1 Solutions (Section4) Load and split the data

1.1 Training data set size: 1500, validation data set size: 500

```
import collections
import copy
import numpy as np
import os
import random

from sklearn.model_selection import train_test_split

POS_DATA_DIR="./data/pos"
NEG_DATA_DIR="./data/neg"
SEQUENCES_TO_IGNORE = set ([',', '.', "'", '"', ';', ':'])
SEED = 5
OBJ_FN_PCT_THRESHOLD = 0.05

# this function returns 2 lists - X, y where each entry of X is a string that has the u
# of a review, and the corresponding element of y is +1 for a positive review, -1 for a
def read_raw_data (data_dir, yval):
    X = []
    files = [os.path.join(data_dir, f) for f in os.listdir (data_dir) if os.path.isfile
    for f in files:
        with open (f, 'r') as x:
            X.append (x.read ().replace ('\n', ''))

    y = [yval] * len (files)
    return X, y

def convert_str_to_bag_of_words (input_str):
    words = set (input_str.split ())
    relevant_words = words - SEQUENCES_TO_IGNORE
    cntr = collections.Counter (relevant_words)
    return cntr

def get_full_dataset ():
    X = []
    y = []

    X_raw, y_partial = read_raw_data (POS_DATA_DIR, 1)
    for x in X_raw:
        X.append (convert_str_to_bag_of_words (x))
    y.extend (y_partial)

    X_raw, y = read_raw_data (NEG_DATA_DIR, -1)
```

```
    for x in X_raw:
        X.append (convert_str_to_bag_of_words (x))
    y.extend (y_partial)

    return X, y

def split_data (X, y):
    X_train, X_val_and_test, y_train, y_val_and_test = \
            train_test_split (X, y, test_size=0.25, random_state=SEED)
    X_val, X_test, y_val, y_test = \
            train_test_split (X_val_and_test, y_val_and_test, test_size=0.0, random_sta
    return X_train, X_val, X_test, y_train, y_val, y_test

# sn: make calls to functions defined above
X, y = get_full_dataset ()
X_train, X_val, X_test, y_train, y_val, y_test = split_data (X, y)
```

2 Solutions (Section5) Bag of words

2.1 Create "bag-of-words" representation from text

The function convert-str-to-bag-of-words listed in the section above discards character sequences defined in the list SEQUENCES-TO-IGNORE and returns a bag of words representation in the form of a Python Counter class object.

3 Solutions (Section 6) - Support Vector Machine via Pegasos

3.1 Solution 6.1 - Gradient of the SVM objective function at a single training point

Objective function: $J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y_i w^T x_i\right\}$

For $y_i w^T x_i > 1$, J_i is 0 (and differentiable) throughout so the gradient is
$\nabla J_i(w) = \nabla(\frac{\lambda}{2}\|w\|^2) = \lambda w$

For $y_i w^T x_i < 1$, J_i is again differentiable at every point:
$\nabla_w J_i(w) = \lambda w - y_i \nabla_w (w^T) x_i$

where y_i is a scalar with value 1 or -1, and x_i is a given (fixed since we selected a single training point) vector in R^d

The gradient $\nabla_w J_i(w)$ thus evaluates to:
$\lambda w - y_i x_i$, given that $y_i w^T x_i > 1$

[$\nabla_w (w^T)$ is 1_d^T by definition]

Finally, at $y_i w^T x_i = 1$, $\nabla_w J_i(w)$ is undefined since the one-sided derivatives at this point are different. In other words, $J_i(w)$ is not differentiable at $y_i w^T x_i = 1$.

3.2 Solution 6.2 - Sub-gradient of SVM objective function

Objective function: $J_i(w) = \frac{\lambda}{2}\|w\|^2 + \max\left\{0, 1 - y_i w^T x_i\right\}$

To show that a sub-gradient of the above objective function is given by:

$$g = \begin{cases} \lambda w - y_i x_i & \text{for } y_i w^T x_i < 1 \\ \lambda w & \text{for } y_i w^T x_i \geq 1 \end{cases}$$

As seen in the above derivation of the expression for gradient, $J_i(w)$ is differentiable at all points but $y_i w^T x_i = 1$. The derived expression for gradient is also the sub-gradient:

$\nabla J_i(w) = \lambda w$ for $y_i w^T x_i > 1$
and, $\nabla J_i(w) = \lambda w - y_i x_i$ for $y_i w^T x_i < 1$

At $y_i w^T x_i = 1$, we will prove that λw is a subgradient using *proof by contradiction*. Let's assume that λw is not a subgradient at $y_i w^T x_i = 1$, and thus violates the definition, giving rise to the following condition

i.e. $\exists$ u such that

$\lambda\|u\|^2 + \max\left\{0, 1 - y_i u^T x_i\right\} < \lambda\|v\|^2 + \max\left\{0, 1 - y_i v^T x_i\right\} + \lambda v^T(u - v)$ given that $y_i v^T x_i = 1$
$=> \|u\|^2 - \|v\|^2 + 2y_i(v^T x_i - u^T x_i) < 2v^T(u - v)$

All the above quantities are scalars. Expanding the norms to matrix products, we get
$u^T u - u^T v + v^T u - v^T v + 2y_i v^T x_i - 2y_i u^T x_i < 2v^T u - 2v^T v$

All these quantities are scalars and $u^T v = v^T u$ and we are looking at the point $y_i v^T x_i = 1$, so we get

$2(y_i v^T x_i - 1) < 0$ for $J_i(w)$ to not be a subgradient, and this is clearly not possible in our setting of $y_i v^T x_i = 1$.

3.3 Solution 6.3 - SGD with step size $\eta_t = 1/(\lambda t)$

The psedu-code in section 6 is equivalent to SGD with the dynamic step size given by $\eta_t = 1/(\lambda t)$. Each pass over the m data points represents an epoch. In SGD, we calculate the prediction function given by w at each data point by taking a step in the direction given by the sub-gradient at that point, as we iterate through the points until a stopping condition is met. This is exactly what the pseudo-code does, with each step being in the direction:

For $y_i w_t^T x_i < 1$,
$\lambda w_t - y_i x_i$ with a step size of $\eta_t = 1/(\lambda t)$. i.e.
$w_{t+1} = w_t - \eta_t * (\lambda w_t - y_i x_i)$
$= (1 - \eta_t \lambda) w_t + \eta_t y_i x_i$, which is the same as pseudo-code

Also, for $y_i w_t^T x_i >= 1$, the sub-gradient is given by λw_t, and with a step size of η_t, SGD sets
$w_{t+1} = w_t - \eta_t \lambda w_t$
$= (1 - \eta_t \lambda) w_t$

Thus, the pseudo code is equivalent to SGD with step size of $\eta_t = 1/(\lambda t)$ done at each point (x_i, y_i).

3.4 Solution 6.4 - Pegasos implementation

```
import copy

OBJ_FN_PCT_THRESHOLD = 0.05

def scale_and_add_sparse_vectors (scale1, d1, scale2=1, d2=None):
    """
    d1 is modified in place to scale by scale1, and if d2 is supplied, it is added to d
    """
    for k, v in d1.items ():
        d1 [k] = scale1 * v

    if not d2:
        return d1

    for k, v in d2.items ():
        if k in d1:
            d1 [k] += scale2 * v
        else:
            d1 [k] = scale2 * v


def dotProduct(d1, d2):
    """
    @param dict d1: a feature vector represented by a mapping from a feature (string) t
    @param dict d2: same as d1
    @return float: the dot product between d1 and d2
    """
    if len(d1) < len(d2):
        return dotProduct(d2, d1)
    else:
        return sum(d1.get(f, 0) * v for f, v in d2.items())
```

```
class PegasosClassifier (object):

    def __init__ (self, lambda_val):
        self._lambda = lambda_val

    def _evaluate_obj_function (self, X, y):
        w_sqr = dotProduct (self._w, self._w)
        reg_term = 0.5 * self._lambda * w_sqr
        i = 0
        correct_conf_predictions = 0
        while i < len (y):
            if y [i] * dotProduct (self._w, X[i]) > 1:
                correct_conf_predictions += 1
            i = i + 1

        pct_incorrect_predictions = (1 - correct_conf_predictions/ len (y))
        an_val = reg_term + pct_incorrect_predictions

        print ("obj_fn_val: %.3f, reg. term: %.3f, pct_incorrect_predictions: %.3f" % \
            (fn_val, reg_term, pct_incorrect_predictions))
        return fn_val, correct_conf_predictions

    def train_model (self, X, y, w={}):
        iterate = True
        self._w = copy.deepcopy (w)
        t = 0
        obj_fn_prev = 1e9
        while iterate:
            w_prev = copy.deepcopy (self._w)
            idx = 0
            while idx < len (y):
                t = t + 1
                eta = 1/ (self._lambda * t)
                if y[idx] * dotProduct (X [idx], self._w) < 1:
                    scale_and_add_sparse_vectors ((1 - eta * self._lambda), self._w, (e
                else:
                    scale_and_add_sparse_vectors ((1 - eta * self._lambda), self._w)
                idx += 1

            obj_fn_cur, correct_predictions_cur = self._evaluate_obj_function (X, y)

            if obj_fn_cur >= obj_fn_prev:
                iterate = False
                self._w = w_prev
            elif obj_fn_cur * (1 + OBJ_FN_PCT_THRESHOLD) > obj_fn_prev:
```

```
                iterate = False
            else:
                iterate = True
                obj_fn_prev = obj_fn_cur

lambda_val = 0.1
classifier = PegasosClassifier (lambda_val)
classifier.train_model (X_train, y_train)
```

3.5 Solution 6.5 - Pegasos implementation using sparse matrices

```
import copy
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer

# sn: this function returns arrays of strings/ text for use by CountVectorizer
def get_full_dataset_as_text ():
    X_raw = []
    y = []

    X_raw_partial, y_partial = read_raw_data (POS_DATA_DIR, 1)
    X_raw.extend (X_raw_partial)
    y.extend (y_partial)

    X_raw, y = read_raw_data (NEG_DATA_DIR, -1)
    X_raw.extend (X_raw_partial)
    y.extend (y_partial)

    return X_raw, y

class VectorizedPegasosClassifier (object):

    def __init__ (self, lambda_val):
        self._lambda = lambda_val

    def _evaluate_obj_function (self, X, y):
        w_sqr = (self._s ** 2) * np.dot (self._w, self._w)
        reg_term = 0.5 * self._lambda * w_sqr
        i = 0
        correct_conf_predictions = 0
        while i < len (y):
            if y [i] * np.dot (self._w, X[i]) > 1:
                correct_conf_predictions += 1
            i = i + 1

        '''
        for feature in self._w:
```

```
            if (self._w [feature] > ABS_WT_THRESHOLD):
                print (feature + ": " + str (self._w [feature]))
        '''
        pct_incorrect_predictions = (1 - correct_conf_predictions/ len (y))
        fn_val = reg_term + pct_incorrect_predictions

        print ("obj_fn_val: %.3f, reg. term: %.3f, pct_incorrect_predictions: %.3f" % \
               (fn_val, reg_term, pct_incorrect_predictions))
        return fn_val, pct_incorrect_predictions

    def train_model (self, X, y):
        # X is received as a bag-of-words dictionary
        iterate = True
        self._w = np.zeros (X.shape [1])
        self._s = 0
        t = 0
        obj_fn_prev = 1e9
        pct_errors_prev = 1
        while iterate:
            w_prev = copy.deepcopy (self._w)
            s_prev = self._s
            idx = 0
            while idx < len (y):
                t = t + 1
                eta = 1/ (self._lambda * t)
                self._s = (1 - eta * self._lambda) * self._s
                if self._s == 0:
                    self._s = 1
                    self._w = self._w = np.zeros (X.shape [1])
                if y[idx] * np.dot (X [idx], self._w) < 1:
                    self._w = self._w + eta * y [idx] * X [idx]/ self._s

                idx += 1

            obj_fn_cur, pct_errors = self._evaluate_obj_function (X, y)

            if obj_fn_cur >= obj_fn_prev:
                iterate = False
                self._w = w_prev
                pct_errors = pct_errors_prev
                obj_fn_cur = obj_fn_prev
            elif obj_fn_cur * (1 + OBJ_FN_PCT_THRESHOLD) > obj_fn_prev:
                iterate = False
            else:
                iterate = True
                obj_fn_prev = obj_fn_cur
```

```
            pct_errors_prev = pct_errors

        return obj_fn_cur, pct_errors

X, y = get_full_dataset_as_text ()
X_train, X_val, X_test, y_train, y_val, y_test = split_data (X, y)
vectorizer = CountVectorizer ()
X_train_vectors = vectorizer.fit_transform (X_train).toarray ()
X_val_vectors = vectorizer.transform (X_val).toarray()

classifier2 = VectorizedPegasosClassifier(0.1)
classifier2.train_model (X_train_vectors, y_train)
```

3.6 Solution 6.6 - Run time comparison

The timt taken by both the versions of the algorithm depends on the choice of λ - less regularization i.e. smaller values lead to longer run times. However,the sparse matric algorithm is several times faster. For $\lambda = 0.1$, the sparse matrix version takes just 3.41 seconds compared to 20.44 seconds taken by the dictionary based implementation.

3.7 Solution 6.7 - Function to return percent error

Such a function is defined in the section above within the implementation of class Vectorized-PegasosClassifier. The function *_evaluate_obj_function* returns both the percent error as well as the objective function value for a given model.

3.8 Solution 6.8 - Search for λ

Lambda	Error rate on (validation data)	Objective Function Value
1.00 e-3	22.10%	-
1.00 e-4	18.99%	0.227
0.50 e-4	19.20%	0.445
0.25 e-4	18.00%	0.883
1.00 e-5	17.20%	2.155

We should choose $\lambda = 1e - 4$ as the error rate on validation data begins to climb up at that point, and this point thus offers a good deal of regularization without sacrificing the model fit.