SVM & T-fold Cross Validation

February 14, 2021

1 Support Vector Machines (SVMs)

In this notebook, we will learn a linear and kernalised method of SVMs, which can be used for both regression and classification. To start with, we will focus on binary classification. We will use stochastic gradient descent (SGD) for the optimisation of the hinge loss.

We will work with the Breast Cancer Wisconsin (Diagnostic) Data Set, which you first need to download and then load in this notebook. If you faced difficulties downloading this data set from Kaggle, you should download the file directly from Blackboard. The data set contains various aspects of cell nuclei of breast screening images of patients with *(malignant)* and without *(benign)* breast cancer. Our goal is to build a classification model that can take these aspects of an unseen breast screening image, and classify it as either malignant or benign.

If you run this notebook locally on your machine, you will simply need to place the `csv` file in the same directory as this notebook. If you run this notebook on Google Colab, you will need to use

`from google.colab import files`

`upload = files.upload()`

and then upload it from your local downloads directory.

[1]:
```python
# necessary imports
import numpy as np
import pandas as pd
```

[2]:
```python
data = pd.read_csv('./data.csv')

# print shape and last 10 rows
print(data.shape)
data.tail(10)
```

```
(569, 33)
```

[2]:
```
         id diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
559  925291         B        11.51         23.93           74.52      403.5
560  925292         B        14.05         27.15           91.38      600.4
561  925311         B        11.20         29.37           70.67      386.0
562  925622         M        15.22         30.62          103.40      716.9
563  926125         M        20.92         25.09          143.00     1347.0
```

564	926424	M	21.56	22.39	142.00	1479.0
565	926682	M	20.13	28.25	131.20	1261.0
566	926954	M	16.60	28.08	108.30	858.1
567	927241	M	20.60	29.33	140.10	1265.0
568	92751	B	7.76	24.54	47.92	181.0

	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	\
559	0.09261	0.10210	0.11120	0.04105	
560	0.09929	0.11260	0.04462	0.04304	
561	0.07449	0.03558	0.00000	0.00000	
562	0.10480	0.20870	0.25500	0.09429	
563	0.10990	0.22360	0.31740	0.14740	
564	0.11100	0.11590	0.24390	0.13890	
565	0.09780	0.10340	0.14400	0.09791	
566	0.08455	0.10230	0.09251	0.05302	
567	0.11780	0.27700	0.35140	0.15200	
568	0.05263	0.04362	0.00000	0.00000	

	…	texture_worst	perimeter_worst	area_worst	smoothness_worst	\
559	…	37.16	82.28	474.2	0.12980	
560	…	33.17	100.20	706.7	0.12410	
561	…	38.30	75.19	439.6	0.09267	
562	…	42.79	128.70	915.0	0.14170	
563	…	29.41	179.10	1819.0	0.14070	
564	…	26.40	166.10	2027.0	0.14100	
565	…	38.25	155.00	1731.0	0.11660	
566	…	34.12	126.70	1124.0	0.11390	
567	…	39.42	184.60	1821.0	0.16500	
568	…	30.37	59.16	268.6	0.08996	

	compactness_worst	concavity_worst	concave points_worst	symmetry_worst	\
559	0.25170	0.3630	0.09653	0.2112	
560	0.22640	0.1326	0.10480	0.2250	
561	0.05494	0.0000	0.00000	0.1566	
562	0.79170	1.1700	0.23560	0.4089	
563	0.41860	0.6599	0.25420	0.2929	
564	0.21130	0.4107	0.22160	0.2060	
565	0.19220	0.3215	0.16280	0.2572	
566	0.30940	0.3403	0.14180	0.2218	
567	0.86810	0.9387	0.26500	0.4087	
568	0.06444	0.0000	0.00000	0.2871	

	fractal_dimension_worst	Unnamed: 32
559	0.08732	NaN
560	0.08321	NaN
561	0.05905	NaN
562	0.14090	NaN

```
563                        0.09873          NaN
564                        0.07115          NaN
565                        0.06637          NaN
566                        0.07820          NaN
567                        0.12400          NaN
568                        0.07039          NaN

[10 rows x 33 columns]
```

We can see that our data set has 569 samples and 33 columns. The column `id` can be taken as an index for our pandas dataframe and `diagnosis` is the label (either **M: malignant** or **B: benign**).

Let's prepare the data set first of all by (i) cleaning it, (ii) separating label from features, and (iii) splitting it into train and test sets.

[3]:
```
# drop last column (extra column added by pd)
data_1 = data.drop(data.columns[-1], axis=1)
# set column id as dataframe index
data_2 = data_1.set_index(data['id']).drop(data_1.columns[0], axis=1)

# check
data_2.tail()
```

[3]:
```
        diagnosis  radius_mean  texture_mean  perimeter_mean  area_mean  \
id
926424          M        21.56         22.39          142.00     1479.0
926682          M        20.13         28.25          131.20     1261.0
926954          M        16.60         28.08          108.30      858.1
927241          M        20.60         29.33          140.10     1265.0
92751           B         7.76         24.54           47.92      181.0

        smoothness_mean  compactness_mean  concavity_mean  \
id
926424          0.11100           0.11590         0.24390
926682          0.09780           0.10340         0.14400
926954          0.08455           0.10230         0.09251
927241          0.11780           0.27700         0.35140
92751           0.05263           0.04362         0.00000

        concave points_mean  symmetry_mean  …  radius_worst  texture_worst  \
id                                          …
926424              0.13890         0.1726  …        25.450          26.40
926682              0.09791         0.1752  …        23.690          38.25
926954              0.05302         0.1590  …        18.980          34.12
927241              0.15200         0.2397  …        25.740          39.42
92751               0.00000         0.1587  …         9.456          30.37

        perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
```

```
id
926424          166.10     2027.0          0.14100            0.21130
926682          155.00     1731.0          0.11660            0.19220
926954          126.70     1124.0          0.11390            0.30940
927241          184.60     1821.0          0.16500            0.86810
92751            59.16      268.6          0.08996            0.06444

        concavity_worst  concave points_worst  symmetry_worst  \
id
926424           0.4107                0.2216          0.2060
926682           0.3215                0.1628          0.2572
926954           0.3403                0.1418          0.2218
927241           0.9387                0.2650          0.4087
92751            0.0000                0.0000          0.2871

        fractal_dimension_worst
id
926424                  0.07115
926682                  0.06637
926954                  0.07820
927241                  0.12400
92751                   0.07039

[5 rows x 31 columns]
```

We do a bit more preparation by converting the categorical labels into 1 for **M** and -1 for **B**.

[4]:
```python
# convert categorical labels to numbers
diag_map = {'M': 1.0, 'B': -1.0}
data_2['diagnosis'] = data_2['diagnosis'].map(diag_map)

# put labels and features in different dataframes
y = data_2.loc[:, 'diagnosis']
X = data_2.iloc[:, 1:]

# check
print(y.tail())
X.tail()
```

```
id
926424    1.0
926682    1.0
926954    1.0
927241    1.0
92751    -1.0
Name: diagnosis, dtype: float64
```

```
[4]:        radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
id
926424        21.56         22.39          142.00     1479.0          0.11100
926682        20.13         28.25          131.20     1261.0          0.09780
926954        16.60         28.08          108.30      858.1          0.08455
927241        20.60         29.33          140.10     1265.0          0.11780
92751          7.76         24.54           47.92      181.0          0.05263

        compactness_mean  concavity_mean  concave points_mean  symmetry_mean  \
id
926424           0.11590         0.24390              0.13890         0.1726
926682           0.10340         0.14400              0.09791         0.1752
926954           0.10230         0.09251              0.05302         0.1590
927241           0.27700         0.35140              0.15200         0.2397
92751            0.04362         0.00000              0.00000         0.1587

        fractal_dimension_mean  …  radius_worst  texture_worst  \
id                              …
926424                 0.05623  …        25.450          26.40
926682                 0.05533  …        23.690          38.25
926954                 0.05648  …        18.980          34.12
927241                 0.07016  …        25.740          39.42
92751                  0.05884  …         9.456          30.37

        perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
id
926424           166.10      2027.0           0.14100            0.21130
926682           155.00      1731.0           0.11660            0.19220
926954           126.70      1124.0           0.11390            0.30940
927241           184.60      1821.0           0.16500            0.86810
92751             59.16       268.6           0.08996            0.06444

        concavity_worst  concave points_worst  symmetry_worst  \
id
926424           0.4107                0.2216          0.2060
926682           0.3215                0.1628          0.2572
926954           0.3403                0.1418          0.2218
927241           0.9387                0.2650          0.4087
92751            0.0000                0.0000          0.2871

        fractal_dimension_worst
id
926424                  0.07115
926682                  0.06637
926954                  0.07820
927241                  0.12400
92751                   0.07039
```

```
[5 rows x 30 columns]
```

As with any data set that has features over different ranges, it's required to standardise the data before.

[5]:
```
## EDIT THIS FUNCTION - DONE
def standardise(X):
  mu = np.mean(X, 0)
  sigma = np.std(X, 0)
  X_std = (X - mu) / sigma ## <-- EDIT THIS LINE - DONE
  return X_std
```

[6]:
```
X_std = standardise(X)

# check
X_std.tail()
```

[6]:
```
        radius_mean  texture_mean  perimeter_mean  area_mean  smoothness_mean  \
id
926424     2.110995      0.721473        2.060786   2.343856         1.041842
926682     1.704854      2.085134        1.615931   1.723842         0.102458
926954     0.702284      2.045574        0.672676   0.577953        -0.840484
927241     1.838341      2.336457        1.982524   1.735218         1.525767
92751     -1.808401      1.221792       -1.814389  -1.347789        -3.112085

        compactness_mean  concavity_mean  concave points_mean  symmetry_mean  \
id
926424          0.219060        1.947285             2.320965      -0.312589
926682         -0.017833        0.693043             1.263669      -0.217664
926954         -0.038680        0.046588             0.105777      -0.809117
927241          3.272144        3.296944             2.658866       2.137194
92751          -1.150752       -1.114873            -1.261820      -0.820070

        fractal_dimension_mean  …  radius_worst  texture_worst  \
id                              …
926424               -0.931027  …      1.901185       0.117700
926682               -1.058611  …      1.536720       2.047399
926954               -0.895587  …      0.561361       1.374854
927241                1.043695  …      1.961239       2.237926
92751                -0.561032  …     -1.410893       0.764190

        perimeter_worst  area_worst  smoothness_worst  compactness_worst  \
id
926424         1.752563    2.015301          0.378365          -0.273318
926682         1.421940    1.494959         -0.691230          -0.394820
926954         0.579001    0.427906         -0.809587           0.350735
```

```
927241        2.303601    1.653171          1.430427            3.904848
92751        -1.432735   -1.075813         -1.859019           -1.207552

        concavity_worst  concave points_worst  symmetry_worst  \
id
926424         0.664512              1.629151       -1.360158
926682         0.236573              0.733827       -0.531855
926954         0.326767              0.414069       -1.104549
927241         3.197605              2.289985        1.919083
92751         -1.305831             -1.745063       -0.048138

        fractal_dimension_worst
id
926424                -0.709091
926682                -0.973978
926954                -0.318409
927241                 2.219635
92751                 -0.751207

[5 rows x 30 columns]
```

[7]:

```
# insert 1 in every row for the intercept b
X_std.insert(loc=len(X_std.columns), column='intercept', value=1)

# split into train and test set
# stacking data X and labels y into one matrix
# the newaxis bit converts y into a column vector instead of a 1D row vector
data_split = np.hstack((X_std, np.array(y)[:, np.newaxis]))

# shuffling the rows
np.random.shuffle(data_split)

# we split train to test as 70:30
split_rate = 0.7
train, test = np.split(data_split, [int(split_rate*(data_split.shape[0]))])

X_train = train[:,:-1]
y_train = train[:, -1]

X_test = test[:,:-1]
y_test = test[:, -1]

y_train = y_train.astype(float)
y_test = y_test.astype(float)
```

1.1 Linear SVM

We start with defining the hinge loss as

$$\mathcal{L}(\boldsymbol{w}) = \frac{1}{2}\|\boldsymbol{w}\|^2 + \frac{\lambda}{n}\sum_{i=1}^{n} \max\left(0, 1 - y_i(\boldsymbol{w} \cdot x_i + b)\right).$$

where $\boldsymbol{w}$ is the vector of weights, λ the regularisation parameter, and b the intercept which is included in our `X` as an additional column of 1's.

[8]:
```
# EDIT THIS FUNCTION - DONE
def compute_cost(W, X, y, regul_strength=1e5):
  n = X.shape[0]
  distances = 1 - y * np.dot(X, W)  ## <-- EDIT THIS LINE - DONE
  distances[distances < 0] = 0  # equivalent to max(0, distance)
  hinge = regul_strength * (np.sum(distances) / n)  ## <-- EDIT THIS LINE - DONE

  # calculate cost
  cost = 1 / 2 * np.dot(W, W) + hinge
  return cost
```

Next, we need the gradients of this cost function.

[9]:
```
# calculate gradient of cost
def calculate_cost_gradient(W, X_batch, y_batch, regul_strength=1e5):
  # if only one example is passed
  if type(y_batch) == np.float64:
      y_batch = np.asarray([y_batch])
      X_batch = np.asarray([X_batch])  # gives multidimensional array

  distance = 1 - (y_batch * np.dot(X_batch, W))
  dw = np.zeros(len(W))

  for ind, d in enumerate(distance):
      if max(0, d)==0:
          di = W
      else:
          di = W - (regul_strength * y_batch[ind] * X_batch[ind])
      dw += di

  dw = dw/len(y_batch)  # average
  return dw
```

Both of the two previous functions are then used in SGD (stochastic gradient descent) to update the weights iteratively with a given learning rate α. We also implement a stop criterion that ends the learning as soon as the cost function has not changed more than a manually determined percentage.

We know that the learning happens through updating the weights according to

$$\boldsymbol{w} = \boldsymbol{w} - \alpha\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}}$$

where $\frac{\partial \mathcal{L}}{\partial w}$ is the gradient of the hinge loss we have computed in the previous cell.

[10]:
```
# EDIT THIS FUNCTION
def sgd(X, y, max_iterations=2000, stop_criterion=0.01, learning_rate=1e-5,␣
 ↪regul_strength=1e5, print_outcome=False):
  # initialise zero weights
  weights = np.zeros(X.shape[1])
  nth = 0
  # initialise starting cost as infinity
  prev_cost = np.inf

  # stochastic gradient descent
  for iteration in range(1, max_iterations):
      # shuffle to prevent repeating update cycles
      np.random.shuffle([X, y])
      for ind, x in enumerate(X):
          ascent = calculate_cost_gradient(weights, x, y[ind], regul_strength)␣
 ↪## <-- EDIT THIS LINE - DONE
          weights = weights - (learning_rate * ascent)

      # convergence check on 2^n'th iteration
      if iteration==2**nth or iteration==max_iterations-1:
          # compute cost
          cost = compute_cost(weights, X, y, regul_strength)  ## <-- EDIT THIS␣
 ↪LINE - DONE
          if print_outcome:
            print("Iteration is: {}, Cost is: {}".format(iteration, cost))
          # stop criterion
          if abs(prev_cost - cost) < stop_criterion * prev_cost:
              return weights

          prev_cost = cost
          nth += 1

  return weights
```

Now, we can take these functions and train a linear SVM with our training data.

[11]:
```
# train the model
W = sgd(X_train, y_train, max_iterations=2000, stop_criterion=0.01,␣
 ↪learning_rate=1e-3, regul_strength=1e3, print_outcome=True)
print("Training finished.")
```

```
Iteration is: 1, Cost is: 643.6344645837801
Iteration is: 2, Cost is: 963.0193861899953
Iteration is: 4, Cost is: 670.9963277398275
Iteration is: 8, Cost is: 836.9168198632913
Iteration is: 16, Cost is: 740.2278402108747
```

```
Iteration is: 32, Cost is: 571.910922482765
Iteration is: 64, Cost is: 778.1430215536361
Iteration is: 128, Cost is: 545.3824443539849
Iteration is: 256, Cost is: 512.7494745850026
Iteration is: 512, Cost is: 453.97725378828716
Iteration is: 1024, Cost is: 742.8344529665217
Iteration is: 1999, Cost is: 655.4247548686748
Training finished.
```

To evaluate the mean accuracy in both train and test set, we write a small function called `score`.

[12]:
```
## EDIT THIS FUNCTION - DONE
def score(W, X, y):
  y_preds = np.array([])
  for i in range(X.shape[0]):
    y_pred = np.sign(np.dot(X[i], W))
    y_preds = np.append(y_preds, y_pred)

  return np.float(sum(y_preds == y)) / float(len(y)) ## <-- EDIT THIS LINE -␣
↪DONE
```

[13]:
```
print("Accuracy on train set: {}".format(score(W, X_train, y_train)))
print("Accuracy on test set: {}".format(score(W, X_test, y_test)))
```

```
Accuracy on train set: 0.9472361809045227
Accuracy on test set: 0.9590643274853801
```

Questions:

1. What are other evaluation metrices besides the accuracy? Implement them and assess the performance of our classification algorithm with them.

- bias, variance, Confusion Matrix

2. What makes other evaluation metrices more appropriate given our unbalanced data set *(we have more benign than malignant examples)*?

- Can get Type 1 and Type 2 errors and adjust the model as appropriate

3. Try different learning rates, regularisation strengths and number of iterations independently. What can you observe? Can you achieve higher accuracies?

- Higher learning rate (closer to 1) leads to fewer iterations till convergence for the cost.
- As regularisation strength increases, so does accuracy (as fewer wrongly classified points are allowed)
- Greater number of iterations leads to higher accuracy (up to convergence of cost, after which increasing iterations doesn't affect accuracy too much)

4. What is your understanding why have we used the hinge loss with this data set of 31 features?

- Idea that a hard margin is not possible

5. Can you think of other loss functions instead of the hinge loss? What is your intuition how they will perform compared to the hinge loss? You could try implementing one and compare the results.

- Square Loss or Logistic Loss
- They give us more information than we need

1.2 *T*-fold cross validation

Now we repeat the same procedure as above but do not only have one train-test split, but multiple in a T-fold cross validation method.

[14]:
```
def cross_val_split(data, num_folds):
  fold_size = int(len(data) / num_folds)
  data_perm = np.random.permutation(data)
  folds = []
  for k in range(num_folds):
    folds.append(data_perm[k*fold_size:(k+1)*fold_size, :])

  return folds
```

[15]:
```
# evaluate
folds = cross_val_split(train, 5)
```

[16]:
```
## EDIT THIS FUNCTION - DONE
def cross_val_evaluate(data, num_folds):

  folds = cross_val_split(data, num_folds)

  train_scores = []
  val_scores = []

  for i in range(len(folds)):
    print('Fold', i+1)
    # define the training set (i.e. selecting all folds and deleting the one␣
↪used for validation)
    train_set = np.delete(np.asarray(folds).reshape(len(folds), folds[0].
↪shape[0], folds[0].shape[1]), i, axis=0)
    train_folds = train_set.reshape(len(train_set)*train_set[0].shape[0],␣
↪train_set[0].shape[1])
    X_train = train_folds[:,:-1]   ## <-- EDIT THIS LINE - DONE
    y_train = train_folds[:, -1]

    # define the validation set
    val_fold = folds[i]   ## <-- EDIT THIS LINE - DONE
    X_val = val_fold[:,:-1]   ## <-- EDIT THIS LINE - DONE
    y_val = val_fold[:, -1]
```

```
    # train the model
    W = sgd(X_train, y_train, max_iterations=1025, stop_criterion=0.01,␣
↪learning_rate=1e-3, regul_strength=1e3)
    print("Training finished.")

    # evaluate
    train_score = score(W, X_train, y_train)
    val_score = score(W, X_val, y_val)
    print("Accuracy on train set #{}: {}".format(i+1, train_score))
    print("Accuracy on validation set #{}: {}".format(i+1, val_score))

    train_scores.append(train_score)
    val_scores.append(val_score)

  return train_scores, val_scores
```

[17]:
```
train_scores, val_scores = cross_val_evaluate(train, 5)
```

```
Fold 1
Training finished.
Accuracy on train set #1: 0.9620253164556962
Accuracy on validation set #1: 0.9493670886075949
Fold 2
Training finished.
Accuracy on train set #2: 0.9620253164556962
Accuracy on validation set #2: 0.9620253164556962
Fold 3
Training finished.
Accuracy on train set #3: 0.9651898734177216
Accuracy on validation set #3: 0.9746835443037974
Fold 4
Training finished.
Accuracy on train set #4: 0.9651898734177216
Accuracy on validation set #4: 0.9493670886075949
Fold 5
Training finished.
Accuracy on train set #5: 0.9841772151898734
Accuracy on validation set #5: 0.8860759493670886
```

Finally, let's compute the mean accuracy.

[18]:
```
print(np.mean(val_scores))
```

```
0.9443037974683545
```