04_soft_svm_new

August 16, 2025

1 Machine Learning LAB 4: Soft SVM

Course 2024/25: *F. Chiariotti*

The notebook contains a simple learning task over which we will implement a **SOFT SUPPORT VECTOR MACHINE**.

Complete all the **required code sections**.

1.0.1 IMPORTANT for the exam:

The functions you might be required to implement in the exam will have the same signature and parameters as the ones in the labs

1.1 Classification of Stayed/Churned Customers

The Customer Churn table contains information on all 3,758 customers from a Telecommunications company in California in Q2 2022. Companies are naturally interested in churn, i.e., in which users are likely to switch to another company soon to get a better deal, and which are more loyal customers.

The dataset contains three features: - **Tenure in Months**: Number of months the customer has stayed with the company - **Monthly Charge**: The amount charged to the customer monthly - **Age**: Customer's age

The aim of the task is to predict if a customer will churn or not based on the three features.

1.2 Import all the necessary Python libraries and load the dataset

1.2.1 The Dataset

The dataset is a `.csv` file containing three input features and a label. Here is an example of the first 4 rows of the dataset:

Tenure in Months	Monthly Charge	Age	Customer Status
9	65.6	37	0
9	-4.0	46	0
4	73.9	50	1
…	…	…	…

Customer Status is 0 if the customer has stayed with the company and 1 if the customer has churned.

[15]:
```
import numpy as np
import pandas as pd
import random as rnd
from matplotlib import pyplot as plt
from sklearn import linear_model, preprocessing
from sklearn.model_selection import train_test_split

np.random.seed(1)

def load_dataset(filename):
    data_train = pd.read_csv(filename)
    #permute the data
    data_train = data_train.sample(frac=1).reset_index(drop=True) # shuffle the␣
 ↪data
    X = data_train.iloc[:, 0:3].values # Get first two columns as the input
    Y = data_train.iloc[:, 3].values # Get the third column as the label
    Y = 2*Y-1 # Make sure labels are -1 or 1 (0 --> -1, 1 --> 1)
    return X,Y

# Load the dataset
X, Y = load_dataset('data/telecom_customer_churn_cleaned.csv')
```

We are going to differentiate (classify) between **class "1" (churned)** and **class "-1" (stayed)**

1.3 Divide the data into training and test sets

[16]:
```
# Compute the splits
m_training = int(0.75*X.shape[0])

# m_test is the number of samples in the test set (total-training)
m_test =  X.shape[0] - m_training
X_training =  X[:m_training]
Y_training =  Y[:m_training]
X_test =   X[m_training:]
Y_test =  Y[m_training:]

print("Number of samples in the train set:", X_training.shape[0])
print("Number of samples in the test set:", X_test.shape[0])
print("Number of churned users in test:", np.sum(Y_test==-1))
print("Number of loyal users in test:", np.sum(Y_test==1))

# Standardize the input matrix
# The transformation is computed on training data and then used on all the 3␣
 ↪sets
scaler = preprocessing.StandardScaler().fit(X_training)
```

```
np.set_printoptions(suppress=True) # sets to zero floating point numbers <␣
 ↪min_float_eps
X_training =  scaler.transform(X_training)
print ("Mean of the training input data:", X_training.mean(axis=0))
print ("Std of the training input data:",X_training.std(axis=0))

X_test =  scaler.transform(X_test)
print ("Mean of the test input data:", X_test.mean(axis=0))
print ("Std of the test input data:", X_test.std(axis=0))
```

```
Number of samples in the train set: 2817
Number of samples in the test set: 940
Number of churned users in test: 479
Number of loyal users in test: 461
Mean of the training input data: [-0.  0. -0.]
Std of the training input data: [1. 1. 1.]
Mean of the test input data: [0.0575483  0.05550169 0.0073833 ]
Std of the test input data: [0.98593187 0.97629659 1.00427583]
```

We will use **homogeneous coordinates** to describe all the coefficients of the model.

Hint: The conversion can be performed with the function *hstack* in *numpy*.

[17]:
```
def to_homogeneous(X_training, X_test):
    Xh_training = np.hstack([np.ones( (X_training.shape[0], 1) ), X_training])
    Xh_test = np.hstack([np.ones( (X_test.shape[0], 1) ), X_test])
    return Xh_training, Xh_test
```

[18]:
```
# convert to homogeneous coordinates using the function above
X_training, X_test = to_homogeneous(X_training, X_test)
print("Training set in homogeneous coordinates:")
print(X_training[:10])
```

```
Training set in homogeneous coordinates:
[[ 1.         -0.3798618  -1.57020044  0.85174963]
 [ 1.         -0.87925308  0.47180292  1.08667766]
 [ 1.         -0.75440526 -0.6130632  -0.26415851]
 [ 1.         -1.12894873  0.09856916 -0.96894261]
 [ 1.         -1.12894873 -0.58486332 -1.20387064]
 [ 1.          1.78416712  1.39908145  0.08823353]
 [ 1.         -0.7960212  -1.0990965  -0.32289052]
 [ 1.          0.20276137 -0.39907585 -0.96894261]
 [ 1.         -0.62955744  0.63934341  0.96921364]
 [ 1.         -0.87925308  1.13201197 -0.02923048]]
```

1.4 Soft SVM with stochastic gradient descent

Now **complete** the function *sgd_soft_svm* and all auxiliary functions. You should select *a single sample*, compute the gradient, and run the soft SVM version.

The input parameters to pass are:

- X: the matrix of input features, one row for each sample
- Y: the vector of labels for the input features matrix X
- $max\_num\_iterations$: the maximum number of iterations for running the soft SVM
- $averaging\_iterations$: the number of iterations to consider when averaging

The output values are:

- $best\_w$: the vector with the coefficients of the best model
- $margin$: the *margin* of the best model
- $outliers$: the number of outliers that are classified correctly by the best model
- $misclassified$: the number of outliers that are misclassified by the best model

1.5 *Important: Professor took the SGD implementation from the textbook directly.*

This is a modified version of SGD that enjoys faster convergence in cases where the objective function is strongly convex - which happens to be our case.

You are not required to know how this update rule is derived. Just remember the following schema:

[19]:
```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg

img = mpimg.imread('images/svm1.png')
fig = plt.figure(figsize = (2*5,2*4))
plt.imshow(img)
plt.axis('off')
plt.show()
fig = plt.figure(figsize = (2*5,2*4))
ximg = mpimg.imread('images/svm2.png')
plt.imshow(ximg)
plt.axis('off')
plt.show()
```

15.5 Implementing Soft-SVM Using SGD

In this section we describe a very simple algorithm for solving the optimization problem of Soft-SVM, namely,

$$\min_{\mathbf{w}} \left(\frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{i=1}^{m} \max\{0, 1 - y\langle \mathbf{w}, \mathbf{x}_i\rangle\} \right). \tag{15.12}$$

We rely on the SGD framework for solving regularized loss minimization problems, as described in Section 14.5.3.

Recall that, on the basis of Equation (14.15), we can rewrite the update rule of SGD as

$$\mathbf{w}^{(t+1)} = -\frac{1}{\lambda t}\sum_{j=1}^{t} \mathbf{v}_j,$$

where $\mathbf{v}_j$ is a subgradient of the loss function at $\mathbf{w}^{(j)}$ on the random example chosen at iteration j. For the hinge loss, given an example $(\mathbf{x}, y)$, we can choose $\mathbf{v}_j$ to be $\mathbf{0}$ if $y\langle \mathbf{w}^{(j)}, \mathbf{x}\rangle \geq 1$ and $\mathbf{v}_j = -y\,\mathbf{x}$ otherwise (see Example 14.2). Denoting $\boldsymbol{\theta}^{(t)} = -\sum_{j<t} \mathbf{v}_j$ we obtain the following procedure.

SGD for Solving Soft-SVM

goal: Solve Equation (15.12)
parameter: T
initialize: $\boldsymbol{\theta}^{(1)} = \mathbf{0}$
for $t = 1, \ldots, T$
  Let $\mathbf{w}^{(t)} = \frac{1}{\lambda t}\boldsymbol{\theta}^{(t)}$
  Choose i uniformly at random from $[m]$
  If $(y_i\langle \mathbf{w}^{(t)}, \mathbf{x}_i\rangle < 1)$
    Set $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} + y_i\mathbf{x}_i$
  Else
    Set $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)}$
output: $\bar{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T} \mathbf{w}^{(t)}$

In particular, notice that:

- The output, `best_w`, is an **aritmetic average** of the last `averaging_iterations` iterations: $\vec{w} = \frac{1}{K}\sum_{t=(T-K+1)}^{T} \vec{w}_t$.

- The derivative of the hinge loss on a single sample $(\vec{x}_i, y_i)$ is:

$$\partial \vec{w}\, l(x_i, y_i, \vec{w}) = \begin{cases} -x_i \cdot y_i, & \text{if } x_i \text{ outlier (missclass. or within margin)} \\ 0 & \text{if } x_i \text{ correctly classified} \end{cases} \tag{1}$$

- We are minimizing the objective, therefore we need to follow **minus** the gradient.

[20]:
```
def count_outliers(current_w, X, Y):
    # TODO Return a tuple containing 2 numbers:
    # First, the number of total outliers (distance below the margin)
    # Second, the number of misclassified outliers
    scaled_distance = np.dot(X, current_w) * Y
    total_outliers = np.sum(np.where(scaled_distance <= 1, 1, 0))
    missclassified_outliers = np.sum(np.where(scaled_distance < 0, 1, 0))
    return total_outliers, missclassified_outliers

def find_margin(current_w):
    # TODO Return the margin for the selected model
    return 1/np.linalg.norm(current_w)

def sgd_soft_svm(X, Y, lambda_par, max_num_iterations, averaging_iterations):
    #Initialize the weights of the algorithm with w=0
    theta = np.zeros(X.shape[1])
    best_w = np.zeros(X.shape[1])
    num_samples = X.shape[0]
    # Loop the SGD algorithm
    for num_iter in range(max_num_iterations):
        # TODO Compute the current weights
        current_w = theta / (lambda_par * (num_iter + 1))
        random_idx = np.random.choice(np.arange(0, num_samples), size = 1)[0]
        # TODO Compute the gradient over a random point
        x, y = X[random_idx, :], Y[random_idx]
        if (np.dot(x, current_w) * y < 1):
            theta += x * y
        if (num_iter >= max_num_iterations - averaging_iterations):
            # TODO Use the current model for averaging
            best_w += current_w
    best_w = best_w / averaging_iterations
    margin = find_margin(best_w)
    outliers, misclassified = count_outliers(best_w, X, Y)
    return best_w, margin, outliers, misclassified
```

Now we use the implementation to learn a model from the training data using 100000 iterations and averaging over the last 10000. Let us consider $\lambda = 1$. Then we use the best model *best_w* to **predict the labels for the test dataset** and print the fraction of outliers in the test set (the test error that is an estimate of the true loss).

[21]:
```
# Now run the Soft SVM with lambda=0.1. Consider 100000 iterations, 10000 of␣
 ↪which are avera
best_w, margin, outliers, misclassified = sgd_soft_svm(X_training, Y_training,␣
 ↪0.1, int(1e5), int(1e4))
print("Soft SVM model: " + str(best_w))
print("Soft SVM margin: " + str(margin))
print("Total outliers: " + str(outliers))
print("Misclassified points: " + str(misclassified))
true_error = np.asarray(count_outliers(best_w, X_test, Y_test)) / len(Y_test)

print("Total outlier fraction (test set): " + str(true_error[0]))
print("True loss (test set): " + str(true_error[1]))
```

```
Soft SVM model: [ 0.01754132 -0.90525444  0.39616412  0.12808265]
Soft SVM margin: 1.0034454894704299
Total outliers: 1940
Misclassified points: 738
Total outlier fraction (test set): 0.7074468085106383
True loss (test set): 0.251063829787234
```

Now let us compare the result with your perceptron function from Lab 2.

[22]:
```
def perceptron_randomized_update(current_w, X, Y):
    # TODO: write the perceptron update function
    current_labels = np.sign(np.dot(X, current_w))
    missclass_mask = np.where(current_labels * Y <= 0, True, False)
    if np.any(missclass_mask) > 0:
        index = np.random.choice(a = np.where(missclass_mask == True)[0], size␣
 ↪= 1)[0]
        new_w = current_w + X[index, :] * Y[index]
    else:
        new_w = current_w
    return new_w


def count_errors(current_w, X, Y):
    # Find all indices which have a different sign from the corresponding labels
    index = np.nonzero(np.sign(np.dot(X, current_w)) - Y)[0]
    n = np.array(index).shape[0]
    if (n == 0):
        # There are no misclassified samples
        return 0, -1
    return n, index

def perceptron_with_randomization(X, Y, max_num_iterations):
    # TODO: write the perceptron main loop
    # The perceptron should run for up to max_num_iterations, or stop if it␣
 ↪finds a solution with ERM=0
```

```
    n_iter = 0
    w = np.zeros(X.shape[1])
    n, _ = count_errors(w, X, Y)
    best_error = n/X.shape[0]
    best_w = w
    while (best_error > 0 and n_iter < max_num_iterations):
        n_iter += 1
        w = perceptron_randomized_update(w, X, Y)
        n , _ = count_errors(w, X, Y)
        error = n/X.shape[0]
        if error < best_error:
            best_w = w
            best_error = error
    return best_w, best_error
```

Let us test the error of the Soft SVM against the perceptron's best model, using 1000 iterations.

[23]:
```
w_found, error = perceptron_with_randomization(X_training, Y_training, 1000)
print("Training Error of perceptron: " + str(error))
print("Best perceptron model: " + str(w_found))
true_loss_estimate =  count_errors(w_found, X_test, Y_test)[0] / len(Y_test)
 ↪# Error rate on the test set
print("Test Error of perceptron: " + str(true_loss_estimate))
```

```
Training Error of perceptron: 0.2463613773517927
Best perceptron model: [-1.         -3.98981197  1.04435555  1.20387064]
Test Error of perceptron: 0.25957446808510637
```

Now we can try to see the effect of λ. Consider values $10, 1, 0.1, 0.01, 0.001$ and run a K-fold cross validation (you can use the code from Lab 3). Plot the margin and outlier count. Use the loss (i.e., the number of **misclassified** points) as a score.

Note: "perf" probably stands for "performance" in the mind of Chiarotti, but here he stores the LOSS inside it, just to make things even more confuse.

Note: you expect that, when increasing `lambda_par`, the svm becomes softer and softer, allowing more points inside the margin. You therefore expect that the total number of outliers will increase with lambda. You should quantify the **loss**, instead, as the number of **missclassified** samples.

[24]:
```
def K_fold(X_training: np.ndarray, Y_training: np.ndarray, lambda_vec: np.
 ↪ndarray, K: int) -> None:
    ## TODO: perform cross-validation
    # Divide training set in K folds
    max_idx = X_training.shape[0]
    fold_points = int(np.floor(max_idx / K)) # I will not use this.
    shuffled_idxs = np.arange(0, max_idx) # preliminary shuffle of the whole
 ↪dataset (to ensure homogeneous folds)
    np.random.shuffle(shuffled_idxs) # remember that shuffle is in-place
```

```
    validation_idxs = np.array_split(shuffled_idxs , K) # this is much cleaner..
 ↪.
    fold_idxs = [np.setdiff1d(shuffled_idxs, validation_idxs[k]) for k in␣
 ↪range(K)] # ... than what he does.

    results = []
    models = []
    best_perf = 10e9 # this variable will store the LOSS
    best = lambda_vec[0]

    for lambda_idx in range(len(lambda_vec)):
        lambda_perf = 0 # stores the LOSS
        for test in range(K):
            # Cross-validation step
            Xt_k, Yt_k = X_training[fold_idxs[test], :],␣
 ↪Y_training[fold_idxs[test]]
            Xv_k, Yv_k = X_training[validation_idxs[test], :],␣
 ↪Y_training[validation_idxs[test]]
            best_w, _, _, _ = sgd_soft_svm(Xt_k, Yt_k, lambda_vec[lambda_idx],␣
 ↪int(1e5), int(1e4))
            _ , misclassified_outliers_v = count_outliers(best_w, Xv_k, Yv_k)
            lambda_perf += misclassified_outliers_v/ Xv_k.shape[0] # accuracy
        lambda_perf /= K
        results.append(lambda_perf)
        models.append(sgd_soft_svm(X_training, Y_training,␣
 ↪lambda_vec[lambda_idx], int(1e5), int(1e4)))
        if (best_perf > lambda_perf):
            # Improvement on the model
            best_perf = lambda_perf
            best = sgd_soft_svm(X_training, Y_training, lambda_vec[lambda_idx],␣
 ↪int(1e5), int(1e4))
    return best, best_perf, models, results
```

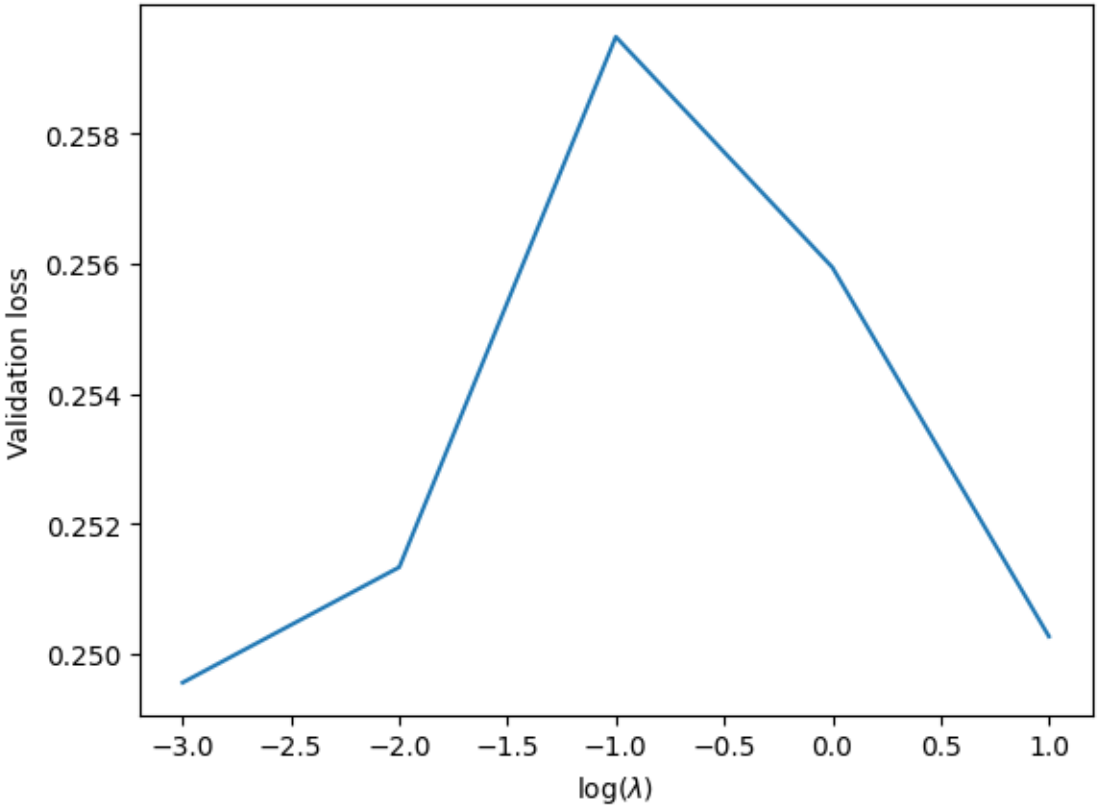
[25]:
```
# Run the training with K-fold cross-validation and plot the score
K = 5
lambda_par = [10, 1, 1e-1, 1e-2, 1e-3]

best_model, best_perf, models, results = K_fold(X_training, Y_training,␣
 ↪lambda_par, K)
print(best_model, results)
plt.plot(np.log10(lambda_par), results)
plt.xlabel('$\log(\lambda)$')
plt.ylabel('Validation loss')
plt.show()
```

```
<>:8: SyntaxWarning: invalid escape sequence '\l'
<>:8: SyntaxWarning: invalid escape sequence '\l'
```

```
/var/folders/vk/kftm8379123bsmwrdp8l0xr00000gn/T/ipykernel_3084/495028130.py:8:
SyntaxWarning: invalid escape sequence '\l'
  plt.xlabel('$\log(\lambda)$')
```

```
(array([-0.06183047, -1.17990577,  0.68486479,  0.2084299 ]),
np.float64(0.7238620390842467), np.int64(1601), np.int64(695))
[np.float64(0.2502708388445889), np.float64(0.2559471171409496),
np.float64(0.2594913268583954), np.float64(0.25133592834737917),
np.float64(0.24956161898643286)]
```



What does this tell you about the margins? Which one should you choose? Now compute the test loss of the best Soft SVM. What does this tell you about the algorithm choice over this problem?

[26]:
```
true_error = np.asarray(count_outliers(best_model[0], X_test, Y_test)) /␣
  ↪len(Y_test)
print("Total outlier fraction (test set): " + str(true_error[0]))
print("True loss (test set): " + str(true_error[1]))
```

```
Total outlier fraction (test set): 0.5808510638297872
True loss (test set): 0.24361702127659574
```

My Answer

In the plot of the validation loss versus lambda, I see that there are no significative variations, as the loss oscillates closely around 0.25. It seems that, for this specific learning task, using a harder (small lambda) or softer (big lambda) SVM does not make a significative difference. The validation loss of SVM is also approximately the same as the test loss of the randomized perceptron.

This indicates that the training set is well representative of the underlying distribution, therefore the algorithm is able to learn an accurate separation boundary independently of the margin width.

Indeed, a further investigation shows that the separating planes learned by the algorithm are all more or less parallel at varying lambda.

In principle, I should choose as best model the one hat minimizes the validation loss, but as said, in this case all models perform similiarly.

The test loss is practically equivalent to the validation loss, which further indicates that the model generalizes well and the SVM is not overfitting on this learning task.

[27]:
```
## CELL ADDED BY ME

best_weights = best_model[0][1:]

print("lambda" + "        " + "cos" + "         " + "angle (degrees)")
for i, model in enumerate(models):
    weights = model[0][1:]
    cos = np.abs(np.dot(best_weights, weights))/(np.linalg.norm(best_weights) *␣
 ↪np.linalg.norm(weights))
    theta = np.arccos(cos) * (180 /np.pi) # radians to degrees
    print(str(lambda_par[i]) + "  " + str(cos) + "  " + str(theta))
```

```
lambda        cos         angle (degrees)
10  0.9869282907176984  9.27423378186751
1  0.9852737116354466  9.845067256541556
0.1  0.9939608997076825  6.300029107477185
0.01  0.9994676113778042  1.8696972215427436
0.001  0.9995800539737584  1.6605408649698736
```