Assignment 1: Sentiment Polarity for Movie Reviews

version 2021.3, details/bugs in this assignment might be patched, you will be notified if this happens

For this assignment, you will implement a classification pipeline for a binary sentiment classification task. The provided dataset includes movie reviews labelled as either positive or negative. Your starting point for the assignment is an implementation of the full pipeline that uses scikit learn and example data.

In most real-world situations, you will have good (and relatively bug free) tools at your disposal. However, for being able to be creative with an ML problem, it is crucial to understand the inner workings of the full pipeline. You will therefore re-implement the model, feature extraction, the learning algorithm, the prediction algorithm, and a simple grid search for hyper parameters. You will find re-implementation tasks marked "implementation task" (in bold) below. In your submission, all code must be written from scratch by you. While you can use the base modules of python (os, string etc) and NumPy, you cannot use sklearn or any equivalent ML library (when in doubt, ask about imports).

Submission

Please submit your code as a notebook through studium. You should include the following:

1. A working implementation of your pipeline, reproducing your principal results when run. Please rerun your notebook as the last thing you do before submitting.
2. Comment the code properly, especially for longer or opaque functionality. Please try to write self documenting code (i.e., choosing descriptive variables names, refactoring to isolate functionality, minimal code duplication etc).
3. A *brief* description (100-200 words in total) of the implementation work that was necessary to complete the different parts, showing how you arrived at your solution and design choices made. You can spread these in the notebook or put them in one place.
4. Comments on what you thought was hard in the assignment, what you think was educational, what took most time and which parts might be unnecessarily tricky.
5. As the submission is anonymous, **all personal information must be removed**.
6. Apart from sections and titles, please remove all unnecessary text and code from the notebook you hand in. Keep only that which strengthens the case that you fulfil the listed requirements. All notebooks containing unnecessary chunks of text from this instruction will receive a U.

Requirements for grade G

To achieve a pass (G) in this assignment, you must solve the following tasks without serious errors.

1. Reimplement the four parts of the assignment. The instructions below will guide you on specifics for each part.
2. When evaluating your model, split the data into a training and a test set. This split can be selected non-randomly before running any training. The split should be 80/20 (i.e., 1600 training documents and 400 test documents). *It is easier for you if the classes are balanced.*
3. Implement the model by following the skeleton API

3. Implement the model as a class following the sklearn API as:

```
class model:
def __init__(self, learning_rate, ...):
 ...

def fit(self, X, y):
 ...

def predict(self, X):
 ...

def score(self, X, y):
 ...
```

4. Include a short (150 words) qualitative analysis in your submission. Discuss the final decision boundary (e.g., why did this word contribute to a negative label etc), the selected hyper-parameters, and design choices.

Additional requirements for grade VG

To achieve a pass with distinction (VG) in this assignment, you must adequately solve the tasks above. In addition, you must:

1. Implement the optimization as *stochastic* gradient descent (SGD)
2. Implement a tf-idf (https://en.wikipedia.org/wiki/Tf%E2%80%93idf) feature model and compare classification performance to bag-of-words (this should also be discussed in your qualitative analysis). Choose your preferred formulation of tf-idf from the literature, but motivate it (in a few sentences).
3. Implement and discuss momentum (https://en.wikipedia.org/wiki/Stochastic_gradient_descent#Momentum) in the SGD optimization.
4. Implement and discuss RMSProp (https://en.wikipedia.org/wiki/Stochastic_gradient_descent#RMSProp) in the SGD optimization.
5. Implement and discuss 10-fold cross validation (https://en.wikipedia.org/wiki/Cross-validation_(statistics)#k-fold_cross-validation) for evaluating your model.
6. Write a longer qualitative analysis (ca 600 words) of your pipeline and results, including some visualization (you can use UMAP or PCA from sklearn for this). The analysis and claims must be essentially correct.

General advice

The task is to predict, for an unseen review, whether it is positive or negative. This is a binary classification task. Work from the given code and change one piece at a time, ensuring functionality throughout your work. You should test your code every couple of lines to make sure your assumptions on functionality and variable content is correct. A good rule of thumb is that a coder will introduce a bug every five lines (even as a professional).

To make sure your code does what it is supposed to do, use `assert` statements to check your

assumptions. Keep the given asserts if you need them. Professional coders sometimes start with writing tests for some functionality instead of starting with the functionality itself. This is called *test-driven development*.

Plagiarism

In code assignments, plagiarism is a tricky concept. A clean cut way would be to demand that you write all the code yourself, from memory, with only the assigned literature as help. This is not how code is developed professionally. It is common to copy and share. However, since this is a learning exercise, you must implement everything on your own, but please look at the course repo, Stack Overflow etc. Moreover, discuss with course mates and TAs to find inspiration and solutions. Code that is *obviously* copied (with minor modifications) will be considered as plagiarized. As a part of the examination, you might be asked to explain any particular part of the functionality in your implementation.

Part 1: Parsing the dataset

For this assignment, we use the Review polarity v2.0 (http://www.cs.cornell.edu/people/pabo/movie-review-data/) data set created by Bo Pang and Lillian Lee at Cornell University. It consists of 2000 movie reviews, 1000 of which are positive and 1000 are negative. *Always check the readme for any dataset before using it.*

The following downloads the dataset (if it's not already present).

In [1]:
```
!wget -N http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz
```

```
--2021-04-16 14:00:26--  http://www.cs.cornell.edu/people/pabo/movie-review-data/review_polarity.tar.gz
Resolving www.cs.cornell.edu (www.cs.cornell.edu)... 132.236.207.36
Connecting to www.cs.cornell.edu (www.cs.cornell.edu)|132.236.207.36|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3127238 (3.0M) [application/x-gzip]
Saving to: ‘review_polarity.tar.gz’

review_polarity.tar 100%[===================>]   2.98M  --.-KB/s    in 0.1s

2021-04-16 14:00:26 (24.1 MB/s) - ‘review_polarity.tar.gz’ saved [3127238/3127238]
```

In order to get the given code to work, a part of the 20 newsgroup dataset is loaded below. This should be replaced by your parsing code in your final submission.

In [2]:
```
from sklearn.datasets import fetch_20newsgroups
X_raw, y = fetch_20newsgroups(subset='train', categories=['rec.sport.baseball', 'rec.sport.hockey'], remove=('headers', 'footers', 'quotes'), return_X_y=True)
y[y==0]=-1
```

```
Downloading 20news dataset. This may take a few minutes.
Downloading dataset from https://ndownloader.figshare.com/files/5975967
```

```
Downloading dataset from https://downloader.figshare.com/files/3975967
(14 MB)
```

In [3]:
```
import numpy as np

# assert len(X_text) == 2000
assert np.all([isinstance(x, str) for x in X_raw])
assert len(X_raw) == y.shape[0]
assert len(np.unique(y))==2
assert y.min() == -1
assert y.max() == 1
```

Implementation task: Implement a parser for the dataset. The output should be a list/array of strings (`X_raw`) and a list/array of labels (`y`) encoded as {-1,1}.

Part 2: Feature extraction

As basic features, we use a binary bag-of-words (BOW) representation of the words in each review. Each review in the data set is described by a vector with one element corresponding to each word in the vocabulary. An element is set to 1 if the review contains its associated word, otherwise it is set to 0.

In [4]:
```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer()
vectorizer.fit(X_raw)             # Creates the vocabulary of the vecto
rizer

X = vectorizer.transform(X_raw)
X = X.todense()                   # sklearn will output a sparse matrix
X[X>1] = 1                        # Turns the count vectors into binary
vectors

ordered_vocabulary = vectorizer.get_feature_names()
vocabulary = set(ordered_vocabulary)
```

If needed, we can do lookup tables going from tokens to feature numbers. Note that most of the elements in any feature vector will be zero.

In [5]:
```
lut = dict()
for i, word in enumerate(ordered_vocabulary):
  lut[word] = i

for word in ['dolphin', 'the', 'coffee']:
  if word in vocabulary:
    print("'%s' is represented as feature dimension %i" %(word, lut[wor
d]))
  else:
    print("'%s' is not in the vocabulary" % word)
```

```
'dolphin' is not in the vocabulary
'the' is represented as feature dimension 12700
'coffee' is represented as feature dimension 3639
```

Implementation task: You should re-implement the feature extraction above. The list/array called

Implementation task: You should re-implement the feature extraction above. The list/array called `ordered_vocabulary` should contain the words for each feature dimension and X should contain the BOW binary vectors.

Hints: Implementing X a numpy array will make your life easier in the comming parts. Also, the `in` operator is way faster for sets than for lists.

We can now look at the data and the words corresponding to feature dimensions.

In [6]:
```
print(ordered_vocabulary[2000:2010])
print(X[:10, 2000:2010])
for w in ['dolphin', 'the', 'coffee']:
  print("'%s' in words: %s" % (w, w in vocabulary))
```

```
['annoying', 'annoys', 'annual', 'anointed', 'another', 'anoynamous',
'anson', 'answer', 'answered', 'answers']
[[0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]
 [0 0 0 0 0 0 0 0 0 0]]
'dolphin' in words: False
'the' in words: True
'coffee' in words: True
```

At this point in the code, X and y are ready for the supervised learning task.

Part 3: Learning framework

The model we will use is a simple hyperplane. This plane will represent the decision boundary through the data space, separating positive from negative ratings.

Implementation task: You should implement your versions of the following parts (you can also find this in the slides):

1. **Hyperplane model**. The model should be a hyperplane as $f(x, \omega) = sgn(\omega^\top x)$, where $sgn(\cdot)$ is the sign function (https://en.wikipedia.org/wiki/Sign_function). Note that x_0 in this notation is the pseudo input 1. When evaluated, this gives us the predicted results as $\hat{y}_i = f(X_i, \omega^{(t)})$, where $\omega^{(t)}$ is the parameter vector at optimization iteration t and $\omega^{(0)}$ is the initial guess for the parameter vector.
2. **Objective function.** The loss function for our model is the hinge loss (https://en.wikipedia.org/wiki/Hinge_loss), which will be used together with l_2 regularization.

   $$\mathcal{Q}(X, y, \omega) = \frac{\lambda}{2}\|\omega\|^2 + \sum_{i=1}^{|X|} \max(0, 1 - y_i \cdot \omega^\top X_i).$$

   Regularization is done by adding a norm on the parameter vector and including that in the objective function. A shorter parameter vector gives a larger margin for this model. The l_2 norm is defined as $\sqrt{\sum_{i=1}^{n} \omega_i^2}$. The regularization always has some positive attenuation parameter $\lambda > 0$ keeping it from dominating the objective function. It symbolizes a trade-off

parameter $\lambda \in \mathbb{R}$ keeping it from dominating the objective function. It symbolizes a trade-off between a more accurate classification and wider margins, while also giving the objective function a unique solution.

3. **Gradient descent**. The update for gradient descent looks like $\omega^{(t)} = \omega^{(t-1)} - \gamma \nabla \mathcal{L}(\omega^{(t-1)})$, where the update gradient is defined as $\nabla \mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \omega_1}, \frac{\partial \mathcal{L}}{\partial \omega_2}, \ldots, \frac{\partial \mathcal{L}}{\partial \omega_n} \right)^\top$. The expression for this gradient $\nabla \mathcal{L}$ is given analytically as:

$$\nabla \mathcal{L}(X, y, \omega) = \lambda \omega + \sum_{i=1}^{|X|} \begin{cases} 0 & \text{if } y_i \omega^\top X_i \geq 1 \\ -y_i X_i & \text{else} \end{cases}$$

In the expression for the gradient, X_i is a vector and y_i is a scalar. These two refer to the i:th data point and its label. The learning rate $\gamma \in \mathbb{R}$ acts as a scaling/dampening factor on the gradient update. This should run until some stopping criteria is met (e.g., $\omega^{(t+1)} \approx \omega^{(t)}$). The default stopping criterion for `SGDClassifier` is when $loss_{current} > loss_{best} - .001$ for five consecutive iterations.

Note that while your code will be runnable, it will likely be much slower than sklearn's code.

In [7]:
```
from sklearn.linear_model import SGDClassifier

# Set hyperparameters (these variables are only here for clarity)
reguliser_dampening = 0.001   # lambda
learning_rate = .1            # gamma

# Create the untrained classifier
model = SGDClassifier(loss='hinge', penalty='l2',
                      alpha=reguliser_dampening, verbose=1,
                      learning_rate='constant', eta0=learning_rate)

# Train the classifier
model.fit(X, y)

# Get the parameter vector
omega = np.concatenate([model.intercept_, model.coef_.ravel()])
```

```
-- Epoch 1
Norm: 12.78, NNZs: 5610, Bias: -1.200000, T: 1197, Avg. loss: 0.542247
Total training time: 0.05 seconds.
-- Epoch 2
Norm: 13.91, NNZs: 6584, Bias: -0.300000, T: 2394, Avg. loss: 0.183976
Total training time: 0.09 seconds.
-- Epoch 3
Norm: 14.00, NNZs: 6827, Bias: -0.700000, T: 3591, Avg. loss: 0.111985
Total training time: 0.13 seconds.
-- Epoch 4
Norm: 13.90, NNZs: 6950, Bias: -0.600000, T: 4788, Avg. loss: 0.078783
Total training time: 0.15 seconds.
-- Epoch 5
Norm: 13.55, NNZs: 7076, Bias: -0.500000, T: 5985, Avg. loss: 0.071252
Total training time: 0.18 seconds.
-- Epoch 6
Norm: 13.33, NNZs: 7176, Bias: -0.700000, T: 7182, Avg. loss: 0.065517
Total training time: 0.21 seconds.
-- Epoch 7
Norm: 13.33, NNZs: 7289, Bias: -0.800000, T: 8379, Avg. loss: 0.063438
Total training time: 0.24 seconds.
```

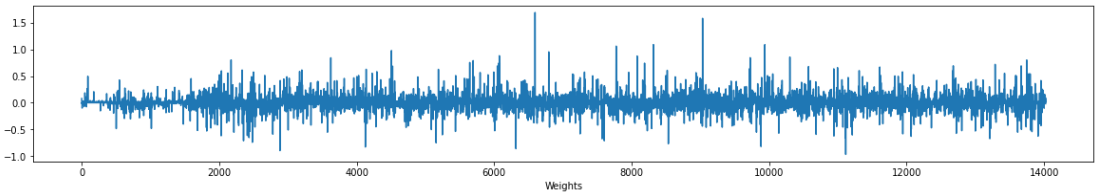
```
-- Epoch 8
Norm: 12.98, NNZs: 7331, Bias: -0.700000, T: 9576, Avg. loss: 0.061281
Total training time: 0.27 seconds.
-- Epoch 9
Norm: 12.65, NNZs: 7410, Bias: -0.400000, T: 10773, Avg. loss: 0.057665
Total training time: 0.29 seconds.
-- Epoch 10
Norm: 12.67, NNZs: 7494, Bias: -1.100000, T: 11970, Avg. loss: 0.063168
Total training time: 0.32 seconds.
-- Epoch 11
Norm: 12.50, NNZs: 7517, Bias: -0.400000, T: 13167, Avg. loss: 0.057011
Total training time: 0.35 seconds.
-- Epoch 12
Norm: 12.80, NNZs: 7631, Bias: -0.700000, T: 14364, Avg. loss: 0.063540
Total training time: 0.38 seconds.
-- Epoch 13
Norm: 12.90, NNZs: 7682, Bias: -0.800000, T: 15561, Avg. loss: 0.064725
Total training time: 0.41 seconds.
-- Epoch 14
Norm: 12.86, NNZs: 7717, Bias: -0.800000, T: 16758, Avg. loss: 0.057303
Total training time: 0.44 seconds.
Convergence after 14 epochs took 0.44 seconds
```

In the above training with verbose=1, note how the loss etc are changing. For your implementation, it can be very good to print out lots of information so that you can see if you get what you expect (e.g. a lowering of the loss).

We can examine the weights by plotting them. Think about how to interpret these weights.

In [8]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 3))
plt.plot(omega[1:])
plt.xlabel("Value")
plt.xlabel("Weights")
plt.show()
```



From the same information, we can plot the words with the strongest influence. Can you see a pattern? Is the model over learning on some words? Why might word frequency be important when analysing their impact?

In [9]:
```
assert (len(omega)-1) == len(vocabulary)

# Sort by absolute value
idx = np.argsort(np.abs(omega[1:]))

print("               Word   Weight  Occurences")
for i in idx[-20:]:   # Pick those with highest 'voting' values
  print("%20s   %.3f\t%i " % (ordered vocabulary[i], omega[i+1], np.sum
```

```
([ordered_vocabulary[i] in d for d in X_raw]))
```

```
      Word   Weight  Occurences
     wings   0.801    6
       ass   0.802    152
  pitching   -0.822   56
      cubs   -0.831   4
     coach   0.843    61
      pens   0.845    32
      puck   0.856    35
      hall   -0.862   19
      lssu   0.875    0
     goals   0.882    54
    braves   -0.899   1
       ice   0.954    188
      runs   -0.969   77
   detroit   0.978    1
     leafs   1.061    1
   playoff   1.088    104
  playoffs   1.089    71
      mask   1.090    13
       nhl   1.583    1
    hockey   1.692    129
```

Part 4: Exploring hyperparameters

For optimization of the hyperparameters, you can search for values on a grid. Trying all combinations is called a grid search and can be implemented with nested `for` loops. A faster alternative is to sample from the grid. Sampling is not as thorough, but most often sufficient (and much faster). Following the current consensus on ML methodology, we must split off a test set before exploring any configuration and use this data only at the very end of the experiment.

Implementation task: Implement code for printing a sorted table of your sampled hyperparameters.

1. **Learning rate**. Try different settings of the learning rate. It is useful to pick values from an exponentially spaced grid (e.g., 0.0001/0.0003/0.001/0.003/0.01/0.03/0.1/0.3/1.0/3.0) where each of the values is about 3 times as large as the previous one. Note what happens when the learning rate gets too small or too large. The best learning rate is as large as possible, but still reliable and stable.
2. **Regulariser dampening**. Try to find how much the regulariser needs to be dampened to get a