homework03

February 7, 2021

0.1 REI602M Machine Learning - Homework 3

0.1.1 Due: Sunday 7.1.2021

Objectives: Classification, support vector machines, text classification

Name: Alexander Guðmundsson, **email:** alg35@hi.is, **collaborators:** (if any)

Please provide your solutions by filling in the appropriate cells in this notebook, creating new cells as needed. Hand in your solution on Gradescope. Make sure that you are familiar with the course rules on collaboration (encouraged) and copying (very, very, bad).

1) [Spam filtering, 30 points - This is based on a problem from Andrew Ng's CS229 machine learning course at Stanford] In recent years, spam on electronic newsgroups has been an increasing problem. Here, you will build a classifier to distinguish between "real" newsgroup messages, and spam messages. For this experiment, a set of spam emails and a set of genuine newsgroup messages have been obtained. Using only the subject line and body of each message, we'll learn to distinguish between the spam and non-spam. All the files for the problem are in the file `email_spam.zip`. In order to get the text emails into a form usable by a off-the shelf classifier, some preprocessing on the messages has already been performed. You can look at two sample spam emails in the files `spam_sample_original`, and their preprocessed forms in the files `spam_sample_preprocessed*`. The first line in the preprocessed format is just the label and is not part of the message. The preprocessing ensures that only the message body and subject remain in the dataset; email addresses (EMAILADDR), web addresses (HTTPADDR), currency (DOLLAR) and numbers (NUMBER) were also replaced by the special tokens to allow them to be considered properly in the classification process. (In this problem, we'll going to call the features "tokens" rather than "words," since some of the features will correspond to special values like EMAILADDR. You don't have to worry about the distinction.) The files `news_sample original` and `news_sample_preprocessed` also give an example of a non-spam mail.

The work to extract feature vectors (i.e. classifier inputs) out of the documents has also been done for you, so you can just load in the design matrices (called document-word matrices in text classification) containing all the data. In a document-word matrix, the i-th row represents the i-th document/email, and the j-th column represents the j-th distinct token. Thus, the (i, j)-entry of this matrix represents the number of occurrences of the j-th token in the i-th document.

For this problem, we've chosen as our set of tokens considered (that is, as our vocabulary) only the medium frequency tokens. The intuition is that tokens that occur too often or too rarely do not have much classification value. (Examples tokens that occur very often are words like "the", "and", and "of", which occur in so many emails and are sufficiently content-free that they aren't worth modeling.) Also, words were stemmed using a standard stemming algorithm; basically, this means that "price," "prices" and "priced" have all been replaced with "price," so that they can be treated as the same word. For a list of the tokens used, see the variable file `tokenlist`. Since the document-word matrix is extremely sparse (has lots of zero entries), we have stored it in our own efficient format to save space. You don't have to worry about this format. The file `read_spam_data.py` provides the function `read_matrix` to read in the document-word matrix and labels.

Train a linear SVM on this dataset using the implementation in scikit-learn, `sklearn.svm.LinearSVC` with parameter $C = 0.1$. Evaluate the accuracy on the test set for training sets of size 50, 100, 200, 400, 800 and 1400 and for the full test set as well. What conclusions can you draw from the results?

Comments:

1) To read the training and test data and the list of tokens behind the features use `python`
   `trainMatrix, tokenlist, trainCategory = readMatrix('MATRIX.TRAIN')`
   `testMatrix, tokenlist, testCategory = readMatrix('MATRIX.TEST')`

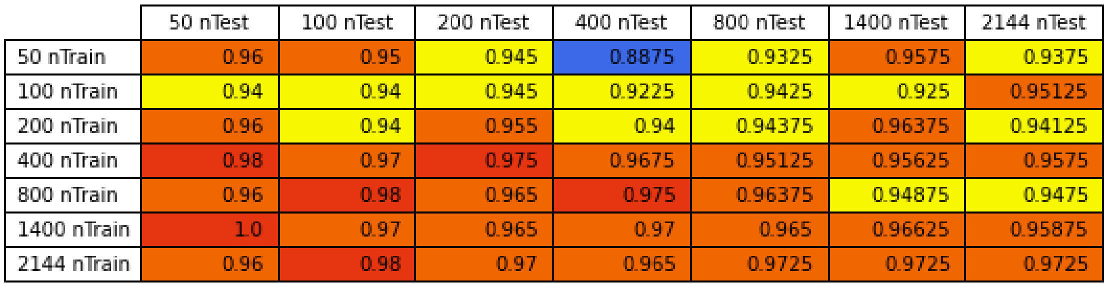
2) To use the `sklearn.svm.LinearSVC` class, you start by calling the `fit` function which solves the SVM optimization problem for the given training set. You then either call `predict` to get predictions for the test set (or other data points) and subsequently evaluate the error rate/accuracy for the classifier "manually"; or you call the `score` function which performs the two operations (prediction and evaluation) in one go. This is completely analogous to how all the classifiers are used in scikit-learn. The Jupyter workbook vika02_logreg.ipynb shows how this is done with the logistic regression classifier.

[3]:
```
import numpy as np
import os
from sklearn.svm import LinearSVC as SVC
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches
from mpl_toolkits.mplot3d import Axes3D
import sys


def data_subsample(X, y, n):
    # Select a random subset of the training data
    perm = np.random.permutation(len(y))
    X_sub=X[perm[0:n],:]
    y_sub=y[perm[0:n]]
    return X_sub, y_sub





sys.path.append("./email_spam")

from read_spam_data import read_matrix
```

```
#insert Data
train_Matrix, tokenlist, trainCategory = read_matrix('./email_spam/MATRIX.
↪TRAIN')
test_Matrix, tokenlist, testCategory = read_matrix('./email_spam/MATRIX.TEST')




unique, counts = np.unique(testCategory, return_counts=True)
print(dict(zip(unique, counts)))

#TestCount
ntests = np.array([50,100,200,400,800,1400, len(train_Matrix)])
scores = [] # [trainsize, testsize, score]



#create labels for visual representation
labels = []
label = lambda c,l: labels.append(mpatches.Patch(color=c,label=l))
label('#E53611','acc >= 0.975')
label('#F06600','0.975 > acc <= 0.95')
label('#f7f702','0.95 > acc <= 0.90')
label('#3C69E7','acc < 0.9')




mpatches.Patch(color='red', label='The red data')


#create an array of scores by sample size
for trainsize in ntests:
    for testsize in ntests:

        fitter = SVC(C = 0.1, max_iter=10000)

        #finding subsamples
        sub_train_Matrix, sub_trainCategory = data_subsample(train_Matrix,␣
↪trainCategory, trainsize)
        sub_test_Matrix, sub_testCategory = data_subsample(test_Matrix,␣
↪testCategory, testsize)

        #adding the sub_training data to linearSVC
        fitter.fit(sub_train_Matrix, sub_trainCategory, )
        #getting score from sub_test data
        score = fitter.score(sub_test_Matrix, sub_testCategory)
        temp = [trainsize, testsize, score]
        scores.append(temp)
scores = np.array(scores)



#create a color vector to paint cells by accuracy
colors = []
for i in scores[:,2]:
    if i >= 0.975:
        colors.append('#E53611')
    elif i < 0.975 and i >= 0.95:
        colors.append('#F06600')

    elif i < 0.95 and i >= 0.9:
        colors.append('#f7f702')
    else:
        colors.append('#3C69E7')


vecn = int(len(scores[:,2])**0.5) #length of m in mxm array from n array
colors = np.array(colors).reshape(vecn,vecn)




celltext = np.reshape(scores[:,2], (vecn,vecn))
table = plt.table(
    cellText = celltext,
    cellColours = colors,
    rowLabels = [str(x) + " nTrain" for x in ntests],
    colLabels = [str(x) + " nTest" for x in ntests],
    loc='center',
)

plt.box(on=None)
plt.axis("off")
table.scale(1.5, 1.5)
plt.legend(handles=labels, bbox_to_anchor=(0.5, 1.5), loc = 'upper right')
plt.show()
```

```
{0: 400, 1: 400}
```

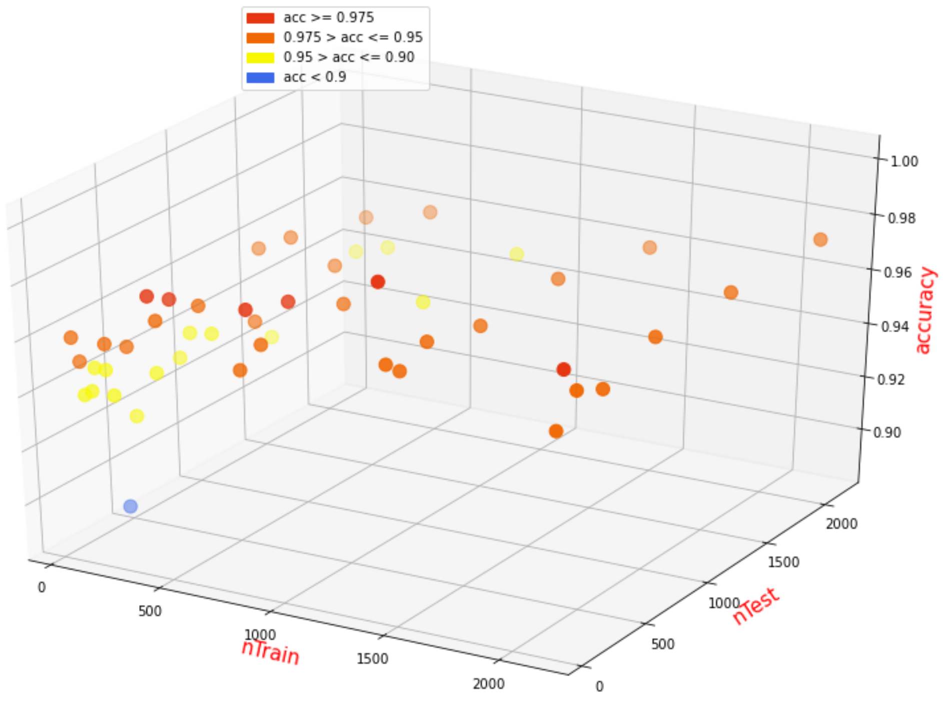
🟥 acc >= 0.975
🟧 0.975 > acc <= 0.95
🟨 0.95 > acc <= 0.90
🟦 acc < 0.9



	50 nTest	100 nTest	200 nTest	400 nTest	800 nTest	1400 nTest	2144 nTest
50 nTrain	0.96	0.95	0.945	0.8875	0.9325	0.9575	0.9375
100 nTrain	0.94	0.94	0.945	0.9225	0.9425	0.925	0.95125
200 nTrain	0.96	0.94	0.955	0.94	0.94375	0.96375	0.94125
400 nTrain	0.98	0.97	0.975	0.9675	0.95125	0.95625	0.9575
800 nTrain	0.96	0.98	0.965	0.975	0.96375	0.94875	0.9475
1400 nTrain	1.0	0.97	0.965	0.97	0.965	0.96625	0.95875
2144 nTrain	0.96	0.98	0.97	0.965	0.9725	0.9725	0.9725

When looking at the output accuracy table, we can see that the fewer the test set, It's more likely for the accuracy to be lower than 0.95, when the training samples are incresed the accuracy goes up, It's also shown below in an 3D scatter plot with number of train values as x axis, number of test values as y axis and accuracy as z axis,

[4]:
```
gfig = plt.figure(figsize=(15,10))
gax = gfig.gca(projection='3d')

gax.scatter(scores[:,0],scores[:,1], scores[:,2], s=100, c = colors.flatten())
gax.set_xlabel('nTrain', fontsize=15, color='r')
gax.set_ylabel('nTest', fontsize=15, color='r')
gax.set_zlabel('accuracy', fontsize=15, color='r')
plt.legend(handles=labels, bbox_to_anchor=(0.5, 1.0), loc = 'upper right')
plt.show()
```



2) [Tweet sentiment, 30 points] Many organizations are interested in analyzing whether given text segments such as news stories and tweets convey positive or negative feeling. In some cases, negative tweets can do significant to company reputation and they are forcde to respond. A system that can automatically analyze text for sentiment is therefore of value. Your task here is to classify tweets sentiment into one of the following categories positive, neutral or negative.

In this exercise you see how raw text containing "tweet extracts" can be converted to feature vectors similar to those that were provided with problem 1). The pandas package is used to read the data from file (`np.genfromtxt` is cumbersome to use here) and scikit-learn used to convert text to features.

a) [20 points] Create a random train/test split using `sklearn.model_selection.train_test_split` and then train a logistic regression classifier on the data using scikit-learn. Use the `multi_class='ovr'` switch to use the one-against-all strategy to handle K>2 classes and set the regularization parameter C to 0.1 to avoid numerical problems during the optimization.

Report the accuracy of your classifier, provide a confusion matrix and comment briefly on the results.

Comments:

1) The data used in this exercise comes from a Machine learning competition on Kaggle. For more details, see here: https://www.kaggle.com/c/tweet-sentiment-extraction You can examine the raw data in e.g. a text editor or Excel (always an important step!)

2) The `CountVectorizer` class counts the occurence of each word in a segment of text. It does

some filtering behind the scenes to remove extremely rare words as well as the most frequent ones. See the documentation for more details. You are free to experiment with the settings if you want.

3) The `LabelEncoder` class is used to convert text labels to integers, e.g. “positive”, “neutral” and “negative” to 1, 0, -1 which are then used as inputs to a classifier.

[5]:
```
# Read sentiment data from text files and convert to vector-based features
import numpy as np
import pandas as pd
from IPython.display import display
from sklearn.preprocessing import LabelEncoder
from sklearn.feature_extraction.text import CountVectorizer

# The text files are somewhat messy, clean-up is probably a good idea
df = pd.read_csv('sentiment_train.csv')
display(df.head())

# Encode tweets using bag-of-words representation
vectorizer = CountVectorizer(min_df=5, max_df=0.95)
X = vectorizer.fit_transform(df['text'].apply(lambda x: np.str_(x)))

# Mapping used to identify original text from feature IDs
inv_map = {v: k for k, v in vectorizer.vocabulary_.items()}


# Encode "positive", neutral" and "negative" labels as integers
le = LabelEncoder()
y = le.fit_transform(df['sentiment'])


# It is important to check for class imbalance
print("X: ", X.shape) # Sanity check
for i in range(3):
    print("Class {} ({}), count={}".format(i, le.classes_[i], np.sum(y == i)))
```

```
       textID                                               text  \
0  a3d0a7d5ad  Spent the entire morning in a meeting w/ a ven...
1  251b6a6766      Oh! Good idea about putting them on ice cream
2  c9e8d1ef1c  says good (or should i say bad?) afternoon!  h...
3  f14f087215         i dont think you can vote anymore! i tried
4  bf7473b12d             haha better drunken tweeting you mean?

                               selected_text sentiment
0  my boss was not happy w/ them. Lots of fun.   neutral
1                                        Good  positive
2  says good (or should i say bad?) afternoon!   neutral
3           i dont think you can vote anymore!  negative
4                                      better  positive
```

```
X:  (27486, 4454)
Class 0 (negative), count=7786
Class 1 (neutral), count=11118
Class 2 (positive), count=8582
```

[6]:
```
from sklearn.model_selection import train_test_split as split
from sklearn.linear_model import LogisticRegression as Logreg
from sklearn.metrics import confusion_matrix



testSize = [0.95, 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, 0.1, 0.05] #testSize

thetas = []
predictions = []
tests = []
for size in testSize:

    X_train, X_test, y_train, y_test = split(X, y, test_size=size,␣
 ↪random_state=10000)
    theta = Logreg(class_weight = 'balanced',multi_class='ovr', C=0.1).
 ↪fit(X_train, y_train)
    thetas.append(theta)
    tests.append([X_test, y_test])
    pred = theta.predict(X_test)
    predictions.append(pred)

thetas = np.array(thetas)
tests = np.array(tests)

print(np.unique(y_test))



for i in range(len(thetas)):
    test_size = testSize[i]
    train_size = format((1 - testSize[i]),".2f")
    score = (thetas[i].score(tests[i,0],tests[i,1]))
    print("accuracy from test size {}, and train size: {} =".format(test_size,␣
 ↪train_size),score)
    print(confusion_matrix(tests[i,1],np.array(predictions[i])))
```

```
[0 1 2]
```

```
accuracy from test size 0.95, and train size: 0.05 = 0.5823376225490197
[[3853 2896  656]
 [2397 6561 1595]
 [ 948 2414 4792]]
accuracy from test size 0.9, and train size: 0.10 = 0.6112862802166708
[[3908 2596  510]
 [2181 6392 1416]
 [ 743 2170 4822]]
accuracy from test size 0.8, and train size: 0.20 = 0.6420937741598072
[[3682 2121  440]
 [1763 5888 1233]
 [ 533 1780 4549]]
accuracy from test size 0.7, and train size: 0.30 = 0.6590094069954784
[[3411 1718  360]
 [1538 5168 1051]
 [ 445 1449 4101]]
accuracy from test size 0.6, and train size: 0.40 = 0.6648071792384186
[[2962 1416  316]
 [1331 4428  906]
 [ 354 1205 3574]]
accuracy from test size 0.5, and train size: 0.50 = 0.6713235829149385
[[2426 1173  273]
 [1057 3771  763]
 [ 277  974 3029]]
accuracy from test size 0.4, and train size: 0.60 = 0.6776716689404275
[[1993  922  197]
 [ 862 3018  599]
 [ 214  750 2440]]
accuracy from test size 0.3, and train size: 0.70 = 0.6837254426388552
[[1483  665  135]
 [ 649 2304  431]
 [ 166  562 1851]]
accuracy from test size 0.2, and train size: 0.80 = 0.6851582393597672
[[1024  417   91]
 [ 441 1499  297]
 [ 111  374 1244]]
accuracy from test size 0.1, and train size: 0.90 = 0.6795198253910513
[[492 229  56]
 [223 739 126]
 [ 58 189 637]]
accuracy from test size 0.05, and train size: 0.95 = 0.6865454545454546
[[235 105  31]
 [107 394  70]
 [ 27  91 315]]
```

```
<ipython-input-6-d89b952d00ba>:22: VisibleDeprecationWarning: Creating an
ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-
tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant
```

```
to do this, you must specify 'dtype=object' when creating the ndarray
  tests = np.array(tests)
```

b) [Feature importance, 10 points] In feature selection (a.k.a. input variable selection) the goal is to identify which features are most relevant for a given classification task. By performing a careful selection of features, the performance of a classifier can often be improved significantly, in particular when data is limited. Alternatively, it can be interesting to identify a minimal set of features for acceptable performance (e.g. due to high costs of collecting/measuring the full set of features). Examining the features most relevant to the classification can also provide valuable insights into the data.

A simple feature selection strategy uses the size of the weights in a linear classifier as measures of feature importance. The larger θ_k is, the larger the role of the corresponding feature in the decision function. The strategy is therefore to rank the features according to θ_k.

The weights are stored in the `coef_` attribute in the LogisticRegression class. This is a n_classes x n_features matrix. For each of the classes, obtain the names of the 10 highest ranking features (in terms of θ's). Comment briefly on the results (you need to consider how the multi-class setting is treated in this case).

[35]:
```
# Insert code here
from itertools import chain
from collections import Counter


coefs = []
for i in thetas:
    coefs.append(i.coef_)
coefs = np.array(coefs)

coefs_sort = []
#coefs sorted
for i in coefs:
    ctemp = []
    for j in range(len(coefs[0])):
        temp = np.argsort(i[j]) #sort the indexes of max rated values to find␣
 ↪best words representing each class.
        ctemp.append(temp)
    coefs_sort.append(ctemp)
    # np.append(indexes, distances_sorted[0])

coefs_sort = np.array(coefs_sort)

n_split,n_classes,n_words = coefs_sort.shape #getting the n*m*l part of the␣
 ↪matrix

n_best_words = n_split*n_classes*10

words_indexes = np.ones(n_best_words).reshape(11,3,10)
```

```
#getting the index value of the most common words


for i in range(len(coefs_sort)):
    for j in range(len(coefs_sort[0])):
        for l in range(10):
            words_indexes[i,j,l] = coefs_sort[i,j,l]


words = np.empty(n_best_words,dtype=object).reshape(11,3,10)
for i in range(len(coefs_sort)):
    for j in range(len(coefs_sort[0])):
        for l in range(10):
            words[i,j,l] = inv_map[words_indexes[i,j,l]] #getting the word by␣
 ↪index


#finding the most common words for each class in all train/test cases
positive = Counter(chain(*words[:,0])).most_common(10)
neutral = Counter(chain(*words[:,1])).most_common(10)
negative = Counter(chain(*words[:,2])).most_common(10)



#showing the top 10 most common words for each classifier
positive = np.array(positive)
neutral = np.array(neutral)
negative = np.array(negative)

print("top 10 words for each class")
print("positive", positive[:,0])
print("neutral", neutral[:,0])
print("negative", negative[:,0])
```

```
top 10 words for each class
positive ['happy' 'love' 'thanks' 'nice' 'awesome' 'great' 'hope' 'thank' 'good'
 'lol']
neutral ['happy' 'thanks' 'great' 'nice' 'awesome' 'love' 'thank' 'good' 'sucks'
 'amazing']
negative ['sad' 'miss' 'hate' 'sorry' 'tired' 'sucks' 'bored' 'sick' 'missed'
 'poor']
```

When looking at the results of top words for each classifier, we see that the top words in neutral is very much like top words in the positive class, and some words like the negative class.

When classifying It's best to first check for the negative words, It's much more unique than the other two.

3) [Stochastic gradient descent for SVM, 40 points]. In this problem you are to implement a stochastic gradient descent algorithm for training a linear SVM. The model is $f_\theta(x) = \theta^T x$ (to include an intercept term you can simply set $x_0 = 1$ as before). The algorithm minimizes the SVM objective function

$$J(\theta) = \frac{\lambda}{2}\theta^T\theta + \frac{1}{n}\sum_{i=1}^{n}\max(0, 1 - y^{(i)}\theta^T x^{(i)}).$$

The hinge loss $\max(0, 1-z)$ is not differentiable at $z = 1$ and this results in an objective function which is not differentiable everywhere, hence the gradient of $J(\theta)$ is not defined everywhere. To deal with this, the SGD algorithm uses the *sub-gradient* of J instead (see below). The algorithm starts from $\theta^{(0)} = 0$ and performs a fixed number of iterations. Step k of the algorithm is as follows:

```
Select $i$ uniformly at random from $[1, n]$
$\alpha^{(k)} = \frac{1}{\lambda k}$
if $y^{(i)}\ (\theta^{(k)})^T x^{(i)} < 1$ then
  $\theta^{(k+1)} = \theta^{(k)} - \alpha^{(k)}(\lambda\theta^{(k)} - y^{(i)}x^{(i)})$
else
  $\theta^{(k+1)} = \theta^{(k)} - \alpha^{(k)}\lambda\theta^{(k)}$
```

where $\theta^{(k)}$ denotes the parameter *vector* in iteration k and $\lambda > 0$ is a regularization hyper-parameter. The step size $\alpha^{(k)}$ decays over the course of iterations (instead to being constant as we've seen previously). This helps to avoid overshooting the minimum.

a) [30 points] Implement the SGD algorithm above. Train an SVM using your algorithm on the data in `synth_train.txt` with $\lambda = 1/100$. Create a scatter plot of the training data (it is a 2D toy data set) and show the decision boundary of the classifier (see Jupyter workbook vika02_logreg). Report the model coefficients and the test set error (fraction of incorrectly classified examples) using the data in `synth_test.txt`

Comments:

1) To sample uniformly at random from $[0, n-1]$ use `np.random.randint`.

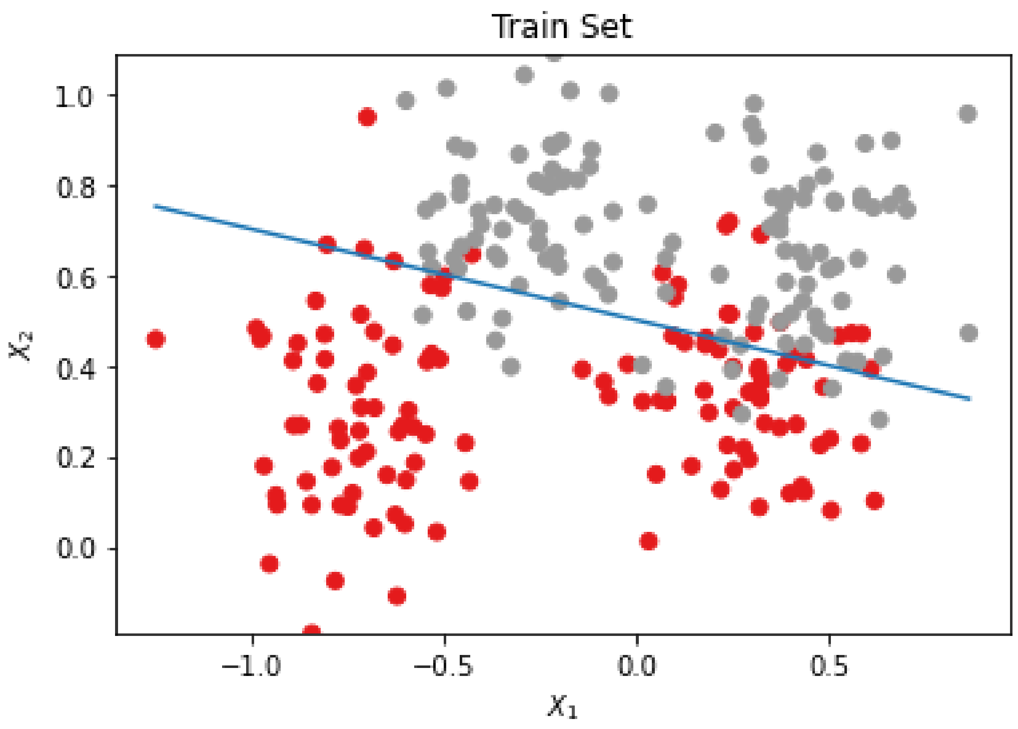
2) Use `np.genfromtxt` to read the data.

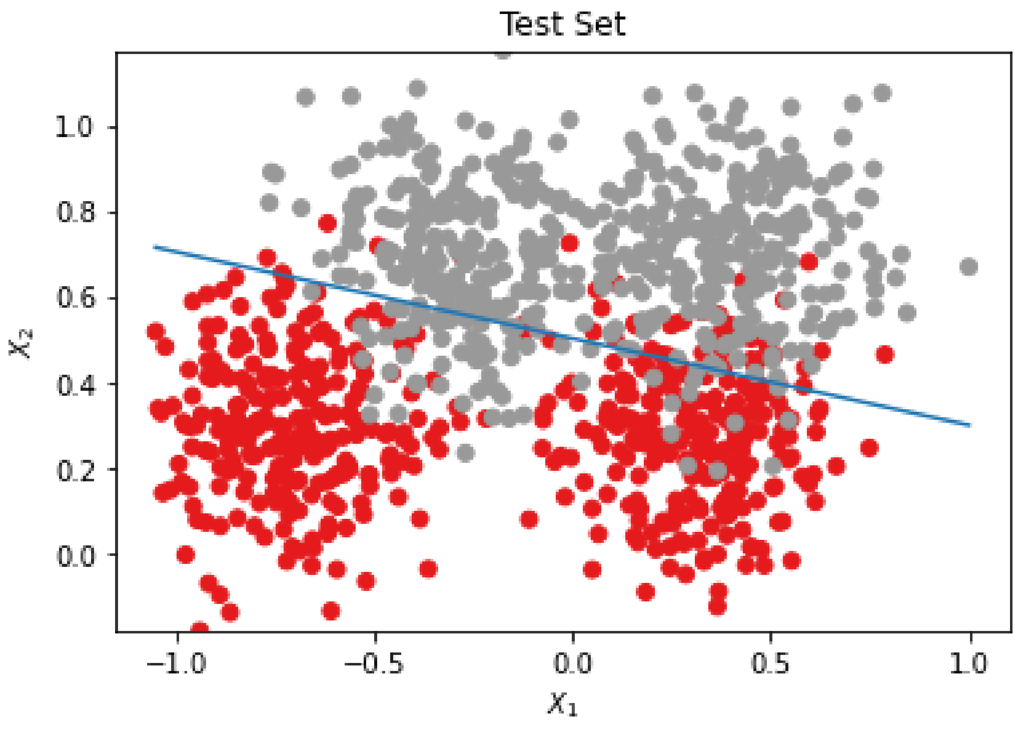
3). A *sub-gradient* is a generalization of the gradient for convex functions which are not necessarily differentiable. Such functions arise quite frequently in machine learning, e.g. when the 1-norm is used for regularization. The sub-gradient of a function at a point is the slope of *a* hyperplane that passes through the point and lies below the graph of the function.

[36]:
```
import numpy as np
import matplotlib.pyplot as plt


def svm_sgd(X, y, lambda_par, max_epochs=10):
    # Inputs:
    #   X ... Input variables (n x p matrix)
    #   y ... Labels (n vector), -1 or 1
    #   lambda_par ... Regularization constant (non-negative)
    #   max_epochs ... Maximum number of passes through the data set
    #
    # Output:
    #   Model coefficients (n vector)

    n,p = X.shape
    theta = np.zeros(p)
    for iter in range(1,max_epochs*n):
        a = 1/(iter*lambda_par)
        i = np.random.randint(n)
        z = y[i]*(theta.T).dot(X[i])
        oldtheta = theta
        if z < 1:
            gradient = (theta.dot(lambda_par) - (X[i].dot(y[i])))
            theta = oldtheta - a*gradient
        else:
            gradient = theta.dot(lambda_par)
            theta = oldtheta - a*gradient


    assert(lambda_par > 0)
    max_iter = max_epochs*X.shape[0]

    # Insert code here

    return theta


#init train
trainData = np.genfromtxt(fname = "synth_train.txt")
X = trainData[:,:-1]
y = trainData[:,-1]

n,p = X.shape

X = np.c_[np.ones(n),X]

#init test
testData = np.genfromtxt(fname = "synth_test.txt")
X_test = testData[:,:-1]
y_test = testData[:,-1]

n_test,p_test = X_test.shape

X_test = np.c_[np.ones(n_test),X_test]
```

```
#finding SVM coeffs
theta = svm_sgd(X,y,0.01)
print("SVM coeffs =", theta)

#finding train y prediction
y_pred = 1 * (X.dot(theta) > 0)
y_pred = np.where(y_pred==0, -1, y_pred) #changing 0 to -1

#showing train set error
print("Training set error rate =",np.average(y != y_pred))

#finding test y prediction
y_test_pred = 1 * (X_test.dot(theta)> 0)
y_test_pred = np.where(y_test_pred==0, -1, y_test_pred) #changing 0 to -1


#showing test set error
print("Test set error rate =", np.average(y_test != y_test_pred))



# Visualize train the data
plt.scatter(X[:,1],X[:,2], c=y, cmap='Set1')
xtmp = np.array(([min(X[:,1]), max(X[:,1])]))
plt.plot(xtmp, -(theta[0]+theta[1]*xtmp)/theta[2])
plt.ylim(min(X[:,2]), max(X[:,2]))
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.title('Train Set')
plt.show()

# Visualize the test data

plt.scatter(X_test[:,1],X_test[:,2], c=y_test, cmap='Set1')
xtmp = np.array(([min(X_test[:,1]), max(X_test[:,1])]))
plt.plot(xtmp, -(theta[0]+theta[1]*xtmp)/theta[2])
plt.ylim(min(X_test[:,2]), max(X_test[:,2]))
plt.xlabel('$X_1$')
plt.ylabel('$X_2$')
plt.title('Test Set')
plt.show()
```

```
SVM coeffs = [-1.92076831  0.77251515  3.82825948]
Training set error rate = 0.136
Test set error rate = 0.111
```


Train Set
1.0
0.8
0.6
0.4
0.2
0.0
X_2
−1.0
−0.5
0.0
0.5
X_1


Test Set
1.0
0.8
0.6
0.4
0.2
0.0
X_2
−1.0
−0.5
0.0
0.5
1.0
X_1

```
[ ]:
```

b) [10 points] Modify the code in a) so that it keeps track of the objective function value during the course of the iterations. Plot the objective function values as a function of iteration number. This is similar to what you did in homework 2 (but with a different objective function).

Comments:

1) Computing the function values and training set error requires a pass through all the training data. This is computationally expensive, so you should compute these values once every T iterations where T could e.g. be 100, 1000 or n.

2) To speed up the computations, use matrix and vector operations instead of *for*-loops where possible. For example, if the training set is in matrix X you can classify all the examples in a single matrix-vector multiplication, $y_{pred} = X\theta$ (why?) This issue is discussed in some detail in http://cs229.stanford.edu/section/vec_demo/Vectorization_Section.pdf

3) Note that the λ parameter in the above SVM formulation is related to the C parameter in the "standard" SVM formulation via $\lambda = 1/(nC)$.