidean Distance Matrices," researchgate.net, Oct. 2014, https://dx.doi.org/10.13140/2.1.4426.1127.

[10] C. Saunders, A. Gammerman, and V. V, "Ridge Regression Learning Algorithm in Dual Variables," in *Proc. ICML*, 1998.

[11] V. Vovk, "Kernel Ridge Regression," in *Empirical Inference*, B. Schölkopf, Z. Luo, and V. Vovk, Eds. Springer, 2013, pp. 105–116.

[12] S. An, W. Liu, and S. Venkatesh, "Face Recognition Using Kernel Ridge Regression," in *Proc. CVPR*, 2007.

[13] C. Bauckhage, "Tensor-Based Filter Design Using Kernel Ridge Regression," in *Proc. ICIP*, 2007.

[14] W. Liu, P. Pokharel, and J. Principe, "The Kernel Least-Mean-Square Algorithm," *IEEE Trans. Signal Processing*, vol. 56, no. 2, pp. 543–554, 2008.