Problem Set 2

Max Ruiz Luyten

Edited by Adityanarayanan Radhakrishnan

January 27, 2023

The goals of this problem set are to (1) demonstrate the simplicity of linear and kernel regression ; and (2) improve understanding of the theory behind linear and kernel regression. Again, the theoretical and coding exercises below review topics from linear algebra, analysis, and probability. We use (T) to denote theory exercises and (C) to denote coding exercises.

Linear Regression

The following problems depend on material through Lecture 2.

Problem 1 (C). Generate a vector $w^* \in \mathbb{R}^{1\times d}$ with entries drawn i.i.d. from a standard normal distribution. In this problem, we will estimate w^* via linear regression on a random set of samples and labels.

(a) For $d = 100, n = 10$, generate a matrix $X \in \mathbb{R}^{d\times n}$ (number of features x number of samples) with the entries of X drawn i.i.d from a standard normal distribution. Lastly, generate labels $y = w^*X$. Compute the minimum norm solution given by $\hat{w} = yX^\dagger$ via the `pinv` function.

(b) Compute the mean squared error (MSE) between $\hat{w}$ and w^*.

(c) Repeat (a) and (b) for $n \in \{1, 10, 20, 30, 40, 50, 60, 70, 80, 90\}$ and generate a plot of the corresponding MSEs vs. n.

(d) Repeat (c) for at least 30 random seeds and average the MSE across seeds for each value of n when generating the plot of MSE vs. n as in (c).

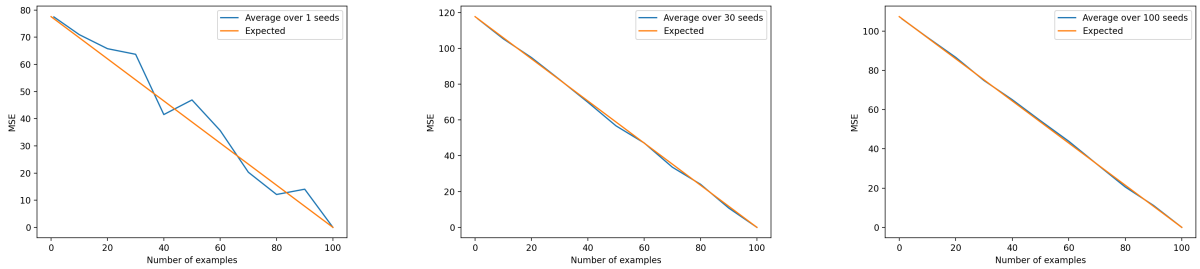
(e) Compare the plot generated in (d) with that of the function $f(n) = \|w^*\|_2^2\left(1 - \frac{n}{100}\right)$.

All the questions are contained in the following code. Figure 1 presents the required plots.

```
import matplotlib.pyplot as plt
import numpy as np

d = 100                              # Input dimensions
seeds = 100                          # Number of seeds for averaging
nit = [1] + list(range(10,101,10))   # Iterator for number of examples
w = np.random.normal(size=(1,d))     # True weights

# COMPUTE MSE
MSE = []
for n in nit:
    X = np.random.normal(size=(seeds,d,n))  # Sample examples
    y = w @ X                               # Compute labels
    wpErr = (w - y@np.linalg.pinv(X))       # Diff btw true and predicted w
    MSE.append(np.mean(np.square(np.linalg.norm(wpErr,axis=-1)))) # Mean of MSE

```

```
# PLOTS
avgL, = plt.plot(nit, MSE)                          # Avg
ExpL, = plt.plot([0, d], [np.inner(w,w), 0])        # Expected
plt.xlabel("Number of examples")
plt.ylabel("MSE")
plt.legend([avgL, ExpL], ['Average over {} seeds'.format(seeds), 'Expected'])
plt.show()
```

Listing 1: Code for problem 2.1



(a) 1 seed (b) 30 seeds (c) 100 seeds

Figure 1: MSE for the estimation $\hat{\omega}$ of ω^* averaged over different number of seeds as a function of the number of examples.

Recall that $\hat{\omega} = yX^\dagger$ and $y = \omega^* X$, so $\hat{\omega} = \omega^* X X^\dagger$. As a consequence $\mathbb{E}[||\hat{\omega} - \omega^*||_2^2] = \mathbb{E}[||\omega^* X X^\dagger - \omega^*||_2^2] = \mathbb{E}[(\omega^*)^T (XX^\dagger - I)^T (XX^\dagger - I)\omega^*]$. Now $(XX^\dagger - I)$ is symmetric and corresponds to the projection operator onto the space generated by the singular vectors that have a zero singular value. This can be seen by taking the SVD (see problem 2 (d)).

As the columns of X are chosen according to an isotropic distribution, it can be seen that the singular vectors are uniformly distributed on the sphere so that $||\omega^*||^2 \mathbb{E}[\frac{(\omega^*)^T}{||\omega^*||}(XX^\dagger - I)\frac{\omega^*}{||\omega^*||}] = ||\omega^*||^2 \left(1 - \frac{n}{d}\right)$, the examples being in $\mathbb{R}^d$.

Problem 2 (T, *). Let $(X, y) = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$ denote a training samples and labels with $(x^{(i)}, y^{(i)}) \in \mathbb{R}^d \times \mathbb{R}$. To find $w \in \mathbb{R}^{1\times d}$ that minimizes the squared loss

$$\mathcal{L}(w) = \frac{1}{2} \sum_{i=1}^{n} \|y^{(i)} - w x^{(i)}\|_2^2 \,, \tag{1}$$

we use gradient descent with learning rate η and initialization $w^{(0)}$.

(a) Write out the gradient descent update at timestep t.

We know that the iterative step for the gradient descent algorithm is defined by

$$\omega^{(t+1)} = \omega^{(t)} - \eta \nabla_\omega \mathcal{L}(\omega^{(t)}), \qquad \eta \in (0, 1). \tag{2}$$

We start by computing the gradient of the L^2 norm with respect to the parameters ω. Notice that $\mathcal{L}(w) = \frac{1}{2}(Y - wX)(Y - wX)^T$. Matrix calculus yields an expression for $\frac{\partial \mathcal{L}(x)}{\partial \omega}$ in terms of X, y, ω (see problem 1.5):

$$\begin{aligned} 2\frac{\partial \mathcal{L}(x)}{\partial \omega} &= \frac{\partial}{\partial \omega}\left[(\omega X - y)(\omega X - y)^T\right] \\ &= \frac{\partial}{\partial \omega}(\omega X X^T \omega^T) - 2\nabla_\omega(\omega X y^T) \\ &= 2\omega X X^T - 2 y X^T. \end{aligned}$$

In the second equality we have used that $yX^T\omega^T$ is a scalar and therefore it is equal to its transpose.

Our iteration 2 can thus be rewritten as

$$\omega^{(t+1)} = \omega^{(t)} - \eta(\omega^{(t)} X X^T - y X^T).$$

(b) Give a closed form for $w^{(t)}$, the solution given by gradient descent at timestep t.
Hint: Write out the first few updates $w^{(1)}, w^{(2)}$ and see if you can proceed via induction.

Let's take the SVD of $X = U\Sigma V^T$, by which $XX^T = U\Sigma^2 U^T$. Using $UU^T = 1$ we get

$$\begin{aligned}\omega^{(t+1)} &= \omega^{(t)} - \eta(\omega^{(t)} U\Sigma^2 U^T - y(U\Sigma V^T)^T) \\ &= \omega^{(t)} U(I - \eta\Sigma^2)U^T + \eta y V\Sigma U^T,\end{aligned}$$

Where I stands for the identity matrix with dimensions $\max\{n,d\} \times \max\{n,d\}$. We now define $\Phi = I - \eta\Sigma^2$ and use the recursion to write $\omega^{(t+1)}$ as a function of $\omega^{(0)}$. By using again the orthonormality of U we can achieve this as follows:

$$\begin{aligned}\omega^{(t+1)} &= \omega^{(t-1)} U\Phi^2 U^T + \eta y V\Sigma(I + \Phi)U^T \\ &= \omega^{(0)} U\Phi^{t+1}U^T + \eta y V\Sigma(I + \Phi + \Phi^2 + \cdots + \Phi^t)U^T\end{aligned}$$

As Φ is diagonal, the series that appears can be treated element-wise, i.e. all non diagonal terms are the sum of zeros and the diagonal ones are just $(\sum_{i=0}^t \Phi^i)_{kk} = \sum_{i=0}^t \Phi_{kk}^i$. Let $r \leq \min(n,d)$ be the rank of X, so that $\Phi_{kk} = 1 - \eta\sigma_k^2$ for $k \leq r$ and $\Phi_{kk} = 1$ for $k > r$. Then, the series can be written in closed form as $(\sum_{i=0}^t \Phi^i)_{kk} = \frac{1-(1-\eta\sigma_k^2)^{t+1}}{\eta\sigma_k^2}$ for $k \leq r$ and $(\sum_{i=0}^t \Phi^i)_{kk} = t+1$ for $k > r$.

However, multiplying this matrix by $\eta\Sigma$ ignores all the 1 terms in the last $\max d, n-r$ rows and gives a square matrix where we have $\frac{1-(1-\eta\sigma_k^2)^{t+1}}{\sigma_k}$ on the diagonal where $\sigma_i \neq 0$ and 0 otherwise. Denoting such matrix as $\tilde{\Sigma}_{t+1}$ we get the closed form

$$\omega^{(t)} = \omega^{(0)} U\Phi^t U^T + y V\tilde{\Sigma}_{t+1} U^T$$

(c) Find the largest $c > 0$ such that when $\eta < c$, $\lim_{t\to\infty} w^{(t)} = w^{(\infty)}$ exists and can be decomposed as:

$$w^{(\infty)} = f_1(w^{(0)}, X) + f_2(X, Y)$$

where $f_1(w^{(0)}, X), f_2(X, y) \in \mathbb{R}^{1\times d}$ such that $f_1(W^{(0)}, X) f_2(X, y)^T = \mathbf{0}$.
Hint: $f_2(X, y)$ should be the minimum ℓ_2 norm solution derived in Lecture 2.

Notice $\Phi^t = diag((1-\eta\sigma_1^2)^t, \ldots, (1-\eta\sigma_r^2)^t, 1, \ldots, 1)$, so it converges if and only if $0 < \eta < \frac{2}{\sigma_i^2}$ for all $i = 1, \ldots, r$, and under such condition, it converges to $diag((0)_{i=1}^r, (1)_{i=r+1}^d) = \Sigma^\perp$. Similarly, the elements of $\tilde{\Sigma}_t$ converge under the same conditions, and they do so to $diag((\frac{1}{\sigma_i})_{i=1}^r, (0)_{i=r+1}^d) = \Sigma^\dagger$. Hence, the second term converges to $yV\Sigma^\dagger U^T = yX^\dagger$.

Noting $f_1(w^{(0)}, X) = w^{(0)} U\Sigma^\perp U^T$, $f_2(X, y)^T = y\Sigma^\dagger$, then $f_1(W^{(0)}, X) f_2(X, y)^T = w^{(0)} U\Sigma^\perp \cancel{U^T U} \Sigma^\dagger V^T y^T = 0$, as $\Sigma^\perp\Sigma^\dagger = 0$. Hence, the largest $c > 0$ for all this to be possible is $c = 2/||\Sigma||_2^2$.

(d) Prove that when $w^{(0)} = \mathbf{0}$, then $w^{(\infty)}$ computed in (c) corresponds to the minimum ℓ_2 norm solution for the loss in Equation (1).
Hint: Assume there exists another solution, $\tilde{w}$. Use an appropriate orthogonal decomposition of $\tilde{w}$ and apply the triangle inequality.

If the initial point is 0 gradient descent converges to $yX^\dagger$. We just have to prove then that $yX^\dagger$ corresponds to the minimum norm solution. Suppose an arbitrary $w \in \mathcal{R}^d$ $||y - wX||_2^2 = ||y - yX^\dagger X + (yX^\dagger - w)X||_2^2 = ||y(I - X^\dagger X) + (yX^\dagger - w)X||_2^2$. Note that $(I - X^\dagger X)^2 = (I - 2X^\dagger X + X^\dagger X X^\dagger X) = I - X^\dagger X$ and $X(I - X^\dagger X) = 0$, so $(I - X^\dagger X)$ acts as the projection operator onto the space orthogonal to the rows of X. Thus $y(I - X^\dagger X)$ lies in the space orthogonal to the rows of X and $(yX^\dagger - w)X$ is in the span of the rows of X, so the Pythagorean theorem gives $||y - wX||_2^2 = ||y(I - X^\dagger X)||^2 + ||(yX^\dagger - w)X||_2^2$. Thus w minimizes $||y - wX||_2^2$ if and only if $(yX^\dagger - w)$ is orthogonal to the columns of X, i.e. $w = yX^\dagger + u$, where $u \in span(\{x^{(i)}\})$.

However, if w is such a vector, then $w = yX^\dagger + w_\perp$, where w lies in the space orthogonal to the span of the examples, and then $||w||_2^2 = ||yX^\dagger||_2^2 + ||w_\perp||$, and thus it is clear that $||w||_2^2 \geq ||yX^\dagger||_2^2$, which proves that $yX^\dagger$ is the minimum norm solution.

Kernel Regression

The following problems depend on material through Lecture 3.

Problem 3 (T). Recall that the feature map $\psi(x)$ is such that $K(x, x') = \langle \psi(x), \psi(x') \rangle$. Write out a feature map $\psi(x)$ for the Gaussian kernel $K(x, x') = \exp\left(-L\|x - x'\|_2^2\right)$ where $x, x' \in \mathbb{R}^d$.
Hint: We will derive a feature map ψ to the space $\ell_2(\mathbb{C})$ with complex inner product in Lecture 4. There is a simpler feature map that can be derived by writing the Taylor series for e^z for $z \in \mathbb{R}$.

We start by using that $e^{-L||x-x'||_2^2} = e^{-L||x||_2^2} e^{-L||x'||_2^2} e^{-2L\langle x, x' \rangle}$.

Now expand the last term by using that exp is analytic (i.e. we use the Taylor series expansion around 0).

$$\begin{aligned} e^{-L\langle x, x' \rangle} &= \sum_{m=0}^{\infty} \frac{L^m}{m!} \langle x, x' \rangle^m \\ &= \sum_{m=0}^{\infty} \frac{L^m}{m!} \left(\sum_{k=0}^{d} x_k x'_k \right)^m \\ &= \sum_{m=0}^{\infty} \frac{L^m}{m!} \sum_{p_1 + \cdots + p_d = m} \binom{m}{p_1, \ldots, p_d} x_1^{p_1} \cdots x_d^{p_d} x_1'^{p_1} \cdots x_d'^{p_d} \end{aligned}$$

Notice that this is equivalent to the "inner product" of the infinite vectors given by the transformation

$$\tilde{\phi}(x) = \left(\sqrt{\frac{(2L)^m}{p_1! p_2! \cdots p_d!}} x_1^{p_1} x_2^{p_2} \cdots x_d^{p_d} \right)_{p_1, \ldots, p_d \in \mathbb{N} \cup \{0\}}$$

Thus we can take the function that also includes the term $e^{-L||x||_2^2}$,

$$\phi(x) = \exp\left(-L||x||_2^2\right) \tilde{\phi}(x)$$

Therefore the Gaussian kernel works on an infinite dimensional space.

Problem 4 (T). Prove the following proposition from Lecture 3:

Proposition 1. *Let $\mathcal{H}$ be a Hilbert space with inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$. Let $K : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$ such that $K(x, \tilde{x}) = \langle \psi(x), \psi(\tilde{x}) \rangle_{\mathcal{H}}$ for $\psi : \mathbb{R}^d \to \mathcal{H}$. Then K is a positive semi-definite kernel.*

Hint: Use Definition 2 from Lecture 3. First try the case of $n = 2$ and write the sum as a square of a quantity.

The exercise follows from the linearity of a real inner product

$$\begin{aligned}\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j K(x^{(i)}, x^{(j)}) &= \sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j \langle \psi(x^{(i)}), \psi(x^{(j)}) \rangle \\ &= \sum_{i=1}^{n}\sum_{j=1}^{n} \langle \psi(c_i x^{(i)}), \psi(c_j x^{(j)}) \rangle \\ &= \langle \sum_{i=1}^{n} \psi(c_i x^{(i)}), \sum_{j=1}^{n} \psi(c_j x^{(j)}) \rangle\end{aligned}$$

Noting $\sum_{i=1}^{n} \psi(c_i x^{(i)}) = v \in \mathcal{H}$ we have

$$\sum_{i=1}^{n}\sum_{j=1}^{n} c_i c_j K(x^{(i)}, x^{(j)}) = ||v||_2^2 \geq 0$$

Problem 5 (C). Let $f : \mathbb{R} \to \mathbb{R}$ such that $f(x) = x^2 + x + 1$. Sample 11 points evenly distributed from -1 to 1 for training data (i.e. take $[-1, -.8, -.6, -.4, -.2, 0, .2, .4, .6, .8, 1]$ for the training samples). Then let $y = f(X)$ (we apply f to each point X).

(a) Using (X, y) as training data solve kernel regression using the Gaussian kernel, $K(x, \tilde{x}) = \exp\left(-L\|x - \tilde{x}\|_2^2\right)$, for $L = \{.01, .05, 1, 10, 100\}$. Plot the predictions of the learned function on 1000 points on the interval $[-1, 1]$.

(b) Using (X, y) as training data solve kernel regression using the Laplace kernel, $K(x, \tilde{x}) = \exp\left(-L\|x - \tilde{x}\|_2\right)$, for $L = \{.01, .05, 1, 10, 100\}$. Plot the predictions of the learned function on 1000 points on the interval $[-1, 1]$.

The required plots are shown in figure 2. Notice that when $L \to 0^+$ the solutions seem to tend to a piecewise linear function, and for $L \to \infty$ the solutions for the Laplacian kernel tend to delta functions, and the solutions for the Guassian kernels also form peaks around the true value at each training example, but each peak has a gaussian shape.

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.gridspec import GridSpec

# Given vectors examples x, y returns an array with ||x_i - y_j||
# Params:
#   x (nxd ndarray): set 1 of examples
#   y (mxd ndarray): set 2 of examples
# Returns: (n x m ndarray) where component ij corresponds to ||x_i - y_j||_2
def outerDifNorm(x,y):
    return np.linalg.norm(x[:, None, :] - y[None, :, :], axis = -1)

# Given x,y and a bandwidth L returns the Gaussian kernel between x and y
def gaussian(x, y, L):
    return np.exp(-L*(outerDifNorm(x,y)**2))

# Given x,y and a bandwidth L returns the Laplace kernel between x and y
def laplacian(x, y, L):
    return np.exp(-L*outerDifNorm(x,y))

Xtr = np.linspace(-1,1,11)
ytr = Xtr**2+Xtr+1

Xtest = np.linspace(-1,1,1000)
```

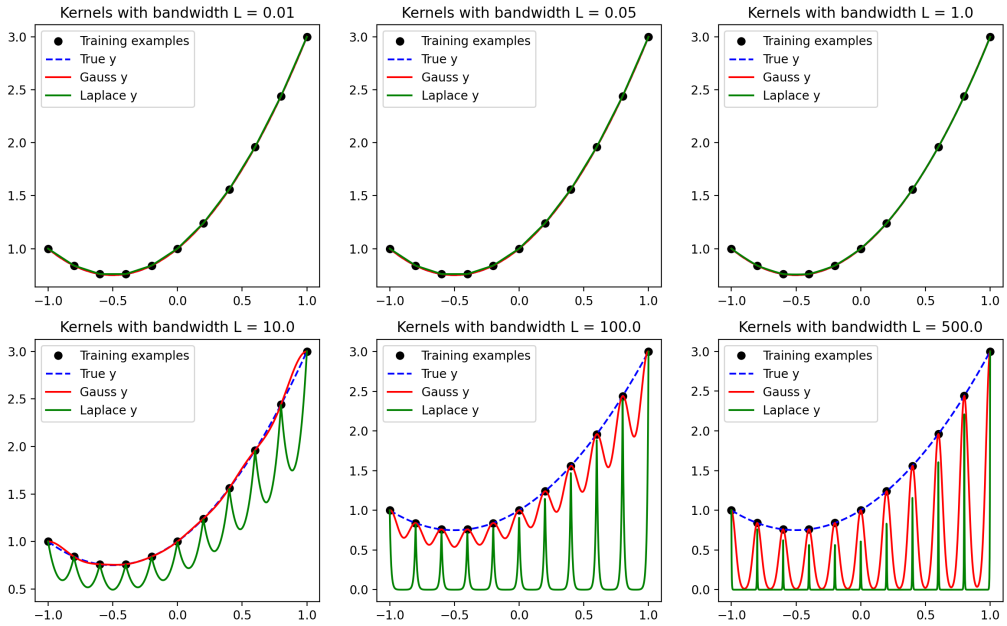


Figure 2: Plot of learned kernel models with the Gaussian and Laplacian kernels for different bandwidths.

```
ytest = Xtest**2+Xtest+1

Lvals = np.array([0.01, 0.05, 1, 10, 100, 500])

fig = plt.figure(figsize=(30, 10))
gs = GridSpec(nrows=2, ncols=3)
i = 0
for L in Lvals:     # We could vectorize over L by sacrificing readabilty
    Kg = gaussian(Xtr[:, None], Xtr[:, None], L)
    Kl = laplacian(Xtr[:, None], Xtr[:, None], L)

    wg = np.linalg.solve(Kg, ytr)
    wl = np.linalg.solve(Kl, ytr)

    ygpred = gaussian(Xtest[:, None], Xtr[:, None], L)@wg
    ylpred = laplacian(Xtest[:, None], Xtr[:, None], L)@wl

    ax = fig.add_subplot(gs[int(i/3),int(i%3)])
    i = i+1
    ax.plot(Xtr, ytr, 'ko', label='Training examples')
    ax.plot(Xtest, ytest, 'b--', label='True y')
    ax.plot(Xtest, ygpred, 'r', label='Gauss y')
    ax.plot(Xtest, ylpred, 'g', label='Laplace y')
    ax.title.set_text('Kernels with bandwidth L = {}'.format(L))
    ax.legend()
plt.show()
```

Listing 2: Code for problem 2.5

Kernel Regression with EigenPro

The following exercise will make use of the EigenPro library [2], which allows for solving kernel regression when the number of samples n is extremely large (on the order of millions of samples). Below, we provide

some links to implementations of EigenPro on the CPU and GPU. Feel free to use whichever implementation is more convenient.

1. Scikit-learn implementation. This implementation runs only on the CPU but is perhaps the easiest one to work with since it follows the standard scikit-learn API.
2. EigenPro Tensorflow and EigenPro Pytorch. These implementations both have GPU compatibility, although the documentation is a bit sparser. An example implementation is provided in the `run_mnist.py`, and we provide additional information about usage of this library in Appendix **??**.

Problem 6 (C. For $d = 100, n = 20000$, generate $X \in \mathbb{R}^{n \times d}$ where the entries are drawn i.i.d. from a standard normal distribution. Let $f : \mathbb{R}^d \to \mathbb{R}$ be the function $f(x) = \sum_{i=1}^{d} \tanh x_i$, and let $y = f(X) \in \mathbb{R}^n$ where $f(X)_j = f(X_{j,:})$ (i.e. f is applied to each row of X). Lastly, for $n_t = 1000$, generate test samples $X_t \in \mathbb{R}^{n_t \times d}$ where the entries are drawn i.i.d. from a standard normal distribution and test labels $y_t = f(X_t) \in \mathbb{R}^{n_t}$. **Make sure to set the random seed in numpy so that experiments are reproducible.**

(a) Initialize an EigenPro regressor model with the following parameters: `n_epoch` $= 10$, `kernel` $=$ 'rbf'. Solve kernel regression using the training data (X, y) and time how long it takes to fit the training data (e.g. with the sklearn implementation, time how long the `fit` function takes). Compute and report the MSE on the training data and test data.

(b) Initialize a sklearn support vector regressor (SVR) with the following parameters `kernel` $=$ 'rbf', `C` $=$ 10000. Solve kernel regression using the training data (X, y) and time how long it takes to fit the training data (e.g. with the sklearn implementation, time how long the `fit` function takes). Compute and report the MSE on the training data and test data.
Remark: If the fit takes more than 15 minutes, report this and move on to the next part.

(c) Returning to the EigenPro regressor model, repeat part (a) using `kernel` $=$ 'laplace' and `gamma` $\in \{1/200, 1, 10\}$. Report the new training and test MSEs.

We present the code for problem 6c below and the code for the remaining parts simply adjusts the paramters in the EigenPro regressor.

```
import sklearn
from sklearn_extra.kernel_methods import EigenProRegressor
import numpy as np
from numpy.linalg import norm, solve
from sklearn.svm import SVR
import time

def f(x):
    return np.tanh(x).sum(axis=-1).reshape(-1, 1)


def main():
    SEED = 17
    np.random.seed(SEED)
    d = 100
    n = 20000

    X = np.random.randn(n, d)
    y = f(X)

    bandwidth = 10
    gamma = 1/ (2 * bandwidth**2)

    model = EigenProRegressor(n_epoch=10, kernel='laplace', gamma=gamma)
    start = time.time()
    model.fit(X, y)
    end = time.time()
    print("EigenPro Time: ", end - start)
```

```
    y_pred = model.predict(X)
    train_error = np.mean(np.square(y_pred - y))
    print("Train Error: ", train_error)

    n_test = 1000
    X_test = np.random.randn(n_test, d)
    y_test = f(X_test)
    y_pred = model.predict(X_test)
    test_error = np.mean(np.square(y_pred - y_test))
    print("Test Error: ", test_error)


    model = SVR(kernel='rbf', C=10000)
    y = y.reshape(-1)
    start = time.time()
    model.fit(X, y)
    end = time.time()
    print("Sklearn SVR Time: ", end - start)
    y_pred = model.predict(X)
    train_error = np.mean(np.square(y_pred - y))
    print("Train Error: ", train_error)

    n_test = 1000
    y_test = y_test.reshape(-1)
    y_pred = model.predict(X_test)
    test_error = np.mean(np.square(y_pred - y_test))
    print("Test Error: ", test_error)

if __name__ == "__main__":
    main()
```

Listing 3: Code for problem 2.6

While not required, the following problem is recommended for those who are interested in applying kernel methods to larger, more realistic datasets. We recommend doing the following problems using a GPU implementation of EigenPro.

Problem 7 (C, *). Load the CIFAR10 [1] image classification dataset. CIFAR10 contains 10 classes of color images of size 32×32. There are 50k training examples and 10k test examples, i.e. the training sample matrix, X, should have shape $50000 \times 32 \times 32 \times 3$ and the training labels, y, should have shape 50000×10 (where each row of y is a one-hot vector indicating the class label).

(a) After vectorizing the training samples, i.e. reshaping X into a matrix $X_v \in \mathbb{R}^{50000 \times 32 \cdot 32 \cdot 3} = \mathbb{R}^{50000 \times 3072}$, use an EigenPro regressor with the Laplace kernel to solve kernel regression. Report the training time, the training MSE, training accuracy, test MSE, and test accuracy.
Remarks: EigenPro should take roughly 10 seconds on the GPU to achieve 100% training accuracy, and the test accuracy should be between 50-60%.

```
from torchvision.datasets import CIFAR10
import torchvision.transforms as transforms
from EigenPro.eigenpro import FKR_EigenPro
from EigenPro.kernel import laplacian
import numpy as np
import torch

PATH_2_CIFAR = "/home/mluyten/raw_datasets"

# Load dataset
train_set = CIFAR10(root=PATH_2_CIFAR, train=True,
                    transform=transforms.ToTensor(), download=False)
test_set = CIFAR10(root=PATH_2_CIFAR, train=False,
                   transform=transforms.ToTensor(), download=False)

# Flatten examples
```

```
trSet = np.array([train_set[j][0].numpy()
                  for j in range(len(train_set))], dtype=np.float32)
tstSet = np.array([test_set[j][0].numpy()
                   for j in range(len(test_set))], dtype=np.float32)
trSet = trSet.reshape((-1, 3*(32**2)))
tstSet = tstSet.reshape((-1, 3*(32**2)))

# One-hot encoding
identity = np.eye(10, dtype=np.float32)
trLab = identity[np.array(train_set.targets)]
tstLab = identity[np.array(test_set.targets)]

# Choose a device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Define kernel
def k_f(x, y): return laplacian(x, y, 10)

# Load model
ydim = trLab.shape[1]
model = FKR_EigenPro(k_f, trSet, ydim, device=device)

# Train model with EigenPro
res = model.fit(x_train=trSet, y_train=trLab, x_val=tstSet, y_val=tstLab,
                epochs=range(9), mem_gb=24, bs=None)
```

Listing 4: Code for problem 2.7

Output of the above code on a GPU, where it takes 3.11s to reach 0% error:

```
train error: 89.80% val error: 90.00% (0 epochs, 0.00 seconds)  train l2: 1.00e-01  val l2: 1.00e-01
train error: 0.34%  val error: 45.10% (1 epochs, 1.55 seconds)  train l2: 1.74e-02  val l2: 6.52e-02
train error: 0.00%  val error: 42.28% (2 epochs, 3.11 seconds)  train l2: 4.22e-03  val l2: 6.06e-02
train error: 0.00%  val error: 41.80% (3 epochs, 4.66 seconds)  train l2: 1.16e-03  val l2: 5.95e-02
train error: 0.00%  val error: 41.70% (4 epochs, 6.22 seconds)  train l2: 3.43e-04  val l2: 5.91e-02
train error: 0.00%  val error: 41.62% (5 epochs, 7.78 seconds)  train l2: 1.16e-04  val l2: 5.90e-02
train error: 0.00%  val error: 41.51% (6 epochs, 9.34 seconds)  train l2: 4.10e-05  val l2: 5.90e-02
train error: 0.00%  val error: 41.52% (7 epochs, 10.90 seconds) train l2: 1.50e-05  val l2: 5.90e-02
train error: 0.00%  val error: 41.46% (8 epochs, 12.47 seconds) train l2: 6.24e-06  val l2: 5.90e-02
```

Thus after 8 epochs we have a training time of 12.47s, a training accuracy of 100%, a train MSE of $6.24 \cdot 10^{-6}$, a test accuracy of 58.54% and a test MSE of $5.90 \cdot 10^{-2}$.

Output of the above code on my CPU (Laptop), where it takes 5min 30s to reach 0% training error:

```
train error: 89.80% val error: 90.00% (0 epochs, 0.00 seconds)  train l2: 1.00e-01  val l2: 1.00e-01
train error: 0.32%  val error: 45.11% (1 epochs, 164.04 seconds)  train l2: 1.73e-02  val l2: 6.52e-02
train error: 0.00%  val error: 42.26% (2 epochs, 327.38 seconds)  train l2: 4.14e-03  val l2: 6.06e-02
train error: 0.00%  val error: 41.80% (3 epochs, 490.43 seconds)  train l2: 1.12e-03  val l2: 5.95e-02
```

References

[1] A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, University of Toronto, 2009.

[2] S. Ma and M. Belkin. Diving into the shallows: a computational perspective on large-scale shallow learning. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2017.