In [0]:
```python
import zipfile
with zipfile.ZipFile("data.zip","r") as zip_ref:
    zip_ref.extractall("data")
```

In [0]:
```python
from util import *
from load import *
import numpy as np
```

Sparse Representation

1. I split the data in to training and validation samples after this function is called in a few pages behind.

In [0]:
```python
def shuffle_data():
#We use the shuffle_helper() function in load.py to load reviews, combine positive and negative,
# and randomly shuffle the data. Here I change the name of the shuffle function in load.py since
# the code could not run with the given shuffle function name in the load.py skeleton code.
    shuffle_helper()
    with open('shuffled.pkl', 'rb') as f:
        data = pickle.load(f)
    print(len(data))
    return data
```

2.

In [0]:

```
from collections import Counter
class reviewInstance:
    """
    Description
    ===========
    Tranform text in review to sparse encoding of words
    """

    def __init__(self,word_list):
        self.word = word_list[:-1]
        self.label = word_list[-1]
        self.word_count = None
        self.tf_idf = None
    """
    Description
    ===========
    - Construtor requires list of words
    - Processing in load.read_data
    """

    def construct_word_dict(self, stop_word=None, count_words=None):
        """
        Description
        ===========
        Count words in word_list to transform to a dict {word: word_count}

        Input
        =====
        stop_word: list
        Words you hope to filter, not included in dict, default set to None

        count_words: list
        Words you hope to keep in the count_dict, if set to None, will keep all the words
        """
        self.word_count = Counter(self.word)
    def transform_to_tfidf(self, idf_dict):
        """
        Description
        ===========
        Construct the {word: tfidf} vector tfidf

        Input
        =====
        Document frequency for each word
        """
        tf = Counter(self.word)
        for k,v in tf:
            tf[k] = v/len(self.word)*idf_dict[k]
        return tf
```

SVM with Pegasos

In [0]:
```python
def scale_counter(s,c):
    scale = Counter()
    for k in c:
        scale[k] = c[k] * s
    return scale
def pegasos_sgd_loss(review_X, review_y, w, reg_lambda):
    """ Implementation of Objective Function"""
    if len(w)==0:
        return 1
    f1 = reg_lambda/2*dotProduct(w,w)
    f2 = max(0,1-review_y*dotProduct(review_X,w))
    return f1+f2
def pegasos_sgd_gradient(review_X, review_y, w, reg_lambda):
    """ Implementation of Gradient of Objective Function"""
    grad = scale_counter(reg_lambda,w)
    if len(w) == 0:
        increment(grad, -1*review_y, review_X)
    elif review_y*dotProduct(review_X,w)<1:
        increment(grad, -1*review_y, review_X)
    return grad
def gradient_checker_for_pegasos(review_X, review_y, weight, reg_lambda, objective_func=pegasos_sgd_loss,
                                 gradient_func=pegasos_sgd_gradient, epsilon=0.01, tolerance=1e-4):
    """ Gradient Checker adapted from Homework 1 """
# Do not know how to adapt this to the sparse representation structure due to the weight dimensions inconsistency.
# Implementing this function is not mandatory so I skipped it.
   # true_gradient = gradient_func(review_X, review_y, weight, reg_lambda) #The true gradient
   # num_features = theta.shape[0]
   # approx_grad = np.zeros(num_features) #Initialize the gradient we approximate
   # for i in range(num_features):
   #     e_i = np.array([0]*num_features)
   #     e_i[i] = 1
   #     approx_grad[i] = (objective_func(review_X, review_y, theta + epsilon * e_i,reg_lambda)-
   #                       objective_func(review_X, review_y, theta -epsilon * e_i,,reg_lambda))/(2*epsilon)
   # increment(true_gradient, -1, approx_grad)
   # if np.sqrt(np.sum(true_gradient**2))>tolerance:
   #     return False
   # return True
```

7.

In [0]:
```python
def accuracy_percent(review_list, weight, tfidf=False):
    """
    Description
    ===========
    Accuracy of predictions for collection of reviews under weights
    """
    corr = 0
    if len(weight)==0:
        print("Warning: weight is 0. Accuracy of predictions: 0")
    for r in review_list:
        pred = svm_predict(r.word_count, weight)
        if pred == r.label:
            corr = corr+1
    return corr/len(review_list)
def loss(review_list, weight, tfidf=False):
    return 1-accuracy_percent(review_list, weight, tfidf=False)
```

In [0]:
```python
def svm_predict(review_X, weight):
    if dotProduct(weight, review_X)>0:
        return 1
    else:
        return -1
```

4.

In [0]:
```python
import random
def pegasos(review_list, max_epoch, lam, watch_list=None, grad_checking=False):
    """
    Description
    ==========
    Implementation of Pegasos Algorithm

    Input
    =====
    review_list: list of reviewInstance's
    list of objects with labels and encoded input from reviews

    max_epoch: int
    stopping condition

    lam: float
    regularization parameter

    watch_list: list or reviewInstance's
    passed to accuracy_percent or magnitude_compare; default None

    grad_checking: bool
    numerical test of gradient of svm objective

    Output
    ======
    weights
    """

    #Initialization
    weight = Counter()
    epoch = 0
    t = 0.
    review_number = len(review_list)
    weight_grad=Counter()

    #Loop
    # Use the util.increment and util.dotProduct functions in update
    while epoch < max_epoch:
        #print("------ epoch",epoch,"------")
        random.seed(0)
        random.shuffle(review_list)
        for i in range(review_number):
            t = t+1
            step = 1/(t*lam)
            Xi = review_list[i].word_count
            yi = review_list[i].label
            weight_grad = pegasos_sgd_gradient(Xi, yi, weight, lam)
            increment(weight, -1*step, weight_grad)
        epoch = epoch+1
    #print("The loss for the last instance is",pegasos_sgd_loss(Xi, yi, weight, lam))
    trainloss = loss(review_list, weight)
    if watch_list != None:
        testloss = loss(watch_list, weight)
    return weight,trainloss,testloss
```

5.

In [0]:
```python
def pegasos_fast(review_list, max_epoch, lam, watch_list=None, grad_checking=False, tfidf= False,con=False):
    """
    Description
    ==========
    Implementation of Pegasos Algorithm

    Input
    =====
    review_list: list of reviewInstance's
    list of objects with labels and encoded input from reviews

    max_epoch: int
    stopping condition

    lam: float
    regularization parameter

    watch_list: list or reviewInstance's
    passed to accuracy_percent or magnitude_compare; default None

    grad_checking: bool
    numerical test of gradient of svm objective

    tfidf: bool
    use tf-idf encoding of text in review_list

    Output
    ======
    weights
    """

    #Initialization
    weight = Counter()
    epoch = 0
    t = 1.
    review_number = len(review_list)
    s = 1.
    l=0
    pre=0
    #Loop
    # Use the util.increment and util.dotProduct functions in update
    while epoch < max_epoch:
        l=0
        random.seed(0)
        random.shuffle(review_list)
        #print("------ epoch",epoch,"------")
        for i in range(review_number):
            t=t+1
            step = 1/(t*lam)
            s = (1-step*lam)*s
            Xi = review_list[i].word_count
            yi = review_list[i].label
            if yi*dotProduct(Xi,scale_counter(s,weight))<1:
                increment(weight, step*yi/s,Xi)
            l=l+pegasos_sgd_loss(Xi, yi, scale_counter(s,weight), lam)
        if con==True:
            l= l/review_number
            if np.abs(l-pre) < 0.1:
                print("Converged at epoch",epoch+1,"with current pegasos-sgd-loss",l)
                w = scale_counter(s,weight)
                trainloss = loss(review_list, w)
                if watch_list != None:
                    testloss = loss(watch_list, w)
                return w,trainloss,testloss
            pre = l
        epoch = epoch+1
    w = scale_counter(s,weight)
    trainloss = loss(review_list, w)
    if watch_list != None:
        testloss = loss(watch_list, w)
    return w,trainloss,testloss
```

6.

In [16]:
```
import time

print ("Loading data")
review_list= shuffle_data()

# Sparse representation Q1
print ("Train validation split")
train_review = list(map(reviewInstance,review_list[:1500]))
validate_review = list(map(reviewInstance,review_list[1500:]))

print ("Build the corpus to get idf")

train_label = np.array([i.label for i in train_review])
validate_label = np.array([i.label for i in validate_review])

print ("In training: %r positive, %r negative" %(np.sum(train_label[train_label>0]), -np.sum(train_label[train_label<0])))
print ("In validation: %r positive, %r negative" %(np.sum(validate_label[validate_label>0]), -np.sum(validate_label[validat
e_label<0])))

# Sparse representaion Q2
print("Build sparse bag-of-word representation")
for i in train_review:
    i.construct_word_dict()
for i in validate_review:
    i.construct_word_dict()
```

```
Loading data
2000
Train validation split
Build the corpus to get idf
In training: 752 positive, 748 negative
In validation: 248 positive, 252 negative
Build sparse bag-of-word representation
```

In [0]:
```
print("Compare pegasos and pegasos algorithm")
train_review2 = []
for r in train_review:
    train_review2.append(r)
t = time.time()
w2,trainl2,testl2 = pegasos_fast(train_review, 10, 1, watch_list=validate_review, tfidf= False)
print("It takes",time.time()-t,"to run pegasos fast algorithm.")
print("Training loss for the fast pegasos algorithm is",trainl2)
print("Validation loss for the fast pegasos algorithm is",testl2)
t = time.time()
w1,trainl1,testl1 = pegasos(train_review2, 10, 1, watch_list=validate_review)
print("It takes",time.time()-t,"to run pegasos algorithm.")
print("Training loss for the pegasos algorithm is",trainl1)
print("Validation loss for the pegasos algorithm is",testl1)
```

```
Compare pegasos and pegasos algorithm
It takes 259.5829644203186 to run pegasos fast algorithm.
Training loss for the fast pegasos algorithm is 0.10733333333333328
Validation loss for the fast pegasos algorithm is 0.20199999999999996
It takes 531.0085389614105 to run pegasos algorithm.
Training loss for the pegasos algorithm is 0.09599999999999997
Validation loss for the pegasos algorithm is 0.19799999999999995
```

It takes to around 53s run pegasos algorithm each epoch and approximately 26s to run pegasos fast algorithm per epoch. The pegasos fast algorithm speeds up the computation. In the pegasos fast algorithm, we scale a number s by $1 - \eta_t \lambda$ instead of scale every entry in w to reconstruct it in every iteration, which saves much time. Obviously, the training and validation loss of these two algorithms are close. Now let's see if these two algorithms give the same result by comparing weights.

In [0]:
```
w1['to']-w2['to']
```

Out[0]:
```
-0.002601817656600853
```

I randomly picked a word and their loss in these two weight results are very close.

In [0]:
```
increment(w1,-1,w2)
dotProduct(w1,w1)
```

Out[0]:
```
0.0012923301940240468
```

We observe that after running 10 epochs, the distance between weight results of these two algorithms is very small. If I run more epochs, the distance should ultimately converge to 0. These two algorithms give essentially the same result. Thus, my implementation is correct.

8.

In [20]:
```
lam_list = [1e-1,1e0,1e1]
w_list = []
trlist = []
telist = []
for la in lam_list:
    w,trainl,testl = pegasos_fast(train_review, 10, la, watch_list=validate_review, con=True)
    w_list.append(w)
    trlist.append(trainl)
    telist.append(testl)
```

```
Converged at epoch 6 with current pegasos-sgd-loss 0.4342515443048777
Converged at epoch 3 with current pegasos-sgd-loss 0.37624569630689797
Converged at epoch 2 with current pegasos-sgd-loss 0.7001482296335814
```

In [21]:
```
print(trlist)
```

```
[0.043333333333333335, 0.16133333333333333, 0.2646666666666667]
```

In [22]:
```
print(telist)
```

```
[0.17200000000000004, 0.22199999999999998, 0.274]
```

We observe that as the λ value increases, the algorithm converges faster, however the loss becomes worse. The convergence criterion is that the the abusolute value of the difference of the objective function loss between two epochs should be smaller than ϵ. When the order of magnitude is -1, we get the optimal result.

In [27]:
```
lam_list = [0.08,0.12]
w_list = []
trlist = []
telist = []
for la in lam_list:
    w,trainl,testl = pegasos_fast(train_review, 10, la, watch_list=validate_review, con=True)
    w_list.append(w)
    trlist.append(trainl)
    telist.append(testl)
```

```
Converged at epoch 6 with current pegasos-sgd-loss 0.5102463223562713
Converged at epoch 5 with current pegasos-sgd-loss 0.4512202981001064
```

In [28]:
```
print(trlist)
```

```
[0.03866666666666663, 0.058666666666666645]
```

In [29]:
```
print(telist)
```

```
[0.17400000000000004, 0.16000000000000003]
```

We observe that after zooming in, when $\lambda = 0.12$, we get the optimal loss.

1003 hw4

Xinmeng Li, xl1575

April 3, 2020

1 Perceptron

1. Since $g \in \partial f_k(x)$, $f_k(z) \geq f_k(x) + g^T(z-x)$. Since $f(x)$ is the pointwise maximum of f_i, $i = 1...m$, $\forall z, f(z) \geq f_k(z)$. Obviously, $f(z) \geq f_k(z) \geq f_k(x) + g^T(z-x) = f(x) + g^T(z-x)$. Therefore, $g \in \partial f(x)$.

2. $\partial J(w) = \partial max(0, 1 - yw^T x)$
Note that if $yw^T x = 1$, we can choose $\partial J(w)$ to be any value between $\partial 1 - yw^T x$ and $\partial 0$(both inclusive). Here, we select $\partial J(w) = \partial 1 - yw^T x$ for $yw^T x = 1$.
We then get

$$= \begin{cases} \partial 0, \text{ if } yw^T x > 1 \\ \partial 1 - yw^T x, \text{ otherwise} \end{cases} = \begin{cases} 0, \text{ if } yw^T x > 1 \\ -yx, \text{ otherwise} \end{cases} = \mathbf{1}(yw^T x <= 1)(-yx)$$

3. Since $\{x | w^T x = 0\}$ is separating the training data, $\forall i = 1...n, y_i x_i^T w > 0$. The average perceptron loss is $\frac{1}{n}\sum_{i=1...n} max(0, -y_i\hat{y}_i) = \frac{1}{n}\sum_{i=1...n} max(0, -y_i x_i^T w) = \frac{1}{n} * n * 0 = 0$. Since the empirical perception loss is always non-negative, 0 is its minimum. Since any separating hyper-plane of D has a 0 average perceptron loss, any separating hyper-plane of D is the empirical loss minimizer for perceptron loss.

4. In the perceptron algorithm, we update w by
$w^{k+1} = w^k + 1(y_i x_i^T w^k <= 0)(y_i x_i)$. In the SSGD, we update w by
$w^{k+1} = w^k - \eta \partial l(y_i, \hat{y}_i)$. Similar to problem 2, $\partial l(y_i, \hat{y}_i) = \partial max(0, -y_i w^{kT} x_i)$. We can select $\partial l(y_i, \hat{y}_i) = \partial(-y_i w^{kT} x_i)$ for $y_i w^{kT} x_i = 0$.
We then get

$$= \begin{cases} \partial 0, \text{ if } yw^T x > 0 \\ \partial(-y_i w^{kT} x_i), \text{ otherwise} \end{cases} = \begin{cases} 0, \text{ if } y_i w^{kT} x_i > 0 \\ -y_i x_i, \text{ otherwise} \end{cases} = \mathbf{1}(y_i w^{kT} x_i <= 0)(-y_i x_i).$$

Let $\eta = 1$, then we get
$w^{k+1} = w^k - \mathbf{1}(y_i x_i^T w^k <= 0)(-y_i x_i)$, which is exactly same to the update rule of the perceptron algorithm. Since we terminate when the training data are separated, here SSGD uses the same stopping criterion with the perceptron algorithm. Therefore, in this case, these two algorithms are exactly same.

5. In the perceptron algorithm, we update the value of w to $w + y_i x_i$ only when $y_i x_i^T w \leq 0$. Since the initial value of w is $(0, ..., 0)$, the returned $w = c_1 y_1 x_1 + ... + c_n y_n x_n$, where c_i is a constant. Since y_i is a number instead of vector, $\alpha_i = c_i y_i \in \mathbf{R}, w = \sum_{i=1...n} \alpha_i x_i$, i.e. w is a linear combination of x_i.

Characteristics: Any support vector x_i associated with nonzero α_i was used to be classified mistakenly and therefore the w is updated by $w = w + y_i x_i$. When $\alpha_i = 0$, the algorithm predicts the true label from the start, i.e. $y_i x_i^T w > 0$ all the time.

2 Spare Representation

Please refer to the code.

3 SVM with Pegasos

1. $\nabla J_i(w) = \nabla \frac{\lambda}{2}||w||^2 + max(0, 1 - y_i w^T x_i)$

$$= \begin{cases} \nabla \frac{\lambda}{2}||w||^2, \text{ if } y_i w^T x_i > 1 \\ \text{undefined, if } y_i w^T x_i = 1 \\ \nabla \frac{\lambda}{2}||w||^2 + 1 - y_i w^T x_i, \text{ if } y_i w^T x_i < 1 \end{cases}$$

$$= \begin{cases} \lambda w, \text{ if } y_i w^T x_i > 1 \\ \text{undefined, if } y_i w^T x_i = 1 \\ \lambda w - y_i x_i, \text{ if } y_i w^T x_i < 1 \end{cases}$$

2. $\partial J_i(w) = \partial(\frac{\lambda}{2}||w||^2 + max(0, 1 - y_i w^T x_i))$
Since $\frac{\lambda}{2}||w||^2$ and $max(0, 1 - y_i w^T x_i)$ are both convex, $\partial J_i(w) = \partial \frac{\lambda}{2}||w||^2 + \partial max(0, 1 - y_i w^T x_i)$.
Since $\lambda > 0$, $\partial \frac{\lambda}{2}||w||^2 = \frac{\lambda}{2}\partial||w||^2 = \lambda w$ (Differentiable).
Note that if $y_i w^T x_i = 1$ (Non-differentiable), we can choose $\partial max(0, 1 - y_i w^T x_i)$ to be any value between $\partial 1 - y_i w^T x_i$ and $\partial 0$(both inclusive). Here, we select $\partial 0$ for $y_i w^T x_i = 1$.
We then get
$\partial J_i(w) = \partial(\frac{\lambda}{2}||w||^2 + \partial max(0, 1 - y_i w^T x_i))$

$$= \begin{cases} \lambda w - y_i x_i, \text{ if } y_i w^T x_i < 1 \\ \lambda w, \text{ if } y_i w^T x_i \geq 1 \end{cases}$$

$= \lambda w + \mathbf{1}(y_i w^{kT} x_i < 1)(-y_i x_i)$

3. In the SVM Pegasos algorithm, we update w by
$w^{k+1} = w^k - \eta_k \lambda w_k + \mathbf{1}(y_i x_i^T w^k < 1)(\eta_k y_i x_i)$.
In the SGD with the subgradient in Q2, we update w by
$w^{k+1} = w^k - \eta_k \partial J_i(w) = w^k - \eta_k \lambda w - \eta_k \mathbf{1}(y_i w^{kT} x_i < 1)(-y_i x_i) = w^k - \eta_k \lambda w + \eta_k \mathbf{1}(y_i w^{kT} x_i < 1)(y_i x_i)$. Since $\eta_k = \frac{1}{\lambda_k}$ in both of these two algorithms, the update rule of the SGD with the subgradient in Q2 is exactly same to the update rule of the SVM Pegasos algorithm.

4. Verification: Since in the fast Pegasos algorithm, $w_t = s_t W_t$, after we plug in those two equations of s_{t+1} and W_{t+1} values, we get
$w_{t+1} = s_{t+1} W_{t+1} = (1 - \eta_t \lambda) s_t (W_t + \frac{1}{(1-\eta_t\lambda)s_t} \eta_t y_i x_i)$
$= (1 - \eta_t \lambda) s_t W_t + \eta_t y_i x_i$
$= (1 - \eta_t \lambda) w_t + \eta_t y_i x_i$, which is the update rule of the Pegasos algorithm.

4 Kernels

1. Let B be a set containing all of the unique words that occur in either x or z. Let ϕ maps x to a vector $\phi(x)$ with $|B|$ entries. Each entry of $\phi(x)$ represents whether the ith word of B

exists in x. i.e. If the ith word in B exists in x, $\phi(x)[i] = 1$, otherwise, $\phi(x)[i] = 0$. Thus, $\phi(x)^T\phi(z) = \sum_{i=1}^{|B|} \phi(x)[i]\phi(z)[i] =$ the number of unique words that occur both in x and z, which indicates that $k(x,z) = \phi(x)^T\phi(z)$ is a kernel function.

2. $k(x,z) = x^T z$ is a kernel $\xrightarrow{by(a)}$ $\frac{x^T z}{||x||_2||z||_2}$ is a kernel $\xrightarrow{by(b)}$ $1 + \frac{x^T z}{||x||_2||z||_2}$ is a kernel $\xrightarrow{by(c)}$ $(1 + \frac{x^T z}{||x||_2||z||_2})(1 + \frac{x^T z}{||x||_2||z||_2})$ is a kernel $\xrightarrow{by(c)}$ $(1 + \frac{x^T z}{||x||_2||z||_2})^2(1 + \frac{x^T z}{||x||_2||z||_2}) = (1 + \frac{x^T}{||x||_2}\frac{z}{||z||_2})^3 = (1 + (\frac{x}{||x||_2})^T(\frac{z}{||z||_2}))^3$ is a kernel.

5 Kernel Pegasos

1. $y_j < w^{(t)}, x_j >= y_j < \sum_{i=1}^n \alpha_i^{(t)} x_i, x_j >= y_j \sum_{i=1}^n < \alpha_i^{(t)} x_i, x_j >$
$= y_j \sum_{i=1}^n \alpha_i^{(t)} < x_i, x_j >= y_j K_{j\cdot}\alpha^{(t)}$

2. $w_{t+1} = (1-\eta_t\lambda)w_t = (1-\eta_t\lambda)\sum_i \alpha_i^{(t)} x_i = \sum_i \alpha_i^{(t+1)} x_i$, where $\alpha_i^{(t+1)} = (1-\eta_t\lambda)\alpha_i^{(t)}$,
thus $\alpha^{(t+1)} = (1-\eta_t\lambda)\alpha^{(t)}$

3. $w_{t+1} = (1-\eta_t\lambda)w_t + \eta_t x_j y_j = (1-\eta_t\lambda)(\sum_i \alpha_i^{(t)} x_i) + \eta_t y_j x_j$,
thus $\alpha_i^{(t+1)} = (1-\eta_t\lambda)\alpha_i^{(t)} + \mathbf{1}(i=j)\eta_t y_j$, $\alpha^{(t+1)} = (1-\eta_t\lambda)\alpha^{(t)} + [0, ..., 1, ...0]\eta_t y_j$, where 1 is at the jth position.
pseodocode:

input: K and $\lambda > 0$.
t=0, $\alpha^{(0)} = (0, ..., 0)$
While termination condition not met
  For $j = 1, ..., m$
    $t = t + 1$
    $\eta^{(t)} = 1/(t\lambda)$
    if $y_j \left\langle K_{j\cdot}, \alpha^{(t-1)} \right\rangle < 1$
      $\alpha^{(t)} = (1-\eta_t\lambda)\alpha^{(t-1)}$
      $\alpha_j^{(t)} = \alpha_j^{(t)} + \eta_t y_j$
    else
      $\alpha^{(t)} = (1-\eta_t\lambda)\alpha^{(t-1)}$
return $\alpha^{(t)}$