Problem Set 2

Name:

Due Midnight on January 21, 2022

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. We use the flag **Required** to denote problems that are required for all students. We lastly indicate more involved problems with a *****, and these will be equivalent to solving 2 non-$*$ problems.

Students should do all the required problems and at least 3 of the remaining problems (for a total of 4 problems). Note that $*$ problems are worth 2 normal problems, i.e. solving one $*$ problem means submitting a total of 3 problems.

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 $f(n) = \|w^*\|_2^2 \left(1 - \frac{n}{100}\right)$.

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

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

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

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

(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.

(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.

(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.

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 $z \in \mathbb{R}$.

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.

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 $[-1, 1]$.

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 A.

Note: When using the GPU version of EigenPro, the parameter γ is computed based on a bandwidth. In particular, one should set the bandwidth B and the parameter $\gamma = \frac{1}{2B^2}$.

Problem 6 (C, Required). 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.

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%.

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.

A EigenPro PyTorch Usage

Creating a Model and Fitting Data

We now describe how to use EigenPro in PyTorch. The GPU implementation of EigenPro in PyTorch is very similar to that of the corresponding scikit-learn implementation. In particular, as shown in the `run_mnist.py`

file, the line:

```
model = eigenpro.FKR_EigenPro(kernel_fn, x_train, n_class, device=device)
```

instantiates an EigenPro regressor model with the following parameters:

1. `kernel_fn`: the kernel function to use.
2. `x_train`: the training data, which will be used for preconditioner computation and should have shape number of samples by number of dimensions.
3. `n_class`: the number of dimensions of the labels y (e.g. for $y^{(i)} \in \mathbb{R}^c$, `n_class` $= c$).
4. `device`: whether the model is on the CPU or GPU.

Next, the line:

```
_ = model.fit(x_train, y_train, x_test, y_test, epochs=[1, 2, 5], mem_gb=12)
```

begins fitting the model to the dataset `x_train`, `y_train`. We describe the arguments to this function below:

1. `x_train`: the training samples, which should have shape number of samples by number of features.
2. `y_train`: the training labels, which should have shape number of samples by number of targets (e.g. number of classes).
3. `x_test`: the test samples, which should have shape number of samples by number of features.
4. `y_test`: the test labels, which should have shape number of samples by number of targets (e.g. number of classes).
5. `epochs`: a list describing the number of epochs and which epochs to print evaluation metrics (e.g. in this case, 5 total epochs and metrics printed on epochs 1, 2, and 5).
6. `mem_gb`: the number of GB of GPU memory available for EigenPro.

The output of this function will be a dictionary containing evaluation metrics (e.g. training error, test error, etc.) for each epoch.

Creating Custom Evaluation Metrics

We now describe how to create custom evaluation metrics (e.g. R^2) in EigenPro. In the file `eigenpro.py`, we see that in the `evaluate` function (lines 145 - 162), multiclass accuracy and mean squared error are already implemented as metrics. For example, the following line implements mean squared error between predictions `p_eval` and labels `y_eval`:

```
eval_metrics['mse'] = np.mean(np.square(p_eval - y_eval))
```

To include other metrics, we need only compute the metric on `p_eval` and `y_eval` and then add a corresponding entry to the `eval_metrics` dictionary.

Adjusting the Pre-conditioner Computation

EigenPro provides a speed-up in solving kernel regression by using pre-conditioned gradient descent. In particular, EigenPro computes an approximate pre-conditioner for a set of data based on a number of subsamples from the total training set.

By default, if the training set contains fewer than 100000 samples, then the number of subsamples will be at most 2000. On the other hand, if the training set contains greater than 100000 samples, by default, the number of subsamples will be 12000. This is reflected in the following lines (171-175) in `eigenpro.py`:

```
        if n_subsamples is None:
            if n_samples < 100000:
                n_subsamples = min(n_samples, 2000)
            else:
```

```
                n_subsamples = 12000

```

Increasing the number of subsamples will lead to faster convergence in terms of number of epochs, but it will take longer to compute the preconditioner matrix. In general, the default values serve as a good reference point for this tradeoff.