Machine Learning

Homework Collection

Disclaimer: The solutions are written by *Yao Xiao* and *not* guaranteed to be correct. If this cause any infringement issues, please contact me and I will delete relevant contents as soon as possible.

Homework 1: Error Decomposition & Polynomial Regression

Due: Wednesday, February 1, 2023 at 11:59pm

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g., LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. The last application is optional.

General considerations (10 Points)

For the first part of this assignment we will consider a synthetic prediction problem to develop our intuition about the error decomposition. Consider the random variables $x \in \mathcal{X} = [0, 1]$ distributed uniformly ($x \sim \text{Unif}([0, 1])$) and $y \in \mathcal{Y} = \mathbb{R}$ defined as a polynomial of degree 2 of x: there exists $(a_0, a_1, a_2) \in \mathbb{R}^3$ such that the values of x and y are linked as $y = g(x) = a_0 + a_1 x + a_2 x^2$. Note that this relation fixes the joint distribution $P_{\mathcal{X}\times\mathcal{Y}}$.

From the knowledge of a sample $\{x_i, y_i\}_{i=1}^N$, we would like to predict the relation between x and y, that is find a function f to make predictions $\hat{y} = f(x)$. We note $\mathcal{H}_d$, the set of polynomial functions on $\mathbb{R}$ of degree d: $\mathcal{H}_d = \left\{f : x \mapsto b_0 + b_1 x + \cdots b_d x^d;\ b_k \in \mathbb{R},\ \forall k \in \{0, \cdots, d\}\right\}$. We will consider the hypothesis classes $\mathcal{H}_d$ varying d. We will minimize the squared loss $\ell(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ to solve the regression problem.

1. (2 Points) Recall the definition of the expected risk $R(f)$ of a predictor f. While this cannot be computed in general note that here we defined $P_{\mathcal{X}\times\mathcal{Y}}$, which function f^* is an obvious Bayes predictor? Make sure to explain why the risk $R(f^*)$ is minimum at f^*.

   Solution. The function $f^* = g$ is an obvious Bayes predictor that achieves the minimal risk 0 among all possible functions. To be more rigorous, we can compute that

$$R(g) = \mathbb{E}_{(x,y)\sim P_{\mathcal{X}\times\mathcal{Y}}}[\ell(g(x), y)] = \mathbb{E}[\ell(g(x), g(x))] = 0. \tag{1}$$

   Moreover, since the squared loss is non-negative, there cannot be a lower risk than $R(g)$.

2. (2 Points) Using $\mathcal{H}_2$ as your hypothesis class, which function $f^*_{\mathcal{H}_2}$ is a risk minimizer in $\mathcal{H}_2$? Recall the definition of the approximation error. What is the approximation error achieved by $f^*_{\mathcal{H}_2}$?

   Solution. The function $f^*_{\mathcal{H}_2} = g$ is a risk minimizer in $\mathcal{H}_2$, since g is a Bayes predictor and is indeed a member of $\mathcal{H}_2$. The approximation error achieved by $f^*_{\mathcal{H}_2} = g$ since $f^* = f^*_{\mathcal{H}_2} = g$, i.e., the risk optimizer in this case is Bayes optimal.

3. (2 Points) Considering now $\mathcal{H}_d$, with $d > 2$. Justify an inequality between $R(f^*_{\mathcal{H}_2})$ and $R(f^*_{\mathcal{H}_d})$. Which function $f^*_{\mathcal{H}_d}$ is a risk minimizer in $\mathcal{H}_d$? What is the approximation error achieved by $f^*_{\mathcal{H}_d}$?

   Solution. The following inequality always holds:

$$R(f^*_{\mathcal{H}_2}) \geq R(f^*_{\mathcal{H}_d}), \qquad \forall d > 2. \tag{2}$$

To see this, note that $H_2 \subseteq H_d$ for all $d > 2$. Therefore, a risk minimizer in $\mathcal{H}_2$ is also in $\mathcal{H}_d$, thus at least the same risk can be attained. A lower risk is also possible since a risk minimizer in $\mathcal{H}_d$ is not necessarily in $\mathcal{H}_2$. Moreover, in this case, the function $f^*_{\mathcal{H}_d} = g$ is a risk minimizer in $\mathcal{H}_d$ for any $d > 2$, since g is a Bayes predictor and $g \in \mathcal{H}_2 \subseteq \mathcal{H}_d$. The approximation error achieved by $f^*_{\mathcal{H}_d} = g$ since $f^* = f^*_{\mathcal{H}_d} = g$, i.e., the risk optimizer in this case is Bayes optimal.

4. (4 Points) For this question we assume $a_0 = 0$. Considering $\mathcal{H} = \{f : x \mapsto b_1 x;\ b_1 \in \mathbb{R}\}$, which function $f^*_{\mathcal{H}}$ is a risk minimizer in $\mathcal{H}$? What is the approximation error achieved by $f^*_{\mathcal{H}}$? In particular what is the approximation error achieved if furthermore $a_2 = 0$ in the definition of true underlying relation $g(x)$ above?

Solution. We can compute the risk of $f \in \mathcal{H}$ as

$$\begin{aligned}
R(f) &= \mathbb{E}_{(x,y)\sim P_{\mathcal{X}\times\mathcal{Y}}}[\ell(f(x), y)] = \mathbb{E}_{x\sim \mathrm{Unif}([0,1])}[\ell(b_1 x, a_1 x + a_2 x^2)] \\
&= \int_0^1 \frac{1}{2}\left((a_1 - b_1)x + a_2 x^2\right)^2 \cdot 1 dx = \int_0^1 \frac{a_2^2 x^4 + (a_1-b_1)^2 x^2 + 2a_2(a_1-b_1)x^3}{2} dx \\
&= \frac{a_2^2 x^5}{10}\bigg|_0^1 + \frac{(a_1-b_1)^2 x^3}{6}\bigg|_0^1 + \frac{a_2(a_1-b_1)x^4}{4}\bigg|_0^1 = \frac{a_2^2}{10} + \frac{(a_1-b_1)^2}{6} + \frac{a_2(a_1-b_1)}{4} \\
&= \frac{10b_1^2 - (15a_2 + 20a_1)b_1 + (6a_2^2 + 10a_1^2 + 15a_1a_2)}{60} \\
&= \frac{1}{6}\left(\left(b_1 - \frac{3a_2+4a_1}{4}\right)^2 + \frac{6a_2^2 + 10a_1^2 + 15a_1a_2}{10} - \frac{9a_2^2 + 16a_1^2 + 24a_1a_2}{16}\right) \\
&= \frac{1}{6}\left(\left(b_1 - \frac{3a_2+4a_1}{4}\right)^2 + \frac{3a_2^2}{80}\right) = \frac{1}{6}\left(b_1 - \frac{3a_2+4a_1}{4}\right)^2 + \frac{a_2^2}{160} \geq \frac{a_2^2}{160},
\end{aligned} \tag{3}$$

and the equality holds if and only if $b_1 = \frac{3a_2+4a_1}{4}$. Therefore, the risk minimizer in $\mathcal{H}$ is

$$f^*_{\mathcal{H}} : x \mapsto \frac{3a_2 + 4a_1}{4} \cdot x, \tag{4}$$

and the corresponding approximation error can be computed as

$$\textsc{Approximation Error} = R(f^*_{\mathcal{H}}) - R(f^*) = \frac{a_2^2}{160} - 0 = \frac{a_2^2}{160}. \tag{5}$$

If furthermore we assume that $a_2 = 0$, the approximation error will be 0.

Polynomial regression as linear least squares (5 Points)

In practice, $P_{\mathcal{X}\times\mathcal{Y}}$ is usually unknown and we use the empirical risk minimizer (ERM). We will reformulate the problem as a d-dimensional linear regression problem. First note that functions in $\mathcal{H}_d$ are parametrized by a vector $\boldsymbol{b} = [b_0, b_1, \cdots b_d]^\top$, we will use the notation $f_{\boldsymbol{b}}$. Similarly we will note $\boldsymbol{a} \in \mathbb{R}^3$ the vector parametrizing $g(x) = f_{\boldsymbol{a}}(x)$. We will also gather data points from the training sample in the following matrix and vector:

$$X = \begin{bmatrix} 1 & x_1 & \cdots & x_1^d \\ 1 & x_2 & \cdots & x_2^d \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_N & \cdots & x_N^d \end{bmatrix}, \quad \boldsymbol{y} = [y_1, y_2, \cdots, y_N]^\top. \tag{6}$$

These notations allow us to take advantage of the very effective linear algebra formalism. X is called the design matrix.

5. (2 Points) Show that the empirical risk minimizer (ERM) $\hat{\boldsymbol{b}}$ is given by the following minimization $\hat{\boldsymbol{b}} = \arg\min_{\boldsymbol{b}} \|X\boldsymbol{b} - \boldsymbol{y}\|_2^2$.

   Proof. We can first compute the empirical risk as

$$\begin{aligned}\hat{R}_N(f) &= \frac{1}{N}\sum_{i=1}^{N}\ell(f(x_i), y_i) = \frac{1}{2N}\sum_{i=1}^{N}(f(x_i) - y_i)^2 = \frac{1}{2N}\sum_{i=1}^{N}\left(\sum_{j=0}^{d} b_j x_i^j - y_i\right)^2 \\ &= \frac{1}{2N}\sum_{i=1}^{N}\left([1, x_i, \cdots, x_i^d]\cdot[b_0, b_1, \cdots, b_d]^\top - y_i\right)^2 = \frac{1}{2N}\|X\boldsymbol{b} - \boldsymbol{y}\|_2^2 . \end{aligned} \tag{7}$$

   Therefore, the empirical risk minimizer $\boldsymbol{b}$ is given by

$$\hat{\boldsymbol{b}} = \arg\min_{\boldsymbol{b}} \hat{R}_N(f) = \arg\min_{\boldsymbol{b}} \ \|X\boldsymbol{b} - \boldsymbol{y}\|_2^2 , \tag{8}$$

   and the proof is complete. □

6. (3 Points) If $N > d$ and X is full rank, show that $\hat{\boldsymbol{b}} = (X^\top X)^{-1}X^\top \boldsymbol{y}$. (Hint: you should take the gradients of the loss above with respect to $\boldsymbol{b}$[1]). Why do we need to use the conditions $N > d$ and X full rank?

   Proof. We can first rewrite the form of $\hat{\boldsymbol{b}}$ as

$$\begin{aligned}\hat{\boldsymbol{b}} &= \arg\min_{\boldsymbol{b}} \ \|X\boldsymbol{b} - \boldsymbol{y}\|_2^2 = \arg\min_{\boldsymbol{b}} \ \left((X\boldsymbol{b} - \boldsymbol{y})^\top (X\boldsymbol{b} - \boldsymbol{y})\right) \\ &= \arg\min_{\boldsymbol{b}} \ \left(\boldsymbol{b}^\top X^\top X\boldsymbol{b} - \boldsymbol{y}^\top X\boldsymbol{b} - b^\top X^\top \mathbf{y} + \mathbf{y}^\top \mathbf{y}\right) \\ &= \arg\min_{\boldsymbol{b}} \ \left(\boldsymbol{b}^\top X^\top X\boldsymbol{b} - 2\boldsymbol{y}^\top X\boldsymbol{b} + \boldsymbol{y}^\top \boldsymbol{y}\right) := \arg\min_{\boldsymbol{b}} f_{\boldsymbol{b}}. \end{aligned} \tag{9}$$

   To obtain the minimum, we require the gradient to be zero, that is,

$$\begin{aligned}\nabla_{\boldsymbol{b}} f_{\boldsymbol{b}} = 0 &\implies \nabla_{\boldsymbol{b}}\left(\boldsymbol{b}^\top X^\top X\boldsymbol{b} - 2\boldsymbol{y}^\top X\boldsymbol{b} + \boldsymbol{y}^\top \boldsymbol{y}\right) = 0 \\ &\implies \left(X^\top X + (X^\top X)^\top\right)\boldsymbol{b} - 2\left(\boldsymbol{y}^\top X\right)^\top = 0 \\ &\implies X^\top X\boldsymbol{b} = X^\top \boldsymbol{y} \implies \hat{\boldsymbol{b}} = (X^\top X)^{-1}X^\top \boldsymbol{y}. \end{aligned} \tag{10}$$

   Note that the conditions $N > d$ and X full rank are necessary. If any of them is violated, then there will be infinitely many solutions to the linear equation $X\boldsymbol{b} = \boldsymbol{y}$ which lead to zero empirical risk. This is clearly not the case given that we are to find an optimal approximation and if so, the conclusion that $\hat{\boldsymbol{b}} = (X^\top X)^{-1}X^\top \boldsymbol{y}$ will no longer hold since there can be infinitely many $\hat{\boldsymbol{b}}$. □

Hands on (7 Points)

Open the source code file *hw1_code_source.py* from the *.zip* folder. Using the function `get_a` get a value for $\boldsymbol{a}$, and draw a sample `x_train, y_train` of size $N = 10$ and a sample `x_test, y_test` of size $N_{\text{test}} = 1000$ using the function `draw_sample`.

[1]You can check the linear algebra review here if needed here.

7. (2 Points) Write a function called `least_square_estimator` taking as input a design matrix $X \in \mathbb{R}^{N\times(d+1)}$ and the corresponding vector $\boldsymbol{y} \in \mathbb{R}^N$ returning $\hat{\boldsymbol{b}} \in \mathbb{R}^{(d+1)}$. Your function should handle any value of N and d, and in particular return an error if $N \leq d$. (Drawing x at random from the uniform distribution makes it almost certain that any design matrix X with $d \geq 1$ we generate is full rank).

   Solution. The function `least_square_estimator` is implemented as follows:

```python
def least_square_estimator(X, y):
    """Computes an empirical risk minimizer with least square loss function.

    Parameters
    ----------
    X : np.ndarray
        The design matrix X of size N times (d+1).
    y : np.array
        The corresponding vector y of size N.

    Returns
    -------
    ERM : np.array
        The empirical risk minimizer b of size (d+1).

    Raises
    ------
    Exception
        If N <= d, then ERM does not exist.
    """
    if X.shape[0] <= X.shape[1] - 1:
        raise Exception("ERM does not exist: N <= d")
    return np.dot(np.dot(np.linalg.inv(np.dot(X.T, X)), X.T), y)
```

8. (1 Points) Recall the definition of the empirical risk $\hat{R}(\hat{f})$ on a sample $\{x_i, y_i\}_{i=1}^N$ for a prediction function $\hat{f}$. Write a function `empirical_risk` to compute the empirical risk of $f_{\boldsymbol{b}}$ taking as input a design matrix $X \in \mathbb{R}^{N\times(d+1)}$, a vector $\boldsymbol{y} \in \mathbb{R}^N$ and the vector $\boldsymbol{b} \in \mathbb{R}^{d+1}$ parametrizing the predictor.

   Solution. The function `empirical_risk` is implemented as follows:

```python
def empirical_risk(X, y, b):
    """Computes the empirical risk of a predictor with respect to squared
    loss.

    Parameters
    ----------
    X : np.ndarray
        The design matrix X of size N times (d+1).
    y : np.array
        The corresponding vector y of size N.
    b : np.array
        The predictor b of size (d+1).

    Returns
    -------
    emp_risk : float
        The empirical risk of the predictor with respect to squared loss.
    """
    return np.sum(np.square(np.dot(X, b) - y)) / y.size / 2
```

9. (3 Points) Use your code to estimate $\hat{\boldsymbol{b}}$ from `x_train, y_train` using $d = 5$. Compare $\hat{\boldsymbol{b}}$ and $\boldsymbol{a}$. Make a single plot (Plot 1) of the plane (x, y) displaying the points in the training set, values of the true underlying function $g(x)$ in $[0, 1]$ and values of the estimated function $f_{\hat{\boldsymbol{b}}}(x)$ in $[0, 1]$. Make sure to include a legend to your plot.

   Solution. Plot 1 is shown as in Figure 1. Note that the true underlying function and the estimated function are indistinguishable in the plot, but they are indeed different.

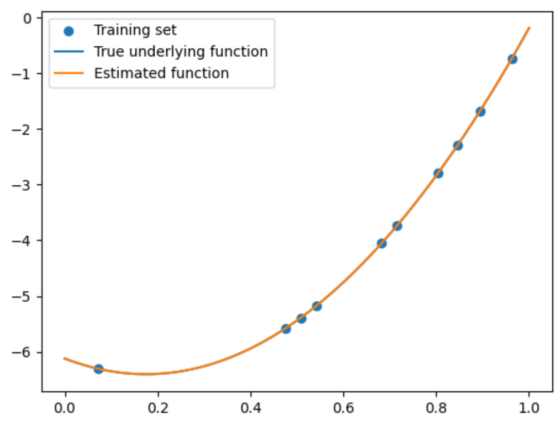


Figure 1: The training set, the true underlying function, and the estimated function.

10. (1 Points) Now you can adjust d. What is the minimum value for which we get a "perfect fit"? How does this result relates with your conclusions on the approximation error above?

   Solution. The minimum value of for which we get a "perfect fit" is $d = 2$, as can be seen from the output below:

```
d=0, R(f)=9.512686082762002
d=1, R(f)=0.11938848709867445
d=2, R(f)=2.443920820733036e-27
d=3, R(f)=1.636629710420283e-25
d=4, R(f)=3.131124948699181e-22
d=5, R(f)=4.446301326088822e-19
```

   Indeed with $d \geq 2$, the approximation error achieved by the risk minimizer in $\mathcal{H}_d$ should be 0, and here the empirical risk (which includes the estimation error and the approximation error) is almost 0 for $d \geq 2$.

In presence of noise (13 Points)

Now we will modify the true underlying $P_{\mathcal{X}\times\mathcal{Y}}$, adding some noise in $y = g(x)+\epsilon$, with $\epsilon \sim \mathcal{N}(0, 1)$ a standard normal random variable independent from x. We will call training error e_t the empirical risk on the train set and generalization error e_g the empirical risk on the test set.

11. (6 Points) Plot e_t and e_g as a function of N for $d < N < 1000$ for $d = 2$, $d = 5$ and $d = 10$ (Plot 2). You may want to use a logarithmic scale in the plot. Include also plots similar to Plot 1 for 2 or 3 different values of N for each value of d.

Solution. Plot 2 is shown as in Figure 2. Note that both axes are of logarithmic scale, i.e., the values in the plots are the natural logarithms of the actual values.

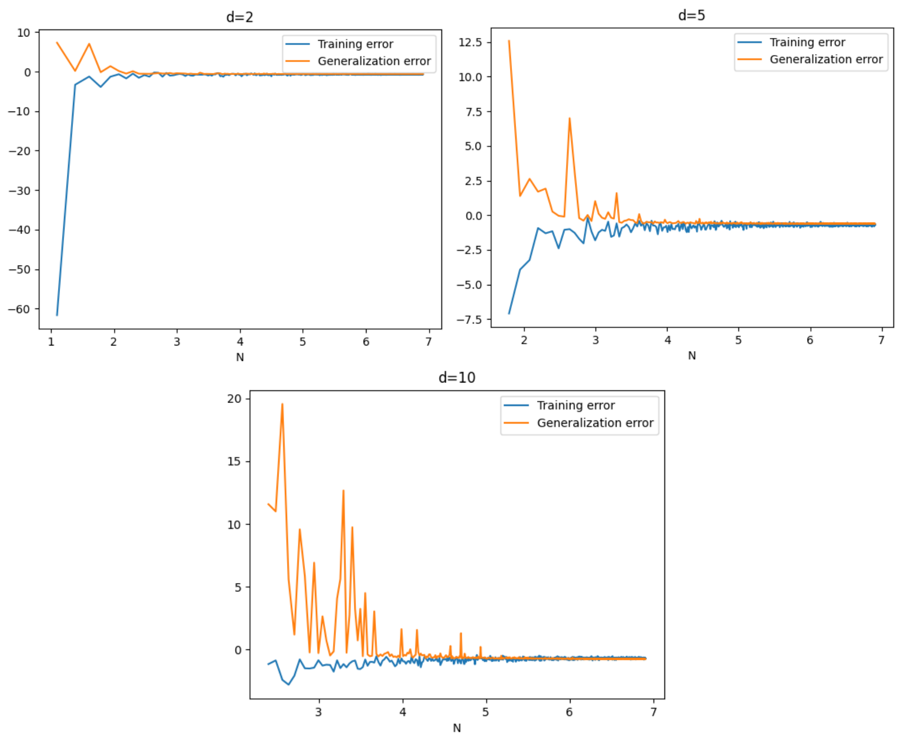


Figure 2: The training error and the generalization error with respect to N for different d.

Moreover, pick sample size $N = 50, 200, 500$, we can respectively plot the true underlying function and the estimated function as is shown in Figure 3.

12. (4 Points) Recall the definition of the estimation error. Using the test set (which we intentionally chose large so as to take advantage of the law of large numbers), give an empirical estimator of the estimation error. For the same values of N and d above plot the estimation error as a function of N (Plot 3).

Solution. Recall the definition of the estimation error that

$$\textsc{Estimation Error} = R(\hat{f}_n) - R(f_{\mathcal{F}}), \tag{11}$$

where $\hat{f}_n$ is the empirical risk minimizer in the hypothesis space and $f_{\mathcal{F}}$ is the risk minimizer in the hypothesis space. By the law of large numbers, the empirical risk estimator $\hat{R}_n(\hat{f}_n)$ approximates the real risk $R(\hat{f}_n)$. Moreover, $\hat{R}_n(\hat{f}_n)$ is represented by the generalization error e_g, and now it suffices to compute $R(f_{\mathcal{F}})$. For any function f, denote $\phi = f - g$ for

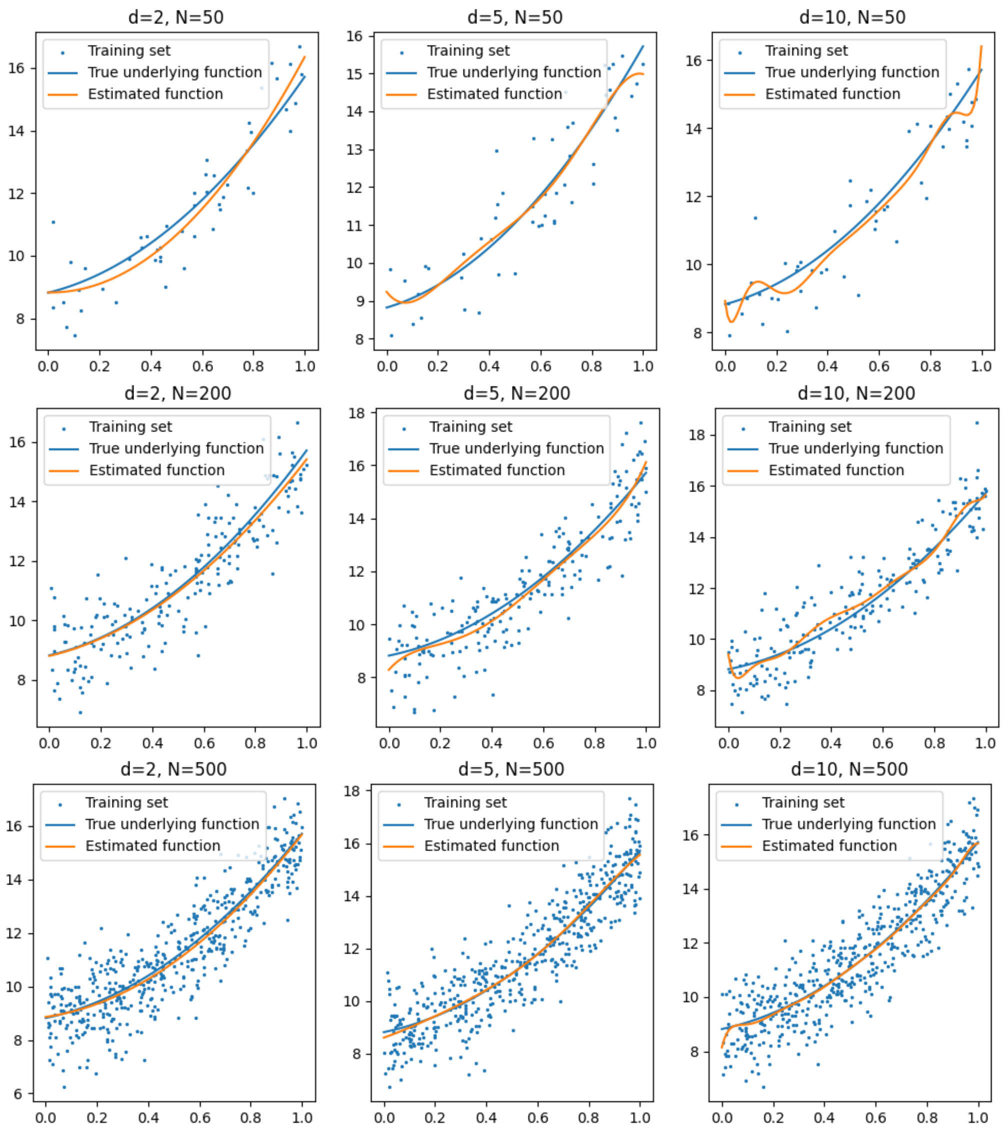


Figure 3: Training sets, true underlying functions, and estimated functions for different combinations of d and N.

simplicity and we have that

$$\begin{aligned}
R(f) &= \mathbb{E}_{(x,y)\sim P'_{\mathcal{X}\times\mathcal{Y}}}[\ell(f(x),y)] = \mathbb{E}_{x\sim\text{Unif}([0,1])}[\ell(f(x),g(x)+\epsilon)] \\
&= \frac{1}{2}\mathbb{E}_{x\sim\text{Unif}([0,1])}[(f(x)-g(x)-\epsilon)^2] = \frac{1}{2}\mathbb{E}_{x\sim\text{Unif}([0,1])}[\phi^2(x)-2\epsilon\phi(x)+\epsilon^2] \\
&= \frac{1}{2}\mathbb{E}_{x\sim\text{Unif}([0,1])}[\phi^2(x)] - \mathbb{E}_{x\sim\text{Unif}([0,1])}[\epsilon\phi(x)] + \frac{1}{2}\mathbb{E}[\epsilon^2] \\
&= \frac{1}{2}\mathbb{E}_{x\sim\text{Unif}([0,1])}[\phi^2(x)] - \mathbb{E}[\epsilon]\cdot\mathbb{E}_{x\sim\text{Unif}([0,1])}[\phi(x)] + \frac{1}{2}\mathbb{E}[\epsilon^2] \qquad (12) \\
&= \frac{1}{2}\mathbb{E}_{x\sim\text{Unif}([0,1])}[\phi^2(x)] + \frac{1}{2} \geq \frac{1}{2}, \qquad (13)
\end{aligned}$$

where the equality can be obtained when $\phi \equiv 0$, i.e., $f \equiv g$. Note that (12) follows from the independence of ϵ from x, and the equality in (13) follows since the first and second moments of a standard normal random variable are 0 and 1, respectively. Therefore, we can see that $f_{\mathcal{F}} = g$ and $R(f_{\mathcal{F}}) = \frac{1}{2}$. As a result, an empirical estimator of the estimation error is given by

$$\text{Estimation Error} \approx e_g - \frac{1}{2}. \tag{14}$$

Using the above observation, Plot 3 is shown as in Figure 4. Note that the x-axis of the plot is of logarithmic scale, i.e., the x-values in the plots are the natural logarithms of the actual values of N.

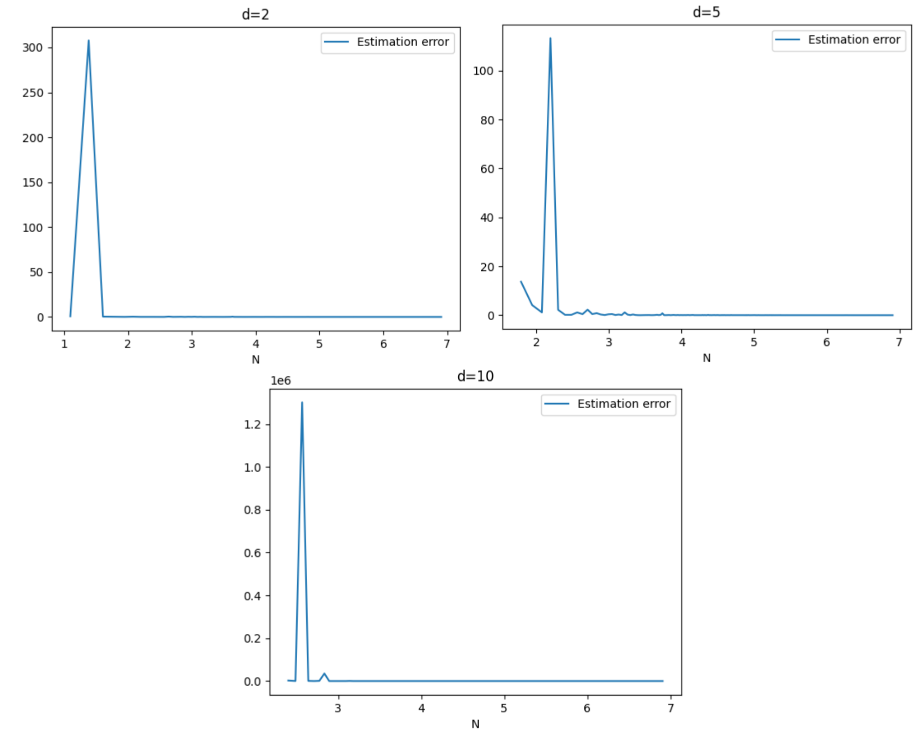


Figure 4: The estimation error with respect to N for different d.

13. (2 Points) The generalization error gives in practice an information related to the estimation error. Comment on the results of (Plot 2 and 3). What is the effect of increasing N? What is the effect of increasing d?

Solution. By analyzing the training error, the generalization error, and the estimation error from Plot 2 and 3, we can see that with N increasing, the training error gets larger, the generalization error gets smaller, and the estimation error gets smaller. On the other hand, when d increases, the training error, the generalization error, and the estimation error all get larger.

14. (1 Points) Besides the approximation and estimation there is a last source of error we have not discussed here. Can you comment on the optimization error of the algorithm we are implementing?

    Solution. The optimization error of the algorithm we are implementing is 0, since we are using the exact empirical risk minimizer rather than a good enough one, which is shown in Question 6.

Application to Ozone data (optional) (2 Points)

You can now use the code we developed on the synthetic example on a real world dataset. Using the command `np.loadtxt("ozone_wind.data")` load the data in the *.zip*. The first column corresponds to ozone measurements and the second to wind measurements. You can try polynomial fits of the ozone values as a function of the wind values.

15. (2 Points) Reporting plots, discuss again in this context the results when varying N (subsampling the training data) and d.

    Solution. The relation of the training error and the generalization error to N and d can be seen in Figure 5 and Figure 6. Note that both axes in Figure 5 are of logarithmic scale, i.e., the values in the plots are the natural logarithms of the actual values, and the y-axis in Figure 6 is of logarithmic scale, i.e., the y-values in the plots are the natural logarithms of the actual values of the errors.

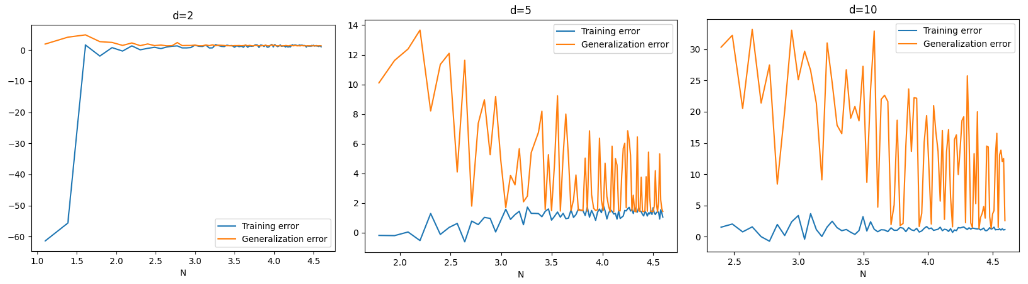


Figure 5: The training error and the generalization error with respect to N for different d.

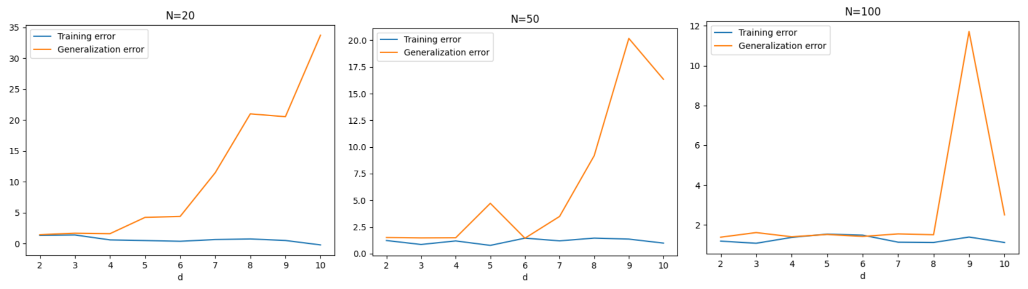


Figure 6: The training error and the generalization error with respect to d for different N.

From the plots, we can see that the training error increases with N and is unclear how it changes with d. The generalization error, on the other hand, decreases with N and increases with d. The sample is small so the conclusions may not be accurate. Moreover, the training set and the estimated function for different combinations of d and N (specifically the cases above) are as shown in Figure 7.

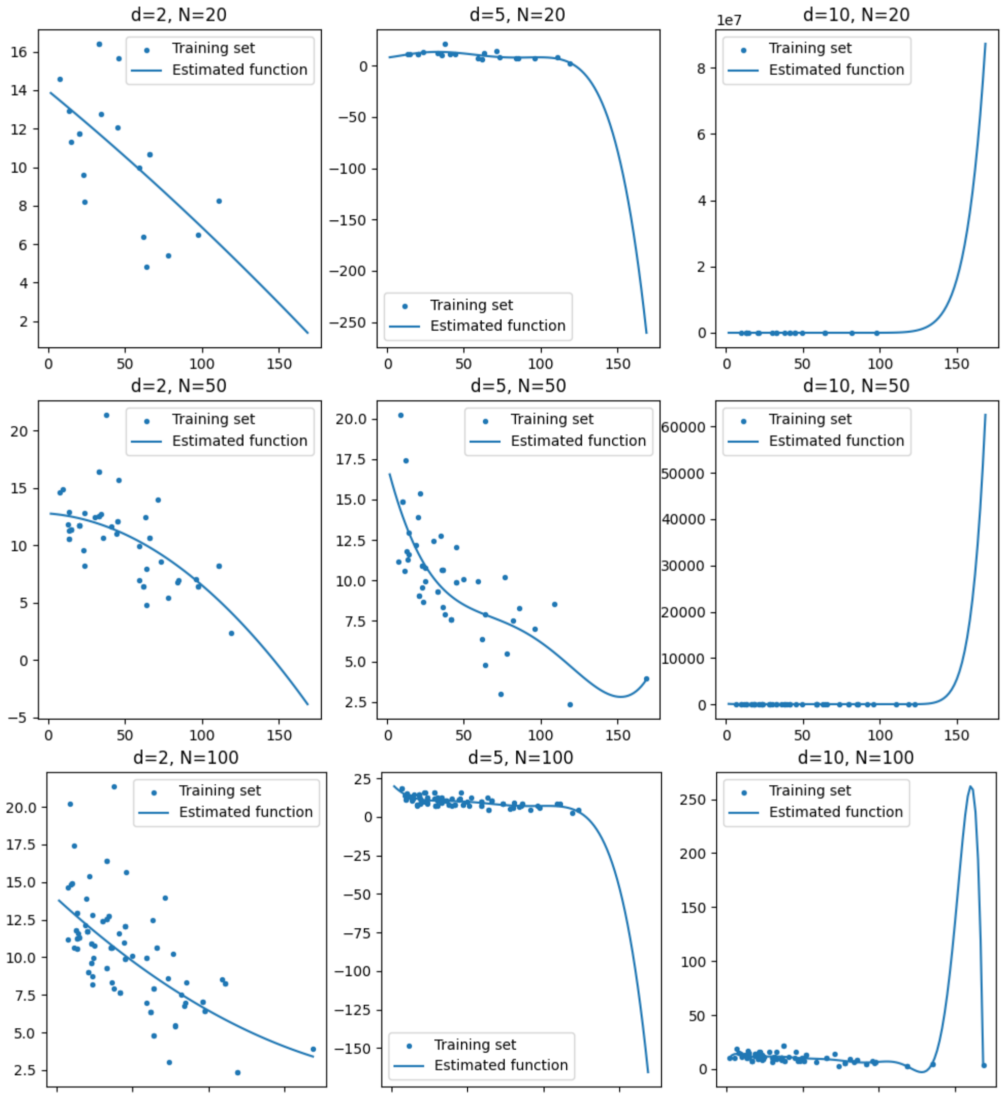


Figure 7: Training sets and estimated functions for different combinations of d and N.

Homework 2: Gradient Descent & Regularization

Due: Wednesday, February 15, 2023 at 11:59pm

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

This second homework features 3 problems and explores statistical learning theory (Week 1 & Week 2), gradient descent algorithms, loss functions (both topics of Week 2), and regularization (topic of Week 3). Following the instructions in the homework you should be able to solve a lot of questions before the lecture of Week 3. Additionally, some parts of this homework are optional. Optional questions will be graded but the points do not count towards this assignment. They instead contribute towards the extra credit you can earn over the entire course (maximum 2%) which can be used to improve the final grade by half a letter (e.g., A- to A).

Statistical Learning Theory

In the last homework, we used training error to determine whether our models have converged. It is crucial to understand what is the source of this training error. We specifically want to understand how it is connected to the noise in the data. In this question, we will compute the expected training error when we use least squares loss to fit a linear function.

Consider a full rank $N \times d$ data matrix X with $N > d$, where the training labels are generated as $y_i = \boldsymbol{b} \cdot \boldsymbol{x}_i + \epsilon_i$, and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ is the noise. From Homework 1, we know that the formula for the ERM is $\hat{\boldsymbol{b}} = (X^\top X)^{-1} X^\top \boldsymbol{y}$.

1. Show that

$$\text{Training Error} = \frac{1}{N} \left\| \left(X(X^\top X)^{-1} X^\top - I \right) \boldsymbol{\epsilon} \right\|_2^2$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2 I_N)$ and training error is defined as $\frac{1}{N}\|X\hat{\boldsymbol{b}} - \boldsymbol{y}\|_2^2$.

Proof. Bringing the formula for the empirical risk minimizer into the definition of the training error, we have that

$$\begin{aligned}
\text{Training Error} &= \frac{1}{N} \left\| X(X^\top X)^{-1} X^\top \boldsymbol{y} - \boldsymbol{y} \right\|_2^2 = \frac{1}{N} \left\| \left(X(X^\top X)^{-1} X^\top - I \right) \boldsymbol{y} \right\|_2^2 \\
&= \frac{1}{N} \left\| \left(X(X^\top X)^{-1} X^\top - I \right) X\boldsymbol{b} + \left(X(X^\top X)^{-1} X^\top - I \right) \boldsymbol{\epsilon} \right\|_2^2 \\
&= \frac{1}{N} \left\| \left(X(X^\top X)^{-1} (X^\top X) - X \right) \boldsymbol{b} + \left(X(X^\top X)^{-1} X^\top - I \right) \boldsymbol{\epsilon} \right\|_2^2 \\
&= \frac{1}{N} \left\| (XI - X) \boldsymbol{b} + \left(X(X^\top X)^{-1} X^\top - I \right) \boldsymbol{\epsilon} \right\|_2^2 = \frac{1}{N} \left\| \left(X(X^\top X)^{-1} X^\top - I \right) \boldsymbol{\epsilon} \right\|_2^2, \quad (1)
\end{aligned}$$

where $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2 I_N)$ since each of its component $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ and are mutually independent. This completes the proof. □

2. Show that the expectation of the training error can be expressed solely in terms of **only** N, d, and σ as:

$$\mathbb{E}\left[\frac{1}{N}\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\boldsymbol{\epsilon}\right\|_2^2\right] = \frac{(N-d)}{N}\sigma^2.$$

Hints:

- Consider $A = X(X^\top X)^{-1}X^\top$. What is $A^\top A$? Is A symmetric? What is A^2?
- For a symmetric matrix A satisfying $A^2 = A$, what are its eigenvalues?
- If X is full rank, then what is the rank of A? What is the eigenmatrix of A?

Proof. Let $A = X(X^\top X)^{-1}X^\top$, then we have that A is idempotent since

$$A^2 = X(X^\top X)^{-1}X^\top X(X^\top X)^{-1}X^\top = XI(X^\top X)^{-1}X^\top = X(X^\top X)^{-1}X^\top = A. \tag{2}$$

Moreover, A is also a symmetric matrix, since

$$\begin{aligned} A^\top &= (X(X^\top X)^{-1}X^\top)^\top = X((X^\top X)^{-1})^\top X^\top = X((X^\top X)^\top)^{-1}X^\top \\ &= X(X^\top X)^{-1}X^\top = A. \end{aligned} \tag{3}$$

Therefore, by definition of the l_2 norm and using the fact that $\operatorname{tr}(M_1M_2) = \operatorname{tr}(M_2M_1)$, we can compute that

$$\begin{aligned} &\mathbb{E}[\text{Training Error}] = \mathbb{E}\left[\frac{1}{N}\left\|\left(X(X^\top X)^{-1}X^\top - I\right)\boldsymbol{\epsilon}\right\|_2^2\right] = \frac{1}{N}\mathbb{E}\left[\|(A-I)\boldsymbol{\epsilon}\|_2^2\right] \\ &= \frac{1}{N}\mathbb{E}\left[((A-I)\boldsymbol{\epsilon})^\top(A-I)\boldsymbol{\epsilon}\right] = \frac{1}{N}\mathbb{E}\left[\boldsymbol{\epsilon}^\top(A-I)^\top(A-I)\boldsymbol{\epsilon}\right] \\ &= \frac{1}{N}\mathbb{E}\left[\boldsymbol{\epsilon}^\top(A^\top - I^\top)(A-I)\boldsymbol{\epsilon}\right] = \frac{1}{N}\mathbb{E}\left[\boldsymbol{\epsilon}^\top(A-I)(A-I)\boldsymbol{\epsilon}\right] \\ &= \frac{1}{N}\mathbb{E}\left[\boldsymbol{\epsilon}^\top(A^2 - AI - IA + I^2)\boldsymbol{\epsilon}\right] = \frac{1}{N}\mathbb{E}\left[\boldsymbol{\epsilon}^\top(I_N - A)\boldsymbol{\epsilon}\right] \\ &= \frac{1}{N}\mathbb{E}\left[\operatorname{tr}(\boldsymbol{\epsilon}^\top(I_N - A)\boldsymbol{\epsilon})\right] = \frac{1}{N}\mathbb{E}\left[\operatorname{tr}((I_N - A)\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top)\right] = \frac{1}{N}\operatorname{tr}((I_N - A)\mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top]). \end{aligned} \tag{4}$$

Note that by definition of the covariance matrix, we have that

$$\sigma^2 I_N = \operatorname{Var}[\boldsymbol{\epsilon}] = \operatorname{Cov}[\boldsymbol{\epsilon},\boldsymbol{\epsilon}] = \mathbb{E}[(\boldsymbol{\epsilon} - \mathbb{E}[\boldsymbol{\epsilon}])(\boldsymbol{\epsilon} - \mathbb{E}[\boldsymbol{\epsilon}])^\top] = \mathbb{E}[\boldsymbol{\epsilon}\boldsymbol{\epsilon}^\top]. \tag{5}$$

Moreover, using the linearity and commutativity property of traces, we can see that

$$\begin{aligned} \operatorname{tr}(I_N - A) &= \operatorname{tr}(I_N) - \operatorname{tr}(A) = N - \operatorname{tr}(X(X^\top X)^{-1}X^\top) \\ &= N - \operatorname{tr}(X^\top X(X^\top X)^{-1}) = N - \operatorname{tr}(I_d) = N - d. \end{aligned} \tag{6}$$

Therefore, (4) can be rewritten as

$$\mathbb{E}[\text{Training Error}] = \frac{1}{N}\operatorname{tr}((I_N - A)\sigma^2 I_N) = \frac{\sigma^2}{N}\operatorname{tr}(I_N - A) = \frac{\sigma^2(N-d)}{N}, \tag{7}$$

which completes the proof of the desired result. $\square$

3. From this result, give a reason as to why the training error is very low when d is close to N, i.e., when we overfit the data.

   Solution. From the previous result, we can easily compute that

$$\lim_{d \to N^-} \mathbb{E}[\text{Training Error}] = \lim_{d \to N^-} \frac{\sigma^2(N-d)}{N} = \frac{\sigma^2(N - \lim_{d\to N^-} d)}{N} = 0, \tag{8}$$

   implying that the expectation of the training error will tend to 0 as d gets closer to N.

Gradient Descent for Ridge(less) Linear Regression

We have provided you with a file called `ridge_regression_dataset.csv`. Columns `x0` through `x47` correspond to the input and column `y` corresponds to the output. We are trying to fit the data using a linear model and gradient based methods. Please also check the supporting code in `skeleton_code.py`. Throughout this problem, we refer to particular blocks of code to help you step by step.

Feature normalization

When feature values differ greatly, we can get much slower rates of convergence of gradient-based algorithms. Furthermore, when we start using regularization, features with larger values are treated as "more important", which is not usually desired.

One common approach to feature normalization is perform an affine transformation (i.e. shift and rescale) on each feature so that all feature values in the training set are in $[0, 1]$. Each feature gets its own transformation. We then apply the same transformations to each feature on the validation set or test set. Importantly, the transformation is "learned" on the training set, and then applied to the test set. It is possible that some transformed test set values will lie outside the $[0, 1]$ interval.

4. Modify function `feature_normalization` to normalize all the features to $[0, 1]$. Can you use numpy's broadcasting here? Often broadcasting can help to simplify and/or speed up your code. Note that a feature with constant value cannot be normalized in this way. Your function should discard features that are constant in the training set.

   Solution. The function `feature_normalization` can be implemented as follows:

```python
def feature_normalization(train, test):
    """Normalize train set features and update test set feature correspondingly.

    Rescale the data so that each feature in the training set is in the interval [0, 1], and apply the same transformations to the test set, using the statistics computed on the training set.

    Parameters
    ----------
    train : np.ndarray
        The training set of size num_instances * num_features.
    test : np.ndarray
        The test set of size num_instances * num_features.

    Returns
    -------
    train_normalized : np.ndarray
        The training set after normalization.
    test_normalized :
        The test set after applying the same transformation as for the
        training set.
    """
    train_min, train_max = np.min(train), np.max(train)
    train_normalized = (train - train_min) / (train_max - train_min)
    test_normalized = (test - train_min) / (train_max - train_min)
    return train_normalized, test_normalized
```

At the end of the skeleton code, the function `load_data` loads, split into a training set and a test set, and normalize the data using your `feature_normalization` function.

Linear regression

In linear regression, we consider the hypothesis space of linear functions

$$h_{\boldsymbol{\theta}} : \mathbb{R}^d \to \mathbb{R}, \qquad \boldsymbol{x} \mapsto \boldsymbol{\theta}^\top \boldsymbol{x},\ \boldsymbol{\theta} \in \mathbb{R}^d,$$

and we choose $\boldsymbol{\theta}$ that minimizes the following "average square loss" objective function:

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2,$$

where $(\boldsymbol{x}_1, y_1), \dots, (\boldsymbol{x}_m, y_m) \in \mathbb{R}^d \times \mathbb{R}$ is our training data.

While this formulation of linear regression is very convenient, it's more standard to use a hypothesis space of affine functions:

$$h_{\boldsymbol{\theta},b}(x) = \boldsymbol{\theta}^\top \boldsymbol{x} + b,$$

which allows a nonzero intercept term b, sometimes called a "bias" term. The standard way to achieve this, while still maintaining the convenience of the first representation, is to add an extra dimension to $\boldsymbol{x}$ that is always a fixed value, such as 1, and use $\boldsymbol{\theta}, \boldsymbol{x} \in \mathbb{R}^{d+1}$. Convince yourself that this is equivalent. We will assume this representation.

5. Let $X \in \mathbb{R}^{m\times(d+1)}$ be the *design matrix*, where the ith row of X is denoted by $\boldsymbol{x}_i$, and let $\boldsymbol{y} = (y_1, \cdots, y_m)^\top \in \mathbb{R}^{m\times 1}$ be the *response*. Write the objective function $J(\boldsymbol{\theta})$ as a matrix/vector expression, without using an explicit summation sign.[1]

   Solution. By definition of the objective function, we can write that

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2 = \frac{1}{m}\sum_{i=1}^{m}\left(\boldsymbol{\theta}^\top \boldsymbol{x}_i - y_i\right)^2 = \frac{1}{m}\left\|X\boldsymbol{\theta} - \boldsymbol{y}\right\|_2^2. \tag{9}$$

6. Write down an expression for the gradient of J without using an explicit summation sign.

[1]Being able to write expressions as matrix/vector expressions without summations is crucial to making implementations that are useful in practice, since you can use numpy (or more generally, an efficient numerical linear algebra library) to implement these matrix/vector operations orders of magnitude faster than naively implementing with loops in Python.

Solution. By definition of gradient of matrices, vectors, and linear forms, we can compute the gradient of J with respect to $\boldsymbol{\theta}$ as

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}}\left(\frac{1}{m}\|X\boldsymbol{\theta}-\boldsymbol{y}\|_2^2\right) = \frac{1}{m}\nabla_{\boldsymbol{\theta}}\left((X\boldsymbol{\theta}-\boldsymbol{y})^\top(X\boldsymbol{\theta}-\boldsymbol{y})\right)\\ &= \frac{1}{m}\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{\theta}^\top X^\top X\boldsymbol{\theta}-\boldsymbol{y}^\top X\boldsymbol{\theta}-\boldsymbol{\theta}^\top X^\top\boldsymbol{y}+\boldsymbol{y}^\top\boldsymbol{y}\right)\\ &= \frac{1}{m}\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{\theta}^\top X^\top X\boldsymbol{\theta}-2\boldsymbol{y}^\top X\boldsymbol{\theta}+\boldsymbol{y}^\top\boldsymbol{y}\right)\\ &= \frac{1}{m}\left(X^\top X+(X^\top X)^\top\right)\boldsymbol{\theta}-\frac{2}{m}\left(\boldsymbol{y}^\top X\right)^\top = \frac{2}{m}\left(X^\top X\boldsymbol{\theta}-X^\top\boldsymbol{y}\right). \qquad (10)\end{aligned}$$

7. Write down the expression for updating the parameter vector $\boldsymbol{\theta}$ in the gradient descent algorithm with step size η.

   Solution. We can update $\boldsymbol{\theta}$ using

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{2\eta}{m}\left(X^\top X\boldsymbol{\theta}-X^\top\boldsymbol{y}\right). \qquad (11)$$

8. Modify the function `compute_square_loss` to compute $J(\boldsymbol{\theta})$ for a given $\boldsymbol{\theta}$. You might want to create a small dataset for which you can compute $J(\boldsymbol{\theta})$ by hand, and verify that your `compute_square_loss` function returns the correct value.

   Solution. The function `compute_square_loss` can be implemented as follows:

```python
def compute_square_loss(X, y, theta):
    """Compute the average square loss for predicting y with X * theta.

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    theta : np.array
        The parameter vector of size num_features.

    Returns
    -------
    loss : float
        The average square loss.
    """
    return np.sum(np.square(np.dot(X, theta) - y)) / y.size
```

9. Modify the function `compute_square_loss_gradient`, to compute $\nabla_\theta J(\theta)$. You may again want to use a small dataset to verify that your `compute_square_loss_gradient` function returns the correct value.

   Solution. The function `compute_square_loss_gradient` can be implemented as follows:

```python
def compute_square_loss_gradient(X, y, theta):
    """Compute the gradient of the average square loss at the point theta.

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    theta : np.array
        The parameter vector of size num_features.

    Returns
    -------
    grad : np.array
        The gradient vector of size num_features.
    """
    return np.dot(X.T, np.dot(X, theta) - y) * 2 / y.size
```

Gradient checker

We can numerically check the gradient calculation. If $J : \mathbb{R}^d \to \mathbb{R}$ is differentiable, then for any vector $\boldsymbol{h} \in \mathbb{R}^d$, the directional derivative of J at $\boldsymbol{\theta}$ in the direction $\boldsymbol{h}$ is given by

$$\lim_{\epsilon \to 0} \frac{J(\boldsymbol{\theta} + \epsilon \boldsymbol{h}) - J(\boldsymbol{\theta} - \epsilon \boldsymbol{h})}{2\epsilon}.$$

It is also given by the more standard definition of directional derivative as

$$\lim_{\epsilon \to 0} \frac{1}{\epsilon} \left[J(\boldsymbol{\theta} + \epsilon \boldsymbol{h}) - J(\boldsymbol{\theta}) \right].$$

The former form gives a better approximation to the derivative when we are using small (but not infinitesimally small) ϵ. We can approximate this directional derivative by choosing a small value of $\epsilon > 0$ and evaluating the quotient above. We can get an approximation to the gradient by approximating the directional derivatives in each coordinate direction and putting them together into a vector. In other words, take $\boldsymbol{h} = (1, 0, 0, \cdots, 0)$ to get the first component of the gradient. Then take $\boldsymbol{h} = (0, 1, 0, \cdots, 0)$ to get the second component, and so on.

10. Complete the function `grad_checker` according to its documentation given in the source file `skeleton_code.py`. Alternatively, you may complete the function `generic_grad_checker` which can work for any objective function.

    Solution. The function `grad_checker` can be implemented as follows:

```
def grad_checker(X, y, theta, epsilon=0.01, tolerance=1e-4):
    """Check whether `compute_square_loss_gradient` function returns the
    correct result.

    Let d be the number of features. Here we numerically estimate the
    gradient by approximating the directional derivative in each of the d
    coordinate directions: (e_1=(1,0,0,...,0,0), e_2=(0,1,0,...,0,0), ...,
    e_d=(0,0,0,...,0,1))

    The approximation for the directional derivative of J at the point theta
     in direction e_i is given by: (J(theta + epsilon * e_i) - J(theta -
    epsilon * e_i)) / (2*epsilon).

    We then look at the Euclidean distance between the gradient computed
    using this approximation and the gradient computed by `
    compute_square_loss_gradient` function. If the Euclidean distance
    exceeds tolerance, we say the gradient is incorrect.

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    theta : np.array
        The parameter vector of size num_features.
    epsilon : float
        The epsilon used in the approximation.
    tolerance : float
        The tolerance error.

    Returns
    -------
    correct : boolean
        Whether the gradient is correct or not.
    """
    true_gradient = compute_square_loss_gradient(X, y, theta)
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features)
    for i in range(num_features):
        direction = np.zeros(num_features)
        direction[i] = epsilon
        approx_grad[i] = (compute_square_loss(X, y, theta + direction)
            - compute_square_loss(X, y, theta - direction)) / 2 / epsilon
    return np.linalg.norm(true_gradient - approx_grad) <= tolerance
```

The more general function `generic_gradient_checker` can be implemented as follows:

```
def generic_gradient_checker(X, y, theta, objective_func, gradient_func,
                             epsilon=0.01, tolerance=1e-4):
    """Check whether the proposed gradient function is correct.

    The functions takes objective_func and gradient_func as parameters, and
    check whether gradient_func(X, y, theta) returned the true gradient for
    objective_func(X, y, theta).

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    theta : np.array
        The parameter vector of size num_features.
    objective_func : function
        The objective function.
    gradient_func : function
        The proposed gradient function of the objective function.
    epsilon : float
        The epsilon used in the approximation.
    tolerance : float
        The tolerance error.

    Returns
    -------
    correct : boolean
        Whether the gradient is correct or not.
    """
    true_gradient = gradient_func(X, y, theta)
    num_features = theta.shape[0]
    approx_grad = np.zeros(num_features)
    for i in range(num_features):
        direction = np.zeros(num_features)
        direction[i] = epsilon
        approx_grad[i] = (objective_func(X, y, theta + direction)
            - objective_func(X, y, theta - direction)) / 2 / epsilon
    return np.linalg.norm(true_gradient - approx_grad) <= tolerance
```

You should be able to check that the gradients you computed above remain correct throughout the learning below.

Batch gradient descent

We will now finish the job of running regression on the training set.

11. Complete `batch_gradient_descent`. Note the phrase *batch* gradient descent distinguishes between *stochastic* gradient descent or more generally *minibatch* gradient descent.

    Solution. The `batch_gradient_descent` algorithm can be implemented as follows:

```
def batch_grad_descent(X, y, alpha=0.1, num_step=1000):
    """Batch gradient descent minimizing average square loss.

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    alpha : float
        The step size in gradient descent.
    num_step : int
        The number of steps to run.

    Returns
    -------
    theta_hist : np.ndarray
        The history of the parameter vector. It is of size (num_step + 1)
        * num_features.
    loss_hist : np.array
        The history of the average square loss on the data. It is of size
        (num_step + 1).
    """
    _, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))
    loss_hist = np.zeros(num_step + 1)
    theta = np.zeros(num_features)
    for step in range(num_step + 1):
        theta_hist[step] = theta
        loss_hist[step] = compute_square_loss(X, y, theta)
        theta -= alpha * compute_square_loss_gradient(X, y, theta)
    return theta_hist, loss_hist
```

12. Now let's experiment with the step size. Note that if the step size is too large, gradient descent may not converge. Starting with a step size of 0.1, try various different fixed step sizes to see which converges most quickly and/or which diverge. As a minimum, try step

sizes 0.5, 0.1, 0.05, and 0.01. Plot the average square loss on the training set as a function of the number of steps for each step size. Briefly summarize your findings.

Solution. The plot of average square loss on the training set versus the number of steps for step sizes 0.01 to 0.05 are shown as in Figure 1. Clearly, within a certain range, larger step sizes lead to faster convergence of loss. However, note that when the step size exceeds approximately 0.066, the average square loss on the training set explodes (diverges), and such cases are not included in the plot.

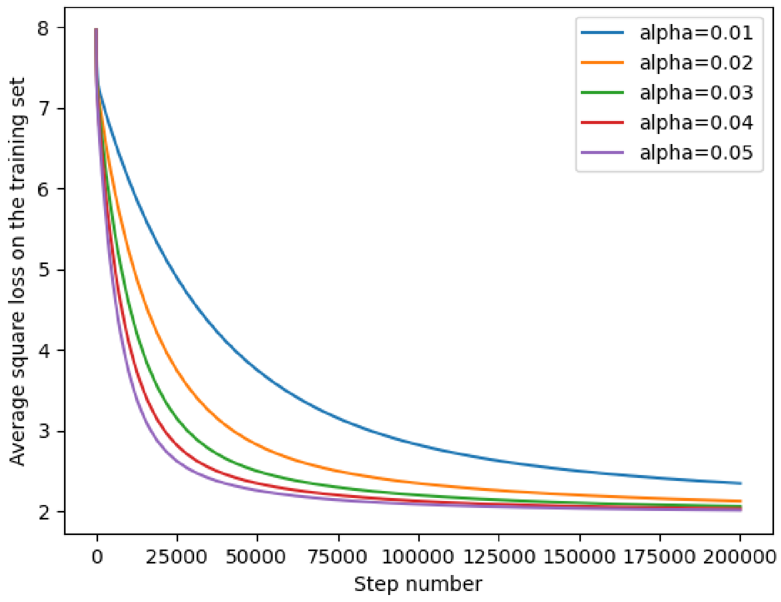


Figure 1: The change of average square loss on the train set with the increase of steps in the batch gradient descent algorithm under different step sizes.

13. For the learning rate you selected above, plot the average test loss as a function of the iterations. You should observe overfitting: the test error first decreases and then increases.

    Solution. The plot of average square loss on the test set versus the number of steps for step sizes 0.01 to 0.05 are shown as in Figure 2. Here we can indeed observe overfitting: while the loss on the train set keeps decreasing, after a number of steps the loss on the test set starts increasing.

Ridge regression

We will add l_2 regularization to linear regression. When we have a large number of features compared to instances, regularization can help control overfitting. *Ridge regression* is just linear regression with l_2 regularization. The regularization term is sometimes called a penalty term. The objective function for ridge regression is

$$J_\lambda(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2 + \lambda\boldsymbol{\theta}^\top\boldsymbol{\theta},$$

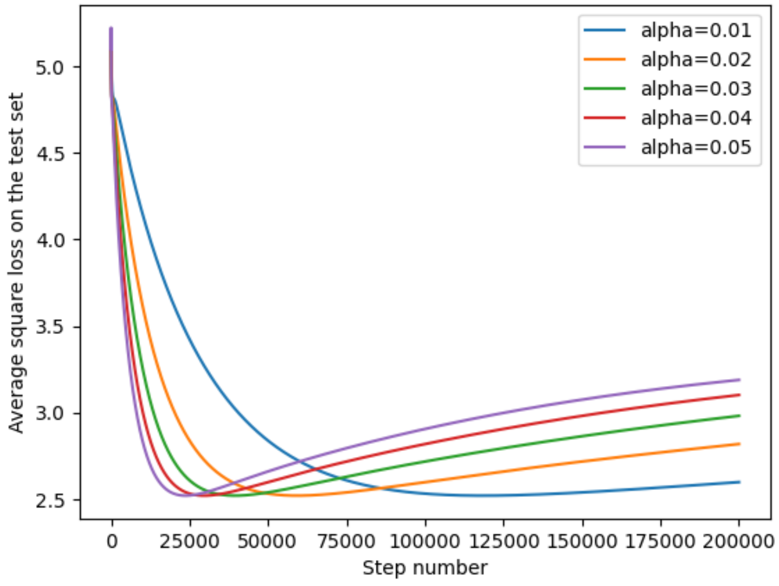


Figure 2: The change of average square loss on the test set with the increase of steps in the batch gradient descent algorithm under different step sizes.

where λ is the regularization parameter, which controls the degree of regularization. Note that the bias term (which we included as an extra dimension in θ) is being regularized as well as the other parameters. Sometimes it is preferable to treat this term separately.

14. Compute the gradient of $J_\lambda(\boldsymbol{\theta})$ and write down the expression for updating $\boldsymbol{\theta}$ in the gradient descent algorithm with matrix/vector expression (without explicit summation).

    Solution. By definition of the objective function and using (10), we can compute the gradient of J_λ with respect to $\boldsymbol{\theta}$ as

    $$\nabla_{\boldsymbol{\theta}} J_\lambda(\boldsymbol{\theta}) = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} \left(\lambda \boldsymbol{\theta}^\top \boldsymbol{\theta}\right) = \frac{2}{m}\left(X^\top X\boldsymbol{\theta} - X^\top \boldsymbol{y}\right) + 2\lambda\boldsymbol{\theta}. \tag{12}$$

    We can thus update $\boldsymbol{\theta}$ using

    $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{2\eta}{m}\left(X^\top X\boldsymbol{\theta} - X^\top \boldsymbol{y}\right) - 2\lambda\boldsymbol{\theta}. \tag{13}$$

15. Implement `compute_regularized_square_loss_gradient`.

    Solution. The function above can be implemented as follows:

```python
def compute_regularized_square_loss_gradient(X, y, theta, lambda_reg):
    """Compute the gradient of l2-regularized average square loss at the
     point theta.

    Parameters
    ----------
    X : np.ndarray
```

```
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    theta : np.array
        The parameter vector of size num_features.
    lambda_reg : float
        The regularization coefficient.

    Returns
    -------
    grad : np.array
        The gradient vector of size num_features.
    """
    return np.dot(X.T, np.dot(X, theta) - y) * 2 / y.size
            + 2 * lambda_reg * theta
```

16. Implement `regularized_grad_descent`.

Solution. The `regularized_grad_descent` algorithm can be implemented as follows:

```
def regularized_grad_descent(X, y, alpha=0.05, lambda_reg=1e-2,
                             num_step=1000):
    """Batch gradient descent minimizing l2-regularized average square loss.

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    alpha : float
        The step size in gradient descent.
    lambda_reg : float
        The regularization coefficient.
    num_step : int
        The number of steps to run.

    Returns
    -------
    theta_hist : np.ndarray
        The history of the parameter vector. It is of size (num_step + 1)
        * num_features.
    loss_hist : np.array
        The history of the average square loss on the data. It is of size
        (num_step + 1).
    """
    _, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_step + 1, num_features))
    loss_hist = np.zeros(num_step + 1)
    theta = np.zeros(num_features)
    for step in range(num_step + 1):
        theta_hist[step] = theta
        loss_hist[step] = compute_square_loss(X, y, theta)
        theta -= alpha
            * compute_regularized_square_loss_gradient(X, y, theta)
    return theta_hist, loss_hist
```

Our goal is to find λ that gives the minimum average square loss on the test set, so you should start your search very broadly, looking over several orders of magnitude. For instance, you can try $\lambda = 10^{-7}, 10^{-5}, 10^{-3}, 10^{-1}, 1, 10, 100$. Then you can zoom in on the best range. Follow the steps below to proceed.

17. Choosing a reasonable step size, plot training average square loss and the test average square loss (just the average square loss part, without the regularization, in each case) as a function of the training iterations for various values of λ. What do you notice in terms of overfitting?

    Solution. Fix the step size as 0.05. The plots of average square loss on the train set and on the test set versus the number of steps for regularization parameter λ between 10^{-6} and 10^{-4} are respectively shown as in Figure 3. Here we can see that the overfitting problem gets alleviated as the regularization parameter λ increases.

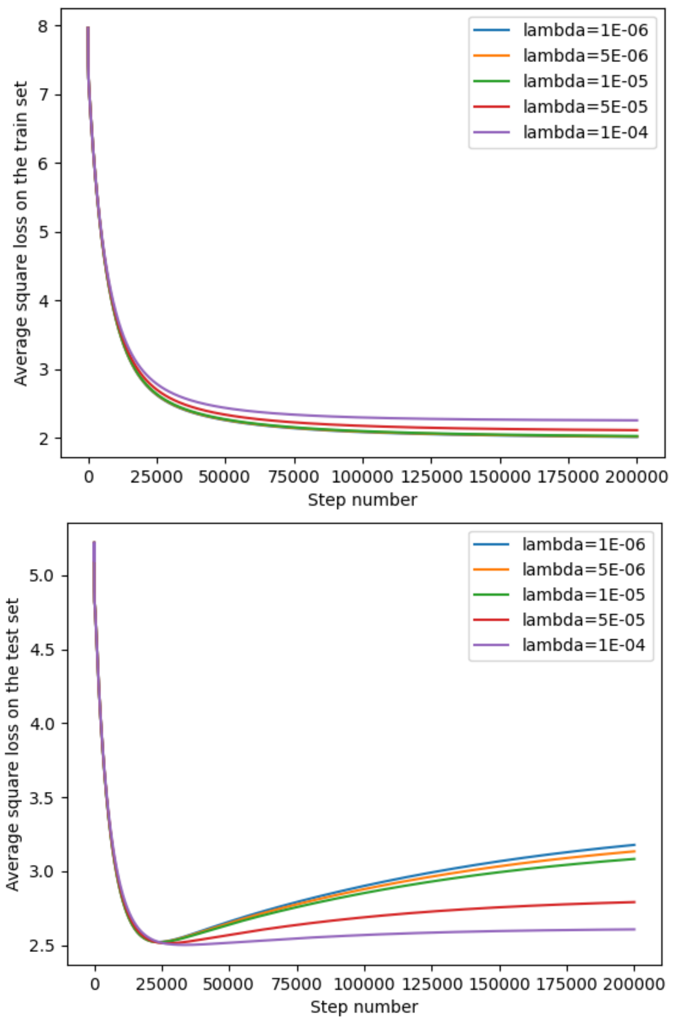


Figure 3: The change of average square loss on the train set (above) and on the test set (below) with the increase of steps in the batch gradient descent algorithm under different regularization parameters and step size 0.05.

18. Plot the training average square loss and the test average square loss at the end of training as a function of λ. You may want to have $\log \lambda$ on the x-axis rather than λ. Which value of λ would you choose?

    Solution. Fix the step size as 0.05. The plot of average square loss on the train set and on the test set after 2×10^6 steps of training versus the regularization parameter λ is shown as in Figure 4. Note that the x-axis of the plot is of logarithmic scale, i.e., the x-values in the plots are the base-10 logarithms of the actual values of λ. I will choose $\lambda \approx 10^{-3.5}$ since the average loss on the test set is minimized around this value.

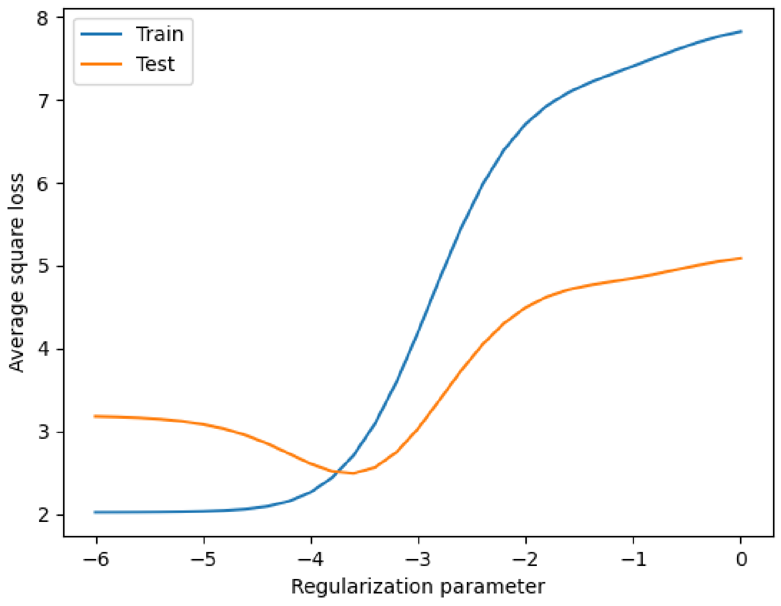


Figure 4: The change of average square loss on the train set and on the test set with the increase of regularization parameter λ and step size 0.05.

19. Another heuristic of regularization is to *early-stop* the training when the test error reaches a minimum. Add to the last plot the minimum of the test average square loss along training as a function of λ. Is the value λ you would select with early stopping the same as before?

    Solution. Fix the step size as 0.05. The plot of average square loss on the test set within 2×10^6 steps of training and with early-stop versus the regularization parameter λ is shown as in Figure 5. Note that the x-axis of the plot is of logarithmic scale, i.e., the x-values in the plots are the base-10 logarithms of the actual values of λ. Though early-stopping has a significant improvement on the average square loss on the test set for small values of λ, I will still choose $\lambda \approx 10^{-3.5}$ since the average loss on the test set is still minimized around this value.

20. What $\boldsymbol{\theta}$ would you select in practice and why?

    Solution. I will first try some random initial $\boldsymbol{\theta}$, then use their solutions as the initial $\boldsymbol{\theta}$ for further experiments as long as they do not produce obviously strange solutions.

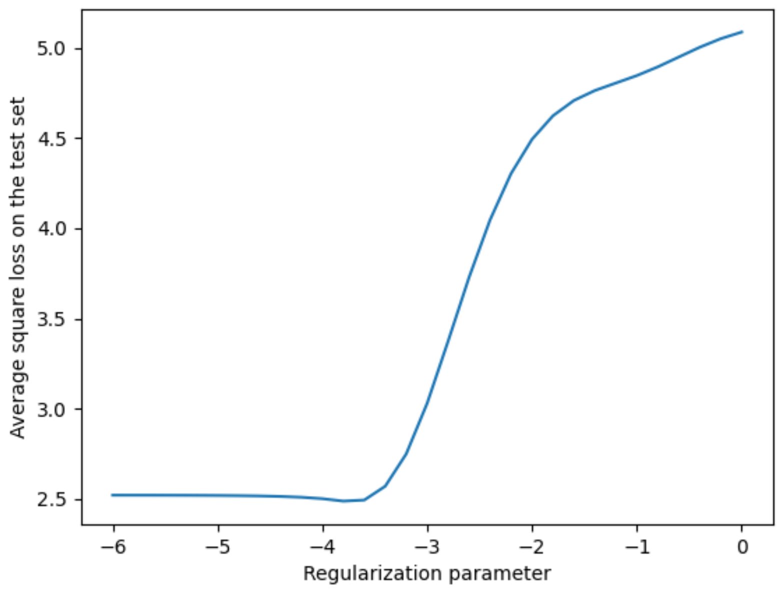


Figure 5: The change of average square loss on the train set and on the test set with the increase of regularization parameter λ and step size 0.05.

Stochastic gradient descent (SGD) (optional)

When the training data set is very large, evaluating the gradient of the objective function can take a long time, since it requires looking at each training example to take a single gradient step. In SGD, rather than taking $-\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ as our step direction to minimize

$$J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{\theta}),$$

we take $-\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta})$ for some i chosen uniformly at random from $\{1, \cdots, m\}$. The approximation is poor, but we will show it is unbiased.

In machine learning applications, each $f_i(\boldsymbol{\theta})$ would be the loss on the ith example. In practical implementations for machine learning, the data points are randomly shuffled, and then we sweep through the whole training set one by one, and perform an update for each training example individually. One pass through the data is called an *epoch*. Note that each epoch of SGD touches as much data as a single step of batch gradient descent. You can use the same ordering for each epoch, though optionally you could investigate whether reshuffling after each epoch affects the convergence speed.

21. Show that the objective function

$$J_\lambda(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} \left(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i\right)^2 + \lambda \boldsymbol{\theta}^\top \boldsymbol{\theta}$$

can be written in the form $J_\lambda(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{\theta})$ by giving an expression for $f_i(\boldsymbol{\theta})$ that makes the two expressions equivalent.

Proof. Note that since the regularization term is independent of i, we have that

$$J_\lambda(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}(h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i)^2 + \lambda\boldsymbol{\theta}^\top\boldsymbol{\theta} = \frac{1}{m}\sum_{i=1}^{m}\left((h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i)^2 + \lambda\boldsymbol{\theta}^\top\boldsymbol{\theta}\right). \tag{14}$$

Therefore, by choosing $f_i(\boldsymbol{\theta}) = (h_{\boldsymbol{\theta}}(\boldsymbol{x}_i) - y_i)^2 + \lambda\boldsymbol{\theta}^\top\boldsymbol{\theta}$ we can make the two expressions equivalent, thus completing the proof. □

22. Show that the stochastic gradient $\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta})$, for i chosen uniformly at random in the range $\{1, \cdots, m\}$, is an *unbiased estimator* of $\nabla_{\boldsymbol{\theta}} J_\lambda(\boldsymbol{\theta})$. In other words, show that $\mathbb{E}\left[\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta})\right] = \nabla_{\boldsymbol{\theta}} J_\lambda(\boldsymbol{\theta})$ for any $\boldsymbol{\theta}$. It will be easier to prove this for a general $J_\lambda(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{\theta})$, rather than the specific case of ridge regression. You can start by writing down an expression for $\mathbb{E}\left[\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta})\right]$.

Proof. Note that since i is chosen uniformly at random, the probability for each i being chosen is $1/m$. Therefore, we have that

$$\mathbb{E}[\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta})] = \sum_{i=1}^{m}\left(\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta}) \cdot \frac{1}{m}\right) = \nabla_{\boldsymbol{\theta}}\left(\frac{1}{m}\sum_{i=1}^{m} f_i(\boldsymbol{\theta})\right) = \nabla_{\boldsymbol{\theta}} J_\lambda(\boldsymbol{\theta}), \tag{15}$$

indicating that the stochastic gradient is an unbiased estimator of $\nabla_{\boldsymbol{\theta}} J_\lambda(\boldsymbol{\theta})$. □

23. Write down the update rule for $\boldsymbol{\theta}$ in SGD for the ridge regression objective function.

Solution. We can compute the stochastic gradient as

$$\begin{aligned}\nabla_{\boldsymbol{\theta}} f_i(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}}\left(\left(\boldsymbol{\theta}^\top\boldsymbol{x}_i - y_i\right)^2 + \lambda\boldsymbol{\theta}^\top\boldsymbol{\theta}\right) = \nabla_{\boldsymbol{\theta}}\left(\left(\boldsymbol{\theta}^\top\boldsymbol{x}_i\right)^2 - 2y_i\left(\boldsymbol{\theta}^\top\boldsymbol{x}_i\right) + \lambda\boldsymbol{\theta}^\top\boldsymbol{\theta}\right)\\ &= 2\boldsymbol{\theta}^\top\boldsymbol{x}_i\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{\theta}^\top\boldsymbol{x}_i\right) - 2y_i\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{\theta}^\top\boldsymbol{x}_i\right) + \lambda\nabla_{\boldsymbol{\theta}}\left(\boldsymbol{\theta}^\top\boldsymbol{\theta}\right)\\ &= \left(2\boldsymbol{\theta}^\top\boldsymbol{x}_i\right)\cdot\boldsymbol{x}_i - 2y_i\cdot\boldsymbol{x}_i + 2\lambda\cdot\boldsymbol{\theta}.\end{aligned} \tag{16}$$

Therefore, we can choose $i \in \{1, \cdots, m\}$ uniformly at random and update $\boldsymbol{\theta}$ using

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - 2\eta\left(\left(\boldsymbol{\theta}^\top\boldsymbol{x}_i\right)\cdot\boldsymbol{x}_i - y_i\boldsymbol{x}_i + \lambda\boldsymbol{\theta}\right). \tag{17}$$

24. Implement `stochastic_grad_descent`.

Solution. The `stochastic_grad_descent` algorithm can be implemented as follows:

```python
def stochastic_grad_descent(X, y, alpha=0.01, lambda_reg=1e-2,
                            num_epoch=1000):
    """Stochastic gradient descent with l2-regularization minimizing average
     square loss.

    Parameters
    ----------
    X : np.ndarray
        The feature vector of size num_instances * num_features.
    y : np.array
        The label vector of size num_instances.
    alpha : float or str
        The step size in gradient descent. Note that in SGD, it is not a
        good idea to use a fixed step size. Usually it is set to 1/sqrt(t)
        or 1/t. If alpha is a float, then the step size in every step is the
        float. If alpha == "1/sqrt(t)", then take alpha= 0.1/sqrt(t). If
        alpha == "1/t", then take alpha = 0.1/t.
    lambda_reg : float
        The regularization coefficient.
    num_epoch : int
        The number of epochs to go through the whole training set.

    Returns
    -------
    theta_hist : np.ndarray
        The history of the parameter vector. It is of size num_epoch
        * num_instances * num_features.
    loss_hist : np.array
        The history of the average square loss on the data. It is of size
        num_epoch * num_instances.

    Raises
    ------
    Exception
        If the alpha is not a float or "1/sqrt(t)" or "1/t".
    """
    if isinstance(alpha, float) or isinstance(alpha, int):
        get_alpha = lambda _: alpha
    elif alpha == "1/sqrt(t)":
        get_alpha = lambda x: 0.1 / np.sqrt(x)
    elif alpha == "1/t":
        get_alpha = lambda x: 0.1 / x
    else:
        raise Exception("Invalid step size specification, use float or \
            '1/sqrt(t)' or '1/t'")
    num_instances, num_features = X.shape[0], X.shape[1]
    theta_hist = np.zeros((num_epoch, num_instances, num_features))
    loss_hist = np.zeros((num_epoch, num_instances))
    theta = np.ones(num_features)
    step = 0
    for epoch in range(num_epoch):
        shuffled_indices = np.array(range(10))
        np.random.shuffle(shuffled_indices)
        for i in range(num_instances):
            step += 1
            theta_hist[epoch][i] = theta
            loss_hist[epoch][i] = compute_square_loss(X, y, theta)
            theta -= 2 * get_alpha(step) * (np.dot(theta.T, X[i]) * X[i]
                - y[i] * X[i] + lambda_reg * theta)
    return theta_hist, loss_hist
```

25. Use SGD to find $\boldsymbol{\theta}_\lambda^*$ that minimizes the ridge regression objective for the λ you selected in the previous problem. (If you could not solve the previous problem, choose $\lambda = 10^{-2}$). Try a few fixed step sizes (at least try $\eta_t \in \{0.05, 0.005\}$). Note that SGD may not converge with fixed step size. Simply note your results. Next try step sizes that decrease with the step number according to the following schedules: $\eta_t = C/t$ and $\eta_t = C/\sqrt{t}$, $C \leq 1$. Please include $C = 0.1$ in your submissions. You are encouraged to try different values of C (see notes below for details). For each step size rule, plot the value of the objective function (or the log of the objective function if that is more clear) as a function of epoch (or step number, if you prefer). How do the results compare?

Solution. Fix $\lambda = 10^{-3.5}$ for this part as determined in previous sections. First we experiment on large values of fixed step size. We will use $\eta = 0.05$ as a baseline for convergence, and experiment for $\eta \in [0.06, 0.064]$, as is shown in Figure 6. Note that the y-axis of the plot is of logarithmic scale, i.e., the y-values in the plots are the base-10 logarithms of the actual values of loss. Also note that when the fixed step size η exceeds this range, the objective function explodes and no longer converges. In fact, as can be seen in Figure 6, as η grows towards the upper bound 0.064, the objective function is increasingly struggling to converge. Experiments suggest that the behaviors of the average square loss on the train set and on the test set with l_2 regularized stochastic gradient descent are almost identical, so we only show the latter for clarity of the plot.

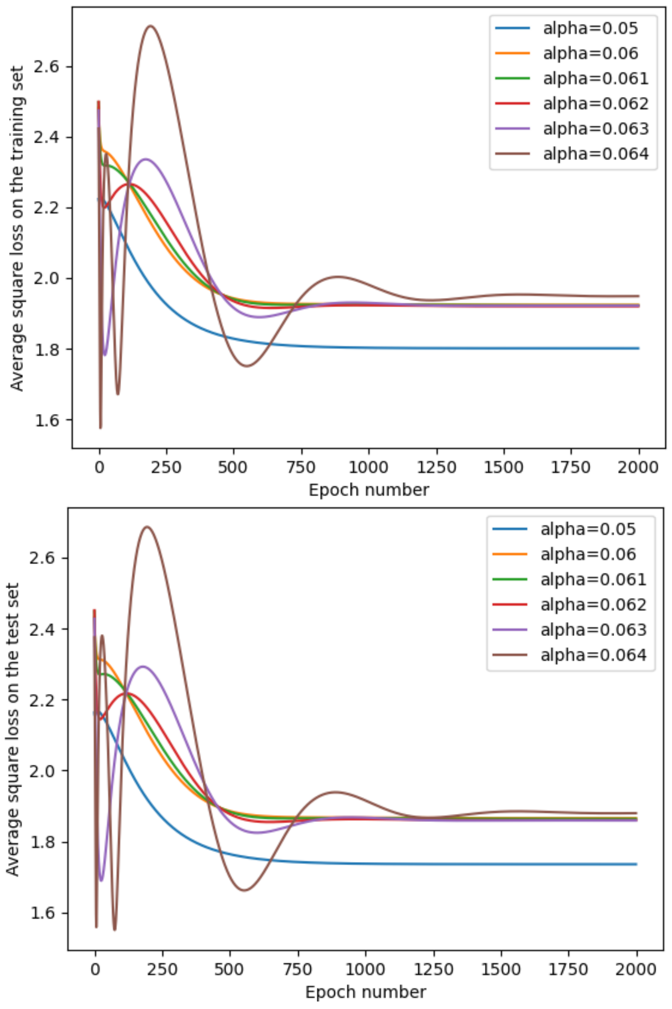


Figure 6: The change of average square loss on the train set (above) and on the test set (below) with the increase of epochs in the l_2 regularized stochastic gradient descent algorithm under different large fixed step sizes and regularization parameter $\lambda = 10^{-3.5}$.

Now we move on to try variable step sizes. We will use fixed step sizes $\eta = 0.02, 0.01, 0.005$

and variable step sizes $\eta_t = 0.1/\sqrt{t}$ and $\eta_t = 0.1/t$, where t is the step number, as is shown in Figure 7. Again, note that the y-axis of the plot is of logarithmic scale, i.e., the y-values in the plots are the base-10 logarithms of the actual values of loss. Here we can see that variable step sizes converges much faster than fixed step sizes in the beginning and slower afterwards.

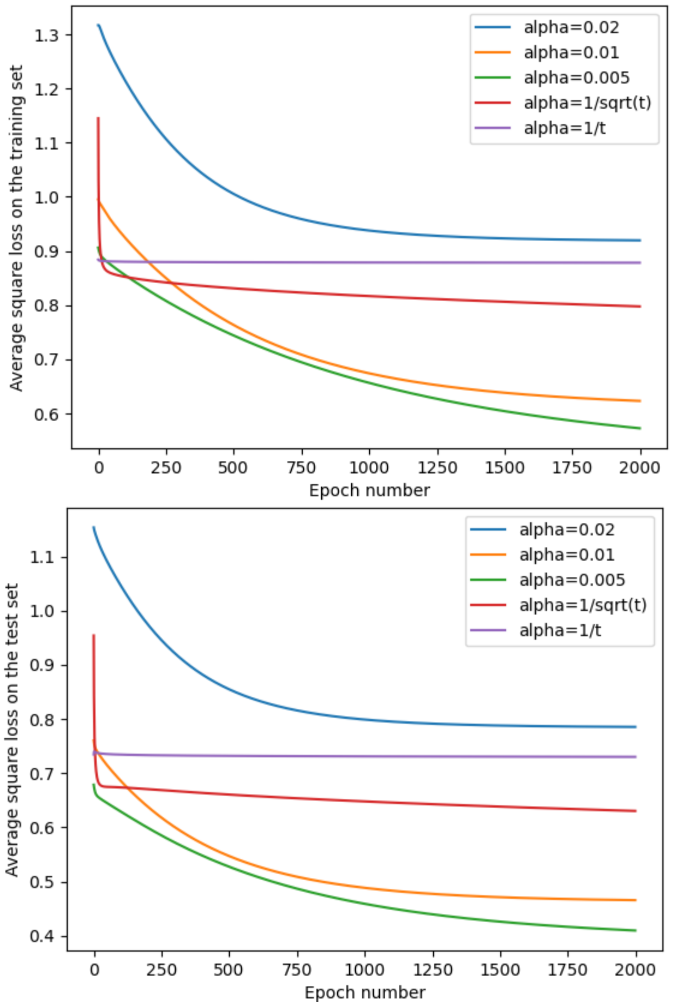
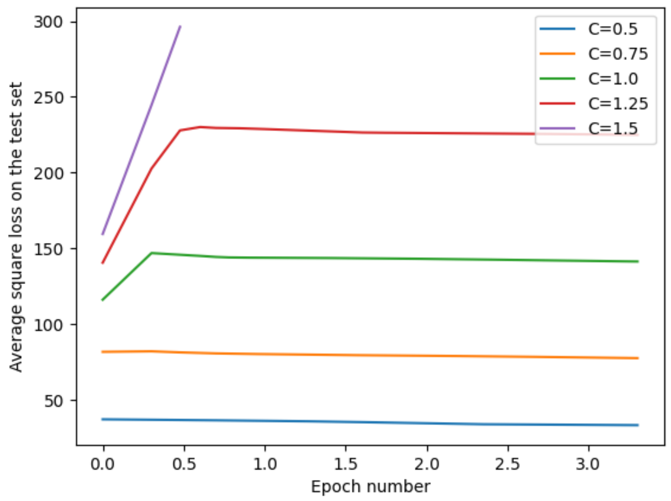


Figure 7: The change of average square loss on the train set (above) and on the test set (below) with the increase of epochs in the l_2 regularized stochastic gradient descent algorithm under different schemes of fixed/variable step sizes and regularization parameter $\lambda = 10^{-3.5}$.

In order to address the effect of C on variable step sizes, we use $\eta_t = C/\sqrt{t}$ as an illustration. We choose large values of C ranging from 0.5 up to 1.5, and small values of C ranging from 0.1 down to 0.01, as are shown in Figure 8. As we can see from the plot above, large values of C are too aggressive in the beginning of the stochastic gradient descent, thus leading to an explosion in loss and is unlikely to converge in reasonable time. As we can see from the plot below, too small values of C are also unfavorable due to too slow convergence in the long term.



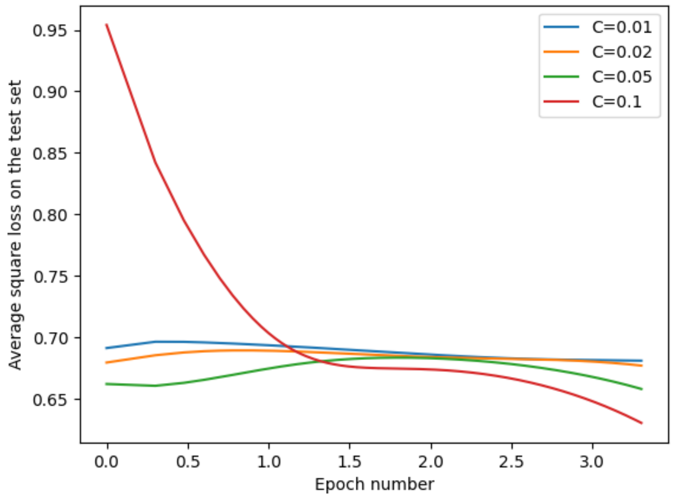


Figure 8: The change of average square loss on test set (below) with the increase of epochs in the l_2 regularized stochastic gradient descent algorithm under different variable step sizes and regularization parameter $\lambda = 10^{-3.5}$.

A few remarks about the question above:

- In this case we are investigating the convergence rate of the optimization algorithm with different step size schedules, thus we're interested in the value of the objective function, which includes the regularization term.
- Sometimes the initial step size (C for C/t and $C/\sqrt{t}$) is too aggressive and will get you into a part of parameter space from which you can't recover. Try reducing C to counter this problem.

- SGD convergence is much slower than GD once we get close to the minimizer (remember, the SGD step directions are very noisy versions of the GD step direction). If you look at the objective function values on a logarithmic scale, it may look like SGD will never find objective values that are as low as GD gets. In statistical learning theory terminology, GD has much smaller *optimization* error than SGD. However, this difference in optimization error is usually dominated by other sources of error (estimation error and approximation error). Moreover, for very large datasets, SGD (or minibatch GD) is much faster (by wall-clock time) than GD to reach a point close enough to the minimizer.

Acknowledgement: This problem set is based on assignments developed by David Rosenberg of NYU and Bloomberg.

Image Classification with Regularized Logistic Regression

In this problem set we will examine a classification problem. To do so we will use the MNIST dataset[2] which is one of the traditional image benchmark for machine learning algorithms. We will only load the data from the 0 and 1 class, and try to predict the class from the image. You will find the support code for this problem in `mnist_classification_source_code.py`. Before starting, take a little time to inspect the data. Load `X_train, y_train, X_test, y_test` with `pre_process_mnist_01()`. Using the function `plt.imshow` from `matplotlib` visualize some data points from `X_train` by reshaping the 764 dimensional vectors into 28×28 arrays. Note how the class labels '0' and '1' have been encoded in `y_train`. No need to report these steps in your submission.

Logistic regression

We will use here again a linear model, meaning that we will fit an affine function,

$$h_{\boldsymbol{\theta},b}(\boldsymbol{x}) = \boldsymbol{\theta}^\top \boldsymbol{x} + b,$$

with $\boldsymbol{x} \in \mathbb{R}^{764}$, $\boldsymbol{\theta} \in \mathbb{R}^{764}$ and $b \in \mathbb{R}$. This time we will use the logistic loss instead of the squared loss. Instead of coding everything from scratch, we will also use the package `scikit-learn` and study the effects of l_1 regularization. You may want to check that you have a version of the package up to date (0.24.1).

26. Recall the definition of the logistic loss between target y and a prediction $h_{\boldsymbol{\theta},b}(\boldsymbol{x})$ as a function of the margin $m = y \cdot h_{\boldsymbol{\theta},b}(\boldsymbol{x})$. Show that given that we chose the convention $y_i \in \{-1, 1\}$, our objective function over the training data $\{\boldsymbol{x}_i, y_i\}_{i=1}^m$ can be rewritten as

$$L(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} \left((1 + y_i) \log \left(1 + e^{-h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)} \right) + (1 - y_i) \log \left(1 + e^{h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)} \right) \right).$$

Proof. Recall that the logistic loss between target y and a prediction $h_{\boldsymbol{\theta},b}(\boldsymbol{x})$ is defined as

$$\ell(\boldsymbol{x}, y) = \log \left(1 + e^{-y h_{\boldsymbol{\theta},b}(\boldsymbol{x})} \right), \tag{18}$$

[2] `http://yann.lecun.com/exdb/mnist/`

Therefore, we can express the objective function as

$$L(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}\log\left(1+e^{-y_i h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)}\right). \tag{19}$$

Note that when $y=1$, we have that

$$\log\left(1+e^{-y h_{\boldsymbol{\theta},b}(\boldsymbol{x})}\right) = \log\left(1+e^{-h_{\boldsymbol{\theta},b}(\boldsymbol{x})}\right) =: f_1(\boldsymbol{x}), \tag{20}$$

and when $y=-1$, we have that

$$\log\left(1+e^{-y h_{\boldsymbol{\theta},b}(\boldsymbol{x})}\right) = \log\left(1+e^{h_{\boldsymbol{\theta},b}(\boldsymbol{x})}\right) =: f_{-1}(\boldsymbol{x}). \tag{21}$$

Therefore, we can rewrite the objective function as

$$\begin{aligned} L(\boldsymbol{\theta}) &= \frac{1}{m}\sum_{i=1}^{m}\left(\frac{1+y_i}{2}f_1(\boldsymbol{x}_i)+\frac{1-y_i}{2}f_{-1}(\boldsymbol{x}_i)\right) \\ &= \frac{1}{2m}\sum_{i=1}^{m}\left((1+y_i)\log\left(1+e^{-h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)}\right)+(1-y_i)\log\left(1+e^{h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)}\right)\right), \end{aligned} \tag{22}$$

since

$$\frac{1+y_i}{2} = \begin{cases}1, & \text{if } y_i = 1,\\ 0, & \text{if } y_i=-1,\end{cases} \qquad \frac{1-y_i}{2} = \begin{cases}0, & \text{if } y_i = 1,\\ 1, & \text{if } y_i=-1,\end{cases} \tag{23}$$

thus concluding the proof. □

27. What will become the loss function if we regularize the coefficients of $\boldsymbol{\theta}$ with an l_1 penalty using a regularization parameter α?

Solution. With the l_1 penalty, the objective function will become

$$L(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}\left((1+y_i)\log\left(1+e^{-h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)}\right)+(1-y_i)\log\left(1+e^{h_{\boldsymbol{\theta},b}(\boldsymbol{x}_i)}\right)\right)+\alpha\sum_{j=1}^{764}|\theta_j|. \tag{24}$$

We are going to use the module `SGDClassifier` from `scikit-learn`. In the code provided we have set a little example of its usage. By checking the online documentation[3], make sure you understand the meaning of all the keyword arguments that were specified. We will keep the learning rate schedule and the maximum number of iterations fixed to the given values for all the problem. Note that `scikit-learn` is actually implementing a fancy version of SGD to deal with the l_1 penalty which is not differentiable everywhere, but we will not enter these details here.

[3]`https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html`

28. To evaluate the quality of our model we will use the classification error, which corresponds to the fraction of incorrectly labeled examples. For a given sample, the classification error is 1 if no example was labeled correctly and 0 if all examples were perfectly labeled. Using the method `clf.predict()` from the classifier, write a function that takes as input an `SGDClassifier` which we will call `clf`, a design matrix `X` and a target vector `y` and returns the classification error. You should check that your function returns the same value as `1 - clf.score(X, y)`.

    Solution. The function `classification_error` can be implemented as follows:

```python
def classification_error(clf, X, y):
    """Compute the classification error.

    Parameters
    ----------
    clf : SGDClassifier
        The SGD classifier object that has been fit to the training data.
    X : np.ndarray
        The design matrix of size n_test_samples * n_features
    y : np.array
        The target vector of size n_test_samples.

    Returns
    -------
    err : float
        The classification error.
    """
    y_pred = clf.predict(X)
    return np.mean(y_pred != y)
```

To speed up computations we will subsample the data. Using the function `sub_sample`, restrict `X_train` and `y_train` to `N_train = 100`.

29. Report the test classification error achieved by the logistic regression as a function of the regularization parameters α (taking 10 values between 10^{-4} and 10^{-1}). You should make a plot with α as the x-axis in log scale. For each value of α, you should repeat the experiment 10 times so as to finally report the mean value and the standard deviation. You should use `plt.errorbar` to plot the standard deviation as error bars.

    Solution. The test classification errors achieved by the logistic regression are as shown in Figure 9. Note that the x-axis of the plot is of logarithmic scale.

30. Which source(s) of randomness are we averaging over by repeating the experiment?

    Solution. The randomness comes from randomly shuffling the samples in the training step within each epoch, since the `SGDClassifier` estimator implements regularized linear models with stochastic gradient descent learning, according to the documentation.

31. What is the optimal value of the parameter α among the values you tested?

    Solution. The optimal value of α is 10^{-1} among all the values that I tested.

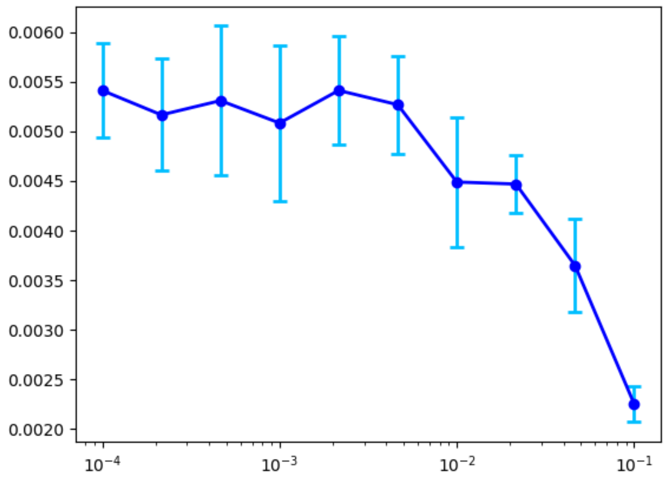


Figure 9: The change the test classification error achieved by the logistic regression with the regularization parameter α. Each data point and the corresponding error bar represent the mean and the standard deviation of the 10 experiments for each value of α.

32. Finally, for one run of the fit for each value of α plot the value of the fitted $\boldsymbol{\theta}$. You can access it via `clf.coef_`, and should reshape the 764 dimensional vector to a 28×28 array to visualize it with `plt.imshow`. Defining `scale = np.abs(clf.coef_).max()`, you can use the following keyword arguments (`cmap=plt.cm.RdBu, vmax=scale, vmin=-scale`) which will set the colors nicely in the plot. You should also use a `plt.colorbar()` to visualize the values associated with the colors.

    Solution. The plots of the fitted $\boldsymbol{\theta}$ for each value of α between 10^{-3} and 10^{-1} are shown as in Figure 10.

33. What can you note about the pattern in $\boldsymbol{\theta}$? What can you note about the effect of the regularization?

    Solution. The red part looks like a circle or zero, and the blue part looks like a line or one. As the regularization parameter α increases, the both the blue part and the red part are vanishing, or in other words, there are increasingly more white parts, indicating increasingly more zero entries in $\boldsymbol{\theta}$. This is reasonable since l_1 regularization tends to yield solutions, i.e., it tends to make entries near 0 to become exactly 0. Therefore, with a larger regularization parameter, more entries will become 0, consistent with the results shown in Figure 10.

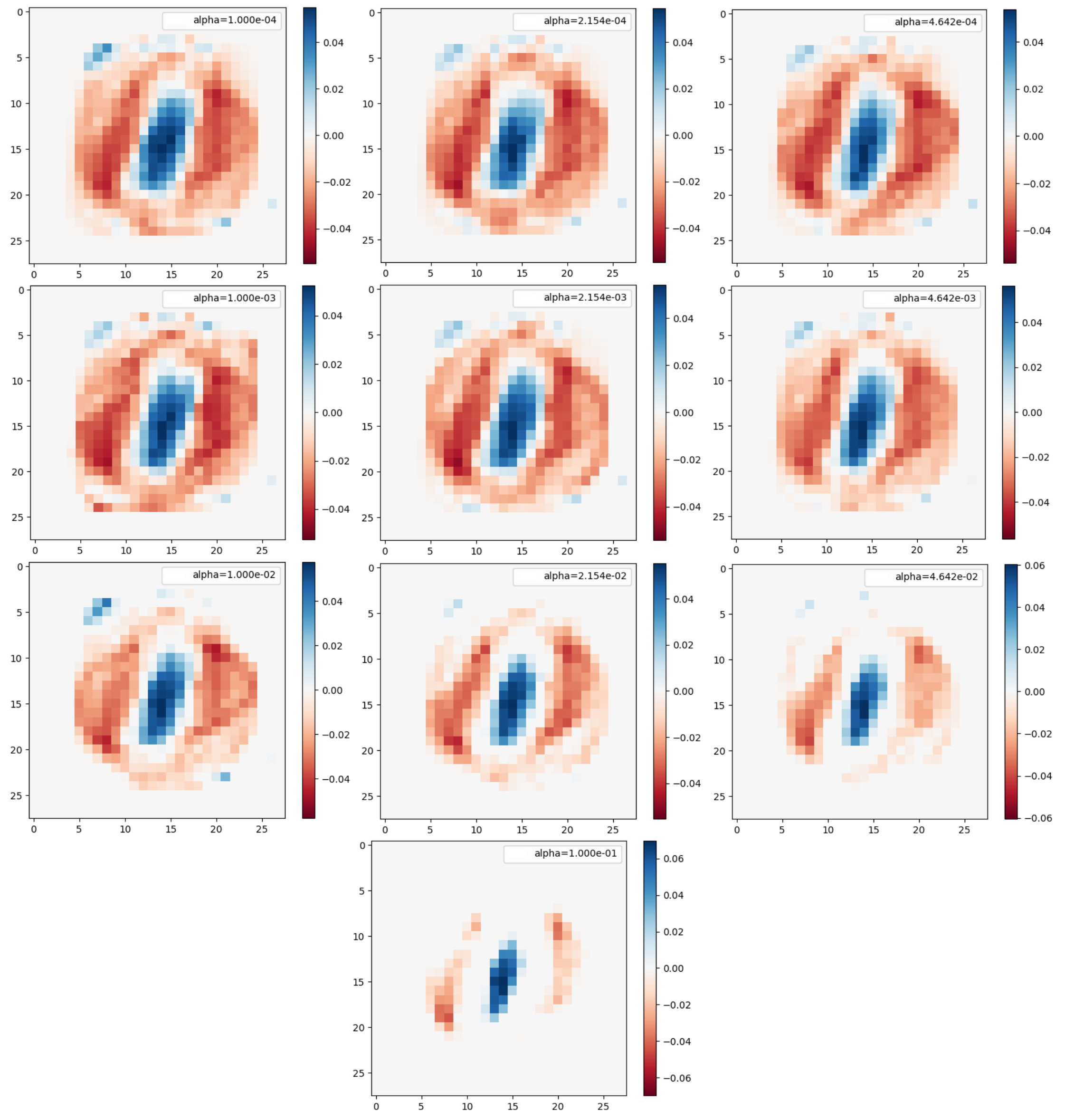


Figure 10: The values of the fitted $\boldsymbol{\theta}$ for the regularization parameter α in the range 10^{-3} to 10^{-1} (increasing from left to right and from top to bottom).

Homework 3: SVMs & Kernel Methods

Due: Wednesday, March 1, 2023 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

In this problem set we will get up to speed with SVMs and Kernels. Long at first glance, the problem set includes a lot of helpers. You will find a review of kernalization. One section will include a revision of ridge regression which you should start to be familiar with. For the second and third problem some codes are provided to save you some time. Finally, some reminders on positive (semi)definite matrices are included in the Appendix.

Support Vector Machines: SVMs with Pegasos

In this first problem we will use Support Vector Machines to predict whether the sentiment of a movie review was *positive* or *negative*. We will represent each review by a vector $\boldsymbol{x} \in \mathbb{R}^d$ where d is the size of the word dictionary and x_i is equal to the number of occurrence of the ith word in the review $\boldsymbol{x}$. The corresponding label is either $y = 1$ for a positive review or $y = -1$ for a negative review. In class we have seen how to transform the SVM training objective into a quadratic program using the dual formulation. Here we will use a gradient descent algorithm instead.

Subgradients

Recall that a vector $\boldsymbol{g} \in \mathbb{R}^d$ is a *subgradient* of $f : \mathbb{R}^d \to \mathbb{R}$ at $\boldsymbol{x}$ if for all $\boldsymbol{z}$,

$$f(\boldsymbol{z}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{z} - \boldsymbol{x}).$$

There may be 0, 1, or infinitely many subgradients at any point. The *subdifferential* of f at a point $\boldsymbol{x}$, denoted $\partial f(\boldsymbol{x})$, is the set of all subgradients of f at $\boldsymbol{x}$. A good reference for subgradients are the course notes on Subgradients by Boyd et al. Below we derive a property that will make our life easier for finding a subgradient of the hinge loss.

1. Suppose $f_1, \cdots, f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(\boldsymbol{x}) = \max_{i=1,\cdots,m} f_i(\boldsymbol{x})$. Let k be any index for which $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, and choose $\boldsymbol{g} \in \partial f_k(\boldsymbol{x})$ (a convex function on $\mathbb{R}^d$ has a non-empty subdifferential at all points). Show that $\boldsymbol{g} \in \partial f(\boldsymbol{x})$.

   Proof. For an arbitrary $\boldsymbol{z} \in \mathbb{R}^d$, we have that

   $$f(\boldsymbol{z}) \geq f_k(\boldsymbol{z}) \geq f_k(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{z} - \boldsymbol{x}) = f(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{z} - \boldsymbol{x}). \quad (1)$$

   By the arbitrariness of $\boldsymbol{z}$, we can conclude that $\boldsymbol{g} \in \partial f(\boldsymbol{x})$. □

2. Give a subgradient of the hinge loss objective $J(\boldsymbol{w}) = \max\left\{0, 1 - y\boldsymbol{w}^\top\boldsymbol{x}\right\}$.

Solution. By the previous question, in order to find a subgradient of the hinge loss objective $J(\boldsymbol{w})$, we only need to find a subgradient of $J_1(\boldsymbol{w}) = 0$ when $y\boldsymbol{w}^\top \boldsymbol{x} > 1$ and a subgradient of $J_2(\boldsymbol{w}) = 1 - y\boldsymbol{w}^\top \boldsymbol{x}$ when $y\boldsymbol{w}^\top \boldsymbol{x} \leq 1$. However, note that J_1 and J_2 are both differentiable, thus their only possible subgradients are their gradients. We have that

$$\nabla_{\boldsymbol{w}} J_1(\boldsymbol{w}) = \boldsymbol{0}, \qquad \nabla_{\boldsymbol{w}} J_2(\boldsymbol{w}) = -y\boldsymbol{x}. \tag{2}$$

Therefore, a subgradient of the hinge loss objective $J(\boldsymbol{w})$ can be

$$\boldsymbol{g} = \begin{cases} \boldsymbol{0}, & \text{if } y\boldsymbol{w}^\top \boldsymbol{x} > 1, \\ -y\boldsymbol{x}, & \text{otherwise.} \end{cases} \tag{3}$$

3. (Optional) Suppose we have function $f : \mathbb{R}^n \to \mathbb{R}$ which is sub-differentiable everywhere, i.e. $\partial f \neq \emptyset$ for all $x \in \mathbb{R}^n$. Show that f is convex. Note, in the general case, a function is convex if for all x, y in the domain of f and for all $\theta \in (0, 1)$,

$$\theta f(\boldsymbol{a}) + (1-\theta) f(\boldsymbol{b}) \geq f(\theta \boldsymbol{a} + (1-\theta)\boldsymbol{b}).$$

Proof. Assume for contradiction that f is not convex, then there exist $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^d$, and $\theta \in (0, 1)$, such that

$$\theta f(\boldsymbol{a}) + (1-\theta) f(\boldsymbol{b}) < f(\theta \boldsymbol{a} + (1-\theta)\boldsymbol{b}). \tag{4}$$

Now since f is sub-differentiable everywhere, there exists a subgradient $\boldsymbol{g}$ at the point $\boldsymbol{x} = \theta\boldsymbol{a} + (1-\theta)\boldsymbol{b}$. By definition of subgradient, we thus have that

$$f(\boldsymbol{a}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{a} - \boldsymbol{x}) = f(\boldsymbol{x}) + \boldsymbol{g}^\top((1-\theta)\boldsymbol{a} - (1-\theta)\boldsymbol{b}), \tag{5}$$

$$f(\boldsymbol{b}) \geq f(\boldsymbol{x}) + \boldsymbol{g}^\top(\boldsymbol{b} - \boldsymbol{x}) = f(\boldsymbol{x}) + \boldsymbol{g}^\top(\theta\boldsymbol{b} - \theta\boldsymbol{a}). \tag{6}$$

Since $\theta \in (0, 1)$, we can rewrite the inequalities above as

$$\boldsymbol{g}^\top(\boldsymbol{a} - \boldsymbol{b}) \leq \frac{f(\boldsymbol{a}) - f(\boldsymbol{x})}{1-\theta}, \tag{7}$$

$$\boldsymbol{g}^\top(\boldsymbol{b} - \boldsymbol{a}) \leq \frac{f(\boldsymbol{b}) - f(\boldsymbol{x})}{\theta}. \tag{8}$$

Adding the two inequalities above, we have that

$$0 \leq \frac{f(\boldsymbol{a}) - f(\boldsymbol{x})}{1-\theta} + \frac{f(\boldsymbol{b}) - f(\boldsymbol{x})}{\theta} = \frac{\theta f(\boldsymbol{a}) + (1-\theta) f(\boldsymbol{b}) - f(\boldsymbol{x})}{\theta(1-\theta)}. \tag{9}$$

However, by the assumption in (4), the right-hand side in the inequality above is strictly less than 0, thus leading to a contradiction. Therefore, f is convex and the proof is complete. □

SVM with the Pegasos algorithm

You will train a Support Vector Machine using the Pegasos algorithm [1]. Recall the SVM objective using a linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x}$ and the hinge loss:

$$\min_{\boldsymbol{w} \in \mathbb{R}^d} \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \frac{1}{n}\sum_{i=1}^{n} \max\left\{0, 1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i\right\},$$

[1] Shalev-Shwartz et al. Pegasos: Primal Estimated sub-GrAdient SOlver for SVM.

where n is the number of training examples and d the size of the dictionary. Note that, for simplicity, we are leaving off the bias term b. Note also that we are using l_2 regularization with a parameter λ. Pegasos is stochastic subgradient descent using a step size rule $\eta_t = 1/(\lambda t)$ for iteration number t. The pseudocode is given below:

Algorithm 1 Pegasos

```
Require: $\lambda > 0$, choose $\boldsymbol{w}_1 = \mathbf{0}$ and $t = 0$.
 1: while termination condition is not met do
 2:   for $j = 1, \cdots, n$ (assuming data is randomly permuted) do
 3:     $t \leftarrow t + 1$;
 4:     $\eta_t \leftarrow 1/(t\lambda)$;
 5:     if $y_i \boldsymbol{w}_t^\top \boldsymbol{x}_j < 1$ then
 6:       $\boldsymbol{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\boldsymbol{w}_t + \eta_t y_j \boldsymbol{x}_j$;
 7:     else
 8:       $\boldsymbol{w}_{t+1} \leftarrow (1 - \eta_t \lambda)\boldsymbol{w}_t$;
 9:     end if
10:   end for
11: end while
```

4. Consider the SVM objective function for a single training point[2]

$$J_i(\boldsymbol{w}) = \frac{\lambda}{2}\|\boldsymbol{w}\|^2 + \max\left\{0, 1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i\right\}.$$

The function $J_i(\boldsymbol{w})$ is not everywhere differentiable. Specify where the gradient of $J_i(\boldsymbol{w})$ is not defined. Give an expression for the gradient where it is defined.

Solution. The function is not differentiable when $y_i \boldsymbol{w}^\top \boldsymbol{x}_i = 0$. Otherwise, we have that

$$\nabla_{\boldsymbol{w}} J_i(\boldsymbol{w}) = \begin{cases} \lambda \boldsymbol{w}, & \text{if } y\boldsymbol{w}^\top \boldsymbol{x} > 1, \\ \lambda \boldsymbol{w} - y\boldsymbol{x}, & \text{if } y\boldsymbol{w}^\top \boldsymbol{x} < 1. \end{cases} \tag{10}$$

5. Show that a subgradient of $J_i(w)$ is given by

$$\boldsymbol{g}_{\boldsymbol{w}} = \begin{cases} \lambda \boldsymbol{w} - y_i \boldsymbol{x}_i, & \text{if } y_i \boldsymbol{w}^\top \boldsymbol{x}_i < 1 \\ \lambda \boldsymbol{w}, & \text{if } y_i \boldsymbol{w}^\top \boldsymbol{x}_i \geq 1. \end{cases}$$

You may use the following facts without proof: (1) If $f_1, \ldots, f_n : \mathbb{R}^d \to \mathbb{R}$ are convex functions and $f = f_1 + \cdots + f_n$, then $\partial f(\boldsymbol{x}) = \partial f_1(\boldsymbol{x}) + \cdots + \partial f_n(\boldsymbol{x})$. (2) For $\alpha \geq 0$, $\partial(\alpha f)(\boldsymbol{x}) = \alpha \partial f(\boldsymbol{x})$.

Hint: Use the first part of this problem.

Proof. Using (3), the subgradient of the function $\max\{1 - y_i \boldsymbol{w}^\top \boldsymbol{x}_i\}$ can be

$$\boldsymbol{h} = \begin{cases} -y\boldsymbol{x}, & \text{if } y\boldsymbol{w}^\top \boldsymbol{x} < 1, \\ \mathbf{0}, & \text{if } y\boldsymbol{w}^\top \boldsymbol{x} \geq 1. \end{cases} \tag{11}$$

[2]Recall that if i is selected uniformly from the set $\{1, \ldots, n\}$, then this objective function has the same expected value as the full SVM objective function.

Moreover, the function $\frac{\lambda}{2}\|\boldsymbol{w}\|^2$ is everywhere differentiable, so its subgradient is

$$\boldsymbol{k} = \nabla_{\boldsymbol{w}}\left(\frac{\lambda}{2}\|\boldsymbol{w}\|^2\right) = \lambda\boldsymbol{w}. \tag{12}$$

Both functions are convex, since they are both everywhere differentiable. Therefore, by the linearity of subgradients, we have that

$$\boldsymbol{g}_{\boldsymbol{w}} = \boldsymbol{h} + \boldsymbol{k} = \begin{cases} \lambda\boldsymbol{w} - y\boldsymbol{x}, & \text{if } y\boldsymbol{w}^\top\boldsymbol{x} < 1, \\ \lambda\boldsymbol{w}, & \text{if } y\boldsymbol{w}^\top\boldsymbol{x} \geq 1, \end{cases} \tag{13}$$

as desired, so the proof is complete. □

Convince yourself that if the step size rule is $\eta_t = 1/(\lambda t)$, then doing SGD with the subgradient direction from the previous question is the same as given in the pseudocode.

Dataset and sparse representation

We will be using the Polarity Dataset v2.0, constructed by Pang and Lee, provided in the `data_reviews/` folder. It has the full text from 2000 movies reviews: 1000 reviews are classified as *positive* and 1000 as *negative*. Our goal is to predict whether a review has positive or negative sentiment from the text of the review. Each review is stored in a separate file: the positive reviews are in a folder called "pos", and the negative reviews are in "neg". We have provided some code in `utils_svm_reviews.py` to assist with reading these files. The code removes some special symbols from the reviews and shuffles the data. Load all the data to have an idea of what it looks like.

A usual method to represent text documents in machine learning is with *bag-of-words*. As hinted above, here every possible word in the dictionnary is a feature, and the value of a word feature for a given text is the number of times that word appears in the text. As most words will not appear in any particular document, many of these counts will be zero. Rather than storing many zeros, we use a *sparse representation*, in which only the nonzero counts are tracked. The counts are stored in a key/value data structure, such as a dictionary in Python. For example, "Harry Potter and Harry Potter II" would be represented as the following Python dictionary: `x = {'Harry':2, 'Potter':2, 'and':1, 'II':1}`.

6. Write a function that converts an example (a list of words) into a sparse bag-of-words representation. You may find Python's `Counter`[3] class to be useful here.

   Solution. We present the function `to_bag` as follows.

```python
def to_bag(words):
    """Convert a list of words into a sparse bag-of-words representation.

    Parameters
    ----------
    words : list
        The list of words to convert.

    Returns
    -------
    bag : dict
```

[3]`https://docs.python.org/3/library/collections.html`

```
        The bag-of-words, with keys as words and values as their
        corresponding counts.

    Example
    -------
    >>> x = ["Harry", "Potter", "and", "Harry", "Potter", "II"]
    >>> to_bag(x)
    {'Harry': 2, 'Potter': 2, 'and': 1, 'II': 1}
    """
    return dict(Counter(words))
```

7. Load all the data and split it into 1500 training examples and 500 validation examples. Format the training data as a list `X_train` of dictionaries and `y_train` as the list of corresponding 1 or -1 labels. Format the test set similarly.

   Solution. We present the function `get_train_and_test` as follows.

```
def get_train_and_test():
    """Split into a train set and a test set.

    Returns
    -------
    X_train : list of dict
        A train set of 1500 bags of words, with keys as words and values as
        their corresponding counts.
    X_test : list of dict
        A test set of 1500 bags of words, with keys as words and values as
        their corresponding counts.
    y_train : list
        A train set of 1500 binary sentiments, 1 as positive and -1 as
        negative.
    y_test : list
        A test set of 1500 binary sentiments, 1 as positive and -1 as
        negative.
    """
    X, y = [], []
    for review in load_and_shuffle_data():
        X.append(to_bag(review[:-1]))
        y.append(review[-1])
    return train_test_split(X, y, test_size=0.25, random_state=42)
```

We will be using linear classifiers of the form $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$, and we can store the vector $\boldsymbol{w}$ in a sparse format as well, such as `w = {‘minimal’:1.3, ‘Harry’:-1.1, ‘viable’:-4.2, ‘and’:2.2, ‘product’:9.1}`. The inner product between $\boldsymbol{w}$ and $\boldsymbol{x}$ would only involve the features that appear in both `x` and `w`, since whatever doesn't appear is assumed to be zero. For this example, the inner product would be `x(‘Harry’) * w(‘Harry’) + x(‘and’) * w(‘and’) = 2*(-1.1) + 1*(2.2)`. To help you along, `utils_svm_reviews.py` includes two functions for working with sparse vectors: (1) a dot product between two vectors represented as dictionaries and (2) a function that increments one sparse vector by a scaled multiple of another vector, which is a very common operation. It is worth reading the code, even if you intend to implement it yourself. You may get some ideas on how to make things faster.

8. Implement the Pegasos algorithm for a sparse data representation. The output should be a sparse weight vector $\boldsymbol{w}$ represented as a dictionary. Note that our Pegasos algorithm starts at $\boldsymbol{w} = \boldsymbol{0}$, which corresponds to an empty dictionary.

Remark 1: With this problem, you will need to take some care to code things efficiently. In particular, be aware that making copies of the weight dictionary can slow down your code significantly. If you want to make a copy of your weights (e.g. for checking for convergence), make sure you don't do this more than once per epoch.

Remark 2: If you normalize your data in some way, be sure not to destroy the sparsity of your data. Anything that starts as zero should stay as zero.

Solution. We implement the Pegasos algorithm as follows.

```python
def pegasos(X, y, w, lambd=1, num_epoch=1000):
    """The Pegasos algorithm.

    Parameters
    ----------
    X : list of dict
        A list of bags of words, with keys as words and values as their
        corresponding counts.
    y : list
        A list of binary sentiments, 1 as positive and -1 as negative.
    w : dict
        A bag of words, the initial value for training.
    lambd : float
        A parameter for the learning rate. For step t, learning rate =
        1 / (lambd * t)
    num_epoch : int
        The number of epochs for training.

    Notes
    -----
    Note that this function modifies w in place.
    """
    t, ran = 1, list(range(len(y)))
    for _ in range(num_epoch):
        random.shuffle(ran)
        for j in ran:
            t += 1
            eta = 1 / (t * lambd)
            if y[j] * dotProduct(w, X[j]) < 1:
                increment(w, -eta * lambd, w)
                increment(w, eta * y[j], X[j])
            else:
                increment(w, -eta * lambd, w)
```

Note that in every step of the Pegasos algorithm, we rescale every entry of $\boldsymbol{w}_t$ by the factor $(1-\eta_t\lambda)$. Implementing this directly with dictionaries is very slow. We can make things significantly faster by representing $\boldsymbol{w} = s\boldsymbol{W}$, where $s \in \mathbb{R}$ and $\boldsymbol{W} \in \mathbb{R}^d$. You can start with $s = 1$ and $\boldsymbol{W}$ all zeros (i.e. an empty dictionary). Note that both updates (i.e. whether or not we have a margin error) start with rescaling $\boldsymbol{w}_t$, which we can do simply by setting $s_{t+1} = (1-\eta_t\lambda)\, s_t$.

9. If the Pegasos update step is $\boldsymbol{w}_{t+1} = (1-\eta_t\lambda)\boldsymbol{w}_t + \eta_t y_j \boldsymbol{x}_j$, verify that it is equivalent to

$$\begin{aligned} s_{t+1} &= (1-\eta_t\lambda)\, s_t, \\ \boldsymbol{W}_{t+1} &= \boldsymbol{W}_t + \frac{1}{s_{t+1}}\eta_t y_j \boldsymbol{x}_j. \end{aligned}$$

Implement the Pegasos algorithm with the $(s, \boldsymbol{W})$ representation described above.[4]

Solution. Since we require that $\boldsymbol{w} = s\boldsymbol{W}$ in each step, with the described update we have

$$\begin{aligned} \boldsymbol{w}_{t+1} = s_{t+1}\boldsymbol{W}_{t+1} &= (1-\eta_t\lambda)s_t\left(\boldsymbol{W}_t + \frac{1}{(1-\eta_t\lambda)s_t}\eta_t y_j \boldsymbol{x}_j\right) \\ &= (1-\eta_t\lambda)s_t\boldsymbol{W}_t + \eta_t y_j \boldsymbol{x}_j = (1-\eta_t\lambda)\boldsymbol{w}_t + \eta_t y_j \boldsymbol{x}_j, \end{aligned} \tag{14}$$

which is exactly the correct Pegasos update step in this case. The implementation of the Pegasos algorithm using this $(s, \boldsymbol{W})$ representation is as follows.

```python
def sw_pegasos(X, y, w, lambd=1, num_epoch=1000):
    """The Pegasos algorithm implemented with the (s,W) representation.

    Parameters
    ----------
    X : list of dict
        A list of bags of words, with keys as words and values as their
        corresponding counts.
    y : list
        A list of binary sentiments, 1 as positive and -1 as negative.
    w : dict
        A bag of words, the initial value for training.
    lambd : float
        A parameter for the learning rate. For step t, learning rate =
        1 / (lambd * t)
    num_epoch : int
        The number of epochs for training.

    Notes
    -----
    Note that this function modifies w in place.
    """
    t, ran, s, W = 1, list(range(len(y))), 1, {}
    for _ in range(num_epoch):
        random.shuffle(ran)
        for j in ran:
            t += 1
            eta = 1 / (t * lambd)
            if y[j] * s * dotProduct(W, X[j]) < 1:
                s *= (1 - eta * lambd)
                increment(W, eta * y[j] / s, X[j])
            else:
                s *= (1 - eta * lambd)
    increment(w, s, W)
```

10. Run both implementations of the Pegasos algorithm on the training data for a couple of epochs. Make sure your implementations are correct by verifying that the two approaches give essentially the same result. Report on the time taken to run each approach.

    Solution. The runtime of the two implementations of the Pegasos algorithm is as follows.

[4]There is one subtle issue with the approach described above: if we ever have $1 - \eta_t\lambda = 0$, then $s_{t+1} = 0$, and we'll have a divide by 0 in the calculation for $\boldsymbol{W}_{t+1}$. This only happens when $\eta_t = 1/\lambda$. With our step-size rule of $\eta_t = 1/(\lambda t)$, it happens exactly when $t = 1$. So one approach is to just start at $t = 2$. More generically, note that if $s_{t+1} = 0$, then $w_{t+1} = 0$. Thus an equivalent representation is $s_{t+1} = 1$ and $\boldsymbol{W} = 0$. Thus if we ever get $s_{t+1} = 0$, simply set it back to 1 and reset $\boldsymbol{W}_{t+1}$ to zero, which is an empty dictionary in a sparse representation.

```
Splitting into training and testing sets... done.
Running intuitive Pegasos... completed in 44.9340 seconds.
Running (s,W)-representation Pegasos... completed in 0.8463 seconds.
```

Though the outputs are not exactly the same, the greatest difference is just a bit over the machine precision 2^{-53}, thus we consider the two approaches to be giving essentially the same result.

11. Write a function `classification_error` that takes a sparse weight vector $\boldsymbol{w}$, a list of sparse vectors X and the corresponding list of labels $\boldsymbol{y}$, and returns the fraction of errors when predicting y_i using $\text{sign}(\boldsymbol{w}^\top \boldsymbol{x}_i)$. In other words, the function reports the 0-1 loss of the linear predictor $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$.

    Solution. The function `classification_error` can be implemented as follows.

```python
def classification_error(X, y, w):
    """Compute the classification error.

    Parameters
    ----------
    X : list of dict
        A list of bags of words, with keys as words and values as their
        corresponding counts.
    y : list
        A list of binary sentiments, 1 as positive and -1 as negative.
    w : dict
        A sparse weight vector, e.g., the prediction result of the Pegasos
        algorithm.

    Returns
    -------
    err : float
        The classification error.
    """
    accum = 0
    for j in range(len(y)):
        if y[j] * dotProduct(w, X[j]) < 0:
            accum += 1
    return accum / len(y)
```

12. Search for the regularization parameter that gives the minimal percent error on your test set. You should now use your faster Pegasos implementation, and run it to convergence. A good search strategy is to start with a set of regularization parameters spanning a broad range of orders of magnitude. Then, continue to zoom in until you're convinced that additional search will not significantly improve your test performance. Plot the test errors you obtained as a function of the parameters λ you tested.

    Hint: The error you get with the best regularization should be closer to 15% than 20%. If not, maybe you did not train to convergence.)

    Solution. The plot of the classification errors on the test set is shown as in Figure 1. From the plot, we can see that the regularization parameter that gives the minimal percent error on the test set is approximately $\lambda = 5 \times 10^{-4}$.

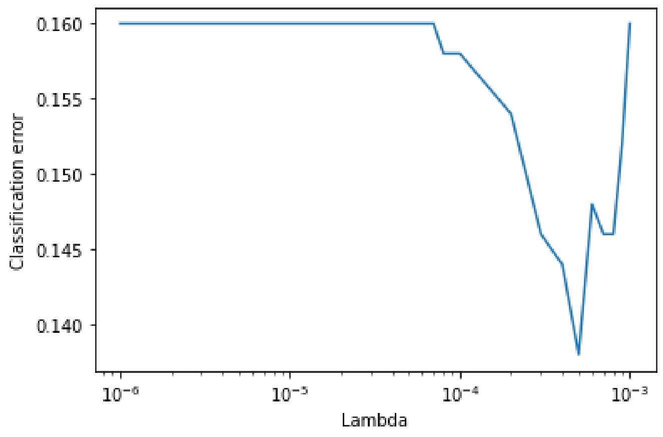


Figure 1: The relation between the classification error of the test set and the regularization parameter λ.

Error Analysis (Optional)

Recall that the *score* is the value of the prediction $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$. We like to think that the magnitude of the score represents the confidence of the prediction. This is something we can directly verify or refute.

13. (Optional) Break the predictions on the test set into groups based on the score (you can play with the size of the groups to get a result you think is informative). For each group, examine the percentage error. You can make a table or graph. Summarize the results. Is there a correlation between higher magnitude scores and accuracy?

    Solution. We break the test set into 6 groups based on the absolute values of the scores, with each group having the same length of range of scores. Specifically in this case,

    ```
    Group #0: (0.51, 49.31)
    Group #1: (49.31, 98.11)
    Group #2: (98.11, 146.90)
    Group #3: (146.90, 195.70)
    Group #4: (195.70, 244.50)
    Group #5: (244.50, 293.30)
    ```

    The classification errors for each group can be seen as in Figure 2. Clearly, we can see that the larger the group number, the less the classification error. In other words, the larger the absolute value of the score, the better confidence we have for that prediction.

In natural language processing one can often interpret why a model has performed well or poorly on a specific example. The first step in this process is to look closely at the errors that the model makes.

14. (Optional) Choose an input example $\boldsymbol{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ that the model got wrong. We want to investigate what features contributed to this incorrect prediction. One way

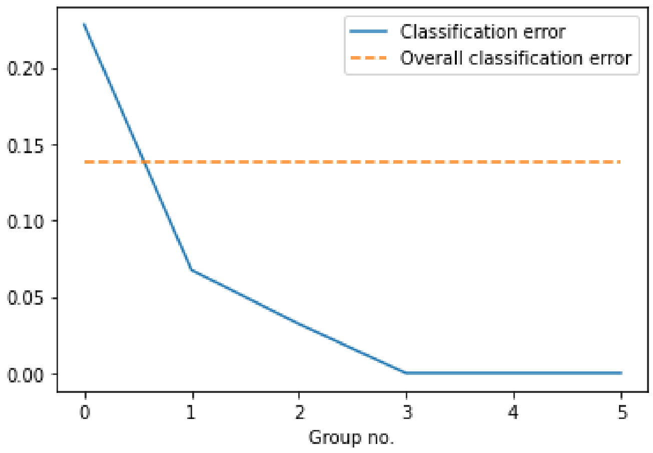


Figure 2: The classification errors of each group. The larger the group number, the greater the absolute values of the scores.

to rank the importance of the features to the decision is to sort them by the size of their contributions to the score. That is, for each feature we compute $|w_i x_i|$, where w_i is the weight of the ith feature in the prediction function, and x_i is the value of the ith feature in the input x. Create a table of the most important features, sorted by $|w_i x_i|$, including the feature name, the feature value x_i, the feature weight w_i, and the product $w_i x_i$. Attempt to explain why the model was incorrect. Can you think of a new feature that might be able to fix the issue? Include a short analysis for at least 2 incorrect examples. Can you think of new features that might help fix a problem? (Think of making groups of words.)

Solution. Let us consider this first example, where the review is negative but misclassified as positive (with score 47.68). The 10 most important features are listed below.

```
     word       |wx|          w   x
0     and  23.146358   1.653311  14
1      he  19.199744   2.399968   8
2    only  19.199744  -6.399915   3
3      to  18.719750  -1.039986  18
4    it's  12.373168   3.093292   4
5   quite  10.879855   5.439927   2
6     bad  10.586526 -10.586526   1
7   movie  10.266530  -1.466647   7
8     the  10.079866   0.239997  42
9  paulie   9.706537   0.693324  14
```

We can see that though there are words like “bad” which have large negative weights and can indicate that the review is negative, there are too many common words like “and” and “he” which are considered as positive. Though these words are commonly assigned only small weights, they appear too often that lead to the misclassification. Now let us consider another example, where the review is positive but misclassified as negative (with score -71.54). The 10 most important features are listed below.

```
            word       |wx|         w   x
0            and  24.799669  1.653311  15
1             to  22.879695 -1.039986  22
2           only  19.199744 -6.399915   3
3           have  19.039746 -2.719964   7
4           even  18.773083 -4.693271   4
5        nothing  18.639751 -6.213250   3
6              i  17.226437  1.013320  17
7         script  13.439821 -4.479940   3
8  unfortunately  11.306516 -5.653258   2
9             as  11.066519  2.213304   5
```

Again, words like “to” and “have” are considered negative and appear too often that lead to the misclassification. One possible fix is to ignore the words that appear too often in both positive and negative reviews (for instance, by assigning them zero weight). Many of the pronouns, articles, etc. clearly fall into this category. In this way, we can minimize (or at least mitigate) the effect of commonly appearing words on the final classification.

Kernel Methods

Kernelization review

Consider the following optimization problem on a data set $(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n) \in \mathbb{R}^d \times \mathcal{Y}$

$$\min_{w \in \mathbb{R}^d} R\left(\sqrt{\langle \boldsymbol{w}, \boldsymbol{w} \rangle}\right) + L\left(\langle \boldsymbol{w}, \boldsymbol{x}_1 \rangle, \ldots, \langle \boldsymbol{w}, \boldsymbol{x}_n \rangle\right),$$

where $\boldsymbol{w}, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_n \in \mathbb{R}^d$, and $\langle \cdot, \cdot \rangle$ is the standard inner product on $\mathbb{R}^d$. The function $R : [0, \infty) \to \mathbb{R}$ is nondecreasing and gives us our regularization term, while $L : \mathbb{R}^n \to \mathbb{R}$ is arbitrary[5] and gives us our loss term. We noted in lecture that this general form includes soft-margin SVM and ridge regression, though not lasso regression. Using the representer theorem, we showed if the optimization problem has a solution, there is always a solution of the form $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$, for some $\boldsymbol{\alpha} \in \mathbb{R}^n$. Plugging this into the our original problem, we get the following “kernelized” optimization problem:

$$\min_{\boldsymbol{\alpha} \in \mathbb{R}^n} R\left(\sqrt{\boldsymbol{\alpha}^\top K \boldsymbol{\alpha}}\right) + L\left(K\boldsymbol{\alpha}\right),$$

where $K \in \mathbb{R}^{n \times n}$ is the Gram matrix (or “kernel matrix”) defined by $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$. Predictions are given by

$$f(x) = \sum_{i=1}^n \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x}),$$

and we can recover the original $\boldsymbol{w} \in \mathbb{R}^d$ by $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i$.

The *kernel trick* is to swap out occurrences of the kernel k (and the corresponding Gram matrix K) with another kernel. For example, we could replace $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle$ by $k'(\boldsymbol{x}_i, \boldsymbol{x}_j) =$

[5]You may be wondering where the y_i's are. They are built into the function L. For example, a square loss on a training set of size 3 could be represented as $L(s_1, s_2, s_3) = \frac{1}{3}\left((s_1 - y_1)^2 + (s_2 - y_2)^2 + (s_3 - y_3)^3\right)$, where each s_i stands for the ith prediction $\langle \boldsymbol{w}, \boldsymbol{x}_i \rangle$.

$\langle \psi(\boldsymbol{x}_i), \psi(\boldsymbol{x}_j)\rangle$ for an arbitrary feature mapping $\boldsymbol{\psi} : \mathbb{R}^d \to \mathbb{R}^d$. In this case, the recovered $\boldsymbol{w} \in \mathbb{R}^d$ would be $\boldsymbol{w} = \sum_{i=1}^n \alpha_i \boldsymbol{\psi}(\boldsymbol{x}_i)$ and predictions would be $\langle \boldsymbol{w}, \boldsymbol{\psi}(\boldsymbol{x}_i)\rangle$.

More interestingly, we can replace k by another kernel $k''(\boldsymbol{x}_i, \boldsymbol{x}_j)$ for which we do not even know or cannot explicitly write down a corresponding feature map $\boldsymbol{\psi}$. Our main example of this is the RBF kernel

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right),$$

for which the corresponding feature map $\boldsymbol{\psi}$ is infinite dimensional. In this case, we cannot recover $\boldsymbol{w}$ since it would be infinite dimensional. Predictions must be done using $\alpha \in \mathbb{R}^n$, with $f(\boldsymbol{x}) = \sum_{i=1}^n \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$.

Your implementation of kernelized methods below should not make any reference to $\boldsymbol{w}$ or to a feature map $\boldsymbol{\psi}$. Your learning routine should return $\boldsymbol{\alpha}$, rather than $\boldsymbol{w}$, and your prediction function should also use $\boldsymbol{\alpha}$ rather than $\boldsymbol{w}$. This will allow us to work with kernels that correspond to infinite-dimensional feature vectors.

Kernel problems

Ridge Regression: Theory

Suppose our input space is $\mathcal{X} = \mathbb{R}^d$ and our output space is $\mathcal{Y} = \mathbb{R}$. Moreover, let us denote $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n)\}$ as a training set from $\mathcal{X} \times \mathcal{Y}$. We will use the "design matrix" $X \in \mathbb{R}^{n\times d}$, which has the input vectors as rows, such that

$$X = \begin{pmatrix} -\boldsymbol{x}_1- \\ \vdots \\ -\boldsymbol{x}_n- \end{pmatrix}.$$

Recall the ridge regression objective function

$$J(\boldsymbol{w}) = ||X\boldsymbol{w} - \boldsymbol{y}||^2 + \lambda||\boldsymbol{w}||^2, \qquad \lambda > 0.$$

15. Show that for $\boldsymbol{w}$ to be a minimizer of $J(\boldsymbol{w})$, we must have $X^\top X\boldsymbol{w} + \lambda I\boldsymbol{w} = X^\top \boldsymbol{y}$. Show that the minimizer of $J(\boldsymbol{w})$ is $\boldsymbol{w} = (X^\top X + \lambda I)^{-1} X^\top \boldsymbol{y}$. Justify that the matrix $X^\top X + \lambda I$ is invertible, for $\lambda > 0$. (You should use properties of positive (semi-)definite matrices. If you need a reminder look up the Appendix.)

    Proof. We can first compute the gradient of the objective function J as

$$\begin{aligned}
\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) &= \nabla_{\boldsymbol{w}}\left((X\boldsymbol{w} - \boldsymbol{y})^\top (X\boldsymbol{w} - \boldsymbol{y}) + \lambda \boldsymbol{w}^\top \boldsymbol{w}\right) \\
&= \nabla_{\boldsymbol{w}}\left(\boldsymbol{w}^\top X^\top X\boldsymbol{w} - \boldsymbol{y}^\top X\boldsymbol{w} - \boldsymbol{w}^\top X^\top \boldsymbol{y} + \boldsymbol{y}^\top \boldsymbol{y} + \lambda \boldsymbol{w}^\top \boldsymbol{w}\right) \\
&= \nabla_{\boldsymbol{w}}\left(\boldsymbol{w}^\top X^\top X\boldsymbol{w} - 2\boldsymbol{y}^\top X\boldsymbol{w} + \boldsymbol{y}^\top \boldsymbol{y} + \lambda \boldsymbol{w}^\top \boldsymbol{w}\right) \\
&= \left(X^\top X + (X^\top X)^\top\right)\boldsymbol{w} - 2\left(\boldsymbol{y}^\top X\right)^\top + 2\lambda \boldsymbol{w} \\
&= 2\left(X^\top X\boldsymbol{w} - X^\top \boldsymbol{y} + \lambda I\boldsymbol{w}\right). \qquad (15)
\end{aligned}$$

    Therefore, in order to minimize $J(\boldsymbol{w})$, we require its gradient to be zero, that is,

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0 \iff X^\top \boldsymbol{y} = X^\top X\boldsymbol{w} + \lambda I\boldsymbol{w}. \qquad (16)$$

Now, note that for any $\boldsymbol{v} \in \mathbb{R}^d \setminus \{\mathbf{0}\}$, we have that

$$\left\langle (X^\top X + \lambda I)\boldsymbol{v}, \boldsymbol{v} \right\rangle = \left\langle X^\top X\boldsymbol{v}, \boldsymbol{v} \right\rangle + \lambda \langle \boldsymbol{v}, \boldsymbol{v} \rangle > \boldsymbol{v}^\top X^\top X\boldsymbol{v} = \langle X\boldsymbol{v}, X\boldsymbol{v} \rangle \geq 0. \tag{17}$$

Therefore, by arbitrariness of $\boldsymbol{v}$, we have that $(X^\top X + \lambda I)\boldsymbol{v} \neq 0$ for all $\boldsymbol{v} \in \mathbb{R}^d \setminus \{\mathbf{0}\}$. Therefore, $X^\top X + \lambda I$ is invertible, and thus we can write that

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}) = 0 \iff X^\top \boldsymbol{y} = (X^\top X + \lambda I)\boldsymbol{w} \iff \boldsymbol{w} = (X^\top X + \lambda I)^{-1} X^\top \boldsymbol{y}, \tag{18}$$

as desired, and the proof is complete. □

16. Rewrite $X^\top X\boldsymbol{w} + \lambda I\boldsymbol{w} = X^\top \boldsymbol{y}$ as $\boldsymbol{w} = \frac{1}{\lambda}(X^\top \boldsymbol{y} - X^\top X\boldsymbol{w})$. Based on this, show that we can write $\boldsymbol{w} = X^\top \boldsymbol{\alpha}$ for some $\boldsymbol{\alpha}$, and give an expression for $\boldsymbol{\alpha}$.

Proof. Clearly, we can write that

$$\boldsymbol{w} = \frac{1}{\lambda}(X^\top \boldsymbol{y} - X^\top X\boldsymbol{w}) = X^\top \left(\frac{1}{\lambda}(\boldsymbol{y} - X\boldsymbol{w}) \right). \tag{19}$$

By taking $\boldsymbol{\alpha} = \frac{1}{\lambda}(\boldsymbol{y} - X\boldsymbol{w})$, we can obtain that $\boldsymbol{w} = X^\top \boldsymbol{\alpha}$, so the proof is complete. □

17. Based on the fact that $\boldsymbol{w} = X^\top \boldsymbol{\alpha}$, explain why we say $\boldsymbol{w}$ is "in the span of the data."

Solution. Since X is the design matrix, we can expand the expression to obtain that

$$\boldsymbol{w} = \begin{pmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{pmatrix} \begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{pmatrix} = \begin{pmatrix} \sum_{i=1}^n x_{i1}\alpha_i \\ \vdots \\ \sum_{i=1}^n x_{id}\alpha_i \end{pmatrix} = \sum_{i=1}^n \alpha_i \begin{pmatrix} x_{i1} \\ \vdots \\ x_{id} \end{pmatrix} = \sum_{i=1}^n \alpha_i \boldsymbol{x}_i, \tag{20}$$

so $\boldsymbol{w}$ can be expressed by some linear combination of $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$, and in order words, we say that $\boldsymbol{w} \in \operatorname{span}(\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n)$.

18. Show that $\boldsymbol{\alpha} = (\lambda I + XX^\top)^{-1}\boldsymbol{y}$. Note that $XX^\top$ is the kernel matrix for the standard vector dot product.
Hint: Replace $\boldsymbol{w}$ by $X^\top \boldsymbol{\alpha}$ in the expression for $\boldsymbol{\alpha}$, and then solve for $\boldsymbol{\alpha}$.

Proof. Plugging in the expression $\boldsymbol{w} = X^\top \boldsymbol{\alpha}$, we can rewrite (16) as

$$X^\top \boldsymbol{y} = X^\top XX^\top \boldsymbol{\alpha} + \lambda X^\top \boldsymbol{\alpha} \implies X^\top \boldsymbol{y} = X^\top (XX^\top + \lambda I)\boldsymbol{\alpha}. \tag{21}$$

Since $X^\top$ is of size $d \times n$ of full rank and with $d \gg n$, we can deduce that

$$\boldsymbol{y} = (XX^\top + \lambda I)\boldsymbol{\alpha} \implies \boldsymbol{\alpha} = (\lambda I + XX^\top)^{-1}\boldsymbol{y}, \tag{22}$$

as desired, and the proof is complete. □

19. Give a kernelized expression for the $X\boldsymbol{w}$, the predicted values on the training points.
Hint: Replace $\boldsymbol{w}$ by $X^\top \boldsymbol{\alpha}$ and $\boldsymbol{\alpha}$ by its expression in terms of the kernel matrix $XX^\top$.

Solution. By previous results, we can write that

$$X\boldsymbol{w} = XX^\top\boldsymbol{\alpha} = XX^\top(\lambda I + XX^\top)^{-1}\boldsymbol{y}. \tag{23}$$

This is indeed a kernelized expression, since all information of the training input $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ is included in $XX^\top$, which is in fact the Gram matrix in this case.

20. Give an expression for the prediction $f(\boldsymbol{x}) = \boldsymbol{x}^\top \boldsymbol{w}^*$ for a new point $\boldsymbol{x}$, not in the training set. The expression should only involve $\boldsymbol{x}$ via inner products with other $\boldsymbol{x}$'s.

    Hint: It is often convenient to define the column vector

$$k_{\boldsymbol{x}} = \begin{pmatrix} \boldsymbol{x}^\top \boldsymbol{x}_1 \\ \vdots \\ \boldsymbol{x}^\top \boldsymbol{x}_n \end{pmatrix}$$

    to simplify the expression.

    Solution. By previous results, we can write that

$$\begin{aligned} f(\boldsymbol{x}) = \boldsymbol{x}^\top X^\top \boldsymbol{\alpha}^* &= \begin{pmatrix} x_1 & \cdots & x_d \end{pmatrix} \begin{pmatrix} | & & | \\ \boldsymbol{x}_1 & \cdots & \boldsymbol{x}_n \\ | & & | \end{pmatrix} \boldsymbol{\alpha}^* \\ &= \begin{pmatrix} \sum_{i=1}^d x_i x_{1i} & \cdots & \sum_{i=1}^d x_i x_{di} \end{pmatrix} \boldsymbol{\alpha}^* = k_{\boldsymbol{x}}^\top \boldsymbol{\alpha}^*. \end{aligned} \tag{24}$$

Kernels and Kernel Machines

There are many different families of kernels. So far we have talked about linear kernels, RBF (Gaussian) kernels, and polynomial kernels. The last two kernel types have parameters. In this section, we will implement these kernels in a way that will be convenient for implementing our kernelized ridge regression later on. For simplicity, we will assume that our input space is $\mathcal{X} = \mathbb{R}$. This allows us to represent a collection of n inputs in a matrix $X \in \mathbb{R}^{n\times 1}$. You should now refer to the jupyter notebook `skeleton_code_kernels.ipynb`.

21. Write functions that compute the RBF kernel

$$k_{\mathrm{RBF}(\sigma)}(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\frac{\|\boldsymbol{x} - \boldsymbol{x}'\|^2}{2\sigma^2}\right),$$

    as well as the polynomial kernel

$$k_{\mathrm{poly}(a,d)}(\boldsymbol{x}, \boldsymbol{x}') = (a + \langle \boldsymbol{x}, \boldsymbol{x}' \rangle)^d.$$

    The linear kernel $k_{\mathrm{linear}}(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle$ has been done for you in the support code. Your functions should take as input two matrices $W \in \mathbb{R}^{n_1\times d}$ and $X \in \mathbb{R}^{n_2\times d}$ and should return a matrix $M \in \mathbb{R}^{n_1\times n_2}$ where $M_{ij} = k(W_{i\cdot}, X_{j\cdot})$. In words, the (i,j)th entry of M should be kernel evaluation between $\boldsymbol{w}_i$ (the ith row of W) and $\boldsymbol{x}_j$ (the jth row of X). For the RBF kernel, you may use the scipy function `cdist(X1, X2, ‘sqeuclidean’)` in the package `scipy.spatial.distance`.

    Solution. The function `RBF_kernel` can be implemented as follows:

```
def RBF_kernel(X1, X2, sigma):
    """Compute the RBF kernel between two sets of vectors.

    Parameters
    ----------
    X1 : np.ndarray
        A matrix of size n1 * d, with vectors x1_1, ..., x1_n1 as its rows.
    X2 : np.ndarray
        A matrix of size n2 * d, with vectors x2_1, ..., x2_n2 as its rows.
    sigma: float
        The bandwidth (i.e., standard deviation) for the RBF kernel.

    Returns
    -------
    K : np.ndarray
        A matrix of size n1 * n2, with the K_ij = k_RBF(x1_i, x2_j).
    """
    return np.exp(-np.square(dist.cdist(X1, X2, "sqeuclidean"))
        / (2 * sigma ** 2))
```

The function `polynomial_kernel` can be implemented as follows:

```
def polynomial_kernel(X1, X2, offset, degree):
    """Compute the inhomogeneous polynomial kernel between two sets of
    vectors.

    Parameters
    ----------
    X1 : np.ndarray
        A matrix of size n1 * d, with vectors x1_1, ..., x1_n1 as its rows.
    X2 : np.ndarray
        A matrix of size n2 * d, with vectors x2_1, ..., x2_n2 as its rows.
    offset : float
        The paremeter a as in k_POLY(x, x') = (a + <x,x'>)^d.
    degree : float
        The paremeter d as in k_POLY(x, x') = (a + <x,x'>)^d.

    Returns
    -------
    K : np.ndarray
        A matrix of size n1 * n2, with K_ij = k_POLY(x1_i, x2_j).
    """
    return (offset + np.dot(X1, X2.T)) ** degree
```

22. Use the linear kernel function defined in the code to compute the kernel matrix on the set of points $x_0 \in \mathcal{D}_X = \{-4, -1, 0, 2\}$. Include both the code and the output.

Solution. The code for computing the kernel matrix on $\mathcal{D}_X$ is as follows.

```
X = np.array([-4, -1, 0, 2]).reshape(-1, 1)  # Construct D_X
K = linear_kernel(X, X)  # Compute the kernel matrix
```

The output, *i.e.*, the kernel matrix `K`, is then

```
[[16  4  0 -8]
 [ 4  1  0 -2]
 [ 0  0  0  0]
 [-8 -2  0  4]]
```

23. Suppose we have the data set $\mathcal{D}_{X,y} = \{(-4,2),(-1,0),(0,3),(2,5)\}$ (in each set of parentheses, the first number is the value of x_i and the second number the corresponding value of the target y_i). Then by the representer theorem, the final prediction function will be in the span of the functions $x \mapsto k(x_0, x)$ for $x_0 \in \mathcal{D}_X = \{-4,-1,0,2\}$. This set of functions will look quite different depending on the kernel function we use. The set of functions $x \mapsto k_{\text{linear}}(x_0, x)$ for $x_0 \in \mathcal{X}$ and for $x \in [-6,6]$ has been provided for the linear kernel.

    (a) Plot the set of functions $x \mapsto k_{\text{poly}(1,3)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and for $x \in [-6,6]$.

    Solution. The plot is shown as in Figure 3.

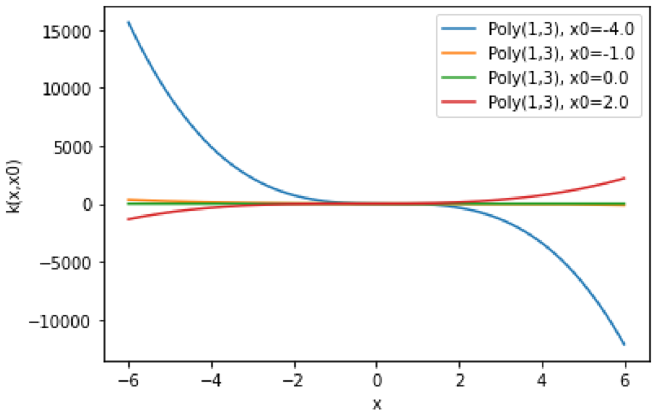


Figure 3: The plot of the set of functions $x \mapsto k_{\text{poly}(1,3)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and $x \in [-6,6]$.

    (b) Plot the set of functions $x \mapsto k_{\text{RBF}(1)}(x_0, x)$ for $x_0 \in \mathcal{X}$ and for $x \in [-6,6]$.

    Solution. The plot is shown as in Figure 4.

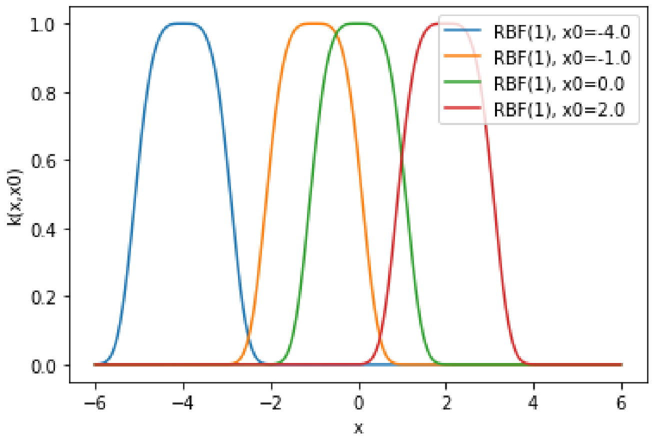


Figure 4: The plot of the set of functions $x \mapsto k_{\text{RBF}(1)}(x_0, x)$ for $x_0 \in \mathcal{D}_X$ and $x \in [-6,6]$.

Note that the values of the parameters of the kernels you should use are given in their definitions in (a) and (b).

24. By the representer theorem, the final prediction function will be of the form $f(x) = \sum_{i=1}^{n} \alpha_i k(x_i, x)$, where $x_1, \cdots, x_n \in \mathcal{X}$ are the inputs in the training set. We will use the class `Kernel_Machine` in the skeleton code to make prediction with different kernels. Complete the `predict` function of the class `Kernel_Machine`. Construct a `Kernel_Machine` object with the RBF kernel ($\sigma = 1$), with prototype points at $-1, 0, 1$ and corresponding weights $\alpha_i = 1, -1, 1$. Plot the resulting function.

Solution. The method `predict` of the class `Kernel_Machine` can be implemented as follows.

```python
class Kernel_Machine:
    def predict(self, X):
        """Evaluate the kernel machine on the points given by the rows of X.

        Parameters
        ----------
        X : np.ndarray
            A matrix with inputs x_1, ..., x_n as its rows, of size n * d.

        Returns
        -------
        prediction : np.array
            The vector of kernel machine evaluations on the n points in X.
        """
        return np.array([np.dot(self.kernel(self.training_points, X)[:, i],
            self.weights) for i in range(X.shape[0])])
```

Initializing the kernel machine using `functools.partial(RBF_kernel, sigma=1)`, prototype points at $-1, 0, 1$, and the corresponding weights $\alpha_i = 1, -1, 1$, the plot of the prediction function $f(x)$ is shown as in Figure 5.

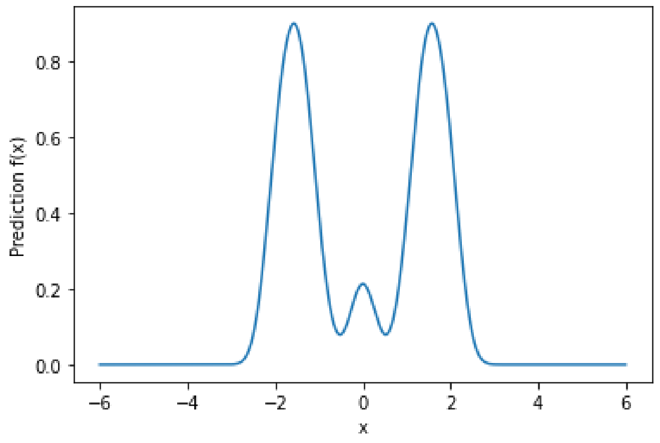


Figure 5: The plot of the prediction function $f(x)$ with respect to x, using the RBF kernel with parameter $\sigma = 1$, prototype points at $-1, 0, 1$, and with corresponding weights $\alpha_i = 1, -1, 1$.

Note: For this last problem, and for other problems below, it may be helpful to use partial application on your kernel functions. For example, if your polynomial kernel function has signature `polynomial_kernel(W, X, offset, degree)`, you can write `k = functools.partial(polynomial_kernel, offset=2, degree=2)`, and then a call to `k(W, X)` is equivalent to `polynomial_kernel(W, X, offset=2, degree=2)`, the advantage being that the extra parameter settings are built into `k(W, X)`. This can be convenient so that you can have a function that just takes a kernel function `k(W, X)` and doesn't have to worry about the parameter settings for the kernel.

Kernel Ridge Regression: Practice

In the `.zip` file for this assignment, we have provided a training set in `krr-train.txt` and a test set in `krr-test.txt` for a one-dimensional regression problem, in which $\mathcal{X} = \mathcal{Y} = \mathcal{A} = \mathbb{R}$. Fitting this data using kernelized ridge regression, we will compare the results using several different kernel functions. Because the input space is one-dimensional, we can easily visualize the results.

25. Plot the training data. You should note that while there is a clear relationship between x and y, the relationship is not linear.

Solution. The plot of the training data is shown as in Figure 6.

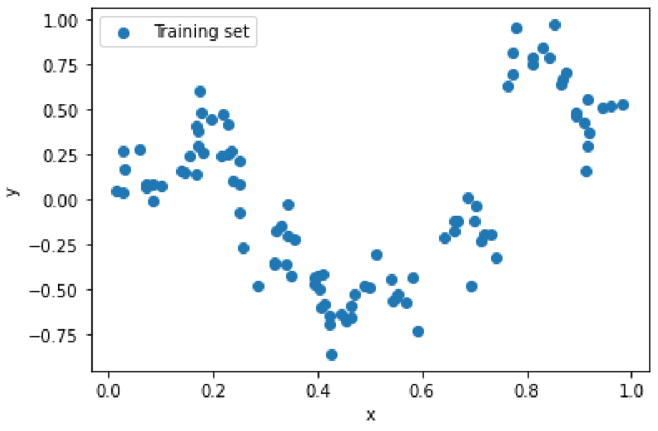


Figure 6: The plot of the data points in the training set.

26. In a previous problem, we showed that in kernelized ridge regression, the final prediction function is $f(\boldsymbol{x}) = \sum_{i=1}^{n} \alpha_i k(\boldsymbol{x}_i, \boldsymbol{x})$, where $\boldsymbol{\alpha} = (\lambda I + K)^{-1}\boldsymbol{y}$ and $K \in \mathbb{R}^{n\times n}$ is the kernel matrix of the training data, such that $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$, for any $i, j \in \{1, \cdots, n\}$. In terms of kernel machines, α_i is the weight on the kernel function evaluated at the training point $\boldsymbol{x}_i$. Complete the function `train_kernel_ridge_regression` so that it performs kernel ridge regression and returns a `Kernel_Machine` object that can be used for predicting on new points.

Solution. The function `train_kernel_ridge_regression` can be implemented as follows.

```python
def train_kernel_ridge_regression(X, y, kernel, l2reg):
    """Return the kernel machine with optimal weight under ridge regression.

    Parameters
    ----------
    X : np.ndarray
        The input of the training set.
    y : np.array
        The output of the training set.
    kernel : function
        The kernel function to use.
    l2reg : float
        The l2 regularization coefficient.

    Returns
    -------
    kernel_machine : Kernel_Machine
        The kernel machine with optimal weights under ridge regression.
    """
    return Kernel_Machine(kernel, X, np.dot(np.linalg.inv(l2reg *
        np.identity(X.shape[0]) + kernel(X, X)), y))
```

27. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed regularization parameter of 0.0001 for 3 different values of sigma: 0.01, 0.1, and 1.0. What values of sigma do you think would be more likely to overfit, and which less?

    Solution. The plot of the fits with the RBF kernel and regularization parameter $\lambda = 10^{-4}$ is shown as in Figure 7. We use different bandwidths $\sigma = 10^{-2}, 10^{-1}, 1.0$ for the RBF kernel. From the plot, we can see that smaller values of σ are more likely to overfit, while larger values of σ are less likely to overfit.

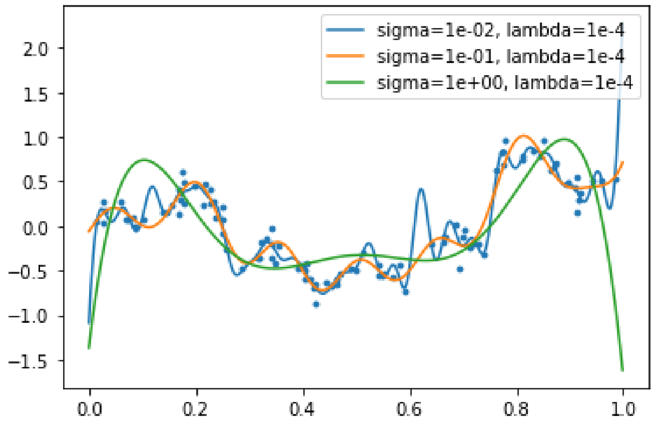


Figure 7: The plot of the data points in the training set, as well as the prediction curves made with the RBF kernel and regularization parameter $\lambda = 10^{-4}$, but with different bandwidths (standard deviations) $\sigma = 10^{-2}, 10^{-1}, 1.0$.

28. Use the code provided to plot your fits to the training data for the RBF kernel with a fixed sigma of 0.02 and 4 different values of the regularization parameter λ: 0.0001, 0.01, 0.1, and 2.0. What happens to the prediction function as $\lambda \to \infty$?

Solution. The plot of the fits with the RBF kernel and bandwidth $\sigma = 0.02$ is shown as in Figure 8. We use different regularization parameters $\lambda = 10^{-4}, 10^{-2}, 10^{-1}, 2.0$ for the RBF kernel. If $\lambda \to \infty$, then the prediction function will become constant zero.

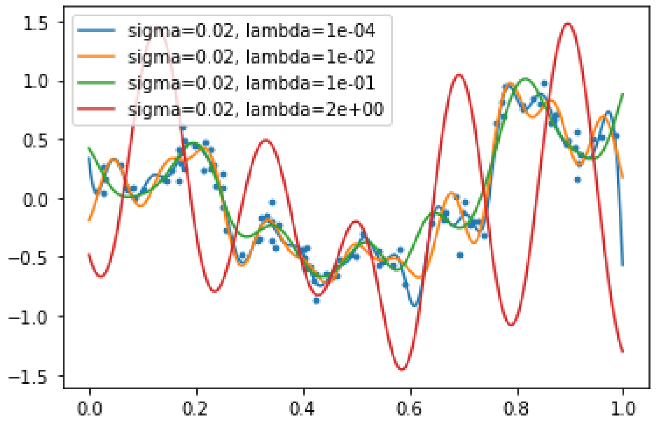


Figure 8: The plot of the data points in the training set, as well as the prediction curves made with the RBF kernel and bandwidth (standard deviation) $\sigma = 0.02$, but with different regularization parameters $\lambda = 10^{-4}, 10^{-2}, 10^{-1}, 2.0$.

29. (Optional) Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use average square loss on the test set to rank the parameter settings. To make things easier for you, we have provided an sklearn wrapper for the kernel ridge regression function we have created so that you can use sklearn's `GridSearchCV`.

    Note: Because of the small dataset size, these models can be fit extremely fast, so there is no excuse for not doing extensive hyperparameter tuning.

    Solution. For the linear kernel, we first try regularization parameters $\lambda \in [10^{-4}, 10]$, and then zoom in to the range $\lambda \in [1, 6]$. We take 50 different values of λ in this range and figure out that the best regularization parameter form the linear kernel is approximately $\lambda = 3.9$. The ten best fitting results are shown as follows.

```
Lambda Kernel Sigma Offset Degree    Testing MSE   Training MSE
   3.9 Linear     -      -      - 1.64509507e-01 2.06560128e-01
   4.0 Linear     -      -      - 1.64509534e-01 2.06562831e-01
   3.8 Linear     -      -      - 1.64509541e-01 2.06557469e-01
   4.1 Linear     -      -      - 1.64509621e-01 2.06565579e-01
   3.7 Linear     -      -      - 1.64509638e-01 2.06554856e-01
   4.2 Linear     -      -      - 1.64509766e-01 2.06568371e-01
   3.6 Linear     -      -      - 1.64509798e-01 2.06552291e-01
   4.3 Linear     -      -      - 1.64509970e-01 2.06571204e-01
```

```
3.5 Linear      -      -      - 1.64510022e-01 2.06549773e-01
4.4 Linear      -      -      - 1.64510230e-01 2.06574080e-01
```

For the RBF kernel, we first try regularization parameters $\lambda \in [10^{-4}, 10]$ and bandwidths $\sigma \in [10^{-4}, 10]$, and then zoom in to the range $\lambda \in [10^{-3}, 3 \times 10^{-3}]$ and $\sigma \in [0.05, 0.1]$. We take 20 different values of each of λ and σ in their ranges respectively, and figure out that the best pair of regularization parameter and bandwidth is approximately $\lambda = 2.1 \times 10^{-3}$ and $\sigma = 0.07$. The ten best fitting results are shown as follows.

```
  Lambda Kernel     Sigma Offset Degree    Testing MSE   Training MSE
0.002158    RBF  0.071053      -      - 1.38077013e-02 1.54797997e-02
0.002263    RBF  0.071053      -      - 1.38101878e-02 1.54594393e-02
0.002053    RBF  0.071053      -      - 1.38192319e-02 1.55253417e-02
0.002368    RBF  0.071053      -      - 1.38220784e-02 1.54563383e-02
0.002474    RBF  0.071053      -      - 1.38406186e-02 1.54655235e-02
0.001947    RBF  0.071053      -      - 1.38530111e-02 1.56094186e-02
0.002579    RBF  0.071053      -      - 1.38640722e-02 1.54837262e-02
0.002684    RBF  0.071053      -      - 1.38912918e-02 1.55087138e-02
0.002789    RBF  0.071053      -      - 1.39214866e-02 1.55389112e-02
0.001842    RBF  0.071053      -      - 1.39249049e-02 1.57561939e-02
```

The polynomial kernel is much trickier since there are three parameters, so my conclusion may not be accurate. We first try regularization parameters $\lambda \in [10^{-5}, 10]$, degrees $d \in [2, 50]$, and offsets $a \in [-100, 100]$, and then zoom in to the range $\lambda \in [10^{-5}, 10^{-1}]$, degrees $d \in [2, 10]$, and offsets $a \in [-50, 50]$. By some more experiments, we find that $d = 6$ seems to be the best degree parameter, and some plots with $d = 6$ are shown as in Figure 9.

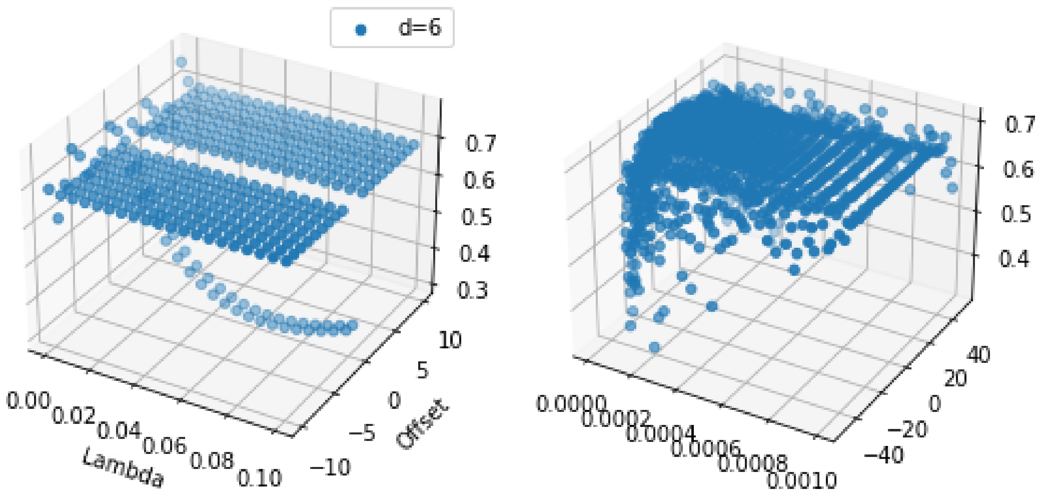


Figure 9: Plots of the polynomial kernel with degree parameter $d = 6$.

Unfortunately, these plots are not very informative, so we perform more experiments with more sets of parameters, and find that the best set of parameters is approximately $\lambda = 10^{-5}$

and $a = -1$, and $d = 6$. Note that these may not be the best sets of parameters, but the best that I can find. The ten best fitting results are shown as follows.

```
  Lambda Kernel Sigma    Offset Degree    Testing MSE   Training MSE
0.000010    Poly     - -1.020408      6 3.22318336e-02 3.93336172e-02
0.000011    Poly     - -1.020408      6 3.23056778e-02 3.93418390e-02
0.000012    Poly     - -1.020408      6 3.23855317e-02 3.93512605e-02
0.000013    Poly     - -1.020408      6 3.24717327e-02 3.93620134e-02
0.000015    Poly     - -1.020408      6 3.25646074e-02 3.93742335e-02
0.000016    Poly     - -1.020408      6 3.26644656e-02 3.93880588e-02
0.000018    Poly     - -1.020408      6 3.27715938e-02 3.94036268e-02
0.000019    Poly     - -1.020408      6 3.28862479e-02 3.94210713e-02
0.000021    Poly     - -1.020408      6 3.30086471e-02 3.94405186e-02
0.000023    Poly     - -1.020408      6 3.31389669e-02 3.94620842e-02
```

30. (Optional) Plot your best fitting prediction functions using the polynomial kernel and the RBF kernel. Use the domain $x \in (-0.5, 1.5)$. Comment on the results.

Solution. The best fitting prediction functions are shown as in Figure 10.

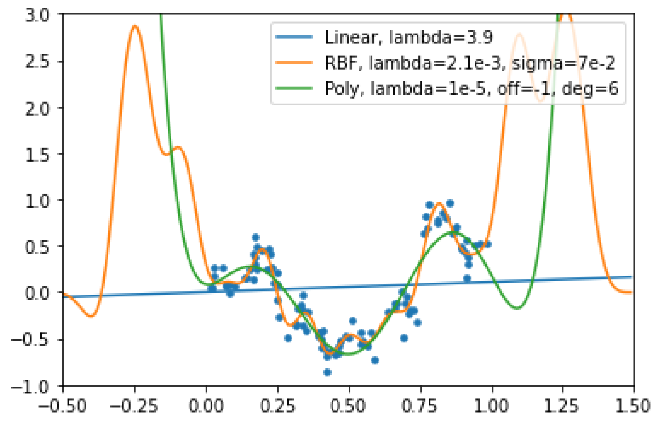


Figure 10: The data points in the training set as well as the fitting curves using linear, RBF, and polynomial kernels, respectively with their best set of parameters as previously determined.

31. (Optional) The data for this problem was generated as follows: A function $f : \mathbb{R} \to \mathbb{R}$ was chosen. Then to generate a point (x, y), we sampled x uniformly from $(0, 1)$ and we sampled $\epsilon \sim \mathcal{N}\left(0, 0.1^2\right)$ (so $\operatorname{Var}[\epsilon] = 0.1^2$). The final point is $(x, f(x) + \epsilon)$. What is the Bayes decision function and the Bayes risk for the loss function $\ell(\hat{y}, y) = (\hat{y} - y)^2$.

Solution. Recall that in this case, a Bayes prediction function $g^* : \mathbb{R} \to \mathbb{R}$ is a function that achieves the minimal risk among all possible functions $g : \mathbb{R} \to \mathbb{R}$. In order to find the Bayes prediction function, we first take an arbitrary function $g : \mathbb{R} \to \mathbb{R}$. For the sake of simplicity, let us denote $\phi = g - f$, then its risk can be computed as

$$\begin{aligned}
R(g) &= \mathbb{E}_{(x,y)\sim P'_{\mathcal{X}\times\mathcal{Y}}}[\ell(g(x),y)] = \mathbb{E}_{x\sim \text{Unif}([0,1])}[\ell(g(x),f(x)+\epsilon)] \\
&= \mathbb{E}_{x\sim \text{Unif}([0,1])}[(g(x)-f(x)-\epsilon)^2] = \mathbb{E}_{x\sim \text{Unif}([0,1])}[\phi^2(x) - 2\epsilon\phi(x) + \epsilon^2] \\
&= \mathbb{E}_{x\sim \text{Unif}([0,1])}[\phi^2(x)] - 2\mathbb{E}_{x\sim \text{Unif}([0,1])}[\epsilon\phi(x)] + \mathbb{E}[\epsilon^2] \\
&= \mathbb{E}_{x\sim \text{Unif}([0,1])}[\phi^2(x)] - 2\mathbb{E}[\epsilon]\cdot\mathbb{E}_{x\sim \text{Unif}([0,1])}[\phi(x)] + \text{Var}[\epsilon] + \mathbb{E}^2[\epsilon] \qquad (25) \\
&= \mathbb{E}_{x\sim \text{Unif}([0,1])}[\phi^2(x)] + 0.1^2 \geq 0.01, \qquad (26)
\end{aligned}$$

where the equality can be obtained when $\phi \equiv 0$, i.e., $g \equiv f$. By the arbitrariness of g, we have found a Bayes prediction function $g^* = f$, with Bayes risk of 0.01.

Kernel SVMs with Kernelized Pegasos (Optional)

32. (Optional) Load the SVM training data in `svm-train.txt` and test data in `svm-test.txt` from the zip file. Plot the training data using the code supplied. Are the data linearly separable? Quadratically separable? What if we used some RBF kernel?

    Solution. The data points in the training set are shown as in Figure 11, with the red plus marks showing data points with +1 labels and blue minus marks showing data points with −1 labels. From the plot, we can clearly see that the data set is neither linearly nor quadratically separable, since the data points with +1 labels are surrounding those with −1 labels. However, it is possible that using the RBF kernel we can reasonably separate the data set, since it is infinitely differentiable (i.e., it is smooth).

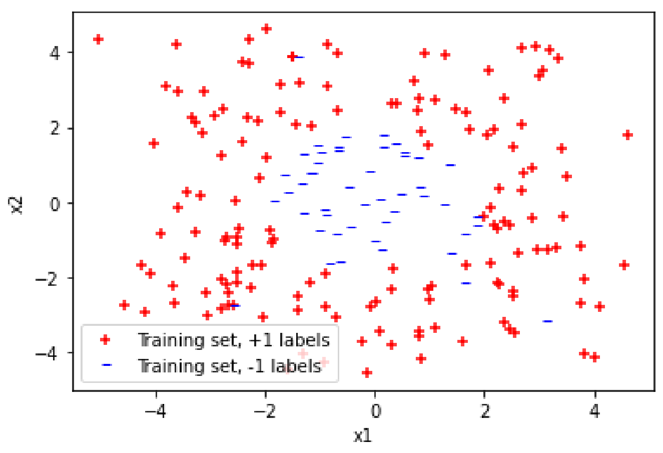


Figure 11: The data points in the training set, with red plus marks showing those with +1 labels and blue minus marks showing those with −1 labels.

33. (Optional) Unlike for kernel ridge regression, there is no closed-form solution for SVM classification (kernelized or not). Implement kernelized Pegasos. Because we are not using a sparse representation for this data, you will probably not see much gain by implementing the "optimized" versions described in the problems above.

Solution. I am not sure what the term "kernelized Pegasos" means. If we want to make use of the Gram matrix, then we will need the dual problem right? So I implemented the following `KernelSVM` class to solve the dual problem. Note that I take the regularization parameter $\lambda = 1$ as in class, since the training process is slow and it is impossible for me to tune so many free parameters.

```python
class KernelSVM:
    def __init__(self, kernel):
        """Create a KernelSVM object.

        Parameters
        ----------
        kernel : function
            The kernel function that returns a cross-kernel matrix.
        """
        self.kernel = kernel
        self._prototypes = None
        self._alphy = None

    def fit(self, X, y, verbose=0):
        """Fit the KernelSVM model to the dataset.

        Parameters
        ----------
        X : np.ndarray
            The feature lists of the data points in the dataset.
        y : np.array
            The labels of the data points in the dataset.
        verbose : int
            The verbose level.

        Raises
        ------
        Exception
            y must be 1-dimensional array, or _gradient() will suspend.
        """
        if y.ndim != 1:
            raise Exception("Error: y must be 1-dimensional array")
        self._gram(X)
        self._alpha_hist, self._obj_hist =
            self._batch_gradient_descent(X, y, verbose)
        self._prototypes, self._alphy = X, self._alpha_hist[-1] * y

    def predict(self, X):
        """Predict for a set of data points.

        Parameters
        ----------
        X : np.ndarray
            The set of data points to predict, with each row a data point.

        Returns
        -------
        pred : np.array
            The array of prediction results for the data points.

        Raises
        ------
        Exception
            Model must be trained via fit() before making predictions.
        """
        if self._alphy is None:
            raise Exception("Error: model has not been trained")
        self._smooth_pred = np.array(
            [np.dot(self.kernel(self._prototypes, X)[:, i], self._alphy)
            for i in range(X.shape[0])])
        pred = []
        for i in range(len(self._smooth_pred)):
            if self._smooth_pred[i] >= 0:
                pred.append(1)
            else:
                pred.append(-1)
        return np.array(pred)

    def _gram(self, X):
        """Build the Gram matrix."""
        self._K = self.kernel(X, X)

    def _objective(self, X, y, alpha):
        """Get the value of the objective function at alpha."""
        alphy =  alpha * y
        return np.dot(np.dot(alphy.T, self._K), alphy) / 2 - np.sum(alpha)

    def _gradient(self, X, y, alpha, epsilon=0.01):
        """Approximate the gradient of the objective function at alpha."""
        num_features = X.shape[0]
        approx_grad = np.zeros(num_features)
        for i in range(num_features):
            direction = np.zeros(num_features)
            direction[i] = epsilon
            approx_grad[i] = (self._objective(X, y, alpha + direction) -
                self._objective(X, y, alpha - direction)) / (2 * epsilon)
        return approx_grad

    def _batch_gradient_descent(self, X, y, verbose, learning_rate=0.01,
                                num_steps=1000):
        """Batch gradient descent algorithm to minimize the objective."""
        num_featues = X.shape[0]
        alpha_hist = np.zeros((num_steps + 1, num_featues))
        obj_hist = np.zeros(num_steps + 1)
        alpha = np.zeros(num_featues)
        for step in range(num_steps + 1):
            if step % 100 == 0 and verbose >= 1:
                print("Gradient Descent: Step {}".format(step))
            alpha_hist[step] = alpha
            obj_hist[step] = self._objective(X, y, alpha)
            alpha -= learning_rate * self._gradient(X, y, alpha)
        return alpha_hist, obj_hist
```

34. (Optional) Find the best hyperparameter settings (including kernel parameters and the regularization parameter) for each of the kernel types. Summarize your results in a table, which gives training error and test error (i.e. average 0-1 loss) for each setting. Include in your table the best settings for each kernel type, as well as nearby settings that show that making small change in any one of the hyperparameters in either direction will cause the performance to get worse. You should use the 0-1 loss on the test set to rank the parameter settings.

Solution. In the previous question (as well as the implementation), I have fixed the regularization parameter $\lambda = 1$ so that there are fewer free parameters to tune. I am sorry about that but my implementation runs too slow so I have to do so. In this case, if we use the linear kernel, there will be no free parameters, and the average 0-1 loss is approximately 49.75%. As for the polynomial kernel, we try degrees $d \in [2, 20]$ and offsets $a \in [-50, 50]$, but the 0-1 loss is always 22.00%. As for the RBF kernel, we start with bandwidths $\sigma \in [10^{-4}, 10]$, and then zoom in to $\sigma \in [10^{-3}, 10^{-2}]$. Some of the results are shown as follows.

```
   Sigma   0-1 Err
7.35e-03    6.750%
7.25e-03    6.875%
7.00e-03    7.000%
7.45e-03    7.000%
7.50e-03    7.000%
6.00e-03    7.125%
8.00e-03    7.250%
5.00e-03    7.500%
1.00e-02    7.625%
```

Therefore, we can see that approximately $\sigma = 7.35 \times 10^{-3}$ is the best bandwidth for the RBF kernel given the regularization parameter $\lambda = 1$.

35. (Optional) Plot your best fitting prediction functions using the linear, polynomial, and the RBF kernel. The code provided may help.

Solution. The plots are shown as in Figure 12, Figure 13, and Figure 14. In each of these plots, plus markers represent the data points with +1 labels and minus markers represent those with −1 labels. Moreover, red data points are predicted to have +1 labels while blue data points are predicted to have −1 labels.

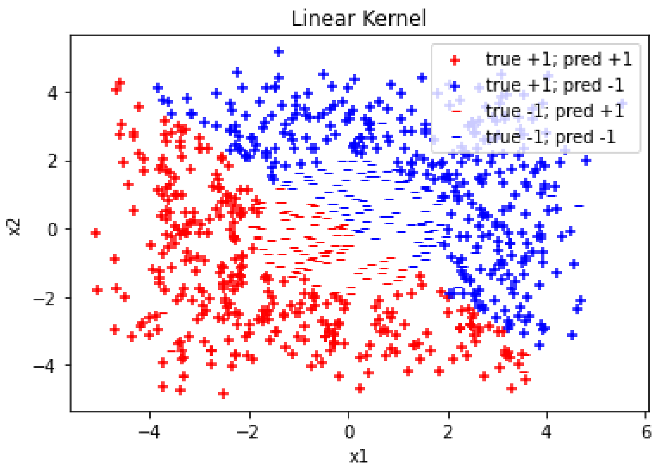


Figure 12: SVM with the linear kernel. Plus markers represent the data points with +1 labels and minus markers represent those with −1 labels. Red data points are predicted to have +1 labels while blue data points are predicted to have −1 labels.

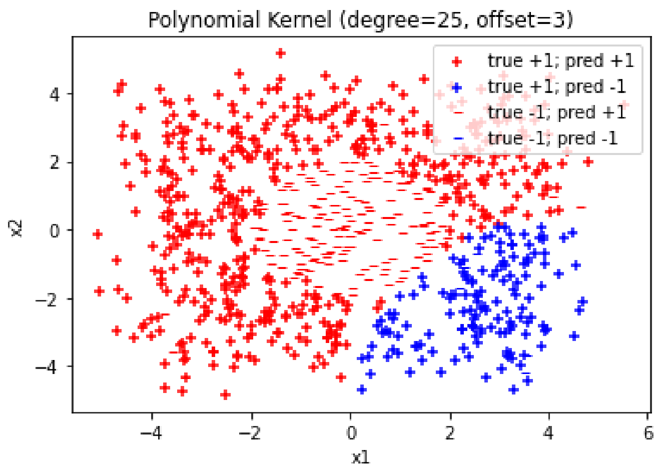


Figure 13: SVM with the polynomial kernel with degree $d = 25$ and offset $a = 3$. Plus markers represent the data points with $+1$ labels and minus markers represent those with -1 labels. Red data points are predicted to have $+1$ labels while blue data points are predicted to have -1 labels.

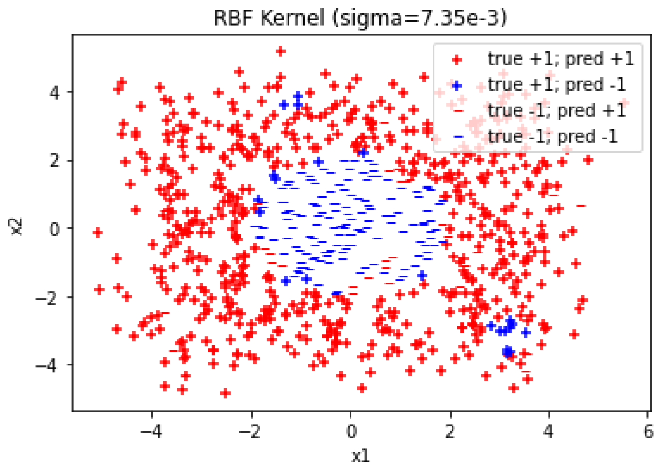


Figure 14: SVM with the RBF kernel with bandwidth $\sigma = 7.35 \times 10^{-3}$. Plus markers represent the data points with $+1$ labels and minus markers represent those with -1 labels. Red data points are predicted to have $+1$ labels while blue data points are predicted to have -1 labels.

Given the marker and color representations, we know that the plus markers are expected to be colored red, and the minus markers are expected to be colored blue. Clearly, the linear kernel does not do well since it can only make linear cuts. The polynomial kernel is not doing well either in my case, but probably since I failed to tune the best set of parameters. The RBF kernel is doing very well, making only a few wrong predictions.

Appendix

Here we are recalling important properties of positive (semi-)definite matrices. The exercises below are for revisions for student who may not feel comfortable with these notions. **None of the appendix is for credit.**

Positive Semidefinite Matrices

In statistics and machine learning, we use positive semidefinite matrices a lot. Let's recall some definitions from linear algebra that will be useful here.

Definition 1. A set of vectors $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n\}$ is **orthonormal** if $\langle \boldsymbol{x}_i, \boldsymbol{x}_i \rangle = 1$ for all $i = 1, \cdots, n$ (i.e. $\boldsymbol{x}_i$ has unit norm), and furthermore, we have for all $i \neq j$ that $\langle \boldsymbol{x}_i, \boldsymbol{x}_j \rangle = 0$ (i.e. $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are orthogonal).

Note that if the vectors are column vectors in a Euclidean space, we can write this as $\boldsymbol{x}_i^\top \boldsymbol{x}_j = \mathbb{1}_{i \neq j}$ for all $i, j \in \{1, \ldots, n\}$.

Definition 2. A matrix is **orthogonal** if it is a square matrix with orthonormal columns.

It follows from the definition that if a matrix $M \in \mathbb{R}^{n \times n}$ is orthogonal, then $M^\top M = I$, where I is the $n \times n$ identity matrix. Thus $M^\top = M^{-1}$, and so $MM^\top = I$ as well.

Definition 3. A matrix M is **symmetric** if $M = M^\top$.

Definition 4. For a square matrix M, if $M\boldsymbol{v} = \lambda v$ for some column vector $\boldsymbol{v}$ and scalar λ, then $\boldsymbol{v}$ is called an **eigenvector** of M and λ is its corresponding **eigenvalue**.

Theorem 1 (Spectral Theorem)**.** *A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ can be diagonalized as $M = Q\Sigma Q^\top$, where $Q \in \mathbb{R}^{n \times n}$ is an orthogonal matrix whose columns are a set of orthonormal eigenvectors of M, and Σ is a diagonal matrix of the corresponding eigenvalues.*

Definition 5. A real, symmetric matrix $M \in \mathbb{R}^{n \times n}$ is **positive semidefinite** if for any $\boldsymbol{x} \in \mathbb{R}^n$,

$$\boldsymbol{x}^\top M \boldsymbol{x} \geq 0.$$

Note that unless otherwise specified, when a matrix is described as positive semidefinite, we are implicitly assuming it is real and symmetric (or complex and Hermitian in certain contexts, though not here). As an exercise in matrix multiplication, note that for any matrix A with columns $\boldsymbol{a}_1, \cdots, \boldsymbol{a}_d$, that is,

$$A = \begin{pmatrix} | & & | \\ \boldsymbol{a}_1 & \cdots & \boldsymbol{a}_d \\ | & & | \end{pmatrix} \in \mathbb{R}^{n \times d},$$

we have that

$$A^\top M A = \begin{pmatrix} \boldsymbol{a}_1^\top M \boldsymbol{a}_1 & \boldsymbol{a}_1^\top M \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_1^\top M \boldsymbol{a}_d \\ \boldsymbol{a}_2^\top M \boldsymbol{a}_1 & \boldsymbol{a}_2^\top M \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_2^\top M \boldsymbol{a}_d \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{a}_d^\top M \boldsymbol{a}_1 & \boldsymbol{a}_d^\top M \boldsymbol{a}_2 & \cdots & \boldsymbol{a}_d^\top M \boldsymbol{a}_d \end{pmatrix}.$$

Therefore, M is positive semidefinite if and only if for any $A \in \mathbb{R}^{n \times d}$, we have that

$$\operatorname{diag}(A^\top M A) = \left(\boldsymbol{a}_1^\top M \boldsymbol{a}_1, \cdots, \boldsymbol{a}_d^\top M \boldsymbol{a}_d\right)^\top \succeq \mathbf{0},$$

where $\succeq$ is the elementwise inequality, and $\mathbf{0}$ is a $d \times 1$ zero column vector.

1. Use the definition of positive definite matrices and the spectral theorem to show that all eigenvalues of a positive semidefinite matrix M are non-negative.

   Hint: By Spectral theorem, $\Sigma = Q^\top MQ$ for some Q. What if you take $A = Q$ in the "exercise in matrix multiplication" described above?

2. In this problem, we show that a positive semidefinite matrix is a matrix version of a non-negative scalar, in that they both have a "square root". Show that a symmetric matrix M can be expressed as $M = BB^\top$ for some matrix B, if and only if M is positive semidefinite.

   Hint: To show $M = BB^\top$ implies M is positive semidefinite, use the fact that for any vector $\boldsymbol{v}$, $\boldsymbol{v}^\top \boldsymbol{v} \geq 0$. To show that M positive semidefinite implies $M = BB^\top$ for some B, use the Spectral Theorem.

Positive Definite Matrices

Definition 6. A real, symmetric matrix $M \in \mathbb{R}^{n\times n}$ is **positive definite** if for any nonzero vector $\boldsymbol{x} \in \mathbb{R}^n$, we have that

$$\boldsymbol{x}^\top M\boldsymbol{x} > 0.$$

1. Show that all eigenvalues of a symmetric positive definite matrix are positive. *Hint:* You can use the same method as you used for positive semidefinite matrices above.

2. Let M be a symmetric positive definite matrix. By the spectral theorem, $M = Q\Sigma Q^\top$, where Σ is a diagonal matrix of the eigenvalues of M. By the previous problem, all diagonal entries of Σ are positive. If $\Sigma = \operatorname{diag}(\sigma_1, \cdots, \sigma_n)$, then $\Sigma^{-1} = \operatorname{diag}\left(\sigma_1^{-1}, \ldots, \sigma_n^{-1}\right)$. Show that the matrix $Q\Sigma^{-1}Q^\top$ is the inverse of M.

3. Since positive semidefinite matrices may have eigenvalues that are zero, we see by the previous problem that not all positive semidefinite matrices are invertible. Show that if M is a positive semidefinite matrix and I is the identity matrix, then $M + \lambda I$ is symmetric positive definite for any $\lambda > 0$, and give an expression for the inverse of $M + \lambda I$.

4. Let M and N be symmetric matrices, with M positive semidefinite and N positive definite. Use the definitions of positive semidefiniteness and positive definiteness to show that $M+N$ is symmetric and positive definite. Thus $M + N$ is invertible.

   Hint: For any nonzero vector $\boldsymbol{x}$, show that $\boldsymbol{x}^\top(M + N)\boldsymbol{x} > 0$.

Homework 4: Probabilistic models

Due: Wednesday, March 22, 2023 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

Logistic Regression

Consider a binary classification setting with input space $\mathcal{X} = \mathbb{R}^d$, outcome space $\mathcal{Y}_\pm = \{-1, 1\}$, and a dataset $\mathcal{D} = \big((\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(n)}, y^{(n)})\big)$.

Equivalence of ERM and probabilistic approaches

In the lecture we derived logistic regression using the Bernoulli response distribution. In this problem you will show that it is equivalent to ERM with logistic loss.

ERM with logistic loss: Consider a linear scoring function in the space

$$\mathcal{F}_{\text{score}} = \left\{\boldsymbol{x} \mapsto \boldsymbol{w}^\top \boldsymbol{x};\ \boldsymbol{w} \in \mathbb{R}^d\right\}.$$

A simple way to make predictions (similar to what we've seen with the perceptron algorithm) is to predict $\hat{y} = 1$ if $\boldsymbol{w}^\top \boldsymbol{x} > 0$, or $\hat{y} = \operatorname{sign}(\boldsymbol{w}^\top \boldsymbol{x})$. Accordingly, we consider margin-based loss functions that relate the loss with the margin, $y\boldsymbol{w}^\top \boldsymbol{x}$. A positive margin means that $\boldsymbol{w}^\top \boldsymbol{x}$ has the same sign as y, i.e., a correct prediction. Specifically, let's consider the **logistic loss** function

$$\ell_{\text{logistic}}(\boldsymbol{x}, y, \boldsymbol{w}) = \log\left(1 + \exp(-y\boldsymbol{w}^\top \boldsymbol{x})\right).$$

This is a margin-based loss function that we have encountered several times. Given the logistic loss, we can now minimize the empirical risk on our dataset $\mathcal{D}$ in order to obtain an estimate of the parameters, $\hat{\boldsymbol{w}}$.

MLE with a Bernoulli response distribution and the logistic link function: As discussed in the lecture, given that $\mathbb{P}[y = 1|\boldsymbol{x}; \boldsymbol{w}] = 1/(1 + \exp(-\boldsymbol{w}^\top \boldsymbol{x}))$, we can estimate $\boldsymbol{w}$ by maximizing the likelihood, or equivalently, minimizing the negative log-likelihood ($\text{NLL}_{\mathcal{D}}(\boldsymbol{w})$ in short) of the data.

1. Show that the two approaches are equivalent, i.e. they produce the same solution for $\boldsymbol{w}$.

   Proof. Since we are modeling $\mathbb{P}[y|\boldsymbol{x}]$ as a Bernoulli distribution, we have that

$$\mathbb{P}[y|\boldsymbol{x};\boldsymbol{w}] = \left(\frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}}}\right)^{\frac{y+1}{2}} \left(1 - \frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}}}\right)^{\frac{1-y}{2}}. \tag{1}$$

Therefore, the negative log-likelihood can be computed as

$$\begin{aligned}
\mathrm{NLL}_{\mathcal{D}}(\boldsymbol{w}) &= -\sum_{i=1}^{n} \log \mathbb{P}[y^{(i)}|\boldsymbol{x}^{(i)};\boldsymbol{w}] \\
&= -\sum_{i=1}^{n}\left(\frac{y^{(i)}+1}{2}\log\left(\frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}\right)+\frac{1-y^{(i)}}{2}\log\left(1-\frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}\right)\right) \\
&= -\sum_{i=1}^{n}\left(\frac{y^{(i)}+1}{2}\log\left(\frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}\right)+\frac{1-y^{(i)}}{2}\log\left(\frac{e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}\right)\right) \\
&= -\sum_{i=1}^{n}\left(\frac{y^{(i)}+1}{2}\log\left(\frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}\right)+\frac{1-y^{(i)}}{2}\log\left(\frac{1}{1+e^{\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}}\right)\right) \\
&= \sum_{i=1}^{n}\underbrace{\left(\frac{y^{(i)}+1}{2}\log\left(1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right)+\frac{1-y^{(i)}}{2}\left(\log\left(1+e^{\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right)\right)\right)}_{=:H_i}. \qquad (2)
\end{aligned}$$

When $y^{(i)} = 1$, the coefficient of the second summand in H_i vanishes so we have that

$$H_i = \log\left(1+e^{-\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right) = \log\left(1+e^{-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right). \qquad (3)$$

When $y^{(i)} = -1$, the coefficient of the first summand in H_i vanishes so again we have that

$$H_i = \log\left(1+e^{\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right) = \log\left(1+e^{-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right). \qquad (4)$$

Therefore, we can substitute the simplified expression of H_i into the expression of the negative log-likelihood, so that

$$\mathrm{NLL}_{\mathcal{D}}(\boldsymbol{w}) = \sum_{i=1}^{n}\log\left(1+e^{-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}}\right) = \sum_{i=1}^{n}\ell_{\text{logistic}}(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{w}). \qquad (5)$$

This is simply n times the ERM objective with logistic loss, so that minimizing the negative log-likelihood with a Bernoulli response distribution and the logistic link function is equivalent to minimizing the ERM objective with logistic loss. Therefore, the two approaches are equivalent, and the proof is complete. □

Linearly Separable Data

In this problem, we will investigate the behavior of MLE for logistic regression when the data is linearly separable.

2. Show that the decision boundary of the logistic regression is given by $\left\{\boldsymbol{x};\, \boldsymbol{w}^\top\boldsymbol{x} = 0\right\}$. Note that the set will not change if we multiply the weights by some constant c.

Proof. The decision boundary of the logistic regression is given by the points $\boldsymbol{x}$ such that

$$\mathbb{P}[1|\boldsymbol{x}] = \mathbb{P}[-1|\boldsymbol{x}] = \frac{1}{2}. \qquad (6)$$

In other words, if a point $\boldsymbol{x}$ is on the decision boundary, we have that

$$\frac{1}{1+e^{-\boldsymbol{w}^\top \boldsymbol{x}}} = \frac{1}{2} \implies e^{-\boldsymbol{w}^\top \boldsymbol{x}} = 1 \implies \boldsymbol{w}^\top \boldsymbol{x} = 0. \tag{7}$$

Therefore, the decision boundary is the set of points $\boldsymbol{x}$ such that $\boldsymbol{w}^\top \boldsymbol{x} = 0$, thus is given by $\{\boldsymbol{x};\, \boldsymbol{w}^\top \boldsymbol{x} = 0\}$, and the proof is complete. □

3. Suppose the data is linearly separable and by gradient descent/ascent we have reached a decision boundary defined by $\hat{\boldsymbol{w}}$ where all examples are classified correctly. Show that we can always increase the likelihood of the data by multiplying a scalar c on $\hat{\boldsymbol{w}}$, which means that MLE is not well-defined in this case.

   Hint: You can show this by taking the derivative of $L(c\hat{\boldsymbol{w}})$ with respect to c, where L is the likelihood function.

Proof. By definition of the log-likelihood, we have that

$$\begin{aligned} l(c\hat{\boldsymbol{w}}) &= \log \mathbb{P}[\mathcal{D}; c\hat{\boldsymbol{w}}] = \log\left(\prod_{i=1}^n \mathbb{P}[y^{(i)}|\boldsymbol{x}^{(i)}; c\hat{\boldsymbol{w}}]\right) \\ &= \sum_{i=1}^n \left(\frac{y^{(i)}+1}{2}\log\left(\frac{1}{1+e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}\right) + \frac{1-y^{(i)}}{2}\log\left(1-\frac{1}{1+e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}\right)\right) \\ &= -\sum_{i=1}^n \left(\frac{y^{(i)}+1}{2}\log\left(1+e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}\right) + \frac{1-y^{(i)}}{2}\left(\log\left(1+e^{c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}\right)\right)\right). \end{aligned} \tag{8}$$

Taking its partial derivative with respect to c, we can deduce that

$$\begin{aligned} \frac{\partial}{\partial c} l(c\hat{\boldsymbol{w}}) &= -\sum_{i=1}^n \left(\frac{y^{(i)}+1}{2}\cdot\frac{-e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}} + \frac{1-y^{(i)}}{2}\cdot\frac{e^{c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}\right) \\ &= \sum_{i=1}^n \underbrace{\left(\frac{y^{(i)}+1}{2}\cdot\frac{\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}} + \frac{1-y^{(i)}}{2}\cdot\frac{-\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}\right)}_{=:H_i}. \end{aligned} \tag{9}$$

When $y^{(i)} = 1$, the coefficient of the second summand in H_i vanishes so we have that

$$H_i = \frac{\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}} = \frac{y^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{cy^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}. \tag{10}$$

When $y^{(i)} = -1$, the coefficient of the first summand in H_i vanishes so again we have that

$$H_i = \frac{-\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{-c\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}} = \frac{y^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{cy^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}. \tag{11}$$

Substituting this expression into the partial derivative then gives

$$\frac{\partial}{\partial c} l(c\hat{\boldsymbol{w}}) = \sum_{i=1}^n \frac{y^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}{1+e^{cy^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)}}}. \tag{12}$$

Since $\hat{\boldsymbol{w}}$ is such that all examples are classified correctly, we know that $y^{(i)}\hat{\boldsymbol{w}}^\top \boldsymbol{x}^{(i)} > 0$ for all $i = 1, \cdots, n$. Therefore, the partial derivative must also be positive. This means that by increasing c, the log-likelihood would also increase. By definition of the log-likelihood, we then have that the likelihood $L(c\hat{\boldsymbol{w}}) = e^{l(c\hat{\boldsymbol{w}})}$. Since the exponential function is monotonically increasing, the likelihood also increases as c gets larger. This implies that the MLE is not well-defined in this case, and thus the proof is complete. □

Regularized Logistic Regression

As we've shown in above, when the data is linearly separable, MLE for logistic regression may end up with weights with very large magnitudes. Such a function is prone to overfitting. In this part, we will apply regularization to fix the problem. The l_2 regularized logistic regression objective function can be defined as

$$J_{\text{logistic}}(\boldsymbol{w}) = \hat{R}_n(\boldsymbol{w}) + \lambda\|\boldsymbol{w}\|^2 = \frac{1}{n}\sum_{i=1}^{n} \log\left(1 + \exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \lambda\|\boldsymbol{w}\|^2.$$

4. Prove that the objective function $J_{\text{logistic}}(\boldsymbol{w})$ is convex. You may use any facts mentioned in the convex optimization notes.

   Proof. To show that $J_{\text{logistic}}(\boldsymbol{w})$ is convex, it suffices to show that the corresponding Hessian matrix is positive definite. First, we compute its first-order partial derivative with respect to an arbitrary entry w_k as

$$\begin{aligned}\frac{\partial}{\partial w_k} J_{\text{logistic}}(\boldsymbol{w}) &= \frac{1}{n}\sum_{i=1}^{n} \frac{\partial}{\partial w_k}\log\left(1 + \exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \lambda \frac{\partial}{\partial w_k}\|\boldsymbol{w}\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{1}{1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)} \cdot \frac{\partial}{\partial w_k}\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right) + 2\lambda w_k \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)}{1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)} \cdot \frac{\partial}{\partial w_k}\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right) + 2\lambda w_k \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{-y^{(i)}x_k^{(i)}}{1+\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)} + 2\lambda w_k. \qquad (13)\end{aligned}$$

   Next, we further compute its partial derivative with respect to an arbitrary entry w_l in order to obtain the second-order partial derivative with respect to w_k and w_l, such that

$$\begin{aligned}\frac{\partial^2}{\partial w_l \partial w_k} J_{\text{logistic}}(\boldsymbol{w}) &= \frac{1}{n}\sum_{i=1}^{n}\left(-y^{(i)}x_k^{(i)}\right)\frac{\partial}{\partial w_l}\left(\frac{1}{1+\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)}\right) + 2\lambda\delta_{kl} \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{y^{(i)}x_k^{(i)}}{\left(1+\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right)^2} \cdot \frac{\partial}{\partial w_l}\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right) + 2\lambda\delta_{kl} \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{y^{(i)}x_k^{(i)}\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)}{\left(1+\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right)^2} \cdot \frac{\partial}{\partial w_l}\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right) + 2\lambda\delta_{kl} \\ &= \frac{1}{n}\sum_{i=1}^{n} \frac{\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)}{\left(1+\exp\left(y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right)^2} \cdot x_k^{(i)}x_l^{(i)} + 2\lambda\delta_{kl}. \qquad (14)\end{aligned}$$

Therefore, the Hessian matrix of the l_2 regularized logistic regression objective function can be expressed as

$$H(\boldsymbol{w}) = \frac{1}{n} \sum_{i=1}^{n} \frac{\exp\left(y^{(i)} \boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)}{\left(1 + \exp\left(y^{(i)} \boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right)^2} \boldsymbol{x}^{(i)} (\boldsymbol{x}^{(i)})^\top + 2\lambda I. \tag{15}$$

Therefore, for any nonzero vector $\boldsymbol{z}$, we have that

$$\boldsymbol{z}^\top H(\boldsymbol{w}) \boldsymbol{z} = \frac{1}{n} \sum_{i=1}^{n} \frac{\exp\left(y^{(i)} \boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)}{\left(1 + \exp\left(y^{(i)} \boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right)^2} \|\boldsymbol{z}^\top \boldsymbol{x}^{(i)}\|^2 + 2\lambda \|\boldsymbol{z}\|^2 > 0, \tag{16}$$

which implies that the Hessian matrix $H(\boldsymbol{w})$ is positive semidefinite. This completes our proof that $J_{\text{logistic}}(\boldsymbol{w})$ is convex. $\square$

5. Complete the `f_objective` function in the skeleton code, which computes the objective function for $J_{\text{logistic}}(\boldsymbol{w})$.

   Hint: you may get numerical overflow when computing the exponential literally, e.g. try e^{1000} in numpy. Make sure to read about the log-sum-exp trick and use the numpy function `logaddexp` to get accurate calculations and to prevent overflow.

   Solution. The function `f_objective` can be implemented as follows.

```python
def f_objective(theta, X, y, l2_param=1):
    """The l2-regularized logistic regression objective.

    Parameters
    ----------
    theta : np.array
        The array of parameters (weights), of size num_features.
    X : np.ndarray
        The design matrix, of size num_instances * num_features.
    y : np.array
        The array of binary outcomes, of size num_instances.
    l2_param : float
        The l2 regularization parameter.

    Returns
    -------
    objective : float
        The scalar value of the objective function.
    """
    n = X.shape[0]
    return np.mean(np.logaddexp(np.zeros(n), -y * np.dot(X, theta))) \
        + l2_param * np.linalg.norm(theta) ** 2
```

6. Complete the `fit_logistic_regression_function` in the skeleton code using the function `minimize` from `scipy.optimize`. Use this function to train a model on the provided data. Make sure to take the appropriate preprocessing steps, such as standardizing the data and adding a column for the bias term.

   Solution. The function `fit_logistic_regression_function` is implemented as follows.

```python
def fit_logistic_reg(X, y, objective_function, l2_param=1):
    """Train a model using l2-regularized logistic regression.

    Parameters
    ----------
    X : np.ndarray
        The design matrix, of size num_instances * num_features.
    y : np.array
        The array of binary outcomes, of size num_instances.
    objective_function : function
        The objective function that takes theta, X, y, and l2_param.
    l2_param : float
        The l2 regularization parameter.

    Returns
    -------
    optimal_theta : np.array
        The optimal array of parameters (weights), of size num_features.
    """
    d = X.shape[1]
    opt = minimize(objective_function, np.zeros(d), args=(X, y, l2_param))
    return opt.x
```

Moreover, we need the following preprocessing steps.

```python
# Preprocessing: change labels from {0, 1} to {-1, 1}
y_train[y_train == 0] = -1
y_val[y_val == 0] = -1

# Preprocessing: add bias term (intercept)
X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
X_val = np.hstack((np.ones((X_val.shape[0], 1)), X_val))

# Preprocessing: normalize to [0, 1]
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_val = scaler.transform(X_val)
```

7. Find the l_2 regularization parameter that minimizes the log-likelihood on the validation set. Plot the log-likelihood for different values of the regularization parameter.

   Solution. The plot of the objective function with respect to different values of the regularization parameter λ is shown as in Figure 1.

   From the plot, we can see that the best regularization parameter is approximately $\lambda = 0.019$.

8. (Optional) It seems reasonable to interpret the prediction $f(\boldsymbol{x}) = \phi(\boldsymbol{w}^\top \boldsymbol{x}) = 1/(1+e^{-\boldsymbol{w}^\top \boldsymbol{x}})$ as the probability that $y = 1$, for a randomly drawn pair $(\boldsymbol{x}, y)$. Since we only have a finite sample (and we are regularizing, which will bias things a bit), there is a question of how well "calibrated" our predicted probabilities are. Roughly speaking, we say $f(\boldsymbol{x})$ is well calibrated if we look at all examples $(\boldsymbol{x}, y)$ for which $f(\boldsymbol{x}) \approx 0.7$ and we find that close to 70% of those examples have $y = 1$, as predicted... and then we repeat that for all predicted probabilities in $(0, 1)$. To see how well-calibrated our predicted probabilities are, break the predictions on the validation set into groups based on the predicted probability (you can play with the size of the groups to get a result you think is informative). For each group,

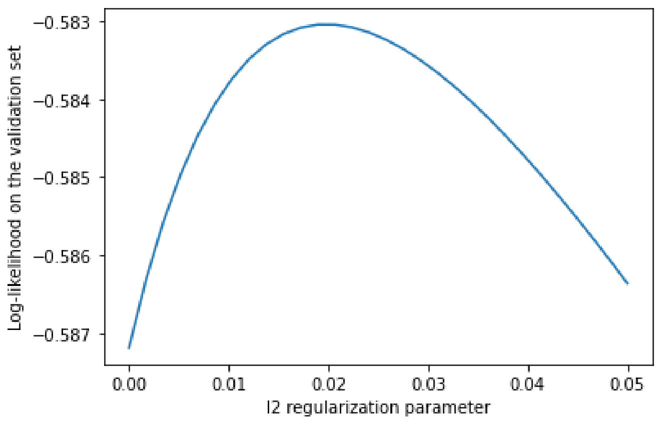


Figure 1: The objective function with respect to different values of the regularization parameter λ, ranging from 0 to 0.05. Both axes are in the log scale.

examine the percentage of positive labels. You can make a table or graph. Summarize the results. You may get some ideas and references from scikit-learn's discussion.

Solution. The calibration plot using regularization parameter $\lambda = 0.019$ and splitting into ten bins is shown as in Figure 2.

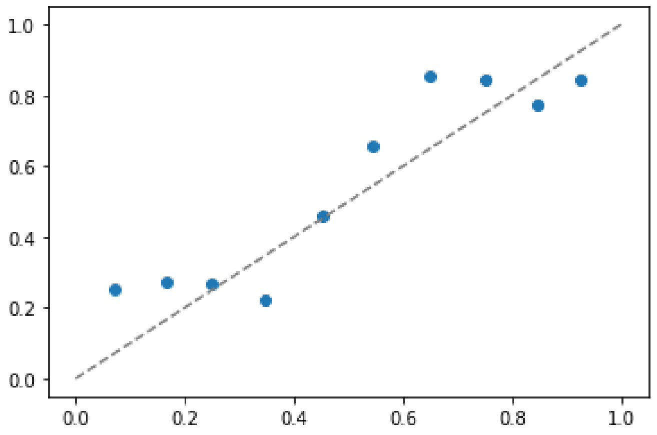


Figure 2: The calibration plot using regularization parameter $\lambda = 0.019$. The x-axis represents the average predicted probability (for +1 label) in each bin. The y-axis represents the fraction of +1 labels among the validation examples in each bin. The dashed line in the middle represents perfect calibration.

From the plot, we can see that the scattered points does not deviate too much from the perfect calibration, indicating that the predicted probabilities are quite well calibrated.

Coin Flipping with Partial Observability

Consider flipping a biased coin where $\mathbb{P}[z = \mathtt{H}; \theta_1] = \theta_1$. However, we cannot directly observe the result z. Instead, someone reports the result to us, which we denote by x. Furthermore, there is a chance that the result is reported incorrectly *if it is a head*. More specifically, we have $\mathbb{P}[x = \mathtt{H} | z = \mathtt{H}; \theta_2] = \theta_2$ and $\mathbb{P}[x = \mathtt{T} | z = \mathtt{T}] = 1$.

9. Show that $\mathbb{P}[x = \mathtt{H}; \theta_1, \theta_2] = \theta_1 \theta_2$.

   Proof. By the law of total probability, we have that

$$\begin{aligned} \mathbb{P}[x = \mathtt{H}; \theta_1, \theta_2] &= \mathbb{P}[x = \mathtt{H} | z = \mathtt{H}; \theta_2] \cdot \mathbb{P}[z = \mathtt{H}; \theta_1] + \mathbb{P}[x = \mathtt{H} | z = \mathtt{T}] \cdot \mathbb{P}[z = \mathtt{T}; \theta_1] \\ &= \theta_2 \cdot \theta_1 + 0 \cdot (1 - \theta_1) = \theta_1 \theta_2, \end{aligned} \tag{17}$$

   so the proof is complete. □

10. Given a set of reported results $\mathcal{D}_r$ of size N_r, where the number of heads is n_h and the number of tails is n_t, write down the likelihood of $\mathcal{D}_r$ as a function of θ_1 and θ_2.

    Solution. The likelihood of $\mathcal{D}_r$ can be written as

$$L(\theta_1, \theta_2) = \mathbb{P}[\mathcal{D}_r; \theta_1, \theta_2] = (\theta_1 \theta_2)^{n_h} (1 - \theta_1 \theta_2)^{n_t}. \tag{18}$$

11. Can we estimate θ_1 and θ_2 using MLE? Explain your judgment.

    Solution. We cannot estimate θ_1 and θ_2 using MLE. The reason is, what we have is just the set of reported results, and we cannot determine whether the change of likelihood is due to the bias of the coin or the incorrectness of the reported results. In other words, we can only estimate $\theta_1 \theta_2$ using MLE, but not θ_1 and θ_2 respectively.

Homework 5: SGD for Multiclass Linear SVM

Due: Wednesday, April 5, 2023 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

Bayesian Modeling

Bayesian Logistic Regression with Gaussian Priors

This question analyzes logistic regression in the Bayesian setting, where we introduce a prior $\mathbb{P}[\boldsymbol{w}]$ on $\boldsymbol{w} \in \mathbb{R}^d$. Consider a binary classification setting with input space $\mathcal{X} = \mathbb{R}^d$, outcome space $\mathcal{Y}_\pm = \{-1, 1\}$, and a dataset $\mathcal{D} = \left((\boldsymbol{x}^{(1)}, y^{(1)}), \cdots, (\boldsymbol{x}^{(n)}, y^{(n)})\right)$.

1. Give an expression for the posterior density $\mathbb{P}[\boldsymbol{w}|\mathcal{D}]$ in terms of the negative log-likelihood function $\mathrm{NLL}_{\mathcal{D}}(\boldsymbol{w})$ and the prior density $\mathbb{P}[\boldsymbol{w}]$ (up to a proportionality constant is fine).

   Solution. The posterior density can be computed by Bayes' rule as

   $$\mathbb{P}[\boldsymbol{w}|\mathcal{D}] = \frac{\mathbb{P}[\mathcal{D}|\boldsymbol{w}]\mathbb{P}[\boldsymbol{w}]}{\mathbb{P}[\mathcal{D}]} = \frac{L_{\mathcal{D}}(\boldsymbol{w})\mathbb{P}[\boldsymbol{w}]}{\mathbb{P}[\mathcal{D}]} = \frac{\exp(-\mathrm{NLL}_{\mathcal{D}}(\boldsymbol{w}))\mathbb{P}[\boldsymbol{w}]}{\mathbb{P}[\mathcal{D}]}, \tag{1}$$

   so that in terms of the negative log-likelihood and the prior density, we can conclude that

   $$\mathbb{P}[\boldsymbol{w}|\mathcal{D}] \propto \exp(-\mathrm{NLL}_{\mathcal{D}}(\boldsymbol{w}))\mathbb{P}[\boldsymbol{w}]. \tag{2}$$

2. Suppose we take a prior on $\boldsymbol{w}$ of the form $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \Sigma)$, which is in the Gaussian family. Is this a conjugate prior to the likelihood given by logistic regression?

   Solution. Since $\boldsymbol{w} \sim \mathcal{N}(\mathbf{0}, \Sigma)$, we have that

   $$\mathbb{P}[\boldsymbol{w}] \propto \exp\left(-\frac{1}{2}\boldsymbol{w}^\top \Sigma^{-1} \boldsymbol{w}\right). \tag{3}$$

   Moreover, recall that the negative log-likelihood given by the logistic regression is

   $$\mathrm{NLL}_{\mathcal{D}}(\boldsymbol{w}) = \sum_{i=1}^{n} \log\left(1 + \exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right). \tag{4}$$

   Therefore, the posterior density can be computed as

   $$\begin{aligned}\mathbb{P}[\boldsymbol{w}|\mathcal{D}] &\propto \exp\left(-\sum_{i=1}^{n}\log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top\boldsymbol{x}^{(i)}\right)\right)\right)\exp\left(-\frac{1}{2}\boldsymbol{w}^\top\Sigma^{-1}\boldsymbol{w}\right) \\ &= \prod_{i=1}^{n}\frac{1}{1+\exp\left(-y^{(i)}\boldsymbol{w}^\top\boldsymbol{x}^{(i)}\right)}\exp\left(-\frac{1}{2}\boldsymbol{w}^\top\Sigma^{-1}\boldsymbol{w}\right). \end{aligned} \tag{5}$$

   This, unfortunately, is not a multivariate Gaussian distribution, thus the given prior is not a conjugate prior to the likelihood given by the logistic regression.

3. Show that there exist a covariance matrix Σ such that the MAP (maximum a posteriori) estimate for $\boldsymbol{w}$ after observing data $\mathcal{D}$ is the same as the minimizer of the regularized logistic regression function, and give its value.

   Hint: Consider minimizing the negative log posterior of $\boldsymbol{w}$. Also, remember you can drop any term from the objective function that does not depend on $\boldsymbol{w}$. You may freely use the results of previous problems.

   Proof. The negative log posterior of $\boldsymbol{w}$, in the above setting, can be computed as

$$-\log \mathbb{P}[\boldsymbol{w}|\mathcal{D}] = \sum_{i=1}^{n} \log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \frac{1}{2}\boldsymbol{w}^\top \Sigma^{-1}\boldsymbol{w} + \texttt{const.} \tag{6}$$

   Also recall that the regularized logistic regression function is defined as

$$J_{\text{logistic}}(\boldsymbol{w}) = \frac{1}{n}\sum_{i=1}^{n} \log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \lambda \left\|\boldsymbol{w}\right\|^2. \tag{7}$$

   By taking $\Sigma = (2\lambda n)^{-1} I$, we can rewrite the negative log posterior of $\boldsymbol{w}$ as

$$\begin{aligned} -\log \mathbb{P}[\boldsymbol{w}|\mathcal{D}] &= \sum_{i=1}^{n} \log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \frac{2\lambda n}{2}\boldsymbol{w}^\top \boldsymbol{w} + \texttt{const.} \\ &= \sum_{i=1}^{n} \log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \lambda n \left\|\boldsymbol{w}\right\|^2 + \texttt{const.} \end{aligned} \tag{8}$$

   Therefore, the MAP estimate for $\boldsymbol{w}$ after observing data $\mathcal{D}$ can be written as

$$\begin{aligned} \hat{\boldsymbol{w}}_{\text{MAP}} &= \arg\max_{\boldsymbol{w}}\ \mathbb{P}[\boldsymbol{w}|\mathcal{D}] = \arg\min_{\boldsymbol{w}} \left(-\log \mathbb{P}[\boldsymbol{w}|\mathcal{D}]\right) \\ &= \arg\min_{\boldsymbol{w}} \left(\sum_{i=1}^{n} \log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \lambda n \left\|\boldsymbol{w}\right\|^2 + \texttt{const.}\right) \\ &= \arg\min_{\boldsymbol{w}} \left(\sum_{i=1}^{n} \log\left(1+\exp\left(-y^{(i)}\boldsymbol{w}^\top \boldsymbol{x}^{(i)}\right)\right) + \lambda n \left\|\boldsymbol{w}\right\|^2\right) \\ &= \arg\min_{\boldsymbol{w}}\ nJ_{\text{logistic}}(\boldsymbol{w}) = \arg\min_{\boldsymbol{w}}\ J_{\text{logistic}}(\boldsymbol{w}). \end{aligned} \tag{9}$$

   Hence, we have shown that there exists a covariance matrix $\Sigma = (2\lambda n)^{-1} I$, such that the MAP estimate for $\boldsymbol{w}$ after observing data $\mathcal{D}$ is the same as the minimizer of the regularized logistic regression function, so the proof is complete. □

4. In the Bayesian approach, the prior should reflect your beliefs about the parameters before seeing the data and, in particular, should be eindependent on the eventual size of your dataset. Imagine choosing a prior distribution $\boldsymbol{w} \sim \mathcal{N}(0, I)$. For a dataset $\mathcal{D}$ of size n, how should you choose λ in our regularized logistic regression objective function so that the ERM is equal to the mode of the posterior distribution of $\boldsymbol{w}$ (i.e. is equal to the MAP estimator)?

   Solution. By the result of the previous question, if we take $\Sigma = I$, then we would require $(2\lambda n)^{-1} = 1$, which solves to $\lambda = (2n)^{-1}$.

Coin Flipping with Partial Observability

This is continuing your analysis done in HW4, you may use the results you obtained in HW4. Consider flipping a biased coin where $\mathbb{P}[z = \texttt{H}; \theta_1] = \theta_1$. However, we cannot directly observe the result z. Instead, someone reports the result to us, which we denote by x. Furthermore, there is a chance that the result is reported incorrectly *if it is a head.* Specifically, we have $\mathbb{P}[x = \texttt{H}|z = \texttt{H}; \theta_2] = \theta_2$ and $\mathbb{P}[x = \texttt{T}|z = \texttt{T}] = 1$.

5. We additionally obtained a set of clean results $\mathcal{D}_c$ of size N_c, where x is directly observed without the reporter in the middle. Given that there are c_h heads and c_t tails, estimate θ_1 and θ_2 by MLE taking the two datasets into account. In this case, note that the likelihood is $L(\theta_1, \theta_2) = \mathbb{P}[\mathcal{D}_r, \mathcal{D}_c; \theta_1, \theta_2]$.

   Solution. From HW4, the likelihood of $\mathcal{D}_r$ can be written as

$$\mathbb{P}[\mathcal{D}_r; \theta_1, \theta_2] = (\theta_1\theta_2)^{n_h}(1-\theta_1\theta_2)^{n_t}. \tag{10}$$

   The likelihood of $\mathcal{D}_c$ depends only on the real result. It can thus be written as

$$\mathbb{P}[\mathcal{D}_c; \theta_1] = \prod_{i=1}^{N_c} \mathbb{P}[z_i = \texttt{H}; \theta_1]^{[z_i=\texttt{H}]}\mathbb{P}[z_i = \texttt{T}; \theta_1]^{[z_i=\texttt{T}]} = \theta_1^{c_h}(1-\theta_1)^{c_t}. \tag{11}$$

   Since the observation on $\mathcal{D}_r$ and on $\mathcal{D}_c$ are independent, we can compute the joint likelihood of $\mathcal{D}_r$ and $\mathcal{D}_c$ as

$$\begin{aligned} L(\theta_1, \theta_2) &= \mathbb{P}[\mathcal{D}_r, \mathcal{D}_c; \theta_1, \theta_2] = \mathbb{P}[\mathcal{D}_r; \theta_1, \theta_2]\mathbb{P}[\mathcal{D}_c; \theta_1] \\ &= \theta_1^{n_h+c_h}\theta_2^{n_h}(1-\theta_1)^{c_t}(1-\theta_1\theta_2)^{n_t}. \end{aligned} \tag{12}$$

   For the sake of simplicity when estimating θ_1 and θ_2, we consider the log-likelihood, which can be written as

$$\log L(\theta_1, \theta_2) = (n_h + c_h)\log\theta_1 + n_h\log\theta_2 + c_t\log(1-\theta_1) + n_t\log(1-\theta_1\theta_2). \tag{13}$$

   The first order condition with respect to θ_1 gives that

$$\begin{aligned} &\frac{n_h + c_h}{\theta_1} - \frac{c_t}{1-\theta_1} - \frac{n_t\theta_2}{1-\theta_1\theta_2} = 0 \\ \implies\ & n_h + c_h + \theta_1^2\theta_2(n_h + c_h + n_t + c_t) = \theta_1\theta_2(n_h + c_h + n_t) + \theta_1(n_h + c_h + c_t). \end{aligned} \tag{14}$$

   The first order condition with respect to θ_2 gives that

$$\frac{n_h}{\theta_2} - \frac{n_t\theta_1}{1-\theta_1\theta_2} = 0 \implies \theta_1\theta_2 n_t = (1-\theta_1\theta_2)n_h \implies \theta_1\theta_2 = \frac{n_h}{n_h + n_t}. \tag{15}$$

   Substituting this result into (14), we have that

$$\begin{aligned} &n_h + c_h + \frac{\theta_1 n_h(n_h + c_h + n_t + c_t)}{n_h + n_t} = \frac{n_h(n_h + c_h + n_t)}{n_h + n_t} + \theta_1(n_h + c_h + c_t) \\ \implies\ & c_h n_t = \theta_1(n_t c_h + n_t c_t) \implies \theta_1 = \frac{c_h}{c_h + c_t}. \end{aligned} \tag{16}$$

Substituting this result into (15), we can further obtain that

$$\theta_2 = \frac{n_h}{n_h + n_t} \cdot \frac{c_h + c_t}{c_h} = \frac{n_h(c_h + c_t)}{c_h(n_h + n_t)}. \tag{17}$$

Therefore, we can conclude that by MLE, the estimates of θ_1 and θ_2 are respectively

$$(\hat{\theta}_1)_{\text{MLE}} = \frac{c_h}{c_h + c_t}, \qquad (\hat{\theta}_2)_{\text{MLE}} = \frac{n_h(c_h + c_t)}{c_h(n_h + n_t)}. \tag{18}$$

6. Since the clean results are expensive, we only have a small number of those and we are worried that we may overfit the data. To mitigate overfitting we can use a prior distribution on θ_1 if available. Let's imagine that an oracle gave use the prior $\mathbb{P}[\theta_1] = \text{Beta}(h, t)$. Derive the MAP estimates for θ_1 and θ_2.

Solution. Given the beta family prior, we can write the posterior distribution as

$$\begin{aligned}\mathbb{P}[\theta_1, \theta_2 | \mathcal{D}_r, \mathcal{D}_c] &\propto \mathbb{P}[\mathcal{D}_r, \mathcal{D}_c; \theta_1, \theta_2]\mathbb{P}[\theta_1] \\ &= \theta_1^{n_h + c_h}\theta_2^{n_h}(1-\theta_1)^{c_t}(1-\theta_1\theta_2)^{n_t} \cdot \theta_1^{h-1}(1-\theta_1)^{t-1} \\ &= \theta_1^{n_h+c_h+h-1}(1-\theta_1)^{c_t+t-1}\theta_2^{n_h}(1-\theta_1\theta_2)^{n_t}. \end{aligned} \tag{19}$$

For the sake of simplicity when estimating θ_1 and θ_2, we consider the log-likelihood, which can be written as

$$\begin{aligned} l(\theta_1, \theta_2) &= \log \mathbb{P}[\theta_1, \theta_2 | \mathcal{D}_r, \mathcal{D}_c] \\ &= (n_h + c_h + h - 1)\log\theta_1 + n_h \log\theta_2 + (c_t + t - 1)\log(1-\theta_1) + n_t \log(1-\theta_1\theta_2), \end{aligned} \tag{20}$$

ignoring the constant additive since it would have no effect on the optimization. The first order condition with respect to θ_1 gives that

$$\begin{aligned} &\frac{n_h + c_h + h - 1}{\theta_1} - \frac{c_t + t - 1}{1-\theta_1} - \frac{n_t\theta_2}{1-\theta_1\theta_2} = 0 \\ \Longrightarrow\ & n_h + c_h + h - 1 + \theta_1^2\theta_2(n_h + c_h + n_t + c_t + h + t - 2) \\ &\quad = \theta_1\theta_2(n_h + c_h + n_t + h - 1) + \theta_1(n_h + c_h + c_t + h + t - 2). \end{aligned} \tag{21}$$

The first order condition with respect to θ_2 gives that

$$\frac{n_h}{\theta_2} - \frac{n_t\theta_1}{1-\theta_1\theta_2} = 0 \implies \theta_1\theta_2 n_t = (1-\theta_1\theta_2)n_h \implies \theta_1\theta_2 = \frac{n_h}{n_h + n_t}. \tag{22}$$

Substituting this result into (21), we have that

$$\begin{aligned} & n_h + c_h + h - 1 + \frac{\theta_1 n_h(n_h + c_h + n_t + c_t + h + t - 2)}{n_h + n_t} \\ &\quad = \frac{n_h(n_h + c_h + n_t + h - 1)}{n_h + n_t} + \theta_1(n_h + c_h + c_t + h + t - 2) \\ \Longrightarrow\ & (c_h + h - 1)n_t = \theta_1(c_h + c_t + h + t - 2)n_t \implies \theta_1 = \frac{c_h + h - 1}{c_h + c_t + h + t - 2}. \end{aligned} \tag{23}$$

Substituting this result into (22), we can further obtain that

$$\theta_2 = \frac{n_h}{n_h + n_t} \cdot \frac{c_h + c_t + h + t - 2}{c_h + h - 1} = \frac{n_h(c_h + c_t + h + t - 2)}{(c_h + h - 1)(n_h + n_t)}. \tag{24}$$

Therefore, we can conclude that by MLE, the estimates of θ_1 and θ_2 are respectively

$$(\hat{\theta}_1)_{\text{MAP}} = \frac{c_h + h - 1}{c_h + c_t + h + t - 2}, \qquad (\hat{\theta}_2)_{\text{MAP}} = \frac{n_h(c_h + c_t + h + t - 2)}{(c_h + h - 1)(n_h + n_t)}. \tag{25}$$

Derivation for Multi-class Modeling

Suppose our output space and our action space are given as follows: $\mathcal{Y} = \mathcal{A} = \{1, \cdots, k\}$. Given a non-negative class-sensitive loss function $\Delta : \mathcal{Y} \times \mathcal{A} \to [0, \infty)$ and a class-sensitive feature mapping $\boldsymbol{\Psi} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^d$, our prediction function $f : \mathcal{X} \to \mathcal{Y}$ is given by

$$f_{\boldsymbol{w}}(\boldsymbol{x}) = \underset{y \in \mathcal{Y}}{\arg\max} \left\langle \boldsymbol{w}, \boldsymbol{\Psi}(\boldsymbol{x}, y) \right\rangle .$$

For training data $(\boldsymbol{x}_1, y_1), \cdots, (\boldsymbol{x}_n, y_n) \in \mathcal{X} \times \mathcal{Y}$, let $J(\boldsymbol{w})$ be the l_2-regularized empirical risk function for the multiclass hinge loss. We can write this as

$$J(\boldsymbol{w}) = \lambda \|\boldsymbol{w}\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max_{y \in \mathcal{Y}} \left(\Delta\left(y_i, y\right) + \left\langle \boldsymbol{w}, \boldsymbol{\Psi}(\boldsymbol{x}_i, y) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i) \right\rangle \right),$$

for some $\lambda > 0$.

7. Show that $J(\boldsymbol{w})$ is a convex function of $\boldsymbol{w}$. You may use any of the rules about convex functions described in our notes on Convex Optimization, in previous assignments, or in the *Boyd and Vandenberghe* book, though you should cite the general facts you are using.

   Hint: If $f_1, \cdots, f_m : \mathbb{R}^n \to \mathbb{R}$ are convex, then their pointwise maximum $f(x) = \max_i f_i(x)$ is also convex.

   Proof that affine mappings are convex. Let $\mathcal{A}(\boldsymbol{x}) = \boldsymbol{a}^\top \boldsymbol{x} + b$ for arbitrary $\boldsymbol{a}$ and b. Then for any $0 \le \lambda \le 1$, we have that

$$\begin{aligned} \mathcal{A}(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) &= \boldsymbol{a}^\top(\lambda \boldsymbol{x} + (1-\lambda)\boldsymbol{y}) + b = \lambda \boldsymbol{a}^\top \boldsymbol{x} + (1-\lambda)\boldsymbol{a}^\top \boldsymbol{y} + \lambda b + (1-\lambda)b \\ &= \lambda(\boldsymbol{a}^\top \boldsymbol{x} + b) + (1-\lambda)(\boldsymbol{a}^\top \boldsymbol{y} + b) = \lambda \mathcal{A}(\boldsymbol{x}) + (1-\lambda)\mathcal{A}(\boldsymbol{y}), \end{aligned} \tag{26}$$

   proving that the affine mapping f is both convex and concave, though neither strictly convex nor strictly concave. Next we are ready to prove Question 7. □

   Proof of Question 7. With respect to $\boldsymbol{w}$, $\Delta(y_i, y)$ and $\boldsymbol{\Psi}(\boldsymbol{x}_i, y) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i)$ are both constant factors. Therefore, we can see that $\Delta(y_i, y) + \boldsymbol{w}^\top (\boldsymbol{\Psi}(\boldsymbol{x}_i, y) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i))$ is an affine mapping on $\boldsymbol{w}$. Therefore, it is convex, though not strictly convex. Taking the pointwise maximum over $y \in \mathcal{Y}$, the hint implies that the result is also convex. Summing up convex functions preserves convexity, and thus

$$\frac{1}{n}\sum_{i=1}^{n} \max_{y\in\mathcal{Y}} \left(\Delta(y_i, y) + \boldsymbol{w}^\top \left(\boldsymbol{\Psi}(\boldsymbol{x}_i, y) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i)\right)\right) \tag{27}$$

is convex. The l_2 norm of $\boldsymbol{w}$ is clearly convex as we have seen multiple times in the past, and given that $\lambda > 0$, we can conclude that the multiclass hinge loss objective is convex, so the proof is complete. $\square$

8. Since $J(\boldsymbol{w})$ is convex, it has a subgradient at every point. Give an expression for a subgradient of $J(\boldsymbol{w})$. You may use any standard results about subgradients, including the result from an earlier homework about subgradients of the pointwise maxima of functions.

   Hint: It may be helpful to refer to $\hat{y}_i = \arg\max_{y\in\mathcal{Y}} \left(\Delta\left(y_i, y\right) + \langle \boldsymbol{w}, \boldsymbol{\Psi}(\boldsymbol{x}_i, y) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i)\rangle\right)$.

   Solution. A subgradient of $J(\boldsymbol{w})$ can be given by

$$\boldsymbol{g}_J(\boldsymbol{w}) = 2\lambda\boldsymbol{w} + \frac{1}{n}\sum_{i=1}^{n}\left(\boldsymbol{\Psi}(\boldsymbol{x}_i, \hat{y}_i) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i)\right). \tag{28}$$

   Now we prove that $\boldsymbol{g}_J$ is indeed a subgradient of the objective function J. First, note that $2\lambda\boldsymbol{w}$ is the gradient vector of $\lambda\|\boldsymbol{w}\|^2$, so it suffices to prove for the rest part. Next, recall that we have shown in *HW3* that

   Suppose $f_1, \cdots, f_m : \mathbb{R}^d \to \mathbb{R}$ are convex functions, and $f(\boldsymbol{x}) = \max_i f_i(\boldsymbol{x})$. Let k be any index for which $f_k(\boldsymbol{x}) = f(\boldsymbol{x})$, and choose $\boldsymbol{g} \in \partial f_k(\boldsymbol{x})$. Then $\boldsymbol{g} \in \partial f(\boldsymbol{x})$.

   In our case, we are taking maximum over $y \in \mathcal{Y}$, so for each point $\boldsymbol{w}$, it suffices to find the subgradient of some function that attains the maximum. The hint has provided us with a good choice which attains the maximum for all $\boldsymbol{w}$. Now note that

$$\nabla_{\boldsymbol{w}}\left(\Delta(y_i, \hat{y}_i) + \boldsymbol{w}^\top\left(\boldsymbol{\Psi}(\boldsymbol{x}_i, \hat{y}_i) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i)\right)\right) = \boldsymbol{\Psi}(\boldsymbol{x}_i, \hat{y}_i) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i), \tag{29}$$

   so by the previous result and linearity of subgradients, we can conclude that $\boldsymbol{g}_J$ is indeed a subgradient of the objective function J.

9. Give an expression for the stochastic subgradient based on the point $(\boldsymbol{x}_i, y_i)$.

   Solution. The update formula for the stochastic gradient descent based on the point $(\boldsymbol{x}_i, y_i)$ can be written as

$$\boldsymbol{w} \leftarrow (1 - 2\lambda\eta)\boldsymbol{w} - \eta\left(\boldsymbol{\Psi}(\boldsymbol{x}_i, \hat{y}_i) - \boldsymbol{\Psi}(\boldsymbol{x}_i, y_i)\right), \tag{30}$$

   where η is the learning rate.

10. Give an expression for a minibatch subgradient descent based on the points $(\boldsymbol{x}_i, y_i), \cdots, (\boldsymbol{x}_{i+m-1}, y_{i+m-1})$.

Solution. The update formula for the minibatch subgradient descent based on the points $(\boldsymbol{x}_i, y_i), \cdots, (\boldsymbol{x}_{i+m-1}, y_{i+m-1})$ can be written as

$$\boldsymbol{w} \leftarrow (1 - 2\lambda\eta)\boldsymbol{w} - \frac{\eta}{m}\sum_{k=i}^{i+m-1} \left(\boldsymbol{\Psi}(\boldsymbol{x}_k, \hat{y}_k) - \boldsymbol{\Psi}(\boldsymbol{x}_k, y_k)\right), \tag{31}$$

where η is the learning rate.

(Optional) Hinge Loss is a Special Case of Generalized Hinge Loss

Let $\mathcal{Y} = \{-1, 1\}$. Let $\Delta(y, \hat{y}) = \mathbb{1}_{y \neq \hat{y}}$. If $g(\boldsymbol{x})$ is the score function in our binary classification setting, then define our compatibility function as

$$h(\boldsymbol{x}, 1) = \frac{g(\boldsymbol{x})}{2}, \qquad h(\boldsymbol{x}, -1) = -\frac{g(\boldsymbol{x})}{2}.$$

11. Show that for this choice of h, the multiclass hinge loss reduces to hinge loss, such that

$$\ell\left(h, (\boldsymbol{x}, y)\right) = \max_{y' \in \mathcal{Y}} \left(\Delta\left(y, y'\right)\right) + h(\boldsymbol{x}, y') - h(\boldsymbol{x}, y)\right) = \max\left\{0, 1 - yg(\boldsymbol{x})\right\}.$$

Proof. Assume that $y = 1$, we can compute that

$$\begin{aligned}\ell(h, (\boldsymbol{x}, y)) &= \max_{y' \in \mathcal{Y}} \left(\mathbb{1}_{y' \neq 1} + h(\boldsymbol{x}, y') - \frac{g(\boldsymbol{x})}{2}\right) \\ &= \max\left\{\frac{g(\boldsymbol{x})}{2} - \frac{g(\boldsymbol{x})}{2}, 1 - \frac{g(\boldsymbol{x})}{2} - \frac{g(\boldsymbol{x})}{2}\right\} = \max\left\{0, 1 - g(\boldsymbol{x})\right\}.\end{aligned} \tag{32}$$

On the other hand, if $y = -1$, we can compute that

$$\begin{aligned}\ell(h, (\boldsymbol{x}, y)) &= \max_{y' \in \mathcal{Y}} \left(\mathbb{1}_{y' \neq -1} + h(\boldsymbol{x}, y') + \frac{g(\boldsymbol{x})}{2}\right) \\ &= \max\left\{1 + \frac{g(\boldsymbol{x})}{2} + \frac{g(\boldsymbol{x})}{2}, -\frac{g(\boldsymbol{x})}{2} + \frac{g(\boldsymbol{x})}{2}\right\} = \max\left\{1 + g(\boldsymbol{x}), 0\right\}.\end{aligned} \tag{33}$$

Therefore, we can conclude that

$$\ell(h, (\boldsymbol{x}, y)) = \max\left\{0, 1 - yg(\boldsymbol{x})\right\}, \tag{34}$$

which means that the multiclass hinge loss reduces to hinge loss with this choice of h, thus the proof is complete. □

Implementation

In this problem we will work on a simple three-class classification example. The data is generated and plotted for you in the skeleton code.

One-vs-All (a.k.a. One-vs-Rest)

First we will implement one-vs-all multiclass classification. Our approach will assume we have a binary base classifier that returns a score, and we will predict the class that has the highest score.

12. Complete the methods `fit`, `decision_function` and `predict` from `OneVsAllClassifier` in the skeleton code. Following the `OneVsAllClassifier` code is a cell that extracts the results of the fit and plots the decision region. You can have a look at it first to make sure you understand how the class will be used.

    Solution. The class `OneVsAllClassifier` can be implemented as follows.

```python
from sklearn.base import BaseEstimator, ClassifierMixin, clone

class OneVsAllClassifier(BaseEstimator, ClassifierMixin):
    """The One-vs-all classifier.

    We assume that the classes will be the integers 0, ..., (n_classes - 1).
    We assume that the estimator provided to the class, after fitting, has a
    `decision_function` method that returns the score of the positive class.
    """
    def __init__(self, estimator, n_classes):
        """Initializes the object.

        Parameters
        ----------
        estimator : object
            A binary base classifier.
        n_classes : int
            The number of classes.
        """
        self.n_classes = n_classes
        self.estimators = [clone(estimator) for _ in range(n_classes)]
        self.fitted = False

    def fit(self, X, y=None):
        """Fits one classifier for each class.

        `self.estimators[i]` should be fit on class i versus the rest.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            The input data.
        y : array-like of shape (n_samples,)
            The class labels.

        Returns
        -------
        self : object
            The estimator itself.
        """
        for i in range(self.n_classes):
            # The ith estimator is for class i versus other classes
            # Where the class label is i, we treat it as 1 and otherwise 0
            self.estimators[i].fit(X, np.where(y == i, 1, 0))
        self.fitted = True
        return self

    def decision_function(self, X):
        """Returns the score of each input for each class.

        We assume that the given estimator implements `decision_function`
        method, and that the estimator has been fitted.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            The input data.

        Returns
        -------
        scores : np.ndarray of shape (n_samples, n_classes)
            The score of each input for each class.
        """
        if not self.fitted:
            raise RuntimeError(
                "You must train the classifier before predicting data."
            )
        if not hasattr(self.estimators[0], "decision_function"):
            raise AttributeError(
                "Base estimator does not support 'decision_function'."
            )
        scores = np.zeros((X.shape[0], self.n_classes))
        for i in range(self.n_classes):
            # The ith column of `scores` corresponds to the class i
            # The ith estimator is for class i versus other classes
            scores[:, i] = self.estimators[i].decision_function(X)
        return scores

    def predict(self, X):
        """Predicts the class with the highest score.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_features)
            The input data.

        Returns
        -------
        pred : np.ndarray of size (n_samples,)
            The predicted classes for each input.
        """
        scores = self.decision_function(X)
        # Example n corresponds to the nth row in `scores`
        # For each example, we pick the class with highest score
        return np.argmax(scores, axis=1)
```

13. Include the results of the test cell in your submission.

Solution. The resulting coefficients of each each classifier are as follows.

```
Coefs 0: [[-1.05852418 -0.90296449]]
Coefs 1: [[-0.26279342 -0.10322927]]
Coefs 2: [[ 0.89085129 -0.82461715]]
```

The confusion matrix is as follows.

```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]], dtype=int64)
```

The plot is shown as in Figure 1.

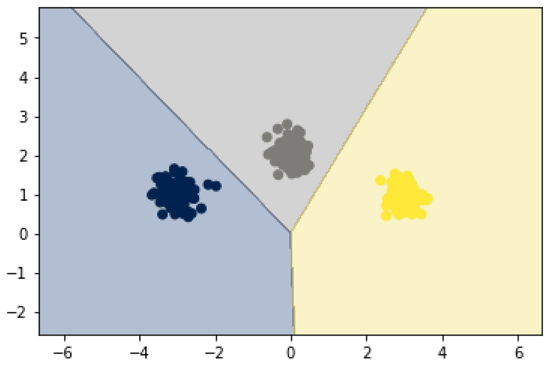


Figure 1: The classification result using `OneVsAllClassifier`.

Multiclass SVM

In this question, we will implement stochastic subgradient descent for the linear multiclass SVM, as described in class and in this problem set. We will use the class-sensitive feature mapping approach with the "multivector construction", as described in the multiclass lecture.

14. Complete the function `featureMap` in the skeleton code.

Solution. The function `featureMap` can be implemented as follows.

```python
def featureMap(X, y, n_classes):
    """Computes the class-sensitive features.

    Parameters
    ----------
    X : array-like of shape (n_samples, n_infeatures) or (n_infeatures,)
        The input features for the input fata.
    y : int
        The target class.

    Returns
    -------
    features : np.ndarray of size (n_samples, n_outfeatures) or
                (n_outfeatures,)
        The class sensitive features for class y.
    """
    one_sample = len(X.shape) == 1
    if one_sample:
        # Reshape to 2D if a 1D array is passed
        n_samples, n_infeatures = 1, X.shape[0]
        X = X.reshape(1, -1)
    else:
        n_samples, n_infeatures = X.shape
    features = np.zeros((n_samples, n_classes * n_infeatures))
    l, r = y * n_infeatures, (y + 1) * n_infeatures
    for n in range(n_samples):
        # Only the indices corresponding the target class are updated
        features[n, l:r] = X[n]
    # Reshape back to 1D array if a 1D array is passed originally
    return features if not one_sample else features.reshape(-1)
```

15. Complete the function `sgd`.

    Solution. The subgradient descent algorithm `sgd` can be implemented as follows.

```python
def sgd(X, y, n_outfeatures, subgd, eta=0.1, T=10000, rng=None):
    """Performs subgradient descent.

    Parameters
    ----------
    X : array-like of shape (n_samples, n_features)
        The input training data.
    y : array-like of shape (n_samples,)
        The class labels.
    n_outfeatures : int
        The number of class-sensitive features.
    subgd : function
        The subgradient taking (x, y, w) and returning the subgradient of
        the objective.
    eta : float
        The learning rate for the SGD.
    T : int
        The maximum number of iterations.
    rng : np.random.RandomState or None
        The random state for reproducing stochastic random behavior.

    Returns
    -------
    w : np.ndarray
        The resulting weight vector.
    """
    n_samples = X.shape[0]
    w = np.zeros(n_outfeatures)
    for _ in range(T):
        # Shuffle the arrays and always pick the first index
        if rng is None:
            indices = np.random.permutation(n_samples)
        else:
            indices = rng.permutation(n_samples)
        X, y = X[indices], y[indices]
        w -= eta * subgd(X[0], y[0], w)
    return w
```

16. Complete the methods `subgradient`, `decision_function` and `predict` from the estimator class `MulticlassSVM`.

    Solution. The class `MulticlassSVM` can be implemented as follows.

```python
class MulticlassSVM(BaseEstimator, ClassifierMixin):
    """The Multiclass SVM estimator."""
    def __init__(
            self,
            n_outfeatures,
            lam=1.0,
            n_classes=3,
            Delta=zeroOne,
            Psi=featureMap,
            random_state=0,
        ):
        """Initializes the estimator.

        Parameters
        ----------
        n_outfeatures : int
            The number of class-sensitive features produced by `Psi`.
        lam : float
            The l2 regularization parameter.
        n_classes : int
            The number of classes, with labels 0, ..., (n_classes - 1).
        Delta : function
            The class-sensitive loss function taking two labels and
            returning the loss.
        Psi : function
            The class-sensitive feature mapping taking x and y and
            returning the feature map.
        """
        self.n_outfeatures = n_outfeatures
        self.lam = lam
        self.n_classes = n_classes
        self.Delta = Delta
        self.Psi = lambda X, y: Psi(X, y, n_classes)
        self.rng = np.random.RandomState(random_state)
        self.fitted = False

    def subgradient(self, x, y, w):
        """The subgradient evaluated at x, y, w.

        Parameters
        ----------
        x : array-like of shape (n_infeatures,)
            The sample input.
        y : int
            The sample class.
        w : array-like of shape (n_infeatures,)
            The parameter vector.

        Returns
        -------
        subgradient : np.array of shape (n_infeatures,)
            The subgradient vector evaluated at x, y given w.
        """
        # Look for the optimal hat{y}
        target, cur = 0, self._subgradient_argmax(x, y, 0, w)
        for cls in range(1, self.n_classes):
            new = self._subgradient_argmax(x, y, cls, w)
            if new > cur:
                target, cur = cls, new
        return 2 * self.lam * w + self.Psi(x, target) - self.Psi(x, y)

    def _subgradient_argmax(self, x, y, target, w):
        """Helper for `self.subgradient`."""
        return self.Delta(y, target) \
            + np.dot(w, self.Psi(x, target) - self.Psi(x, y))

    def fit(self, X, y, eta=0.1, T=10000):
        """Fits the estimator.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_infeatures)
            The input training data.
        y : array-like of shape (n_samples,)
            The input classess.
        eta : float
            The learning rate of the SGD.
        T : int
            The maximum number of iterations.

        Returns
        -------
        self : object
            The estimator itself.
        """
        self.coef_ = sgd(
            X, y, self.n_outfeatures, self.subgradient, eta, T, self.rng
        )
        self.fitted = True
        return self

    def decision_function(self, X):
        """Returns the score of each input for each class.

        Assume that the estimator has been fitted.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_infeatures)
            The input data.

        Returns
        -------
        scores : np.ndarray of shape (n_samples, n_classes)
            The score of each input for each class.
        """
        if not self.fitted:
            raise RuntimeError(
                "You must train the classifier before predicting data."
            )
        scores = np.zeros((X.shape[0], self.n_classes))
        for i in range(self.n_classes):
            # The ith column of `scores` corresponds to the class i
            scores[:, i] = np.dot(self.Psi(X, i), self.coef_)
        return scores

    def predict(self, X):
        """Predicts the class with the highest score.

        Parameters
        ----------
        X : array-like of shape (n_samples, n_infeatures)
            The input data to predict.

        Returns
        -------
        pred : np.ndarray of shape (n_samples,)
            The class labels predicted for each example.
        """
        scores = self.decision_function(X)
        # Example n corresponds to the nth row in `scores`
        # For each example, we pick the class with highest score
        return np.argmax(scores, axis=1)
```

17. Following the multiclass SVM implementation, we have included another block of test code. Make sure to include the results from these tests in your assignment, along with your code.

Solution. The resulting coefficients are as follows.

```
[-0.33354761 -0.05890081 -0.01473713  0.10689075  0.34828473 -0.04798994]
```

The confusion matrix is as follows.

```
array([[100,   0,   0],
       [  0, 100,   0],
       [  0,   0, 100]], dtype=int64)
```

The plot is shown as in Figure 2.

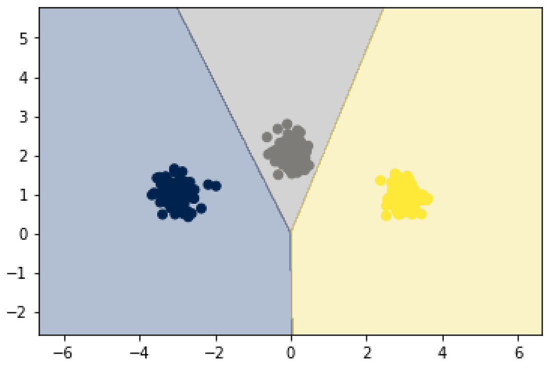


Figure 2: The classification result using `MulticlassSVM`.

Homework 6: Decision Trees and Boosting

Due: Wednesday, April 19th, 2023 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g.LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better. **The optional problems should not take you too much time and help you navigate the material, consider taking a shot at them.**

Decision Tree Implementation

In this problem we'll implement decision trees for both classification and regression. The strategy will be to implement a generic class, called `Decision_Tree`, which we'll supply with the loss function we want to use to make node splitting decisions, as well as the estimator we'll use to come up with the prediction associated with each leaf node. For classification, this prediction could be a vector of probabilities, but for simplicity we'll just consider hard classifications here. We'll work with the classification and regression data sets from previous assignments.

1. Complete the `compute_entropy` and `compute_gini` functions.

   Solution. The function `compute_entropy` can be implemented as follows.

```python
def compute_entropy(y_label):
    """Compute the entropy of a label list.

    Parameters
    ----------
    y_label : np.ndarray of shape (n_samples, 1)
        The label list to compute entropy of.

    Returns
    -------
    entropy : float
        The entropy of the given label list.
    """
    n_samples, _ = y_label.shape
    # Obtain the prevalence of each label
    _, counts = np.unique(y_label, return_counts=True)
    prevalence = counts / n_samples
    # Compute the entropy by -sum (p * log(p))
    return -np.sum(prevalence * np.log(prevalence))
```

   The function `compute_gini` can be implemented as follows.

```python
def compute_gini(y_label):
    """Compute the Gini index of a label list.

    Parameters
    ----------
    y_label : np.ndarray of shape (n_samples, 1)
        The label list to compute gini index of.

    Returns
    -------
    gini : float
        The gini index of the given label list.
    """
    n_samples, _ = y_label.shape
    # Obtain the prevalence of each label
    _, counts = np.unique(y_label, return_counts=True)
    prevalence = counts / n_samples
    # Compute the Gini index by sum (p * (1-p))
    return np.sum(prevalence * (1 - prevalence))
```

2. Complete the class `Decision_Tree`, given in the skeleton code. The intended implementation is as follows: Each object of type `Decision_Tree` represents a single node of the tree. The depth of that node is represented by the variable `self.depth`, with the root node having depth 0. The main job of the `fit` function is to decide, given the data provided, how to split the node or whether it should remain a leaf node. If the node will split, then the splitting feature and splitting value are recorded, and the left and right subtrees are fit on the relevant portions of the data. Thus tree-building is a recursive procedure. We should have as many `Decision_Tree` objects as there are nodes in the tree. We will not implement pruning here. Some additional details are given in the skeleton code.

   Solution. The class `Decision_Tree` can be implemented as follows.

```
class Decision_Tree(BaseEstimator):

    def __init__(
        self,
        split_loss_function,
        leaf_value_estimator,
        depth=0,
        min_sample=5,
        max_depth=10,
    ):
        """Initialize a Decision Tree.

        Parameters
        ----------
        split_loss_function : callable
            A function taking y_label and returning the loss.
        leaf_value_estimator : callable
            A function taking y_label and estimating the leaf value.
        depth : int
            The depth indicator. 0 represents the root node.
        min_sample : int
            If an internal node has no more than this many sample points,
            it cannot be splitted.
        max_depth : int
            The maximum depth of the decision tree.
        """
        self.split_loss_function = split_loss_function
        self.leaf_value_estimator = leaf_value_estimator
        self.depth = depth
        self.min_sample = min_sample
        self.max_depth = max_depth
        self.is_leaf = False

    def fit(self, X, y):
```

```python
        """Fit the decision tree.

        This is a recursive tree building procedure.

        Parameters
        ----------
        X : np.ndarray of shape (n_samples, n_features)
            The training data.
        y : np.ndarray of shape (n_samples, 1)
            The labels of the training samples.

        Returns
        -------
        self : object
            Returns the fitted decision tree itself.
        """
        n_samples, n_features = X.shape
        # If we have reached the maximum depth or we cannot split the node
        # Mark as leaf node and compute the prediction value
        if self.depth == self.max_depth or n_samples <= self.min_sample:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self
        # Compute the loss of not splitting at all as baseline
        split_feature, split_index, cur_loss = \
            None, None, self.split_loss_function(y)
        for i in range(n_features):
            # For each feature, try to split into xi <= t and xi > t
            # Sort the label list based on the ith feature of X
            indices = np.argsort(X[:, i])
            for n in range(1, n_samples):
                # Pick a split index and check the weighted loss
                # If better then baseline, then update split info
                y_left, y_right = y[indices[:n]], y[indices[n:]]
                new_loss = (
                    self.split_loss_function(y_left) * len(y_left) \
                    + self.split_loss_function(y_right) * len(y_right)
                ) / n_samples
                if new_loss < cur_loss:
                    split_feature, split_index, cur_loss = i, n, new_loss
        # If none of the splits is better than baseline
        # Mark as leaf node and compute the prediction value
        if split_feature is None:
            self.is_leaf = True
            self.value = self.leaf_value_estimator(y)
            return self
        # Set the attributes split_id and split_value
        indices = np.argsort(X[:, split_feature])
        self.split_id, self.split_value = \
            split_feature, X[indices[split_index], split_feature]
        # Create left and right child nodes, and recursively fit subtrees
        self.left = Decision_Tree(
            self.split_loss_function,
            self.leaf_value_estimator,
            depth=self.depth + 1,
            min_sample=self.min_sample,
            max_depth=self.max_depth,
        )
        self.right = Decision_Tree(
            self.split_loss_function,
            self.leaf_value_estimator,
            depth=self.depth + 1,
            min_sample=self.min_sample,
            max_depth=self.max_depth,
        )
        self.left.fit(X[indices[:split_index]], y[indices[:split_index]])
        self.right.fit(X[indices[split_index:]], y[indices[split_index:]])
        return self

    def predict_instance(self, instance):
        """Predict the label according to the decision tree.

        Parameters
        ----------
        instance : np.ndarray of shape (1, n_features)
            The new data to predict.

        Returns
        -------
        pred : object
            The prediction made by the corresponding child node.
        """
        if self.is_leaf:
            return self.value
        if instance[self.split_id] <= self.split_value:
            return self.left.predict_instance(instance)
        else:
            return self.right.predict_instance(instance)
```

3. Run the code provided that builds trees for the two-dimensional classification data. Include the results. For debugging, you may want to compare results with sklearn's decision tree (code provided in the skeleton code). For visualization, you'll need to install `graphviz`.

   Solution. The plot of the classification results using the classification tree estimator implemented with `Decision_Tree` and with different maximum depths is shown as in Figure 1.

4. Complete the function `mean_absolute_deviation_around_median` (MAE). Use the code provided to fit the `Regression_Tree` to the krr dataset using both the MAE loss and median predictions. Include the plots for the 6 fits.

   Solution. The function `mean_absolute_deviation_around_median` can be implemented as follows.

```
def mean_absolute_deviation_around_median(y):
    """Compute the mean absolute deviation around median.

    Parameters
    ----------
    y : np.ndarray of shape (n_samples, 1)
        The target array to compute MAE of.

    Returns
    -------
    mae : float
        The mean absolute deviation around median.
    """
    med = np.median(y)
    return np.mean(np.abs(y - med))
```

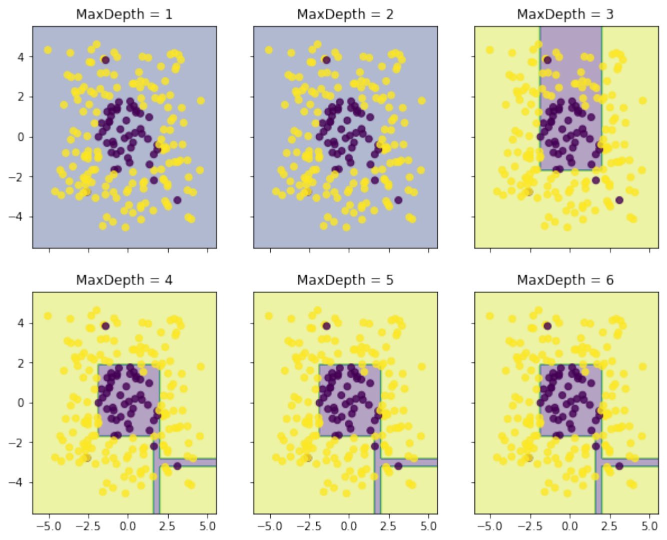


Figure 1: The classification results using the classification tree estimator implemented with the `Decision_Tree` class and with maximum depths ranging from 1 to 6.

The plot of the regression results using the regression tree estimator implemented with `Decision_Tree` and with different maximum depths is shown as in Figure 2.

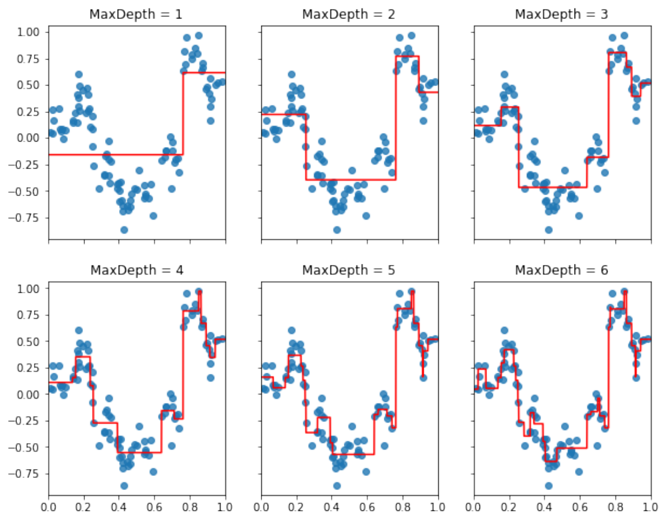


Figure 2: The regression results using the regression tree estimator implemented with the `Decision_Tree` class and with maximum depths ranging from 1 to 6.

Ensembling

Recall the general gradient boosting algorithm, for a given loss function ℓ and a hypothesis space $\mathcal{F}$ of regression functions (i.e. functions mapping from the input space to $\mathbb{R}$):

```
1: Initialize $f_0(x) = 0$;
2: for $m = 1$ to $M$ do
3:     Compute
           $$\boldsymbol{g}_m = \left( \frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i)\right) \right)_{j=1}^{n};$$
4:     Fit regression model to $-\boldsymbol{g}_m$, such that
           $$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left( (-\boldsymbol{g}_m)_i - h(x_i) \right)^2;$$
5:     Choose fixed step size $\nu_m = \nu = (0, 1]$, or take
           $$\nu_m = \arg\min_{\nu > 0} \sum_{i=1}^{n} \ell\left(y_i, f_{m-1}(x_i) + \nu h_m(x_i)\right);$$
6:     Take the step $f_m(x) = f_{m-1}(x) + \nu_m h_m(x)$;
7: end for
8: return $f_M$;
```

This method goes by many names, including gradient boosting machines (GBM), generalized boosting models (GBM), AnyBoost, and gradient boosted regression trees (GBRT), among others. One of the nice aspects of gradient boosting is that it can be applied to any problem with a subdifferentiable loss function.

Gradient Boosting Regression Implementation

First we'll keep things simple and consider the standard regression setting with square loss. In this case the we have $\mathcal{Y} = \mathbb{R}$, our loss function is given by $\ell(\hat{y}, y) = (\hat{y} - y)^2/2$, and at the mth round of gradient boosting, we have

$$h_m = \arg\min_{h \in \mathcal{F}} \sum_{i=1}^{n} \left((y_i - f_{m-1}(x_i)) - h(x_i) \right)^2.$$

5. Complete the `Gradient_Boosting` class. As the base regression algorithm to compute the argmin, you should use sklearn's regression tree. You should use the square loss for the tree splitting rule (`criterion` keyword argument) and use the default sklearn leaf prediction rule from the `predict` method[1]. We will also use a constant step size ν.

   Solution. The class `Gradient_Boosting` can be implemented as follows.

[1]The sklearn `DecisionTreeRegressor` documentation. Examples of usage are provided in the skeleton code.

```python
class Gradient_Boosting:

    def __init__(
        self,
        n_estimator,
        pseudo_residual_func,
        learning_rate=0.01,
        min_sample=5,
        max_depth=5,
    ):
        """Initialize a gradient boosting machine.

        Parameters
        ----------
        n_estimator : int
            The number of estimators, i.e., the number of rounds of boosting
    .
        pseudo_residual_func : callable
            The function taking true and pred labels and returning the
    pseudo-residual.
        learning_rate : float
            The learning rate, i.e., the step size of gradient descent.
        min_sample : int
            If an internal node has no more than this many sample points,
            it cannot be splitted.
        max_depth : int
            The maximum depth of the decision tree.
        """
        self.n_estimator = n_estimator
        self.pseudo_residual_func = pseudo_residual_func
        self.learning_rate = learning_rate
        self.min_sample = min_sample
        self.max_depth = max_depth
        self.estimators = []
        self.fitted = False

    def fit(self, X, y):
        """Fit the gradient boosting machine.

        Parameters
        ----------
        X : np.ndarray of shape (n_samples, n_features)
            The training data.
        y : np.array of shape (n_samples,)
            The training targets.

        Returns
        -------
        self : object
            Returns the fitted gradient boosting machine itself.
        """
        n_samples, _ = X.shape
        # The prediction function depends only on the training points
        # Therefore, we characterize it via current predictions
        # The prediction function is initialized as constant zero
        cur_preds = np.zeros(n_samples)
        for _ in range(self.n_estimator):
            # Compute the pseudo-residuals, i.e., negative gradients
            residuals = self.pseudo_residual_func(y, cur_preds)
            # Fit a base learner with squared loss using x and residuals
            # We will use the sklearn decision tree regressor, using
            # square loss as the splitting rule (by default `criterion`)
            base_learner = DecisionTreeRegressor(
                min_samples_split=self.min_sample, max_depth=self.max_depth
            )
            base_learner.fit(X, residuals)
            # Update the prediction function via updating the predictions
            # at the training points.
            cur_preds += self.learning_rate * base_learner.predict(X)
            self.estimators.append(base_learner)
        self.fitted = True
        return self

    def predict(self, X):
        """Make prediction using the trained gradient boosting machine.

        Parameters
        ----------
        X : np.ndarray of shape (n_samples, n_features)
            The data for to make predictions on.

        Returns
        -------
        pred : np.array
            The predictions.
        """
        if not self.fitted:
            raise Exception("Model has to be trained before predicting")
        n_samples, _ = X.shape
        # The prediction should be made by the final prediction function
        # but we characterized it only on the training data when fitting
        # We need to retrieve the prediction function function via the
        # trained estimators, but making predictions on the given data
        pred = np.zeros(n_samples)
        for base_learner in self.estimators:
            pred += self.learning_rate * base_learner.predict(X)
        return pred
```

6. Run the code provided to build gradient boosting models on the regression data sets `krr-train.txt`, and include the plots generated. For debugging you can use the sklearn implementation of `GradientBoostingRegressor`[2].

   Solution. The plot of the regression results using the gradient boosting machine with different numbers of estimators (rounds of boosting) is shown as in Figure 3.

Classification of images with Gradient Boosting

In this problem we will consider the classification of MNIST, the dataset of handwritten digits images, with ensembles of trees. For simplicity, we only retain the "0" and "1" examples and perform binary classification. First we'll derive a special case of the general gradient boosting framework: BinomialBoost. Let's consider the classification framework, where $\mathcal{Y} = \{-1, 1\}$. In lecture, we noted that AdaBoost corresponds to forward stagewise additive modeling with the exponential loss, and that the exponential loss is not very robust to outliers (i.e. outliers can have a large effect on the final prediction function). Instead, let's consider the logistic loss

$$\ell(m) = \ln\left(1 + e^{-m}\right),$$

where $m = yf(\boldsymbol{x})$ is the margin.

[2]The sklearn `GradientBoostingRegressor` documentation.

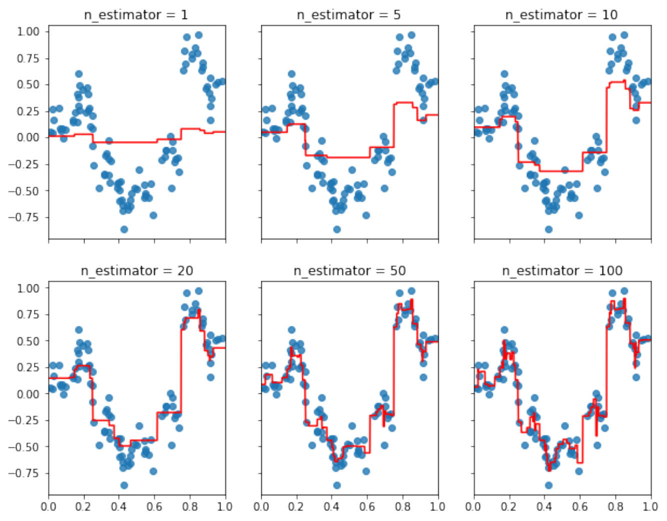


Figure 3: The regression results using the gradient boosting machine with different numbers of estimators (rounds of boosting) ranging from 1 to 100.

7. Give the expression of the negative gradient step direction, or pseudo residual, $-\boldsymbol{g}_m$ for the logistic loss as a function of the prediction function f_{m-1} at the previous iteration and the dataset points $\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$. What is the dimension of $\boldsymbol{g}_m$?

   Solution. We can compute the pseudo-residual for the jth example as

$$\begin{aligned} r_{m,j} &= \frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^n \ell\left(y_i, f_{m-1}(x_i)\right) = \frac{\partial}{\partial f_{m-1}(x_j)} \sum_{i=1}^n \ln\left(1 + \exp\left(-y_i f_{m-1}(x_i)\right)\right) \\ &= \frac{\partial \ln\left(1 + \exp\left(-y_j f_{m-1}(x_j)\right)\right)}{\partial f_{m-1}(x_j)} = \frac{y_j \exp\left(-y_j f_{m-1}(x_j)\right)}{1 + \exp\left(-y_j f_{m-1}(x_j)\right)} = \frac{y_j}{1 + \exp\left(y_j f_{m-1}(x_j)\right)}. \end{aligned} \tag{1}$$

   Therefore, the negative gradient direction $\boldsymbol{g}_m$ would be given by

$$-\boldsymbol{g}_m = -\left(r_{m,j}\right)_{j=1}^n = -\left(\frac{y_j}{1 + \exp\left(y_j f_{m-1}(x_j)\right)}\right)_{j=1}^n, \tag{2}$$

   thus of dimension n.

8. Write an expression for h_m as an argmin over functions h in $\mathcal{F}$.

   Solution. The basis function h_m is obtained simply by fitting a regression model to $-\boldsymbol{g}_m$ (in the l^2 sense), thus is given by

$$h_m = \underset{h \in \mathcal{F}}{\arg\min} \sum_{i=1}^n \left((-\boldsymbol{g}_m)_i - h(x_i)\right)^2 = \underset{h \in \mathcal{F}}{\arg\min} \sum_{i=1}^n \left(\frac{-y_i}{1 + \exp\left(y_i f_{m-1}(x_i)\right)} - h(x_i)\right)^2. \tag{3}$$

9. Load the MNIST dataset using the helper preprocessing function in the skeleton code.Using the scikit learn implementation of `GradientBoostingClassifier`, with the logistic loss (`loss="deviance"`) and trees of maximum depth 3, fit the data with 2, 5, 10, 100 and 200 iterations (estimators). Plot the train and test accuracy as a function of the number of estimators.

   Solution. Using the sklearn implementation of the gradient boosting classifier with maximum depth 3, the change of training and testing accuracies with respect to the number of estimators (the number of boosting rounds) is shown as in Figure 4.

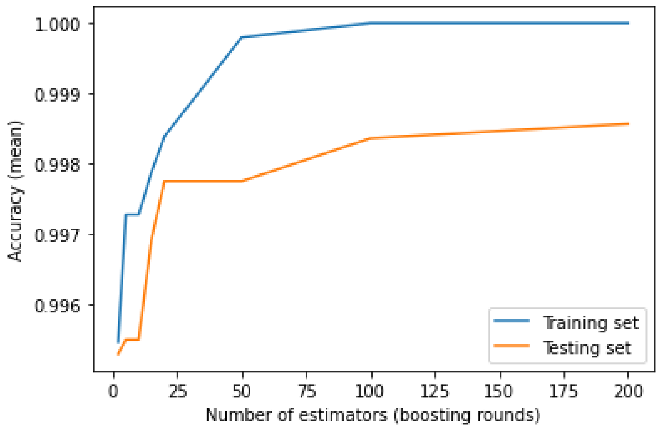


Figure 4: The change of training and testing accuracies using `GradientBoostingClassifier` with maximum depth 3, with respect to the number of estimators (the number of boosting rounds), ranging from 2 estimators to 200 estimators.

Classification of images with Random Forests (Optional)

10. Another type of ensembling method we discussed in class are random forests. Explain in your own words the construction principle of random forests.

    Solution. As in bagging, we draw bootstrap samples and treat them as independent, then build a collection of decision trees independently. However, we modify the tree-growing procedure by adding an additional layer of randomness, that is, when constructing each tree node, we restrict the choice of the splitting variable to a randomly chosen subset of features. This would prevent situations in which all trees are dominated by the same small number of strong features and therefore too similar to each other, thus mitigating overfitting and improving diversity.

11. Using the scikit learn implementation of `RandomForestClassifier`[3], with the entropy loss (`criterion="entropy"`) and trees of maximum depth 3, fit the preprocessed binary MNIST dataset with 2, 5, 10, 50, 100 and 200 estimators.

    Solution. Using the sklearn implementation of the random forest classifier with maximum depth 3, the change of training and testing accuracies with respect to the number of estimators is shown as in Figure 5.

[3]The sklearn `RandomForestClassifier` documentation.

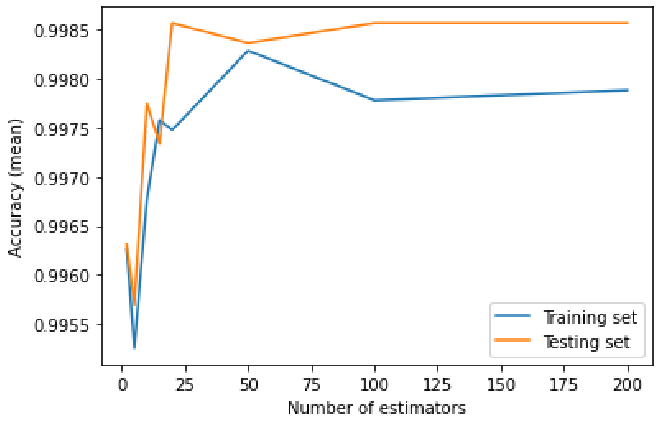


Figure 5: The change of training and testing accuracies using `RandomForestClassifier` with maximum depth 3, with respect to the number of estimators, ranging from 2 estimators to 200 estimators.

12. What general remark can you make on overfitting for Random Forests and Gradient Boosted Trees? Which method achieves the best train accuracy overall? Is this result expected? Can you think of a practical disadvantage of the best performing method? How do the algorithms compare in term of test accuracy?

    Solution. As for the **overfitting problem**, the random forests are prone to overfitting while the gradient boosting trees are more robust. As we can see from Figure 4 using the gradient boosting trees, though the testing set accuracy does not increase a lot as the number of boosting rounds increases, it does not drop either. However, as is shown in Figure 5 using the random forests, the testing set accuracy is the best with approximately 50 estimators, but increasing the number of estimators does not result in any improvement and even a drop. As for the **training set accuracy**, the gradient booting trees achieves overall the best training set accuracy, with a perfect score achieved at 100 and 200 estimators. This is expected since gradient boosting trees are typically known for their high accuracy and ability to fit complex datasets. In contrast, though the random forests also achieve high training set accuracy ($> 99.5\%$), it does not achieve perfect training set accuracy, possibly because the dataset is very noisy or contains outliers. A **practical disadvantage** of the gradient boosting trees, however, is that it is computationally expensive especially on large datasets. For instance on the MNIST dataset, using random forests with 200 estimators took me no more than 3 seconds, but using gradient boosting trees with 200 estimators took me over half a minute. As for the **testing set accuracy**, both algorithms achieve high testing set accuracy, with one outperforming the other in some cases and vice versa in the others. Yet this is not a general conclusion, since overfitting problems may arise using a different dataset.

Homework 7: Computation Graphs, Back-propagation, and Neural Networks

Due: Wednesday, May 3rd, 2023 at 11:59PM EST

Instructions: Your answers to the questions below, including plots and mathematical work, should be submitted as a single PDF file. It's preferred that you write your answers using software that typesets mathematics (e.g. LaTeX, LyX, or MathJax via iPython), though if you need to you may scan handwritten work. You may find the minted package convenient for including source code in your LaTeX document. If you are using LyX, then the listings package tends to work better.

Introduction

There is no doubt that neural networks are a very important class of machine learning models. Given the sheer number of people who are achieving impressive results with neural networks, one might think that it's relatively easy to get them working. This is a partly an illusion. One reason so many people have success is that, thanks to GitHub, they can copy the exact settings that others have used to achieve success. In fact, in most cases they can start with "pre-trained" models that already work for a similar problem, and "fine-tune" them for their own purposes. It's far easier to tweak and improve a working system than to get one working from scratch. If you create a new model, you're kind of on your own to figure out how to get it working: there's not much theory to guide you and the rules of thumb do not always work. Understanding even the most basic questions, such as the preferred variant of SGD to use for optimization, is still a very active area of research.

One thing is clear, however: If you do need to start from scratch, or debug a neural network model that doesn't seem to be learning, it can be immensely helpful to understand the low-level details of how your neural network works – specifically, back-propagation. With this assignment, you'll have the opportunity to linger on these low-level implementation details. Every major neural network type (RNNs, CNNs, Resnets, etc.) can be implemented using the basic framework we'll develop in this assignment.

To help things along, Philipp Meerkamp, Pierre Garapon, and David Rosenberg have designed a minimalist framework for computation graphs and put together some support code. The intent is for you to read, or at least skim, every line of code provided, so that you'll know you understand all the crucial components and could, in theory, create your own from scratch. In fact, creating your own computation graph framework from scratch is highly encouraged – you'll learn a lot.

Computation Graph Framework

To get started, please read the tutorial on the computation graph framework we'll be working with. (Note that it renders better if you view it locally.) The use of computation graphs is not specific to machine learning or neural networks. Computation graphs are just a way to represent a function that facilitates efficient computation of the function's values and its gradients with respect to inputs. The tutorial takes this perspective, and there is very little in it about machine learning, per se.

To see how the framework can be used for machine learning tasks, we've provided a full implementation of linear regression. You should start by working your way through the `__init__` of the `LinearRegression` class in `linear_regression.py`. From there, you'll want to review the node class definitions in `nodes.py`, and finally the class `ComputationGraphFunction` in `graph.py`. `ComputationGraphFunction` is where we repackage a raw computation graph into something that's more friendly to work with for machine learning. The rest of `linear_regression.py` is fairly routine, but it illustrates how to interact with the `ComputationGraphFunction`.

As we've noted earlier in the course, getting gradient calculations correct can be difficult. To help things along, we've provided two functions that can be used to test the backward method of a node and the overall gradient calculation of a `ComputationGraphFunction`. The functions are in `test_utils.py`, and it's recommended that you review the tests provided for the linear regression implementation in `linear_regression.t.py`. (You can run these tests from the command line with `python3 linear_regression.t.py`.) The functions actually doing the testing, `test_node_backward` and `test_ComputationGraphFunction`, may seem a bit intricate, but they're implementing the exact same `gradient_checker` logic we saw in the second homework assignment.

Once you've understood how linear regression works in our framework, you're ready to start implementing your own algorithms. To help you get started, please make sure you are able to execute the following commands:

- `cd /path/to/hw7`
- `python3 linear_regression.py`
- `python3 linear_regression.t.py`

Ridge Regression

When moving to a new system, it's always good to start with something familiar. But that's not the only reason we're doing ridge regression in this homework. In ridge regression the parameter vector is "shared", in the sense that it's used twice in the objective function. In the computation graph, this can be seen in the fact that the node for the parameter vector has two outgoing edges. This sharing is common in many popular neural networks (RNNs and CNNs), where it is often referred to as *parameter tying*. `ridge_regression.py` provides a skeleton code and `ridge_regression.t.py` is a test code, which you should eventually be able to pass.

1. Complete the class `L2NormPenaltyNode` in `nodes.py`. If your code is correct, you should be able to pass `test_L2NormPenaltyNode` in `ridge_regression.t.py`. Please attach a screenshot that shows the test results for this question.

   Solution. The class `L2NormPenaltyNode` can be implemented as follows.

```python
class L2NormPenaltyNode(object):
    def __init__(self, l2_reg, w, node_name):
        """Initialize a node computing l2 penalty.

        Parameters
        ----------
        l2_reg : np.array of size (1,)
            The l2 regularization parameter, with value being nonnegative.
        w : node object with w.out being np.array
```

```
            Reference to the node providing input to this node.
        node_name : str
            The name of this node.

        Attributes
        ----------
        out : np.array
            The output of this node.
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        self.l2_reg = np.array(l2_reg)
        self.w = w
        self.node_name = node_name
        # Setting additional attributes as documented
        self.out = None
        self.d_out = None

    def forward(self):
        """Forwards one step.

        This sets `out` of the current node, so we move forward one step.
        It also initializes `d_out` to be updated in the backward round.
        It returns the output of this node which is to be computed.

        Returns
        -------
        out : np.array
            The output of this node.
        """
        # The output of w is the input of the current node
        # The current node then outputs lambda * norm(w)^2
        self.out = self.l2_reg * (np.linalg.norm(self.w.out) ** 2)
        # The partial derivative should have the same dimension as the
        # output of this node (with respect to which derivative is taken)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        """Backwards one step.

        This sets `d_out` of its predecessor providing input to this node.
        It returns partial derivative of the graph output (i.e., objective)
        with respect to this node which is computed by its successor.

        Returns
        -------
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        # By chain rule, we can compute pJ/pin = pJ/pout * pout/pin
        # pJ/pout is just `d_out` of the current node
        # pout/pin is 2 * lambda * w
        self.w.d_out += self.d_out * (2 * self.l2_reg * self.w.out)
        return self.d_out

    def get_predecessors(self):
        """Gets the predecessors.

```

```python
        Returns
        -------
        predecessors : list (of nodes)
            The list of predecessors, i.e., the nodes that provide input to
            the current node. Normally the l2 regularization penalty node
            has only one predecessor.
        """
        return [self.w]
```

The result of running `test_L2NormPenaltyNode` is shown as follows.

```
> python ridge_regression.t.py TestAll.test_L2NormPenaltyNode
DEBUG: (Node l2 norm node) Max rel error for partial deriv w.r.t. w is
1.7066627461369896e-08.
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

2. Complete the class `SumNode` in `nodes.py`. If your code is correct, you should be able to pass `test_SumNode` in `ridge_regression.t.py`. Please attach a screenshot that shows the test results for this question.

Solution. The class `SumNode` can be implemented as follows.

```python
class SumNode(object):
    def __init__(self, a, b, node_name):
        """Initialize a node computing l2 penalty.

        Parameters
        ----------
        a : node object with a.out being np.array
            Reference to one node providing input to this node.
        b : node object with b.out being np.array
            Reference to the other node providing input to this node.
            It must have the same shape as a.
        node_name : str
            The name of this node.

        Attributes
        ----------
        out : np.array
            The output of this node.
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        self.a = a
        self.b = b
        self.node_name = node_name
        # Setting additional attributes as documented
        self.out = None
        self.d_out = None

    def forward(self):
        """Forwards one step.

        This sets `out` of the current node, so we move forward one step.
        It also initializes `d_out` to be updated in the backward round.
        It returns the output of this node which is to be computed.

        Returns
        -------
        out : np.array
            The output of this node.
        """
        # The outputs of a and b are the inputs of the current node
        # The current node then outputs a + b
        self.out = self.a.out + self.b.out
        # The partial derivative should have the same dimension as the
        # output of this node (with respect to which derivative is taken)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        """Backwards one step.

        This sets `d_out` of its predecessors providing inputs to this node.
        It returns partial derivative of the graph output (i.e., objective)
        with respect to this node which is computed by its successor.

        Returns
        -------
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        # By chain rule, we can compute pJ/pain = pJ/pout * pout/pain
        # pJ/pout is just `d_out` of the current node
        # pout/pain is 1
        self.a.d_out += self.d_out
        # The case for b is completely symmetric to that for a
        self.b.d_out += self.d_out
        return self.d_out

    def get_predecessors(self):
        """Gets the predecessors.

        Returns
        -------
        predecessors : list (of nodes)
            The list of predecessors, i.e., the nodes that provide input to
            the current node. Normally the sum node has two predecessors.
        """
        return [self.a, self.b]
```

The result of running `test_SumNode` is shown as follows.

```
> python ridge_regression.t.py TestAll.test_SumNode
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. a is
1.6365788753567756e-09.
DEBUG: (Node sum node) Max rel error for partial deriv w.r.t. b is
1.6365788753567756e-09.
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

3. Implement ridge regression with w regularized and b unregularized. Do this by completing the `__init__` method in `ridge_regression.py`, using the classes created above. When complete, you should be able to pass the tests in `ridge_regression.t.py`. Report the average square error on the **training** set for the parameter settings given in the `main()` function.

   Solution. The `__init__` method of the class `RidgeRegression` can be implemented as follows. It constructs and stores the `ComputationGraphFunction` instance in `self.graph`.

```python
class RidgeRegression(BaseEstimator, RegressorMixin):
    def __init__(self, l2_reg=1, step_size=5e-3, max_num_epochs=5000):
        """Initializes a ridge regression object with computation graph.

        We want to build a computation graph for the ridge regression
        objective :math:`J = lambda * norm(w)^2 + |(w^Tx + b) - y|^2`.

        Parameters
        ----------
        l2_reg : real
            The l2 regularization parameter, which should be nonnegative.
        step_size : real
            The learning rate, which should be nonnegative.
        max_num_epochs : integral
            The maximum number of epochs, which should be nonnegative.

        Attributes
        ----------
        graph : ComputationGraphFunction object
            The computation graph instance for ridge regression.
        """
        self.l2_reg = l2_reg
        self.step_size = step_size
        self.max_num_epochs = max_num_epochs

        # Constructing the input nodes
        # The input vector x
        self._x = nodes.ValueNode(node_name="x")
        # The response scalar y
        self._y = nodes.ValueNode(node_name="y")

        # Constructing the parameter nodes
        # The parameter vector w
        self._w = nodes.ValueNode(node_name="w")
        # The scalar bias parameter b
        self._b = nodes.ValueNode(node_name="b")

        # Constructing the interim computation nodes
        # Compute :math:`hat{y} = w^Tx + b`
        self._prediction = nodes.VectorScalarAffineNode(
            x=self._x, w=self._w, b=self._b, node_name="prediction"
        )
        # Compute :math:`l = |hat{y} - y|^2`
        self._loss = nodes.SquaredL2DistanceNode(
            a=self._prediction, b=self._y, node_name="loss",
        )
        # Compute :math:`r = lambda * norm(w)^2`
        self._reg = nodes.L2NormPenaltyNode(
            l2_reg=self.l2_reg, w=self._w, node_name="regularization",
        )
        # Compute :math:`J = r + l`
        self._objective = nodes.SumNode(
            a=self._reg, b=self._loss, node_name="objective"
        )

        # Constructing the computation graph
        self.graph = graph.ComputationGraphFunction(
            inputs=[self._x],
            outcomes=[self._y],
            parameters=[self._w, self._b],
            prediction=self._prediction,
            objective=self._objective
        )
```

The result of running `test_ridge_regression_gradient` is shown as follows.

```
> python ridge_regression.t.py TestAll.test_ridge_regression_gradient
DEBUG: (Parameter w) Max rel error for partial deriv
2.0535835581448403e-10.
DEBUG: (Parameter b) Max rel error for partial deriv
1.4230151508246912e-10.
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

Using regularization parameter $\lambda = 1$ with step size 5×10^{-5} and 2000 epochs, the average square error on the training set is 0.20166182530478097. Using regularization parameter $\lambda = 0$ with step size 5×10^{-4} and 500 epochs, the average square error on the training set is 0.042898700925240965. The regression results are shown as in Figure 1.

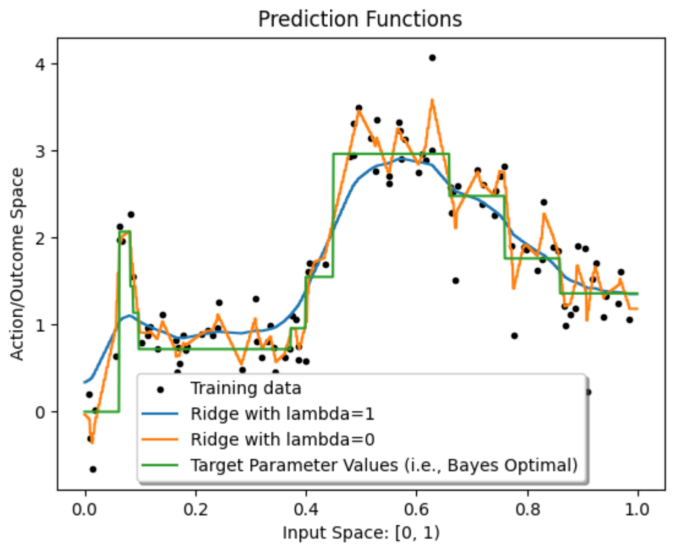
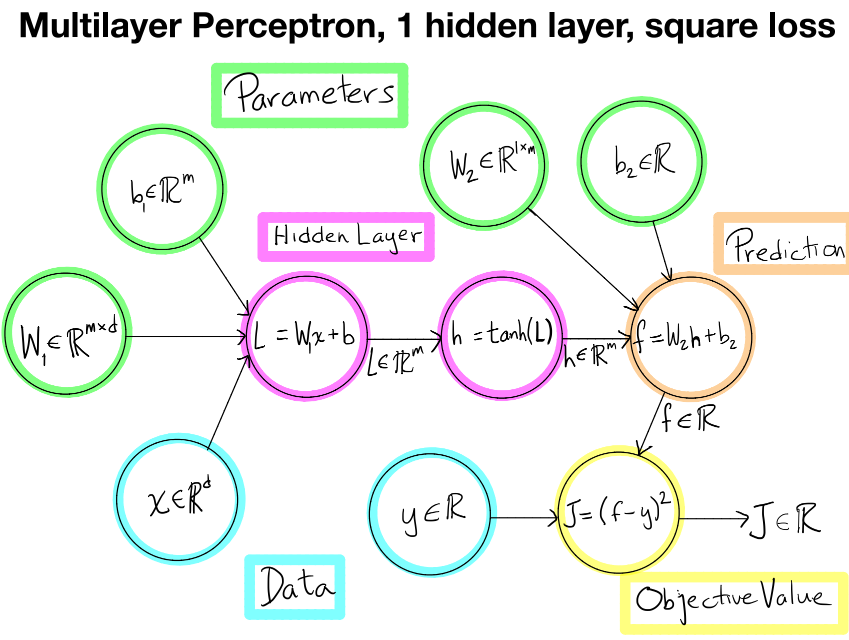


Figure 1: The result of ridge regression.

Multilayer Perceptron

Let's now turn to a multilayer perceptron (MLP) with a single hidden layer and a square loss. We'll implement the computation graph illustrated below:



The crucial new piece here is the nonlinear **hidden layer**, which is what makes the multilayer perceptron a significantly larger hypothesis space than linear prediction functions.

The Standard Nonlinear Layer

The multilayer perceptron consists of a sequence of "layers" implementing the following nonlinear function

$$h(\boldsymbol{x}) = \sigma\left(W\boldsymbol{x} + \boldsymbol{b}\right),$$

where $\boldsymbol{x} \in \mathbb{R}^d$, $W \in \mathbb{R}^{m\times d}$, and $\boldsymbol{b} \in \mathbb{R}^m$, and where m is often referred to as the number of hidden units or hidden nodes. σ is some nonlinear function, typically tanh or ReLU, applied elementwise to the argument of σ. Referring to the computation graph illustration above, we will implement this nonlinear layer with two nodes, one implementing the affine transform $L = \boldsymbol{W}_1^\top \boldsymbol{x} + b_1$, and the other implementing the nonlinear function $h = \tanh(L)$. In this problem, we'll work out how to implement the backward method for each of these nodes.

The Affine Transformation

In a general neural network, there may be quite a lot of computation between any given affine transformation $W\boldsymbol{x} + \boldsymbol{b}$ and the final objective function value J. We will capture all of that in a function $f : \mathbb{R}^m \to \mathbb{R}$, for which $J = f(W\boldsymbol{x} + \boldsymbol{b})$. Our goal is to find the partial derivative of J with respect to each element of W, namely $\partial J/\partial W_{ij}$, as well as the partials $\partial J/\partial b_i$, for each element of $\boldsymbol{b}$. For convenience, let $\boldsymbol{y} = W\boldsymbol{x} + \boldsymbol{b}$, so we can write $J = f(\boldsymbol{y})$. Suppose we have already computed the partial derivatives of J with respect to the entries of $\boldsymbol{y} = (y_1, \cdots, y_m)^\top$, namely $\partial J/\partial y_i$ for $i = 1, \cdots, m$. Then by the chain rule, we have that

$$\frac{\partial J}{\partial W_{ij}} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}}.$$

4. Show that $\partial J/\partial W_{ij} = x_j \cdot \partial J/\partial y_i$, where $\boldsymbol{x} = (x_1, \cdots, x_d)^\top$.

   Hint: Although not necessary, you might find it helpful to use the Kronecker delta noation δ_{ij}. So, for example, $\partial_{x_j} \left(\sum_{i=1}^n x_i^2\right) = 2x_i\delta_{ij} = 2x_j$.

   Proof. By the chain rule, we can compute that

$$\begin{aligned}\frac{\partial J}{\partial W_{ij}} &= \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial y_r}{\partial W_{ij}} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \frac{\partial}{\partial W_{ij}} \left(\sum_{l=1}^{d} W_{rl} x_l + b_r \right) = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \sum_{l=1}^{d} \frac{\partial \left(W_{rl} x_l\right)}{\partial W_{ij}} \\ &= \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \sum_{l=1}^{d} \delta_{ir} \delta_{jl} x_l = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r} \delta_{ir} x_j = x_j \cdot \frac{\partial J}{\partial y_i}, \qquad (1)\end{aligned}$$

   as desired, so the proof is complete. □

5. Now let's vectorize this. Let's write $\partial J/\partial \boldsymbol{y} \in \mathbb{R}^{m\times 1}$ for the column vector whose ith entry is $\partial J/\partial y_i$. Let's also define the matrix $\partial J/\partial W \in \mathbb{R}^{m\times d}$, whose (i, j)th entry is $\partial J/\partial W_{ij}$. Generally speaking, we'll always take $\partial J/\partial A$ to be an array of the same size ("shape" in numpy) as A. Give a vectorized expression for $\partial J/\partial W$ in terms of the column vectors $\partial J/\partial \boldsymbol{y}$ and $\boldsymbol{x}$.

   Hint: Outer product.

   Solution. In the previous question, we have shown that

$$\frac{\partial J}{\partial W_{ij}} = x_j \cdot \frac{\partial J}{\partial y_i}. \qquad (2)$$

   By definition of the vectorized expressions, we can thus write that

$$\left(\frac{\partial J}{\partial W}\right)_{ij} = \frac{\partial J}{\partial W_{ij}} = x_j \cdot \frac{\partial J}{\partial y_i} = (\boldsymbol{x})_j \cdot \left(\frac{\partial J}{\partial \boldsymbol{y}}\right)_i. \qquad (3)$$

   We note that $(\boldsymbol{a} \otimes \boldsymbol{b})_{ij} = a_i b_j$. Therefore, the expression above implies that

$$\frac{\partial J}{\partial W} = \frac{\partial J}{\partial \boldsymbol{y}} \otimes \boldsymbol{x}, \qquad (4)$$

   where $\otimes$ denotes the vector outer product.

6. In the usual way, define $\partial J/\partial \boldsymbol{x} \in \mathbb{R}^d$, whose ith entry is $\partial J/\partial x_i$. Show that

$$\frac{\partial J}{\partial \boldsymbol{x}} = W^\top \frac{\partial J}{\partial \boldsymbol{y}}.$$

   Note: If $\boldsymbol{x}$ is just data, technically we won't need this derivative. However, in a multilayer perceptron, $\boldsymbol{x}$ may actually be the output of a previous hidden layer, in which case we will need to propagate the derivative through $\boldsymbol{x}$ as well.

Proof. By the chain rule, we can can compute that

$$\begin{aligned}\frac{\partial J}{\partial x_i} &= \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial x_i} = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\frac{\partial}{\partial x_i}\left(\sum_{l=1}^{d} W_{rl}x_l + b_r\right) = \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\sum_{l=1}^{d}\frac{\partial\left(W_{rl}x_l\right)}{\partial x_i} \\ &= \sum_{r=1}^{m} \frac{\partial J}{\partial y_r}\sum_{l=1}^{d}\delta_{il}W_{rl} = \sum_{r=1}^{m}\frac{\partial J}{\partial y_r}W_{ri}. \end{aligned} \tag{5}$$

By definition of the vectorized expression, we can thus write that

$$\left(\frac{\partial J}{\partial \boldsymbol{x}}\right)_i = \frac{\partial J}{\partial x_i} = \sum_{r=1}^{m}\frac{\partial J}{\partial y_r}W_{ri} = \sum_{r=1}^{m}\left(W^\top\right)_{ir}\left(\frac{\partial J}{\partial \boldsymbol{y}}\right)_r = \left(W^\top\frac{\partial J}{\partial \boldsymbol{y}}\right)_i. \tag{6}$$

The expression above then implies that

$$\frac{\partial J}{\partial \boldsymbol{x}} = W^\top\frac{\partial J}{\partial \boldsymbol{y}}, \tag{7}$$

as desired, so the proof is complete. □

7. Show that $\partial J/\partial \boldsymbol{b} = \partial J/\partial \boldsymbol{y}$, where $\partial J/\partial \boldsymbol{b}$ is defined in the usual way.

Proof. By the chain rule, we can can compute that

$$\begin{aligned}\frac{\partial J}{\partial b_i} &= \sum_{r=1}^{m}\frac{\partial J}{\partial y_r}\frac{\partial y_r}{\partial b_i} = \sum_{r=1}^{m}\frac{\partial J}{\partial y_r}\frac{\partial}{\partial b_i}\left(\sum_{l=1}^{d}W_{rl}x_l + b_r\right) = \sum_{r=1}^{m}\frac{\partial J}{\partial y_r}\frac{\partial b_r}{\partial b_i} \\ &= \sum_{r=1}^{m}\frac{\partial J}{\partial y_r}\delta_{ir} = \frac{\partial J}{\partial y_i}. \end{aligned} \tag{8}$$

By definition of the vectorized expression, we can thus write that

$$\left(\frac{\partial J}{\partial \boldsymbol{b}}\right)_i = \frac{\partial J}{\partial b_i} = \frac{\partial J}{\partial y_i} = \left(\frac{\partial J}{\partial \boldsymbol{y}}\right)_i. \tag{9}$$

The expression above then implies that

$$\frac{\partial J}{\partial \boldsymbol{x}} = \frac{\partial J}{\partial \boldsymbol{y}}, \tag{10}$$

as desired, so the proof is complete. □

Elementwise Transformers

Our nonlinear activation function nodes take an array (e.g. a vector, matrix, higher-order tensor, etc), and apply the same nonlinear transformation $\sigma : \mathbb{R} \to \mathbb{R}$ to every element of the array. Let's abuse notation a bit, as is usually done in this context, and write $\sigma(A)$ for the array that results from applying $\sigma(\cdot)$ to each element of A. If σ is differentiable at $x \in \mathbb{R}$, then we'll write $\sigma'(x)$ for the derivative of σ at x, with $\sigma'(A)$ defined analogously to $\sigma(A)$.

Suppose the objective function value J is written as $J = f(\sigma(A))$, for some function $f : S \mapsto \mathbb{R}$, where S is an array of the same dimension as $\sigma(A)$ and A. As before, we want to find the array

$\partial J/\partial A$ for any A. Suppose for some A we have already computed the array $\partial J/\partial S = \partial f(S)/\partial S$ for $S = \sigma(A)$. At this point, we'll want to use the chain rule to figure out $\partial J/\partial A$. However, because we're dealing with arrays of arbitrary shapes, it can be tricky to write down the chain rule. Appropriately, we'll use a tricky convention: We'll assume all entries of an array A are indexed by a single variable. So, for example, to sum over all entries of an array A, we'll just write $\sum_i A_i$.

8. Show that $\partial J/\partial A = \partial J/\partial S \odot \sigma'(A)$, where we're using $\odot$ to represent the **Hadamard product**. If A and B are arrays of the same shape, then their Hadamard product $A \odot B$ is an array with the same shape as A and B, and for which $(A \odot B)_i = A_i B_i$. That is, it's just the array formed by multiplying corresponding elements of A and B. Conveniently, in numpy if `A` and `B` are arrays of the same shape, then `A * B` is their Hadamard product.

   Proof. By the chain rule, we can compute that

$$\frac{\partial J}{\partial A_i} = \sum_j \frac{\partial J}{\partial S_j}\frac{\partial S_j}{\partial A_i} = \frac{\partial J}{\partial S_i}\frac{\partial S_i}{\partial A_i} = \frac{\partial J}{\partial S_i}\sigma'(A_i). \tag{11}$$

   By definition of the vectorized expressions, we can thus write that

$$\left(\frac{\partial J}{\partial A}\right)_i = \frac{\partial J}{\partial A_i} = \frac{\partial J}{\partial S_i}\sigma'(A_i) = \left(\frac{\partial J}{\partial S} \odot \sigma'(A)\right)_i. \tag{12}$$

   The expression above then implies that

$$\frac{\partial J}{\partial A} = \frac{\partial J}{\partial S} \odot \sigma'(A), \tag{13}$$

   as desired, so the proof is complete. □

MLP Implementation

9. Complete the class `AffineNode` in `nodes.py`. Be sure to propagate the gradient with respect to $\boldsymbol{x}$ as well, since when we stack these layers, $\boldsymbol{x}$ will itself be the output of another node that depends on our optimization parameters. If your code is correct, you should be able to pass `test_AffineNode` in `mlp_regression.t.py`. Please attach a screenshot that shows the test results for this question.

   Solution. The class `AffineNode` can be implemented as follows.

```python
class AffineNode(object):
    def __init__(self, W, x, b, node_name):
        """Initialize a node computing affine transformation.

        Parameters
        ----------
        W : node object with W.out being np.ndarray of shape (m, d)
            Reference to the node providing matrix input to this node.
        x : node object with x.out being np.array of shape (d,)
            Reference to the node providing vector input to this node.
        b : node object with b.out being np.array of shape (m,)
            Reference to the node providing offset input to thie node.
        node_name : str
            The name of this node.

        Attributes
        ----------
        out : np.array
            The output of this node.
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node
        """
        self.W = W
        self.x = x
        self.b = b
        self.node_name = node_name
        # Setting additional atttributes as documented
        self.out = None
        self.d_out = None

    def forward(self):
        """Forwards one step.

        This sets `out` of the current node, so we move forward one step.
        It also initialized `d_out` to be updated in the backward round.
        It returns the output of this node which is to be computed.

        Returns
        -------
        out : np.array
            The output of this node.
        """
        # The outputs of W, x, and b are the inputs of the current node
        # The current node then outputs Wx + b
        self.out = np.dot(self.W.out, self.x.out) + self.b.out
        # The partial derivative should have the same dimension as the
        # output of this node (with respect to which derivative is taken)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        """Backwards one step.

        This sets `d_out` of its predecessors providing inputs to this node.
        It returns partial derivative of the graph output (i.e., objective)
        with respect to this node which is computed by its successor.

        Returns
        -------
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        # pJ/pWin is given by outer(pJ/pout, xin) :cite:`HW7 Q5`
        # pJ/pout is just `d_out` of the current node
        self.W.d_out += np.outer(self.d_out, self.x.out)
        # pJ/pxin is given by dot(W^T, pJ/pout) :cite:`HW7 Q6`
        self.x.d_out += np.dot(self.W.out.T, self.d_out)
        # pJ/pbin is given by pJ/pout :cite:`HW7 Q7`
        self.b.d_out += self.d_out
        return self.d_out

    def get_predecessors(self):
        """Gets the predecessors.

        Returns
        -------
        predecessors : list (of nodes)
            The list of predecessors, i.e., the nodes that provide input to
            the current node. Normally the affine transformation node has
            three predecessors.
        """
        return [self.W, self.x, self.b]
```

The result of running `test_L2NormPenaltyNode` is shown as follows.

```
> python mlp_regression.t.py TestNodes.test_AffineNode
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. W is
1.5544356312464695e-08.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. x is
2.256338571093939e-08.
DEBUG: (Node affine) Max rel error for partial deriv w.r.t. b is
2.8043131803233403e-09.
.
----------------------------------------------------------------------
Ran 1 test in 0.003s
OK
```

10. Complete the class `TanhNode` in `nodes.py`. As you'll recall, $\tanh'(x) = 1 - \tanh^2 x$. Note that in the forward pass, we'll already have computed tanh of the input and stored it in `self.out`. So make sure to use `self.out` and not recalculate it in the backward pass. If your code is correct, you should be able to pass `test_TanhNode` in `mlp_regression.t.py`. Please attach a screenshot that shows the test results for this question.

Solution. The class `TanhNode` can be implemented as follows.

```python
class TanhNode(object):
    def __init__(self, a, node_name):
        """Initialize a node computing hyperbolic tangent function.

        Parameters
        ----------
        a : node object with a.out being np.array
            Reference to the node providing input to this node.
        node_name : str
            The name of this node.

        Attributes
        ----------
        out : np.array
            The output of this node.
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        self.a = a
        self.node_name = node_name
        # Setting additional attributes as documented
        self.out = None
        self.d_out = None

    def forward(self):
        """Forwards one step.

        This sets `out` of the current node, so we move forward one step.
        It also initializes `d_out` to be updated in the backward round.
        It returns the output of this node which is to be computed.

        Returns
        -------
        out : np.array
            The output of this node.
        """
        # The output of a is the input of the current node
        # The current node then outputs tanh(a)
        self.out = np.tanh(self.a.out)
        # The partial derivative should have the same dimension as the
        # output of this node (with respect to which derivative is taken)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        """Backwards one step.

        This sets `d_out` of its predecessor providing input to this node.
        It returns partial derivative of the graph output (i.e., objective)
        with respect to this node which is computed by its successor.

        Returns
        -------
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        # By chain rule, we can compute pJ/pin = pJ/pout * pout/pin
        # pJ/pout is just `d_out` of the current node
        # pJ/pin is tanh'(x) = 1 - tanh^2(x), and since the output of this
        # node is tanh(x), it can be simplified as 1 - out^2
        self.a.d_out += self.d_out * (1 - self.out ** 2)
        return self.d_out

    def get_predecessors(self):
        """Gets the predecessors.

        Returns
        -------
        predecessors : list (of nodes)
            The list of predecessors, i.e., the nodes that provide input to
            the current node. Normally the tanh node has only one
    predecessor.
        """
        return [self.a]
```

The result of running `test_TanhNode` is shown as follows.

```
> python mlp_regression.t.py TestNodes.test_TanhNode
DEBUG: (Node tanh) Max rel error for partial deriv w.r.t. a is
1.268954386373119e-08.
.
------------------------------------------------------------------------
```

```
Ran 1 test in 0.001s
OK
```

11. Implement an MLP by completing the skeleton code in `mlp_regression.py` and making use of the nodes above. Your code should pass the tests provided in `mlp_regression.t.py`. Note that to break the symmetry of the problem, we initialize our weights to small random values, rather than all zeros, as we often do for convex optimization problems. Run the MLP for the two settings given in the `main()` function and report the average **training** error. Note that with an MLP, we can take the original scalar as input, in the hopes that it will learn nonlinear features on its own, using the hidden layers. In practice, it is quite challenging to get such a neural network to fit as well as one where we provide features.

    Solution. The `__init__` method of the class `MLPRegression` can be implemented as follows. It constructs and stores the `ComputationGraphFunction` instance in `self.graph`.

```python
class MLPRegression(BaseEstimator, RegressorMixin):
    def __init__(
        self,
        num_hidden_units=10,
        step_size=5e-3,
        init_param_scale=0.01,
        max_num_epochs=5000,
    ):
        """Initializes an MLP regression object with computation graph.

        We want to build a computation graph for the multilayer perceptron
        objective, with a single hidden layer and a square loss.

        - Take parameter b1 of size (m,), parameter W1 of size (m, d),
          and data x of size (d,), and make the affine transform to get
          the first hidden layer :math:`L = W1x + b1`;

        - Take interim result L of size (m,) and make the tanh transform
          to get the second hidden layer :math:`h = tanh(L)`;

        - Take parameter b2 of size (1,), parameter w2 of size (1, m),
          and interim result h of size (m,), and make the affine transform
          to get the prediction :math:`f = w2h + b2`;

        - Take data y of size (1,) and prediction f of size (1,), and take
          the squared difference to get the objective :math:`J = (f-y)^2`.

        Parameters
        ----------
        num_hidden_units : integral
            The number of hidden units in each hidden layer. We consider a
            single hidden layer in this class.
        step_size : real
            The learning rate, which should be nonnegative.
        init_param_scale : real
            The global scaler for the initial parameters, which should be
            nonnegative.
        max_num_epochs : integral
            The maximum number of epochs, which should be nonnegative.

        Attributes
        ----------
        graph : ComputationGraphFunction object
```

```
            The computation graph instance for MLP regression.
        """
        self.num_hidden_units = num_hidden_units
        self.step_size = step_size
        self.init_param_scale = init_param_scale
        self.max_num_epochs = max_num_epochs

        # Constructing the input nodes
        # The input vector x
        self._x = nodes.ValueNode(node_name="x")
        # The response scalar y
        self._y = nodes.ValueNode(node_name="y")

        # Constructing the parameter nodes
        # The parameter matrix W1 (the 1st affine)
        self._W1 = nodes.ValueNode(node_name="W1")
        # The vector bias parameter b1 (the 1st affine)
        self._b1 = nodes.ValueNode(node_name="b1")
        # The parameter vector w2 (the 2nd affine)
        self._w2 = nodes.ValueNode(node_name="w2")
        # The scalar bias parameter b2 (the 2nd affine)
        self._b2 = nodes.ValueNode(node_name="b2")

        # Constructing the interim computation nodes
        # Compute :math:`L = W1x + b1`
        self._hidden_L = nodes.AffineNode(
            W=self._W1, x=self._x, b=self._b1, node_name=":hidden:"
        )
        # Compute :math:`h = tanh(L)`
        self._hidden_h = nodes.TanhNode(
            a=self._hidden_L, node_name=":hidden:"
        )
        # Compute :math:`f = w2h + b2`
        self._prediction = nodes.VectorScalarAffineNode(
            x=self._hidden_h, w=self._w2, b=self._b2, node_name="prediction"
        )
        # Compute :math:`J = (f-y)^2`
        self._objective = nodes.SquaredL2DistanceNode(
            a=self._prediction, b=self._y, node_name="objective"
        )

        # Constructing the computation graph
        self.graph = graph.ComputationGraphFunction(
            inputs=[self._x],
            outcomes=[self._y],
            parameters=[self._W1, self._b1, self._w2, self._b2],
            prediction=self._prediction,
            objective=self._objective,
        )
```

The result of running `test_mlp_regression_gradient` is shown as follows.

```
> python mlp_regression.t.py TestNodes.test_mlp_regression_gradient
DEBUG: (Parameter W1) Max rel error for partial deriv
1.4189237423709666e-06.
DEBUG: (Parameter b1) Max rel error for partial deriv
1.6946877248656965e-07.
DEBUG: (Parameter w2) Max rel error for partial deriv
1.0886442160606574e-09.
```

```
DEBUG: (Parameter b2) Max rel error for partial deriv
8.704641947693862e-10.
.
----------------------------------------------------------------------
Ran 1 test in 0.004s
OK
```

Using 10 hidden units with no features, step size 10^{-3}, initial parameter scale 5×10^{-4}, and 5000 epochs, the average square error on the training set is 0.2593825310338179. Using 10 hidden units with no features, step size 5×10^{-4}, initial parameter scale 10^{-2}, and 500 epochs, the average square error on the training set is 0.0844137569476599. The regression results are shown as in Figure 2.

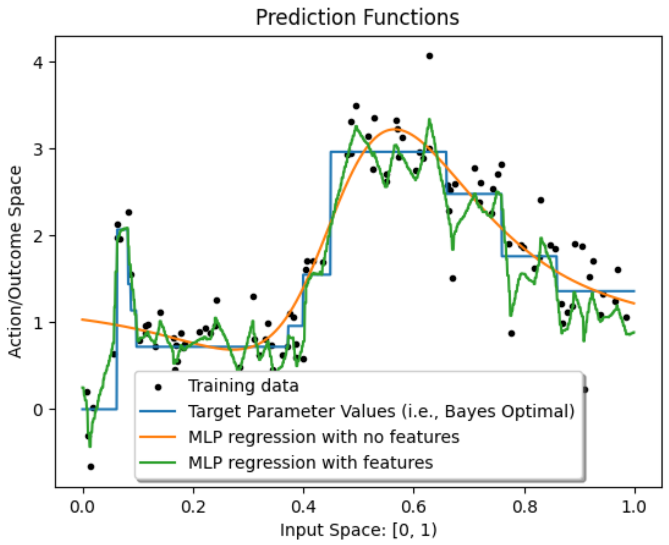


Figure 2: The result of multilayer perceptron regression.

Multiclass classification with an MLP (Optional)

We consider a generic classification problem with K classes over inputs $\boldsymbol{x}$ of dimension d. Using a MLP we will compute a K-dimensional vector $\boldsymbol{z}$ representing scores, such that

$$\boldsymbol{z} = W_2 \tanh(W_1 \boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2,$$

with $W_1 \in \mathbb{R}^{m \times d}$, $\boldsymbol{b}_1 \in \mathbb{R}^m$, $W_2 \in \mathbb{R}^{K \times m}$ and $\boldsymbol{b}_2 \in \mathbb{R}^K$. Our model assumes that $\boldsymbol{x}$ belongs to class k with probability

$$\frac{\exp(z_k)}{\sum_{k=1}^{K} \exp(z_k)},$$

which corresponds to applying a Softmax to the scores. Given this probabilistic model we can train the model by minimizing the negative log-likelihood.

12. Implement a Softmax node. We have provided skeleton code for the class `SoftmaxNode` in `nodes.py`. If your code is correct, you should be able to pass `test_SoftmaxNode` in `multiclass.t.py`. Please attach a screenshot that shows the test results for this question.

Solution. The class `SoftmaxNode` can be implemented as follows.

```python
class SoftmaxNode(object):
    def __init__(self, z, node_name):
        """Initialize a node computing the softmax function.

        Parameters
        ----------
        z : node object with z.out being np.array
            Reference to the node providing input to this node.
        node_name : str
            The name of this node.

        Attributes
        ----------
        out : np.array
            The output of this node.
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        self.z = z
        self.node_name = node_name
        # Setting additional attributes as documented
        self.out = None
        self.d_out = None

    def forward(self):
        """Forwards one step.

        This sets `out` of the current node, so we move forward one step.
        It also initializes `d_out` to be updated in the backward round.
        It returns the output of this node which is to be computed.

        Returns
        -------
        out : np.array
            The output of this node.
        """
        # The output of z is the input of the current node
        # The current node then outputs softmax(z)
        # In order to avoid overflow, we refer to the implementation of
        # `scipy.special.softmax` v1.10.1 in scipy/special/_logsumexp.py
        exp_z_shifted = np.exp(self.z.out - np.max(self.z.out))
        self.out = exp_z_shifted / np.sum(exp_z_shifted)
        # The partial derivative should have the same dimension as the
        # output of this node (with respect to which derivative is taken)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        """Backwards one step.

        This sets `d_out` of its predecessor providing input to this node.
        It returns partial derivative of the graph output (i.e., objective)
        with respect to this node which is computed by its successor.

        Returns
        -------
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        # pJ/pin is given by dot(M, pJ/pout), where the ith diagonal entry
        # of M is :math:`S_i * (1-S_i)`, and the (i, j)th entry for i != j
        # of M is :math:`S_i * S_j`
        # pJ/pout is just `d_out` of the current node
        # S is the output of the current node, i.e., `out`
        M = np.zeros((self.z.out.size, self.out.size))
        for i in range(self.z.out.size):
            for j in range(self.out.size):
                M[i, j] = -self.out[i] * self.out[j] if i != j \
                    else self.out[i] * (1 - self.out[i])
        self.z.d_out += np.dot(M, self.d_out)
        return self.d_out

    def get_predecessors(self):
        """Gets the predecessors.

        Returns
        -------
        predecessors : list (of nodes)
            The list of predecessors, i.e., the nodes that provide input to
            the current node. Normally the softmax node has only one
    predecessor.
        """
        return [self.z]
```

The result of running `test_SoftmaxNode` is shown as follows.

```
> python multiclass.t.py TestNodes.test_SoftmaxNode
DEBUG: (Node softmax) Max rel error for partial deriv w.r.t. z is
2.580295092005085e-09.
.
----------------------------------------------------------------------
Ran 1 test in 0.001s
OK
```

13. Implement a negative log-likelihood loss node for multiclass classification. We provided skeleton code for class `NLLNode` in `nodes.py`. The test code for this question is combined with the test code for the next question.

    Solution. The class `NLLNode` can be implemented as follows.

```
class NLLNode(object):
    def __init__(self, f, y, node_name):
        """Initialize a node computing the negative log-likelihood.

        Parameters
        ----------
        f : node object with f.out being np.array
            Reference to the node providing predictions input to this node.
        y : node object with y.out being np.array
            Reference to the node providing responses input to this node.
        node_name : str
            The name of this node

        Attributes
        ----------
        out : np.array
```

```
            The output of this node.
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node
        """
        self.f = f
        self.y = y
        self.node_name = node_name
        # Setting additional attributes as documented
        self.out = None
        self.d_out = None

    def forward(self):
        """Forwards one step.

        This sets `out` of the current node, so we move forward one step.
        It also initialized `d_out` to be updated in the backward round.
        It returns the output of this node which is to be computed.

        Returns
        -------
        out : np.array
            The output of this node.
        """
        # The outputs of f and y are the inputs of the current node
        # The current node then outputs the negative log-likelihood
        # This is taking negative log on the likelihood of the true class
        self.out = -np.log(self.f.out[self.y.out])
        # The partial derivative should have the same dimension as the
        # output of this node (with respect to which derivative is taken)
        self.d_out = np.zeros(self.out.shape)
        return self.out

    def backward(self):
        """Backwards one step.

        This sets `d_out` of its predecessors providing inputs to this node.
        It returns partial derivative of the graph output (i.e., objective)
        with respect to this node which is computed by its successor.

        Returns
        -------
        d_out : np.array
            The partial derivatives of the graph output (i.e., objective)
            with respect to the output of this node.
        """
        # By chain rule, we can compute pJ/pfin = pJ/pout * pout/pfin
        # pJ/pout is just `d_out` of the current node
        # pJ/pin is -1/f[y] for the true class y and 0 otherwise
        pred = np.zeros(self.f.out.shape)
        pred[self.y.out] = 1 / self.f.out[self.y.out]
        self.f.d_out -= self.d_out * pred
        # Whether we need to update y is unclear
        return self.d_out

    def get_predecessors(self):
        """Gets the predecessors.

        Returns
        -------
        predecessors : list (of nodes)
```

```python
            The list of predecessors, i.e., the nodes that provide input to
            the current node. Normally the NLL node has two predecessors.
        """
        return [self.f, self.y]
```

14. Implement an MLP instance for multiclass classification by completing the skeleton code in `multiclass.py`. Your code should pass the tests in `test_multiclass` provided in `multiclass.t.py`. Please attach a screenshot that shows the test results for this question.

    Solution. The `__init__` method of the class `MulticlassClassifier` can be implemented as follows. It constructs and stores the `ComputationGraphFunction` instance in `self.graph`.

```python
class MulticlassClassifier(BaseEstimator, RegressorMixin):
    def __init__(
        self,
        num_hidden_units=10,
        step_size=5e-3,
        init_param_scale=0.01,
        max_num_epochs = 1000,
        num_class=3,
    ):
        """Initializes an MLP multiclass classifier with computation graph.

        We want to build a computation graph for the multilayer perceptron
        objective, with a single hidden layer and a square loss.

        - Take parameter b1 of size (m,), parameter W1 of size (m, d),
          and data x of size (d,), and make the affine transform to get
          the first hidden layer :math:`L = W1x + b1`;

        - Take interim result L of size (m,) and make the tanh transform
          to get the second hidden layer :math:`h = tanh(L)`;

        - Take parameter b2 of size (K,), parameter W2 of size (K, m),
          and interim result h of size (m,), and make the affine transform
          to get the third hidden layer :math:`z = W2h + b2`;

        - Take interim result z of size (K,) and make the softmax transform
          to get the prediction :math:`f = Softmax(z)`

        - Take data y of size (1,) and prediction f of size (K,), and take
          the NLL to get the objective :math:`J = NLL(f, y)`.

        Parameters
        ----------
        num_hidden_units : integral
            The number of hidden units in each hidden layer. We consider a
            single hidden layer in this class.
        step_size : real
            The learning rate, which should be nonnegative.
        init_param_scale : real
            The global scaler for the initial parameters, which should be
            nonnegative.
        max_num_epochs : integral
            The maximum number of epochs, which should be nonnegative.
        num_class : integral
            The number of classes.

        Attributes
        ----------
        graph : ComputationGraphFunction object
            The computation graph instance for MLP regression.
        """
        self.num_hidden_units = num_hidden_units
        self.step_size = step_size
        self.init_param_scale = init_param_scale
        self.max_num_epochs = max_num_epochs
        self.num_class = num_class

        # Constructing the input nodes
        # The input vector x
        self._x = nodes.ValueNode(node_name="x")
        # The response scalar y
        self._y = nodes.ValueNode(node_name="y")

        # Constructing the parameter nodes
        # The parameter matrix W1 (the 1st affine)
        self._W1 = nodes.ValueNode(node_name="W1")
        # The vector bias parameter b1 (the 1st affine)
        self._b1 = nodes.ValueNode(node_name="b1")
        # The parameter matrix w2 (the 2nd affine)
        self._W2 = nodes.ValueNode(node_name="W2")
        # The scalar bias parameter b2 (the 2nd affine)
        self._b2 = nodes.ValueNode(node_name="b2")

        # Constructing the interim computation nodes
        # Compute :math:`L = W1x + b1`
        self._hidden_L = nodes.AffineNode(
            W=self._W1, x=self._x, b=self._b1, node_name=":hidden:"
        )
        # Compute :math:`h = tanh(L)`
        self._hidden_h = nodes.TanhNode(
            a=self._hidden_L, node_name=":hidden:"
        )
        # Compute :math:`z = W2h + b2`
        self._hidden_z = nodes.AffineNode(
            W=self._W2, x=self._hidden_h, b=self._b2, node_name=":hidden:"
        )
        # Compute :math:`f = Softmax(z)`
        self._prediction = nodes.SoftmaxNode(
            z=self._hidden_z, node_name="prediction"
        )
        # Compute :math:`J = NLL(f, y)`
        self._objective = nodes.NLLNode(
            f=self._prediction, y=self._y, node_name="objective"
        )

        # Constructing the computation graph
        self.graph = graph.ComputationGraphFunction(
            inputs=[self._x],
            outcomes=[self._y],
            parameters=[self._W1, self._b1, self._W2, self._b2],
            prediction=self._prediction,
            objective=self._objective,
        )
```

The result of running `test_multiclass` is shown as follows.

```
> python multiclass.t.py TestNodes.test_multiclass
DEBUG: (Parameter W1) Max rel error for partial deriv
```

```
5.1741662292183015e-08.
DEBUG: (Parameter b1) Max rel error for partial deriv
8.059693440926508e-08.
DEBUG: (Parameter W2) Max rel error for partial deriv
2.7081779821974406e-07.
DEBUG: (Parameter b2) Max rel error for partial deriv
5.995142302081188e-08.
.
----------------------------------------------------------------------
Ran 1 test in 0.004s
OK
```