REI602M Machine Learning - Homework 3

Due: Sunday 7.1.2021

Objectives: Classification, support vector machines, text classification

Name: Þorsteinn Sigurðsson, **email:** ths251@hi.is, **collaborators:** Ágúst Logi, Jón Ingimarsson, Arnar Leó

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

```
trainMatrix, tokenlist, trainCategory = readMatrix('MATRIX.TRAIN')
    testMatrix, tokenlist, testCategory = readMatrix('MATRIX.TEST')
```

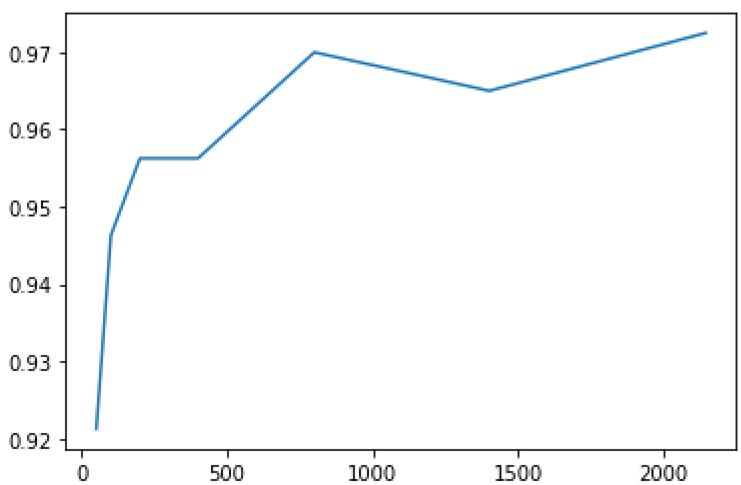
2) To use the `sklearn.svm.LinearSVC` class, you start by calling the `fit` function which solves the SVM optimization problem for the given training set. You then either call `predict` to get predictions for the test set (or other data points) and subsequently evaluate the error rate/accuracy for the 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.

In [2]:
```
#imports
import numpy as np
import matplotlib.pyplot as plt
from sklearn.svm import LinearSVC
from email_spam.read_spam_data import read_matrix as readMatrix
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
```

In [18]:
```
testMatrix, tokenlist, testCategory = readMatrix('MATRIX.TEST')

fylki = [50, 100, 200, 400, 800, 1400, 2144]
fylkiStrengur = [".50", ".100", ".200", ".400", ".800", ".1400", ""]


score = []
for i in range(len(fylki)):
    trainMatrix, tokenlist, trainCategory  = readMatrix("MATRIX.TRAIN"+fylkiSt
rengur[i])
    clf = LinearSVC(C = 0.1,max_iter = 100000).fit(trainMatrix, trainCategory)
    scoregildi = clf.score(testMatrix,testCategory)
    score.append(scoregildi)

print(score)

plt.plot(fylki, score)
```

```
[0.92125, 0.94625, 0.95625, 0.95625, 0.97, 0.965, 0.9725]
```

Out[18]:
```
[<matplotlib.lines.Line2D at 0x1c1874ab688>]
```



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 (https://www.kaggle.com/c/tweet-sentiment-extraction)](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.

In [19]:
```
# 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	selected_text	sentiment
0	a3d0a7d5ad	Spent the entire morning in a meeting w/ a ven...	my boss was not happy w/ them. Lots of fun.	neutral
1	251b6a6766	Oh! Good idea about putting them on ice cream	Good	positive
2	c9e8d1ef1c	says good (or should i say bad?) afternoon! h...	says good (or should i say bad?) afternoon!	neutral
3	f14f087215	i dont think you can vote anymore! i tried	i dont think you can vote anymore!	negative
4	bf7473b12d	haha better drunken tweeting you mean?	better	positive

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

In [20]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, rand
om_state=42)
clf = LogisticRegression(C=0.1, multi_class='ovr',solver='lbfgs').fit(X_train,
y_train)


score = clf.score(X_test, y_test)
print(score)


y_pred = clf.predict(X_test)

confusion_matrix(y_pred, y_test)

##Ekki með gott accuracy vegna þess að það er erfitt að meta útkomu með stikor
ðum úr tweeti.
```

```
0.6793076838275824
```

Out[20]:
```
array([[1454,  441,  125],
       [1002, 2803,  794],
       [ 122,  425, 1905]], dtype=int64)
```

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).

In [21]:
```
# Insert code here
# ...

clf = LogisticRegression(C = 0.1, multi_class='multinomial',solver = 'lbfgs').
fit(X_train, y_train)
a = clf.coef_.argsort()[-10:][::-1]
for i in range(3):
    print('Class:', le.classes_[i])
    for j in range(10):
        print(inv_map[a[i][j]])
```

```
Class: negative
sorry
sad
miss
missed
sucks
hate
tired
bored
bad
not
Class: neutral
happy
thanks
missing
cute
nice
thank
enjoy
awesome
perfect
headache
Class: positive
love
thanks
awesome
glad
hope
excited
great
better
nice
thank
```

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 $y^{(i)}\ (\theta^{(k)})^T x^{(i)} < 1 \text{ then}$

$\theta^{(k+1)} = \theta^{(k)} - \alpha^{(k)}(\lambda\theta^{(k)} - y^{(i)}x^{(i)})$

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`

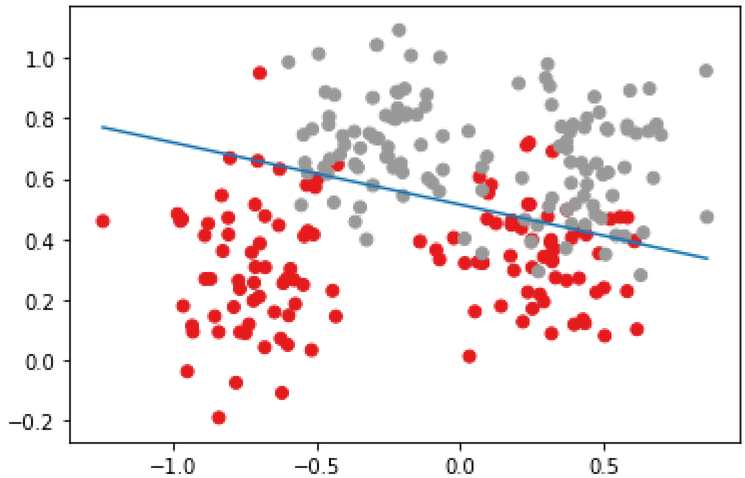
Comments:

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.

In [28]:
```
def svm_sgd(X, y, lambdapar, max_epochs=10):
    assert(lambdapar > 0)
    max_iter = max_epochs*X.shape[0]

    n,m = X.shape
    t = np.zeros([m,1])

    for k in range(max_iter):
        a = 1/(lambdapar*(1+k))
        i = np.random.randint(0,n-1)
        if (y[i])*(np.matmul(t.T,X[i,:])) < 1:

            t = t - a*(np.dot(lambdapar,t)-np.dot(y[i],X[i,:].reshape([-1,1
])))
            #.reshape([-1,1])
        else:
            t = t -a*lambdapar*t

        theta = t
    return theta


def testError(X,y,t):
    counter = 0
    for k in range(0,X.shape[0]):
        if (y[k]==1 and t[0]+t[1]*X[k,1]+t[2]*X[k,2]<0):
            counter+=1
        if (y[k]==-1 and t[0]+t[1]*X[k,1]+t[2]*X[k,2]>0):
            counter+=1
    return counter/X.shape[0]


SYNTH_TRAIN = np.genfromtxt('synth_train.txt')
SYNTH_TEST = np.genfromtxt('synth_test.txt')


X = np.c_[np.ones(SYNTH_TRAIN.shape[0]), SYNTH_TRAIN[:,0:-1]]
y = SYNTH_TRAIN[:,-1]
lambdapar = 1/100
theta = svm_sgd(X, y, lambdapar)
testSetError = testError(X,y,theta)

plt.scatter(X[:,1], X[:,2], c = y, cmap = 'Set1')
xtmp = np.array([min(X[:,1]),max(X[:,1])])

np.array([min(X[:,1]),max(X[:,1])])
plt.plot(xtmp, -(theta[0]+theta[1]*xtmp)/theta[2])

plt.show()



print(f'Theta: {theta}')
print(f'Test set Error: {testSetError}')
```




```
Theta: [[-2.        ]
 [ 0.79933295]
 [ 3.88911237]]
Test set Error: 0.128
```

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).

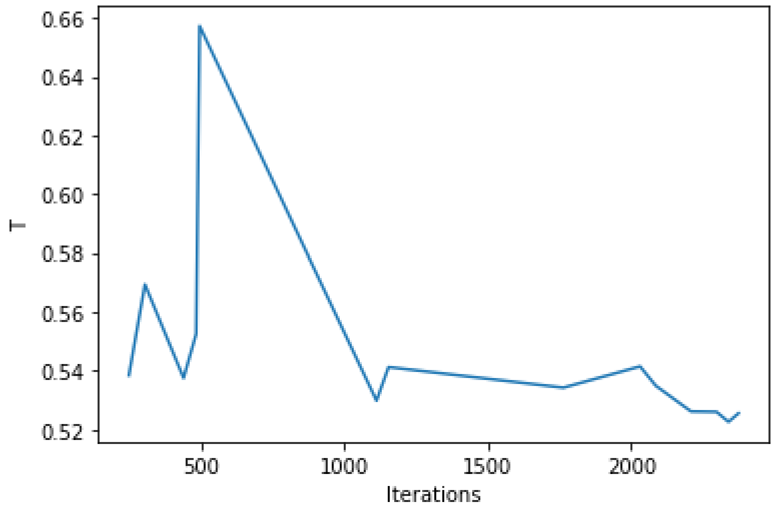
Comments:

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 (http://cs229.stanford.edu/section/vec_demo/Vectorization_Section.pdf)](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 $\lambda = 1/(nC)$.

In [29]:
```
def cost_func(X,y,t,lp):
    lambdatTt = np.dot(lp,np.matmul(t.T,t))[0][0]/2
    ytX = np.multiply(y.reshape([-1,1]),np.matmul(X,t))
    lytX = np.ones([ytX.shape[0],1])-ytX
    lytX[lytX<0] = 0
    T = lambdatTt + np.sum(lytX)/X.shape[0]
    return T


def svm_sgdOpt(X, y, lambdapar, max_epochs=10):
    assert(lambdapar > 0)
    max_iter = max_epochs*X.shape[0]

    n,m = X.shape
    t = np.zeros([m,1])
    T = []
    itranir = []
    for k in range(max_iter):
        a = 1/(lambdapar*(1+k))
        i = np.random.randint(0,n)
        if(i%1000 == 0):

            itranir.append(k)
            T.append(cost_func(X,y,t,lambdapar))

        if (y[i])*(np.matmul(t.T,X[i,:])) < 1:
            t = t - a*(np.dot(lambdapar,t)-np.dot(y[i],X[i,:].reshape([-1,1
])))

        else:
            t = t - a*np.dot(lambdapar,t)
        theta = t

    return T, itranir

lambdapar = 1/100
x,y = svm_sgdOpt(X, y, lambdapar)
plt.plot(y,x)
plt.ylabel('T')
plt.xlabel('Iterations')
plt.show()
```


0.66
0.64
0.62
0.60
0.58
0.56
0.54
0.52
T
500
1000
1500
2000
Iterations

In [ ]:

In [ ]: