LAB1: SVM

21304219	
刘文婧	

Requirements

> 使用SVM训练一个二分类器，使其能够根据样本的像素值分辨出其label是0还是1
>
> 1. 考虑两种不同的核函数：i) 线性核函数; ii) 高斯核函数
> 2. 可以直接调用现成 SVM 软件包来实现
> 3. 手动实现采用 hinge loss 和 cross-entropy loss 的线性分类模型，并比较它们的优劣

Overview of SVM model

SVM的一般理论、Hinge Loss角度的SVM、与cross-entropy loss比较

Fundamental Theory

> Maximum-Margin

1. 首先，最基本的线性回归想法是，如下的超平面把样本分为了2类：

$$Hyperplane : \{\vec{x} | \vec{w}^\top \vec{x} + b = 0\},\ normal\ vector : \frac{\vec{w}}{||\vec{w}||}$$

2. 然后，我们希望超平面分割的两个半空间中，训练样本点距离这个超平面的“余量”尽量地大，我们定义margin如下（某种意义上样本到超平面的“距离”，利用超平面和法向量的性质就容易写出来）：

$$margin = \min_{l} \frac{y^{(l)}(\vec{w}^\top \vec{x}^{(l)} + b)}{||\vec{w}||}$$

3. 我们的目标是**<u>最大化margin</u>**，引入放缩因子κ后的$\{\vec{x}|\kappa\vec{w}^\top\vec{x}+\kappa b=0\}$和$\{\vec{x}|\vec{w}^\top\vec{x}+b=0\}$是平行的（其实是同一个），且其实margin值也一样，于是不妨利用κ，使得问题满足2个约束条件（如下）。所以构建出最原始的优化问题形式：

$$\begin{gathered}\max_{\vec{w},b} \quad \frac{1}{||\vec{w}||}\min_{l} y^{(l)}(\vec{w}^\top\vec{x}^{(l)}+b), \\ \text{s.t.} \quad y^{(l)}(\vec{w}^\top\vec{x}^{(l)}+b) \geq 1, l=1,2...N, \\ at\ least\ 1 \text{“}=\text{”} exists\end{gathered}$$

即，

$$\begin{gathered}\min_{\vec{w},b} \quad \frac{1}{2}||\vec{w}||^2, \\ \text{s.t.} \quad y^{(l)}(\vec{w}^\top\vec{x}^{(l)}+b) \geq 1, l=1,2...N\end{gathered}$$

4. 该优化问题的 dual 形式如下（拉格朗日函数，然后分别对w、b求偏导（grad=0），代入）：

$$\begin{gathered}\max_{\vec{a}} \quad g(\vec{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i a_j y^{(i)} y^{(j)} \vec{x}^{(i)\top} \cdot \vec{x}^{(j)}, \\ \text{s.t.} \quad \vec{a} \geq 0, \quad \sum_{i=1}^{N} a_i y^{(i)} = 0\end{gathered}$$

> ## Soft-Maximum-Margin

5. 但是，并不是所有的训练样本，都能被超平面理想地分割，于是我们引入松弛的$\xi_n, \xi_n \geq 0$，所有样本都满足

$$y^{(l)}(\vec{w}^\top \vec{x}^{(l)} + b) \geq 1 - \xi_l, \quad l = 1, 2 ... N$$

问题变成了

$$\begin{aligned} \min_{\vec{w}, b} \quad & \frac{1}{2}||\vec{w}||^2 + C \sum_{n=1}^{N} \xi_n, \\ \text{s.t.} \quad & y^{(l)}(\vec{w}^\top \vec{x}^{(l)} + b) \geq 1 - \xi_l, \quad \xi_l \geq 0, \quad l = 1, 2 ... N \end{aligned}$$

然后对偶形式：

$$\begin{aligned} \max_{\vec{a}} \quad & g(\vec{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j y^{(i)} y^{(j)} \vec{x}^{(i)\top} \cdot \vec{x}^{(j)}, \\ \text{s.t.} \quad & \vec{a} \geq 0, \quad \vec{a} \leq C, \quad \sum_{i=1}^{N} a_i y^{(i)} = 0 \end{aligned}$$

Support-Vector-Machine

6. 为了处理非线性可分，或者希望映射到更高维的特征空间内，我们让

$$\vec{x} \rightarrow \vec{\phi}(\vec{x})$$

然后就是SVM的形式了：

$$\begin{aligned} \min_{\vec{w}, b, \vec{\xi}} \quad & \frac{1}{2}||\vec{w}||^2 + C \sum_{n=1}^{N} \xi_n, \\ \text{s.t.} \quad & y^{(l)}(\vec{w}^\top \vec{\phi}(\vec{x}^{(l)}) + b) \geq 1 - \xi_l, \quad \xi_l \geq 0, \quad l = 1, 2 ... N \end{aligned}$$

分类器：

$$\hat{y} = \text{sign}(\vec{w}^* \cdot \vec{\phi}(\vec{x}) + b^*)$$

对偶形式：

$$
\begin{aligned}
\max_{\vec{a}} \quad & g(\vec{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j y^{(i)} y^{(j)} [\vec{\phi}(\vec{x}^{(i)})^\top \cdot \vec{\phi}(\vec{x}^{(j)})], \\
\text{s.t.} \quad & \vec{a} \geq 0, \quad \vec{a} \leq C, \quad \sum_{i=1}^{N} a_i y^{(i)} = 0
\end{aligned}
$$

分类器：

$$
\hat{y} = \text{sign}\left(\sum_{l=1}^{N} a_l^* y^{(l)} \langle \vec{\phi}(\vec{x}^{(l)}), \vec{\phi}(\vec{x}) \rangle + b^* \right)
$$

核函数

7. $\langle \vec{\phi}(\vec{x}^{(n)}), \vec{\phi}(\vec{x}^{(m)}) \rangle$内积，如果我们希望特征空间，即$\vec{\phi}$的维度数很大，甚至可以有无穷维，直接这样计算显然不合适，根据相关理论（Mercer Theorem等），直接用Kernel Function来表示它：

$$
\begin{aligned}
\max_{\vec{a}} \quad & g(\vec{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} a_i a_j y^{(i)} y^{(j)} K(\vec{x}^{(i)}, \vec{x}^{(j)}), \\
\text{s.t.} \quad & \vec{a} \geq 0, \quad \vec{a} \leq C, \quad \sum_{i=1}^{N} a_i y^{(i)} = 0
\end{aligned}
$$

分类器：

$$
\hat{y} = \text{sign}\left(\sum_{l=1}^{N} a_l^* y^{(l)} K(\vec{x}^{(l)}, \vec{x}) + b^* \right)
$$

Hinge Loss

8. 上面指出了，现在SVM的优化目标是

$$
\begin{aligned}
\min_{\vec{w}, b, \vec{\xi}} \quad & \frac{1}{2} \|\vec{w}\|^2 + C \sum_{n=1}^{N} \xi_n, \\
\text{s.t.} \quad & y^{(l)} (\vec{w}^\top \vec{\phi}(\vec{x}^{(l)}) + b) \geq 1 - \xi_l, \quad \xi_l \geq 0, \quad l = 1, 2 \ldots N
\end{aligned}
$$

- 需要minimize的公式，其实也是某种需要minimize的loss，我们就从loss function的角度写一下，把 $||\vec{w}||$ 项看作正则化项，然后对于后面 ξ_n 有关的，回想一下 ξ_n 的来源：
- 满足 $y^{(l)}(\vec{w}^\top\vec{\phi}(\vec{x}^{(l)})+b)\geq 1$ 的样本不需要，ξ_n 是0；
- 不满足 $y^{(l)}(\vec{w}^\top\vec{\phi}(\vec{x}^{(l)})+b)\geq 1$ 的样本需要 ξ_n ，且 $y^{(l)}(\vec{w}^\top\vec{\phi}(\vec{x}^{(l)})+b)$ 比1差的越远，ξ_n就要补偿的越大。
- 就定义这个函数，来表达对输入 $y^{(l)}(\vec{w}^\top\vec{\phi}(\vec{x}^{(l)})+b)$ 时，相应的ξ_n：

$$E_{SV}(z)=\max(0,1-z)$$

- 于是，SVM模型其实是在训练模型降低 hinge loss，hinge损失函数如下：

$$L(\vec{w},b)=\sum_{\ell=1}^{N}E_{SV}\left[y^{(\ell)}\left(\vec{w}^T\vec{\phi}(\vec{x}^{(\ell)})+b\right)\right]+\lambda\|\vec{w}\|_2^2$$

or

$$L(\vec{w},b)=\sum_{\ell=1}^{N}E_{SV}\left(y^{(\ell)}h^{(\ell)}\right)+\lambda\|\vec{w}\|_2^2$$

Cross-Entropy Loss

类似于上面写出的hinge loss形式，传统的cross-entropy loss是：

$$L(\vec{w},b)=-\sum_{n=1}^{N}\left[\tilde{y}^{(n)}\log\sigma(h^{(n)})+(1-\tilde{y}^{(n)})\log\left(1-\sigma(h^{(n)})\right)\right]+\lambda\|\vec{w}\|^2,$$
$$\tilde{y}\in\{0,1\}$$

也即

$$L(\vec{w},b)=\sum_{\ell=1}^{N}E_{LR}(y^{(\ell)}h^{(\ell)})+\lambda\|\vec{w}\|^2,\quad y\in\{-1,1\}$$

其中，

$$E_{LR}(z) = \log\left(1 + \exp(-z)\right)$$

Different Kernel Functions

考虑两种不同的核函数：i) 线性核函数; ii) 高斯核函数

- 要maximize的拉格朗日对偶函数是：

$$g(\vec{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i a_j y^{(i)} y^{(j)} K(\vec{x}^{(i)}, \vec{x}^{(j)})$$

分类器是：

$$\hat{y} = \operatorname{sign}\left(\sum_{l=1}^{N} a_l^* y^{(l)} K(\vec{x}^{(l)}, \vec{x}) + b^*\right)$$

线性核函数

$$K(\vec{x}^{(i)}, \vec{x}^{(j)}) = \vec{x}^{(i)^\top} \vec{x}^{(j)}$$

高斯核函数（RBF 核函数）

$$K(\vec{x}^i, \vec{x}^j) = \exp\left(-\frac{\|\vec{x}^{(i)} - \vec{x}^{(j)}\|^2}{2\sigma^2}\right)$$

Idea && Code

调用SVM软件包，考虑线性核函数和高斯核函数

> 根据scikit-learn.svm的documentation，使用其中的SVC类就好

```python
2 usages
class SVM:
    def __init__(self, kernel, train_samples, train_labels):
        if kernel == "Gaussian":
            self.kernel = "rbf"
        elif kernel == "Linear":
            self.kernel = "linear"
        self.train_samples = train_samples
        self.train_labels = train_labels
        self.clf = SVC(kernel=self.kernel)

    1 usage
    def train(self):
        self.clf.fit(self.train_samples, self.train_labels)

    1 usage
    def predict(self, test_samples):
        """ return type is 'numpy.ndarray' """
        return self.clf.predict(test_samples)

    1 usage
    def evaluate(self, test_samples, test_labels):
        predictions = self.predict(test_samples)
        accuracy = np.mean(predictions == test_labels)
        print(f"Accuracy: {accuracy:.6%}")
        return accuracy
```

关于hinge loss的分类模型与SVM模型之间的关系

> 理论角度分析

- 根据上面“Overview of SVM model”中的第8点想法，其实二者的训练目标/优化目标是意义一致的。
- 从优化（需要minimize）的目标公式来看，hinge loss显然公式没有SVM模型中的 $g(\vec{a}) = \sum_{i=1}^{N} a_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} a_i a_j y^{(i)} y^{(j)} K(\vec{x}^{(i)}, \vec{x}^{(j)})$那么复杂
- 但可以看到，SVM模型直接使用核函数计算，避开了在特征空间 $\vec{\phi}$ 维度下的计算，于是，能够处理特征空间 $\vec{\phi}$ 是极其高维的情况，或者说，甚至能够使用无穷维的 $\vec{\phi}$

，比如高斯核函数对应的 $\vec{\phi}$ 。

- SVM模型，处理 **样本维度**N 的计算就好。
- 而hinge loss的公式中，$h^{(\ell)}=\left(\vec{w}^T\vec{\phi}(\vec{x}^{(\ell)})+b\right)$需要在特征空间 $\vec{\phi}$ 维度下计算，当$\vec{\phi}$ 维度数很大，计算就很复杂，或者说，没办法在有限计算资源下使用太高维度的 $\vec{\phi}$ 。
- 只是线性模型，不需要非线性化时，$\vec{\phi}(\vec{x})$ 取 $\vec{x}$ 就好。
- **SVM是通过求解QP优化问题来训练的，不像hinge loss的分类模型是通过反向传播（梯度下降）来更新参数从而训练模型的。**

> 实际训练效果角度分析

- 见下面的 Result && Analysis 部分

手动实现hinge loss和cross-entropy loss的线性分类模型

> 由于要手动实现线性分类模型，这里写一下训练过程中要用到的，两种Loss函数分别对w、b的偏导数（用于梯度下降更新）

- Note：实验中实现的是线性模型，所以下面的$\vec{\phi}(\vec{x})$ 取 $\vec{x}$ 就好。

1. hinge loss对w的偏导数：

$$\frac{\partial L}{\partial \vec{w}}=2\lambda\vec{w}+\sum_{n=1}^{N}\alpha(\vec{x}^{(n)})$$

其中，

$$\alpha(\vec{x}^{(n)})=\begin{cases}0, & if \quad y^{(n)}\left(\vec{w}^T\vec{\phi}(\vec{x}^{(n)})+b\right)\geq 1\\ -y^{(n)}\vec{\phi}(\vec{x}^{(n)}), & if \quad y^{(n)}\left(\vec{w}^T\vec{\phi}(\vec{x}^{(n)})+b\right)<1\end{cases}$$

2. hinge loss对b的偏导数：

$$\frac{\partial L}{\partial b} = \sum_{n=1}^{N} b(\vec{x}^{(n)})$$

其中，

$$b(\vec{x}^{(n)}) = \begin{cases} 0, & if \quad y^{(n)}\left(\vec{w}^T\vec{\phi}(\vec{x}^{(n)}) + b\right) \geq 1 \\ -y^{(n)}, & if \quad y^{(n)}\left(\vec{w}^T\vec{\phi}(\vec{x}^{(n)}) + b\right) < 1 \end{cases}$$

3. cross-entropy loss对w、b的偏导数：

$$\frac{\partial L}{\partial \vec{w}} = \sum_{n=1}^{N} \vec{\phi}(\vec{x}^{(n)})\left(\sigma(h^{(n)}) - \tilde{y}^{(n)}\right) + 2\lambda\vec{w}, \quad \tilde{y} \in \{0, 1\}$$

$$\frac{\partial L}{\partial b} = \sum_{n=1}^{N} \left(\sigma(h^{(n)}) - \tilde{y}^{(n)}\right)$$

上面的 h 是：$h^{(\ell)} = \left(\vec{w}^T\vec{\phi}(\vec{x}^{(\ell)}) + b\right)$

- 实际代码实现时，对loss进行了除以样本数的取平均数操作。
- 大致代码如下：

```
1 usage
def hinge_loss(labels: np.ndarray, samples: np.ndarray, weights: np.ndarray, bias, lamda):
    """
    implementation of hinge loss

    :param labels: one-dim vector, (N, ), and each y in {-1, 1}
    :param samples: matrix, (N, m)
    :param weights: one-dim vector, (m, )
    :param bias: scalar
    :param lamda: regularization strength
    :return: hinge loss(a scalar) result of N samples
    """
    predict = np.dot(weights, samples.T) + bias
    vec_yh = np.multiply(labels, predict)
    ESV_result = np.maximum(0, 1 - vec_yh)
    loss_result = np.sum(ESV_result) / len(labels) + lamda * np.sum(weights ** 2)  # average
    return loss_result

```

```python
def grad_hinge(labels: np.ndarray, samples: np.ndarray, weights: np.ndarray, bias, lamda):
    """
    Partial derivative of the loss function with respect to w,b

    :param labels: one-dim vector, (N, ), and each y in {-1, 1}
    :param samples: matrix, (N, m)
    :param weights: one-dim vector, (m, )
    :param bias: scalar
    :param lamda: regularization strength
    :return: (grad with respect to w, grad with respect to b)
    """
    predict = np.dot(weights, samples.T) + bias
    compare_with_1_bool = np.multiply(labels, predict) < 1  # return a bool array
    compare_with_1 = compare_with_1_bool.astype(int)  # return a int array
    compare_column = compare_with_1[:, np.newaxis]  # add newaxis for later computing
    labels_column = labels[:, np.newaxis]  # add newaxis for later computing
    matrix_yx = - (samples * labels_column)
    grad_w = 2 * lamda * weights + (np.sum(matrix_yx * compare_column, axis=0) / len(labels))  # average
    grad_b = np.dot(-labels, compare_with_1)
    return grad_w, grad_b
```

```python
def sigmoid(x):
    """
    x can be scalar or vector, since "/" and 'np.exp' support broadcasting
    """
    return 1 / (1 + np.exp(-x))


1 usage
def cross_entropy_loss(labels: np.ndarray, samples: np.ndarray, weights: np.ndarray, bias, lamda):
    """
    Implementation of cross entropy loss

    :param labels: one-dim vector, (N, ), and each y in {0, 1}
    :param samples: matrix, (N, m)
    :param weights: one-dim vector, (m, )
    :param bias: scalar
    :param lamda: regularization strength
    :return: result of CE loss(a scalar)
    """
    predict = sigmoid(np.dot(weights, samples.T) + bias)
    result_no_regularize = - np.sum(labels * np.log(predict + 1e-9) + (1 - labels) * np.log(1 - predict + 1e-9))
    result = result_no_regularize / len(labels) + lamda * np.sum(weights ** 2)  # average
    return result

```

```python
def grad_cross_entropy(labels: np.ndarray, samples: np.ndarray, weights: np.ndarray, bias, lamda):
    """

    :param labels: one-dim vector, (N, ), and each y in {0, 1}
    :param samples: matrix, (N, m)
    :param weights: one-dim vector, (m, )
    :param bias: scalar
    :param lamda: regularization strength
    :return: grad of CE loss with respect to w, b
    """
    predict = sigmoid(np.dot(weights, samples.T) + bias)
    grad_w = np.dot(samples.T, predict - labels) / len(labels) + 2 * lamda * weights  # average
    grad_b = np.sum(predict - labels) / len(labels)  # average
    return grad_w, grad_b
```

两种线性分类的预测器如下

1. 使用hinge loss的：

$$\hat{y} = \mathrm{sign}(\vec{w}^T\vec{x} + b)$$

2. 使用cross-entropy loss的：

$$\hat{y} = \begin{cases} 1, & if \quad \mathrm{sigmoid}(\vec{w}^T\vec{x} + b) \geq 0.5 \\ 0, & if \quad \mathrm{sigmoid}(\vec{w}^T\vec{x} + b) < 0.5 \end{cases}$$

```python
def hinge_predict(self, input_samples):
    """predict result in {-1, 1}"""
    predict = np.sign(np.dot(self.weights, input_samples.T) + self.bias)
    return predict

1 usage
def sigmoid_predict(self, input_samples):
    """predict result in {0, 1}"""
    predict = sigmoid(np.dot(self.weights, input_samples.T) + self.bias)
    result = np.where(predict >= 0.5, 1, 0)
    return result
```

训练过程

数据预处理

1. 要先把csv文件，转换成样本的属性（像素值）向量$\vec{x}^{(\ell)}$，和对应的label $y^{(\ell)}$，然后存好到向量和矩阵中
2. 进行normalization，考虑两种如下：

- 第一种（Standardization）是对每个特征维度，把所有样本减去该特征上的样本均值，并除以标准差（方差开根号），比如现在每一行是一个样本：

$$column^{(\ell)} = \frac{column^{(\ell)} - mean^{(\ell)}}{standard(\ell)}$$

- 这样，每一列都被normalize成了 $\mu = 0$，$\sigma^2 = 1$ 的分布
- 第二种（Min-Max Normalization）是对每个特征维度，把所有样本减去该特征上的最小值，并除以最大值与最小值之差，比如现在每一行是一个样本：

$$column^{(\ell)} = \frac{column^{(\ell)} - min^{(\ell)}}{max^{(\ell)} - min^{(\ell)}}$$

- 这样，每一列的特征都被缩放到了[0, 1]区间内，且最小值是0，最大值是1

```
def standardization(samples):
    mean = np.mean(samples, axis=0)
    std = np.std(samples, axis=0)
    # "samples = (samples - mean) / std" leads to Nan when std = 0
    # samples = np.where(std == 0, 0, (samples - mean) / std) might lead to "RuntimeWarning"
    std_adjust = np.where(std == 0, 1, std)
    samples = (samples - mean) / std_adjust
    return samples


1 usage
def min_max_normalization(samples):
    col_min = np.min(samples, axis=0)
    col_max = np.max(samples, axis=0)
    # still need to consider whether (col_max-col_min)==0
    # difference_adjust = np.where((col_max - col_min) == 0, 1, (col_max - col_min))
    # samples = (samples - col_min) / difference_adjust
    samples = np.where((col_max - col_min) == 0, 0.5, (samples - col_min) / (col_max - col_min))
    return samples
```

3. 随机初始化/零初始化权重向量$\vec{w}$和偏置b，且让$\vec{w} \sim N(\mu = 0, \sigma^2 = 0.01^2)$

```python
def para_initialize(self):
    """
    initialize weights and bias
    """
    np.random.seed(4219)
    self.weights = np.random.randn(self.samples.shape[1]) * 0.01  # normal distribution N(0.0.01^2)
    self.bias = 0.0  # still 0.0
```

训练模型

1. 对于基于不同loss函数的线性分类模型而言：重复epochs次数，用梯度下降方式更新（参数 = 参数 - 学习率*Loss对参数求导），每个epoch内可以采用mini-batch方式。
2. 对于利用SVM包的模型而言：SVC调用 fit 方法可以直接实现整个参数训练过程，且SVM是基于二次规划优化问题求解的。
3. 对于线性分类模型，最后可以利用matplotlib包绘制loss变化曲线。
4. 在main函数中，只需要依次调用这些方法就好：

```python
def main():
    arguments = args()
    train_labels, train_samples_before = extract_data('mnist_01_train.csv')
    train_samples = standardization(train_samples_before)
    # train_samples = min_max_normalization(train_samples_before)
    test_labels, test_samples_before = extract_data('mnist_01_test.csv')
    test_samples = standardization(test_samples_before)
    # test_samples = min_max_normalization(test_samples_before)

    if arguments.model == "linear":
        model = LinearClassifier(arguments.loss_func, train_samples, train_labels, arguments.learning_rate, arguments.
        model.para_initialize()
        model.train()
        model.evaluate(test_samples, test_labels)
        model.plot_loss()
    elif arguments.model == "SVM":
        model = SVM(arguments.kernel_func, train_samples, train_labels)
        model.train()
        model.evaluate(test_samples, test_labels)
```

训练技巧（包括相关超参数的设置）

- **使用mini-batch**：每一个epoch内，把数据集按batch_size划分为多个小批次，然后遍历所有批次，每次只使用小批次内的数据去更新参数：

```python
def data_shuffle(samples, labels):
    indices = np.arange(len(labels))
    np.random.shuffle(indices)
    shuffled_samples = samples[indices]
    shuffled_labels = labels[indices]
    return shuffled_samples, shuffled_labels
```

```python
def train(self):
    """
    train our model with weights already initialized, data already normalized
    """
    for epoch in range(self.epochs + 1):
        # firstly shuffle our data
        shuffled_samples, shuffled_labels = data_shuffle(self.samples, self.labels)

        # then traverse every batch to update
        for j in range(0, len(self.labels), self.batch_size):
            batch_samples = shuffled_samples[j: min(j+self.batch_size, len(self.labels))]
            batch_labels = shuffled_labels[j: min(j+self.batch_size, len(self.labels))]
            self.para_update(batch_labels, batch_samples)

        if epoch % 10 == 0:
            curr_loss = self.curr_loss()
```

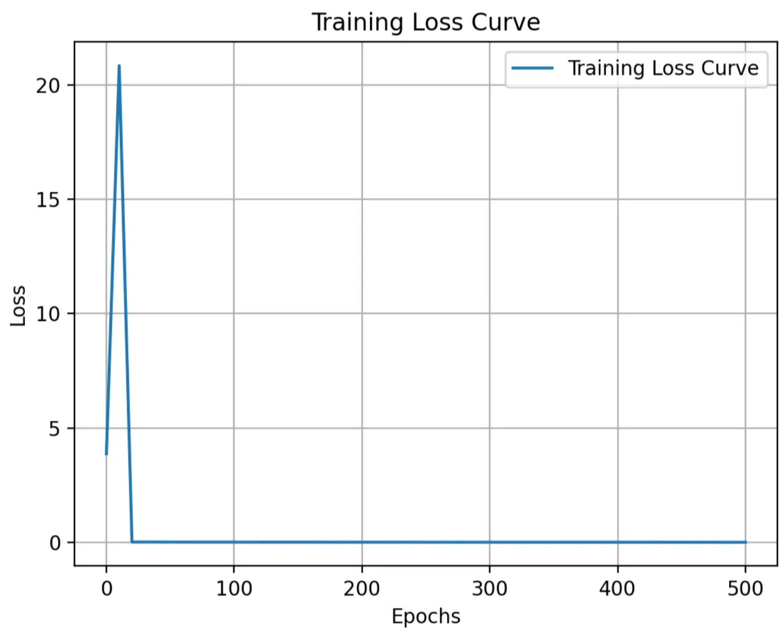
通过使用mini-batch，与没有mini-batch的代码训练效果相比，我发现，由于mini-batch更频繁地更新了模型的参数，此时模型收敛速度得到了明显的加快。

另外，使用了mini-batch后，我发现模型更加稳定了，学习率较大的时候，不会像之前不用mini-batch时出现Loss突然变成几十的情况。

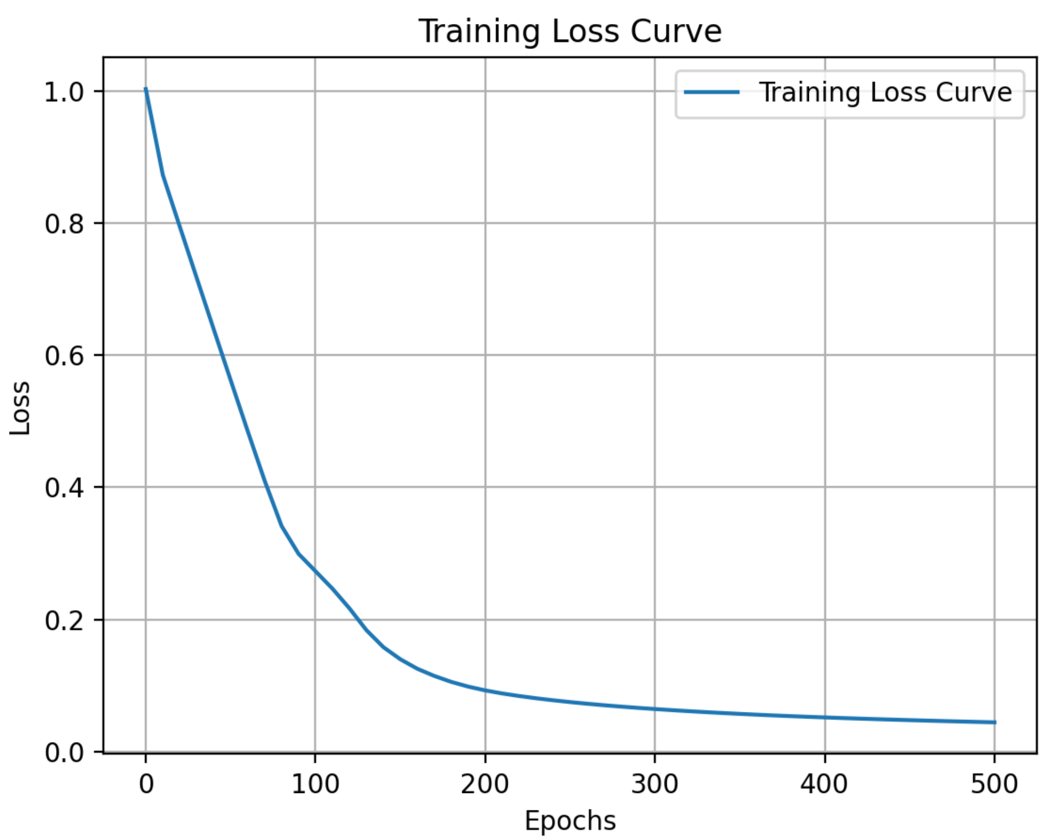
- 在数据预处理中，**standardization的效果比min_max_normalization效果更好，相同epochs内收敛到的Loss更小**。
- **在进行log计算时，添加一个小的常数（如 `1e-9`）**来确保数值运算的稳定性，防止出现比如 $\log(0)$导致的 NaN 或 `-inf` 错误。

```python
result_no_regularize = - np.sum(labels * np.log(predict + 1e-9) + (1 - labels) * np.log(1 - predict + 1e-9)
```

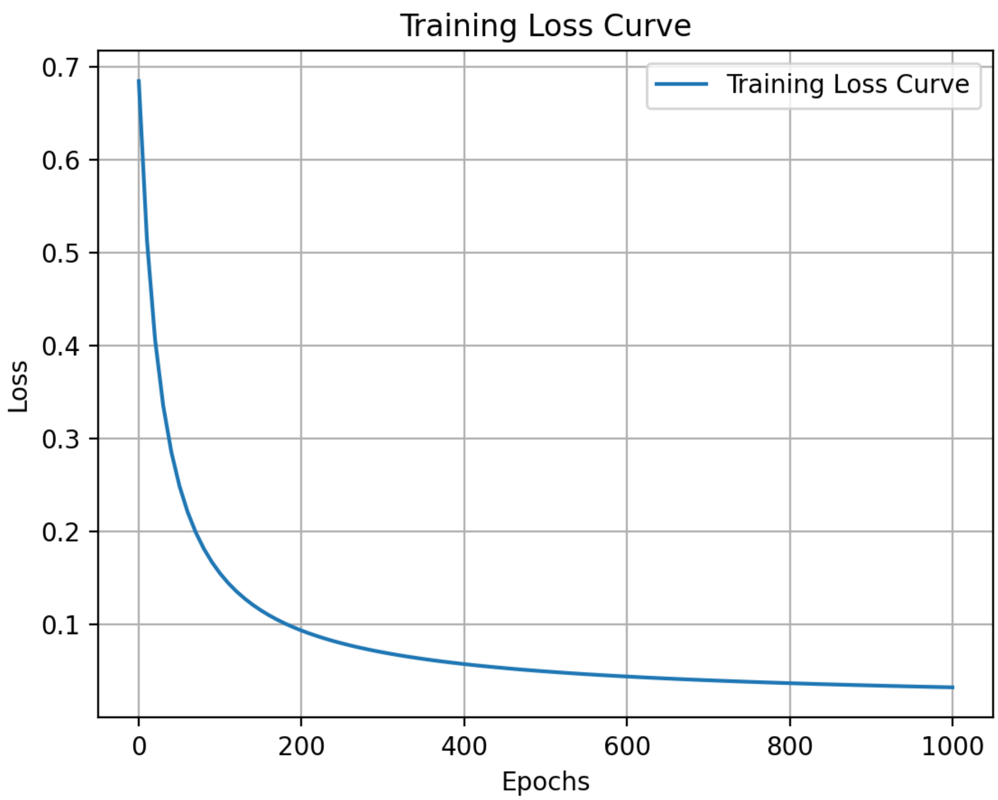
- normalization和standardization 要注意处理**除法分母为0**的情况，否则会出现loss: Nan，或者导致可能的RuntimeWarning。
- **超参数的设置，要防止学习率过大、更新步长过长**，比如，还没有使用mini-batch的时候，使用hinge loss的learning rate设置成1e-2时，训练loss会如下：


Training Loss Curve
Training Loss Curve
Loss
Epochs
0
5
10
15
20
0
100
200
300
400
500

- 改成1e-4就会好很多：


Training Loss Curve
Training Loss Curve
Loss
Epochs
0.0
0.2
0.4
0.6
0.8
1.0
0
100
200
300
400
500

- 而对于cross-entropy，初步测试1e-3就挺好的：



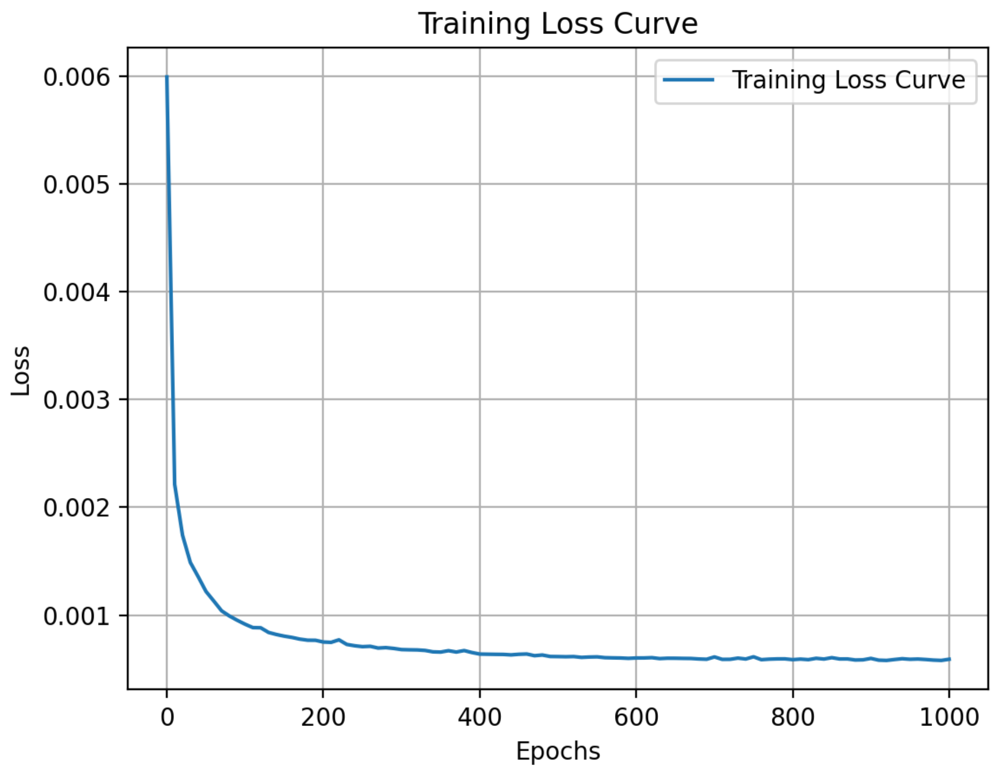
- 多次尝试后，在最终用了mini-batch的代码中，1000 epochs 训练比较合适的一些超参数设置是：（后续测试发现，其实100epochs就能达到一样的accuracy了）

`python main.py --learning_rate 1e-3 --model "linear" --loss_func "hinge" --regular_strength 1e-3 --epochs 1000`

`python main.py --learning_rate 1e-2 --model "linear" --loss_func "hinge" --regular_strength 1e-3 --epochs 1000`

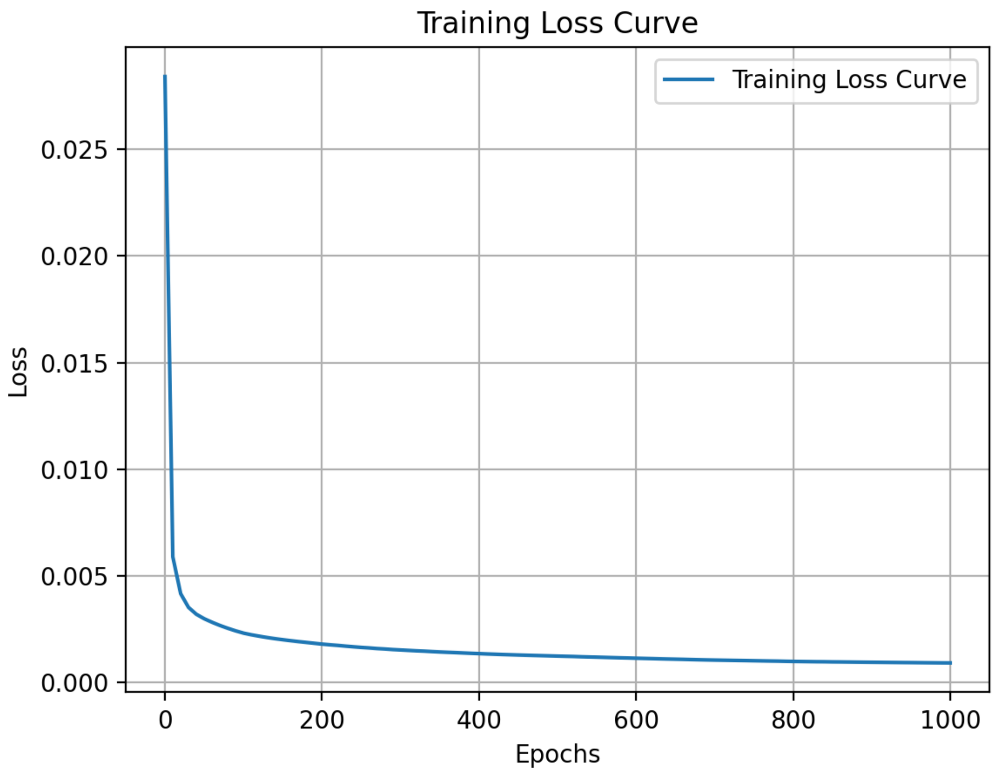
`python main.py --learning_rate 1e-2 --model "linear" --loss_func "cross-entropy" --regular_strength 1e-3 --epochs 1000`

- 对于hinge loss，学习率1e-2最后的loss更小，但可以从曲线上看出来Loss还是会稍有抖动，如下：




```
Epochs 600, Loss: 0.0006017328847017977
Epochs 700, Loss: 0.0006114306222366303
Epochs 800, Loss: 0.0005861576923749647
Epochs 900, Loss: 0.0005978390391112246
Epochs 1000, Loss: 0.0005899729759066
Accuracy: 99.952719%
```

- 学习率1e-3最后的loss大一点，但更稳定，曲线是比较光滑的：




```
Epochs 600, Loss: 0.0011271464006935858
Epochs 700, Loss: 0.001039829538981689
Epochs 800, Loss: 0.000978301679720123
Epochs 900, Loss: 0.0009372693254934227
Epochs 1000, Loss: 0.0009055911885773684
Accuracy: 99.952719%
```

- 比起hinge loss，交叉熵损失在面对较大的学习率时会更稳定一些，但loss会比hinge loss更大。

主要关键代码（所有代码见Code文件夹）

- 这里仅给出上面截图没有出现的LinearClassifier类部分：

```python
class LinearClassifier:
    """
    Linear Classifier using hinge loss or cross-entropy loss
    """
    def __init__(self, loss_func, samples, labels, lr, epochs, regu_strength, batch_size):
        self.loss_func = loss_func
        self.samples = samples
        if loss_func == "hinge":
            self.labels = np.where(labels == 0, -1, labels)
        elif loss_func == "cross-entropy":
            self.labels = labels
        else:
            raise ValueError("Invalid Loss Function")
        self.lr = lr
        self.epochs = epochs
        self.regu_strength = regu_strength
        self.batch_size = batch_size
        self.weights = np.zeros(samples.shape[1])
        self.bias = 0.0  # float
        self.training_loss = []  # for future drawing

```

```python
def para_initialize(self):
    """
    initialize weights and bias
    """
    np.random.seed(4219)
    self.weights = np.random.randn(self.samples.shape[1]) * 0.01  # normal distribution N(0.0.01^2)
    self.bias = 0.0  # still 0.0

1 usage
def para_update(self, batch_labels, batch_samples):
    """
    update weights and bias with gradient descendant method
    """
    if self.loss_func == "hinge":
        # grad_w, grad_b = grad_hinge(self.labels, self.samples, self.weights, self.bias, self.regu_strength)
        grad_w, grad_b = grad_hinge(batch_labels, batch_samples, self.weights, self.bias, self.regu_strength)
    elif self.loss_func == "cross-entropy":
        # grad_w, grad_b = grad_cross_entropy(self.labels, self.samples, self.weights, self.bias, self.regu_strength)
        grad_w, grad_b = grad_cross_entropy(batch_labels, batch_samples, self.weights, self.bias, self.regu_strength)
    else:
        raise ValueError("Invalid Loss Function")
    # then update
    self.weights = self.weights - self.lr * grad_w
    self.bias = self.bias - self.lr * grad_b
```

```python
    1 usage
    def curr_loss(self):
        """
        :return: current loss of the model
        """
        if self.loss_func == "hinge":
            loss_result = hinge_loss(self.labels, self.samples, self.weights, self.bias, self.regu_strength)
        elif self.loss_func == "cross-entropy":
            loss_result = cross_entropy_loss(self.labels, self.samples, self.weights, self.bias, self.regu_strength)
        else:
            raise ValueError("Invalid Loss Function")
        return loss_result

    1 usage
    def hinge_predict(self, input_samples):
        """predict result in {-1, 1}"""
        predict = np.sign(np.dot(self.weights, input_samples.T) + self.bias)
        return predict

    1 usage
    def sigmoid_predict(self, input_samples):
        """predict result in {0, 1}"""
        predict = sigmoid(np.dot(self.weights, input_samples.T) + self.bias)
        result = np.where(predict >= 0.5, 1, 0)
        return result
```

```python
    def train(self):
        """
        train our model with weights already initialized, data already normalized
        using mini-batch
        """
        for epoch in range(self.epochs + 1):
            # firstly shuffle our data
            shuffled_samples, shuffled_labels = data_shuffle(self.samples, self.labels)

            # then traverse every batch to update
            for j in range(0, len(self.labels), self.batch_size):
                batch_samples = shuffled_samples[j: min(j+self.batch_size, len(self.labels))]
                batch_labels = shuffled_labels[j: min(j+self.batch_size, len(self.labels))]
                self.para_update(batch_labels, batch_samples)

            if epoch % 10 == 0:
                curr_loss = self.curr_loss()
                self.training_loss.append(curr_loss)
                if epoch % 100 == 0:
                    print(f"Epochs {epoch}, Loss: {curr_loss}")
```

```python
def predict(self, test_samples):
    """predict result in {0, 1}"""
    if self.loss_func == "hinge":
        predict_0 = self.hinge_predict(test_samples)
        predict = np.where(predict_0 == -1, 0, 1)
    elif self.loss_func == "cross-entropy":
        predict = self.sigmoid_predict(test_samples)
    else:
        raise ValueError("Invalid Loss Function")
    return predict

1 usage
def evaluate(self, test_samples, test_labels):
    """
    evaluate accuracy of our model on testing dataset
    """
    test_predict = self.predict(test_samples)
    accuracy = np.mean(test_predict == test_labels)
    print(f"Accuracy: {accuracy:.6%}")
    return accuracy

1 usage
def plot_loss(self):
    """plot training loss curve"""
    epochs_points = [i * 10 for i in range(len(self.training_loss))]
    plt.figure()
    plt.plot( *args: epochs_points, self.training_loss, label="Training Loss Curve")
```

Result && Analysis

实验结果、分析及讨论

不同核函数的SVM模型性能比较与分析

1. 使用高斯核函数

```
(SVMLab) PS E:\2024-1\ML\Assignment1\LAB1> python main.py --model "SVM" --kernel_func "Gaussian"
Accuracy: 99.432624%
```

2. 使用线性核函数

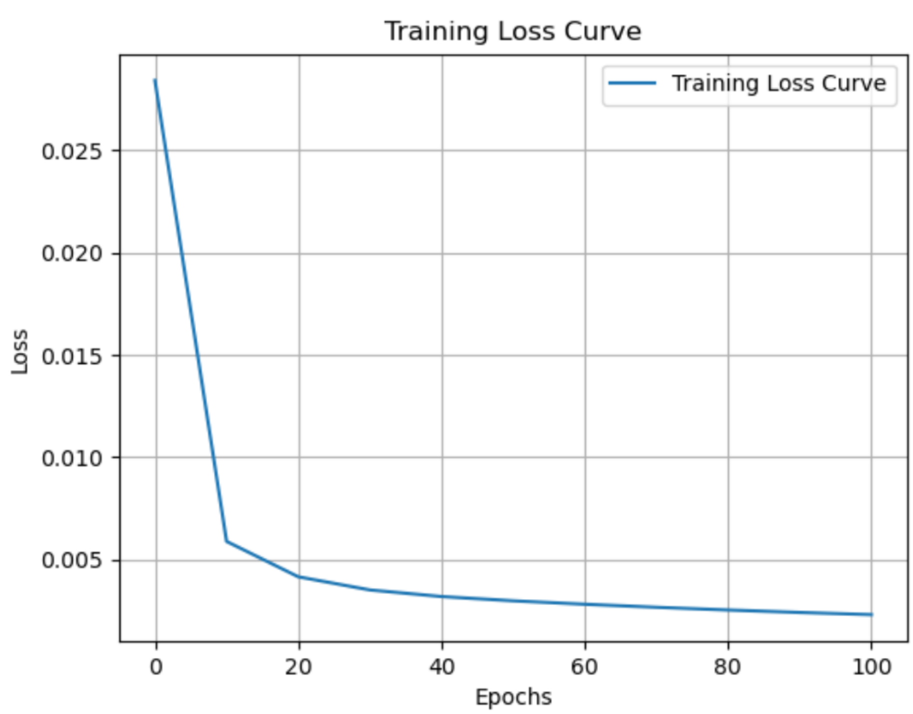
```
(SVMLab) PS E:\2024-1\ML\Assignment1\LAB1> python main.py --model "SVM" --kernel_func "Linear"
Accuracy: 99.905437%
```

1. 发现，这里是使用线性核函数的SVM模型表现更好，在测试集上表现的准确率会比使用高斯核的模型更高一些。
2. 分析原因：高斯核更适用于处理**非线性可分、**更复杂的数据情况，但此时可能我们的数据更偏向于**线性可分**，所以线性核就可以很好地拟合了、且泛化性能好；高斯核反而可能导致在训练集上稍微过拟合，从而泛化性能稍弱，在测试集上的准确率小于高斯核的。
3. 此外，其实准确率还可能与SVC类中参数C、gamma的设置选择有关。
4. 另外，训练时发现，使用高斯核函数的模型出结果要更久一些，也就是使用高斯核函数的模型所需训练时间更长。
5. 分析原因：高斯核函数比线性核函数的表达式更复杂一点，所需计算更多。

> ### SVM模型和 hinge loss，cross-entropy loss实现的模型的比较与分析

1. 使用hinge loss的线性二分类模型：

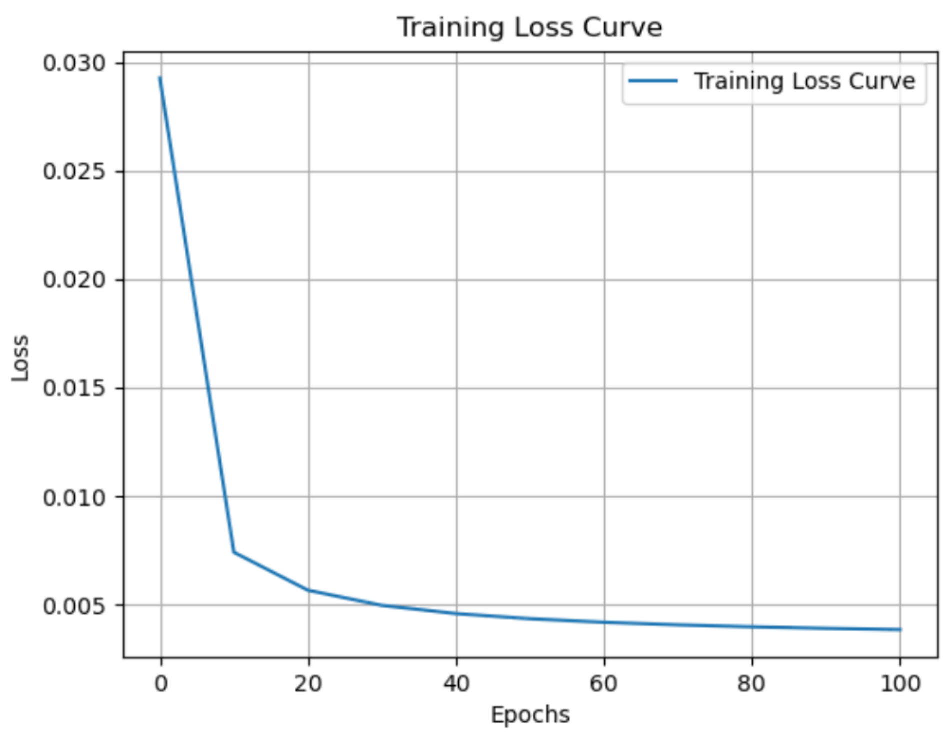
`python main.py --learning_rate 1e-3 --model "linear" --loss_func "hinge" --regular_strength 1e-3 --epochs 100`



```
(SVMLab) PS E:\2024-1\ML\Assignment1\LAB1> python main.py --learning_rate 1e-3 --model "linear" --loss_func "hinge" --regular_str
ength 1e-3 --epochs 100
Epochs 0, Loss: 0.02841984791950114
Epochs 20, Loss: 0.0041555411575420025
Epochs 40, Loss: 0.003189265669150 81
Epochs 60, Loss: 0.002814085027011993
Epochs 80, Loss: 0.0025319229309998586
Epochs 100, Loss: 0.002305287837142966
Accuracy: 99.952719%
```

2. 使用cross-entropy loss的线性二分类模型：

`python main.py --learning_rate 1e-2 --model "linear" --loss_func "cross-entropy" --regular_strength 1e-3 --epochs 100`



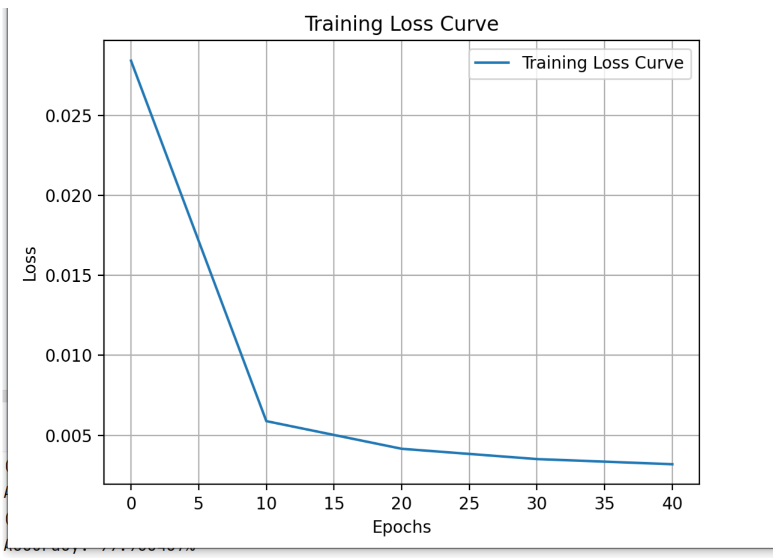

```
(SVMLab) PS E:\2024-1\ML\Assignment1\LAB1> python main.py --learning_rate 1e-2 --model "linear" --loss_func "cross-entropy" --reg
ular_strength 1e-3 --epochs 100
Epochs 0, Loss: 0.029275589416401012
Epochs 20, Loss: 0.005655548164427508
Epochs 40, Loss: 0.004579249319680917
Epochs 60, Loss: 0.004178693779189645
Epochs 80, Loss: 0.0039688536304814045
Epochs 100, Loss: 0.003840023399188348
Accuracy: 99.952719%
```

- 两种线性二分类模型，在100epochs的训练后，就都能有较小的loss和较高的预测准确率了。

- 相比较之下，在学习率相应地恰当选取、模型的loss稳定时，同样的100 epochs训练轮数内，二者预测准确率相同，hinge loss的下降幅度更大，最后的loss更小一些。

再与SVM模型比较：

- 线性核函数的SVM模型得到99.905437%的预测准确率大概需要3秒
- 高斯核函数的SVM模型得到99.432624%的预测准确率大概需要6秒
- 而hinge loss的线性模型得到9.905437%的预测准确率大概需要6~十几秒（20epochs/40epochs）




```
func "Gaussian"
func "Linear"
(SVMLab) PS E:\2024-1\ML\Assignment1\LAB1> python main.py --learning_rate 1e-3 --model "linear" --loss_func "hinge" --regular_strength 1e-3 --epochs 40
Epochs 0, Loss: 0.02841984791950114
Epochs 20, Loss: 0.0041555411575420025
Epochs 40, Loss: 0.00318926566915081
Accuracy: 99.905437%
```

- 可见，直接使用线性核SVM，在现在的数据集情况下会更高效一些。

Conclusion

- 通过梳理SVM模型理论、调用SVM软件包、手动实现hinge loss和cross-entropy loss的线性分类模型，以及相关数据标准化预处理、mini-batch技巧的实现，我对SVM、二分类机器学习模型更加熟悉了。
- 本来想仿照Adam方式进行参数更新，但发现Adam的真正实现要比课件上多一些细节，要考虑训练初期的偏差修正问题，还需进一步探究。