Multi-class classification with Linear Support Vector Machine from Scratch

In this project, we will perform three tasks:

1. Code a binary Support Vector Machine classifier with linear kernel from scratch.
2. Build a multi-class linear SVM classifier on top of the linear SVM to enable the classifier to handle multi-class classification tasks.
3. Compare the multi-class linear SVM classifier with the off-the-shelf linear SVM classifier from scikit-learn.

In [1]:

```python
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score, confusion_matrix, classification_report
from sklearn.preprocessing import StandardScaler

from autograd import grad
import time
```

1. Implementing linear SVM from scratch for binary classification

Background

The SVM for binary classification defines a hyper-plane in a high dimensional space to separate the data points belonging to two classes.

Denote by X the matrix with n observations and p features, w is a vector orthogonal to the hyperplane and b is the bias, defining the hyperplane relative to the origin in the high-dimensional space, the separation equation is:

$$w^Tx + b = 0, w \in \mathbb{R}^p, x \in \mathbb{R}^p, b \in \mathbb{R}$$

The **support vectors** are the data points with the minimum distance to the hyperplane. The distance between the support vector and the hyperplane is called the **margin**. To optimize a SVM, the objective is to maximize the margin:

$$\max_{w,b} M$$

subject to:

- $y_i(w^Tx_i + b) \geq M, \forall i = 1..n$
- $\|w\| = 1$

For $y_i \in \{-1, 1\}$, $y_i(w^Tx_i + b)$ defines the absolute distance between the ith data point to the hyperplane.

Equivalently, by changing the condition on the norm of w such that $\|w\| = \frac{1}{M}$, we can set the objective to be

$$\min_{w,b} \frac{1}{2}\|w\|^2$$

subject to: $y_i(w^Tx_i + b) \geq 1, \forall i = 1..n$

However, the requirement of $y_i(w^Tx_i + b) \geq 1, \forall i = 1..n$ may not always be possible to satisfy since it does not allow any misclassifications. In the case where it can not be satisfied, we will fail to find a hyperplane to separate the data points.

Therefore, to allow for misclassification, we relax the hard constraint and update the objective function as follows:

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\xi_i$$

subject to:

- $\xi_i \geq 0$
- $y_i(w^Tx_i + b) \geq 1 - \xi_i, \forall i = 1..n$

where ξ_i represents the distance to the correct margin when the data point is misclassfied and C is the regularization parameter that controls the level of penality applied when data points are misclassfied.

To optimize the objective function, we write the loss function as follows:

$$L(w, b) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\max(0, 1 - y_i(w^Tx_i + b))$$

We will use the gradient descent algorithm to minimize the loss function.

Implementation

To implement a binary linear SVM, we write a Python class which is initiated with 4 parameters:

- C: the regularization parameter
- alpha: learning rate for the gradient descent algorithm
- max_its: number of iterations for updating the weights
- random_state: random seed for initializing w_0

If the `fit` function is called, the loss function of the model will be optimized or minimized using gradient descent. We record the optimal weights and bias that lead to the minimum loss (cost), for prediction use.

If the `predict` function is called, we will compute the model's predicted values for the input observations using the best fitted weights and bias. We use $w^Tx + b = 0$ as the separating criterion: if $w^Tx + b \geq 0$, predicted label is 1; else -1.

In [2]:

```python
class BinaryLinearSVM:

    def __init__(self, C, alpha, max_its, random_state):

        self.C = C
        self.alpha = alpha
        self.max_its = max_its
        self.random_state = random_state
        self.weight = None
        self.bias = None

    def fit(self, X, y):

        N = X.shape[1]
        X = X.T
        y = y.T

        def model(w, x):

            return w[:-1].T @ x + w[-1]

        def loss(w):

            # loss function
            L = 0.5 * w[:-1].T * w[:-1] + self.C * np.sum(np.maximum(0, 1 - y * model(w, X)))

            return L[0]

        def gradient_descent(g, alpha, max_its, w):

            gradient = grad(g)
            weight_history = [w]     # container for weight history
            cost_history = [g(w)]    # container for corresponding cost history

            for i in range(max_its):
                # evaluate the gradient, store current weights and cost function value
                grad_eval = gradient(w)

                # take gradient descent step
                w = w - alpha * grad_eval

                # record weight and cost
                weight_history.append(w)
                cost_history.append(g(w))

            return weight_history,cost_history

        if self.random_state != None:
            np.random.seed(123)

        # randomly intialize w
        w0 = np.random.normal(size=N+1)

        # perform gradient descent
        w_hist, c_hist = gradient_descent(g=loss, alpha=self.alpha, max_its=self.max_its, w=w0)

        # w_best should have the smallest cost value
        ind = np.argmin(c_hist)
        w_best = w_hist[ind]

        self.weight = w_best[:-1]
        self.bias = w_best[-1]


    def predict(self, X):

        X = X.T
        preds = self.weight.T @ X + self.bias
        labels = [1 if pred >= 0 else -1 for pred in preds]
        pred_labels = np.array(labels)

        return pred_labels
```

Importing data

We will use the breast cancer dataset from scikit-learn to train and test our binary classifier. The breast cancer dataset has two classes: benign (1) and malignant (0).

In [3]:

```python
data = load_breast_cancer()
X, y = data.data, data.target
print(X.shape, y.shape)
```

```
(569, 30) (569,)
```

In [4]:

```python
pd.Series(y).value_counts()
```

Out[4]:

```
1    357
0    212
dtype: int64
```

Change 0 to -1 so that $y_i \in \{-1, 1\}$.

In [5]:

```python
for i in range(len(y)):
    if y[i] == 0:
        y[i] = -1
```

In [6]:

```python
pd.Series(y).value_counts()
```

Out[6]:

```
 1    357
-1    212
dtype: int64
```

Train and test split

We keep 70% of the data as training set and 30% of the data as test set.

In [7]:

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(398, 30) (171, 30) (398,) (171,)
```

We scale the data.

In [8]:

```python
ss = StandardScaler()
X_train_ss = ss.fit_transform(X_train)
X_test_ss = ss.transform(X_test)
```

Linear SVM model fitting

We create a binary linear SVM classifier object from the `BinaryLinearSVM` class and use it to fit the standardized training data `X_train_ss` and `y_train`. We set the regularization parameter to 1.0, gradient descent learning rate to 0.01, and number of iterations to 2000.

In [9]:

```python
binarySVM = BinaryLinearSVM(C=1, alpha=0.01, max_its=2000, random_state=123)
binarySVM.fit(X_train_ss, y_train)
```

Use the fitted `binarySVM` to make predictions for `X_test_ss`.

In [10]:

```python
y_pred = binarySVM.predict(X_test_ss)
```

Model performance:

In [11]:

```python
print(f"accuracy score: {accuracy_score(y_test, y_pred)}")
print(f"F1 score: {f1_score(y_test, y_pred)}")
```

```
accuracy score: 0.9590643274853801
F1 score: 0.9671361502347416
```

In [12]:

```python
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

          -1       0.92      0.97      0.95        63
           1       0.98      0.95      0.97       108

    accuracy                           0.96       171
   macro avg       0.95      0.96      0.96       171
weighted avg       0.96      0.96      0.96       171
```

We obtained a good model performance using our implementation of linear SVM for binary classfication. Now we are going to build a multi-class classifier using the linear SVM.

2. Building a Multi-class Classfier using linear SVM

We will use the One-vs-Rest approach. If the number of classes is n, then we will have n hyperplanes, each separating one class from the rest.

Implementation

To implement a multi-class linear SVM classfier, we write a Python class which is initiated with 4 parameters (same as the binary linear SVM class):

- C: the regularization parameter
- alpha: learning rate for the gradient descent algorithm
- max_its: number of iterations for updating the weights
- random_state: random seed for initializing w_0

If the `fit` function is called on a dataset with n classes, n binary linear SVMs will be fit. Each binary SVM predicts whether a data point belongs to one class (1) or not (-1). The n set of fitted weights (w) and biases (b) will be recorded for prediction use.

If the `predict` function is called, we compute n one dimensional array of predicted values with: $w^Tx + b$. For each data point, it will be classified as the class that produces the largest value of $w^Tx + b$.

In [13]:

```python
class MulticlassLinearSVM():

    def __init__(self, C, alpha, max_its, random_state):

        self.C = C
        self.alpha = alpha
        self.max_its = max_its
        self.random_state = random_state
        self.weights = []
        self.biases = []
        self.binarySVMs = []


    def fit(self, X, y):

        # fit n_classes SVMs and record and fitted weights and biases
        self.classes = list(set(y))
        for cls in self.classes:

            y_bin = np.array([1 if c==cls else -1 for c in y])

            binarySVM = BinaryLinearSVM(C=self.C, alpha=self.alpha, max_its=self.max_its, random_state=self.random_state)
            binarySVM.fit(X, y_bin)

            self.binarySVMs.append(binarySVM)
            self.weights.append(binarySVM.weight)
            self.biases.append(binarySVM.bias)


    def predict(self, X):

        n_classes = len(self.classes)
        preds_matrix = np.zeros((n_classes, X.shape[0])) # n_classes x n_observations

        X = X.T
        for i in range(n_classes):

            # compute value of wx+b for each class for all the data points, store in preds_matrix
            preds = self.weights[i].T @ X + self.biases[i]
            preds_matrix[i, :] = preds

        # data point will be classified as the class that has the largest value of wx+b
        pred_labels = [self.classes[i] for i in np.argmax(preds_matrix, axis=0)]

        return pred_labels
```

Importing data

We will use the iris dataset from scikit-learn. The iris dataset has three classes for three types of irises.

In [14]:

```python
data = load_iris()
X, y = data.data, data.target
print(X.shape, y.shape)
```

```
(150, 4) (150,)
```

In [15]:

```python
pd.Series(y).value_counts()
```

Out[15]:

```
0    50
1    50
2    50
dtype: int64
```

Train and test split

We keep 70% of the data as training set and 30% of the data as test set.

In [16]:

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
print(X_train.shape, X_test.shape, y_train.shape, y_test.shape)
```

```
(105, 4) (45, 4) (105,) (45,)
```

We scale the data:

In [17]:

```python
ss = StandardScaler()
X_train_ss = ss.fit_transform(X_train)
X_test_ss = ss.transform(X_test)
```

Multi-class SVM model fitting

We create a multi-class linear SVM classifier object from the `MulticlassLinearSVM` class and use it to fit the standardized training data `X_train_ss` and `y_train`. We set the regularization parameter to 1.0, gradient descent learning rate to 0.01, and number of iterations to 2000.

In [18]:

```python
multiclassSVM = MulticlassLinearSVM(C=1, alpha=0.01, max_its=2000, random_state=123)
multiclassSVM.fit(X_train_ss, y_train)
```

Use the fitted `multiclassSVM` to make predictions for `X_test_ss`.

In [19]:

```python
y_pred = multiclassSVM.predict(X_test_ss)
```

Model performance:

In [20]:

```python
print(f"accuracy score: {accuracy_score(y_test, y_pred)}")
print(f"F1 score: {f1_score(y_test, y_pred, average='weighted')}")
```

```
accuracy score: 0.9777777777777777
F1 score: 0.9776182336182336
```

In [21]:

```python
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.95      1.00      0.97        19
           1       1.00      0.92      0.96        13
           2       1.00      1.00      1.00        13

    accuracy                           0.98        45
   macro avg       0.98      0.97      0.98        45
weighted avg       0.98      0.98      0.98        45
```

As you can see, the performance of our fitted `multiclassSVM` is very good. It has a high precision and recall score for all three classes.

3. Comparing the Multi-class SVM Classifier with Off-shelf SVM

We will compare the performance of our `MulticlassLinearSVM` class with the off-the-shelf linear SVM classifier from scikit-learn on the iris dataset, in terms of training time and error metrics.

We import the svm from the `sklearn` package. The `LinearSVC` model also tries to minimizes the squared hinge loss, and it also use the One-vs-Rest approach to handle multi-class classification. The methodology is similar to our implementation.

In [22]:

```python
from sklearn import svm
```

Compare training time

Both will train for 2000 iterations

In [23]:

```python
t_start = time.time()
off_shelf_svm = svm.LinearSVC(C=1.0, max_iter=2000)
off_shelf_svm.fit(X_train_ss, y_train)
t_end = time.time()
print(f"Off-the-shelf SVM classifier took {np.round(t_end - t_start, 2)} seconds to fit the training data")
```

```
Off-the-shelf SVM classifier took 0.0 seconds to fit the training data
```

In [28]:

```python
t_start = time.time()
multiclassSVM = MulticlassLinearSVM(C=1, alpha=0.1, max_its=2000, random_state=123)
multiclassSVM.fit(X_train_ss, y_train)
t_end = time.time()
print(f"Multi-class linear SVM classifier took {np.round(t_end - t_start, 2)} seconds to fit the training data")
```

```
Multi-class linear SVM classifier took 19.06 seconds to fit the training data
```

Compare performance metrics

In [25]:

```python
y_pred_1 = off_shelf_svm.predict(X_test_ss)
print("Off-the-shelf SVM classifier:\n")
print(f"accuracy score: {accuracy_score(y_test, y_pred_1)}")
print(f"F1 score: {f1_score(y_test, y_pred_1, average='weighted')}")
print(classification_report(y_test, y_pred_1))
```

```
Off-the-shelf SVM classifier:

accuracy score: 0.9555555555555556
F1 score: 0.9552910052910052
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        19
           1       1.00      0.85      0.92        13
           2       0.87      1.00      0.93        13

    accuracy                           0.96        45
   macro avg       0.96      0.95      0.95        45
weighted avg       0.96      0.96      0.96        45
```

In [29]:

```python
y_pred_2 = multiclassSVM.predict(X_test_ss)
print("Multi-class linear SVM classifier:\n")
print(f"accuracy score: {accuracy_score(y_test, y_pred_2)}")
print(f"F1 score: {f1_score(y_test, y_pred_2, average='weighted')}")
print(classification_report(y_test, y_pred_2))
```

```
Multi-class linear SVM classifier:

accuracy score: 0.9111111111111111
F1 score: 0.9072631072631072
              precision    recall  f1-score   support

           0       0.83      1.00      0.90        19
           1       1.00      0.69      0.82        13
           2       1.00      1.00      1.00        13

    accuracy                           0.91        45
   macro avg       0.94      0.90      0.91        45
weighted avg       0.93      0.91      0.91        45
```

Surprisingly, even though our hand-crafted multi-class linear SVM classifier takes significantly more time to train, it produces a similar model performance to the off-the-shelf SVM (just slightly inferior), given the same regularization value and the same number of iterations.

Reflection

It's apparent that the training of our hand-crafted model takes too long compared to the off-the-shelf model.

To reduce the training time of our hand-crafted model, one method we could use is early-stopping. We could add a `tolerance` parameter and a `early_stopping_rounds` parameter to our model class. They can work together in the following way:

If, for `early_stopping_rounds` iterations, the cost function of our model still hasn't improved (reduced) by `tolerance` amount, we will early-stop the training.

These two parameters can be tuned to help our hand-crafted model achieve better performance.