In [1]:
```
# Lets import all the libraties needed
import numpy as np
from sklearn import preprocessing
import random
from sklearn.datasets import load_digits
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import f1_score, accuracy_score
from keras.datasets import fashion_mnist

import time
```

```
/home/anjan/anaconda3/lib/python3.6/site-packages/h5py/__init__.py:36: Future
Warning: Conversion of the second argument of issubdtype from `float` to `np.
floating` is deprecated. In future, it will be treated as `np.float64 == np.d
type(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Loading Pegassos and Mercer_Pegasos

Classes Pegassos and Mercer_Pegasos are declared in the file pegasos.py

So instead of declaring these classes in the following cells you can just uncomment the below line

In [2]:
```
# from pegasos import Pegasos, Mercer_Pegasos
```

Lets see a simple Pegasos class and its functions

In [3]:
```
class Pegasos(object):
    def __init__(self, n_iter=10, lambda1=1, projection=False, bias = False,
objective='hinge', margin = 0.2):
#         Initialise the variables needed
        self.n_iter = n_iter # number of iterations
        self._lambda = lambda1 # parameter that controls the descent
        self.projection = projection # Optional projection step
        self.labelEncoder = None # encodesthe labels to 0, 1
        self.bias = bias # Set it to true if a bias variable has to be added.
        self.objective = objective # Objective function. can take values ['hi
nge', 'sigmoid', 'margin']
        self.margin = margin # size of the margin needed if the objective fun
ction is set to 'margin' (epislon value)

    def fit(self, X, Y, verbose=False):
        starttime = time.time()

#         Encode the labels
        self.labelEncoder = preprocessing.LabelEncoder()
        self.labelEncoder.fit(Y)
        Y_labels = 2*self.labelEncoder.transform(Y) - 1

#         Add 1 at the end if bias is needed
        if (self.bias):
            X = np.append(X, np.ones(X.shape[0]).reshape(-1, 1), axis = 1)

#         initialize w
        self.w = np.zeros((X.shape[1]))

        for t in range(self.n_iter):
            i_t =  random.randint(0, X.shape[0] - 1)
            x = X[i_t]
            y = Y_labels[i_t]

            eta = 1.0/float(self._lambda * (t+1))

#             Depending on the different loss functions, we update self.w wit
h respective sub-gradients
            if(self.objective == 'hinge'):
                if((y*( self.w.dot(x)))  < 1):
                    self.w = (1 - eta*self._lambda)*self.w + eta*y*((x))
                else:
                    self.w = (1 - eta*self._lambda)*self.w

            elif (self.objective == 'sigmoid'):
                self.w = (1 - eta*self._lambda)*self.w + eta* (y/(1 + np.exp(
(y*( self.w.dot(x))))))) * x


            elif (self.objective == 'margin'):
                if(self.w.dot(x) - y > self.margin):
                    self.w = (1 - eta*self._lambda)*self.w + ((x))
                elif(y - self.w.dot(x) > self.margin):
                    self.w = (1 - eta*self._lambda)*self.w - ((x))

            if(self.projection):
                self.w = (np.min((((1.0/ np.sqrt(self._lambda))/np.linalg.nor
m(self.w)), 1.0))) * self.w

        if(verbose):
            print("Total time take to train data of size " + str(X.shape[0])
+ " datapoints with " +
             str(X.shape[1]) + " features is " + str(time.time() - starttime)
+ " seconds")


    def predict(self, X):
        if(self.bias) :
```

Now lets see the entire procedure.

Subgradient Update

In the function fit() in the above cell, self.w is updated in the below form

```
eta = 1.0/float(self._lambda * (t+1))
self.w = (1 - eta*self._lambda)*self.w + eta*(sub_gradient)
```

This process is strickingly similar to SGD. But here the step size is carefully chosen so that we can have an upperbound on the number of iterations needed. We can see that as the number of timesteps t increases, eta decreases and we keep taking smaller steps towards the goal. It makes sense because intially we are far off the goal, so we have to take higher steps, but more we go towards the goal, we need to keep taking fewer steps.

Projection Step

```
self.w = (np.min((((1.0/ np.sqrt(self._lambda))/np.linalg.norm(self.w)), 1.0))) * self.w
```

In cases of outliers, they give us a gradient direction, thats very far off. So in order to handle cases like this, we restrict set of admissable values to a circle if radius (1/sqrt(lambda))

Objective functions

Objective functions or loss functions tell you how far off you are from the goal. Our objective is the minimize this fucntion globally. But in algorithms like this, we can only achieve local minima. Given below are some objective functions and their subgradients.

Loss function	Subgradient
$\ell(z, y_i) = \max\{0, 1 - y_i z\}$	$\mathbf{v}_t = \begin{cases} -y_i \mathbf{x}_i & \text{if } y_i z < 1 \\ \mathbf{0} & \text{otherwise} \end{cases}$
$\ell(z, y_i) = \log(1 + e^{-y_i z})$	$\mathbf{v}_t = -\frac{y_i}{1+e^{y_i z}} \mathbf{x}_i$
$\ell(z, y_i) = \max\{0, \lvert y_i - z \rvert - \epsilon\}$	$\mathbf{v}_t = \begin{cases} \mathbf{x}_i & \text{if } z - y_i > \epsilon \\ -\mathbf{x}_i & \text{if } y_i - z > \epsilon \\ \mathbf{0} & \text{otherwise} \end{cases}$
$\ell(z, y_i) = \max_{y \in \mathcal{Y}} \delta(y, y_i) - z_{y_i} + z_y$	$\mathbf{v}_t = \phi(\mathbf{x}_i, \hat{y}) - \phi(\mathbf{x}_i, y_i)$ where $\hat{y} = \arg\max_y \delta(y, y_i) - z_{y_i} + z_y$
$\ell(z, y_i) = \log\left(1 + \sum_{r \neq y_i} e^{z_r - z_{y_i}}\right)$	$\mathbf{v}_t = \sum_r p_r \phi(\mathbf{x}_i, r) - \phi(\mathbf{x}_i, y_i)$ where $p_r = e^{z_r} / \sum_j e^{z_j}$

In this code, the first three are implemented. as 'hinge', 'sigmoid', 'margin' losses.

Now Lets run some tests

In [4]:
```
# Load the data

((trainX, trainY), (testX, testY)) = fashion_mnist.load_data()
trainX = trainX.reshape(trainX.shape[0], -1)
testX = testX.reshape(testX.shape[0], -1)

trainX = np.copy(trainX)
trainY = np.copy(trainY)
testX = np.copy(testX)
testY = np.copy(testY)

trainY[trainY > 0] = 1
testY[testY > 0] = 1
```

In [5]:
```
# Call a linear  Pagasos object
clf = Pegasos( n_iter=100, objective='hinge')

# Fit the algortihm.
# Set verbose=True if you want to display fitting time and dataset informatio
n.
clf.fit(trainX, trainY, verbose=True)
```

```
Total time take to train data of size 60000 datapoints with 784 features is 0
.003719806671142578 seconds
```

In [6]:
```
# Predict on a given testdata
Ypred = clf.predict(testX)
```

```
/home/anjan/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/label
.py:151: DeprecationWarning: The truth value of an empty array is ambiguous.
Returning False, but in future this will result in an error. Use `array.size
> 0` to check that an array is not empty.
  if diff:
```

In [7]:
```
#Run metrics on the given testdata
# Set verbose=True if you want to display inference time and dataset informat
ion.
clf.test(testX, testY, verbose=True)
```

```
Total time take to test on  data of size 10000 datapoints with 784 features i
s 0.08300113677978516 seconds
Accuracy is : 0.9419
F1 score is : 0.9675690761931341
```

```
/home/anjan/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/label
.py:151: DeprecationWarning: The truth value of an empty array is ambiguous.
Returning False, but in future this will result in an error. Use `array.size
> 0` to check that an array is not empty.
  if diff:
```

Results

Now lets see the results with different objective fucntions

	10		50		100		1000		5000	
Hinge	0.16 0.13	0.005	0.59 0.70	0.003	0.90 0.95	0.004 43	0.928 0.961	0.020	0.923 0.958	0.083
Sigmoid	0.9 0.94	0.015	0.90 0.95	0.004 556	0.90 0.95	0.006	0.947 0.971	0.036 56	0.891 0.936	0.159 997
Margin	0.1 0.0	0.002 8	0.1 0.0	0.003 66	0.1 0.0	0.005 13	0.9 0.947	0.017 35	0.1 0.114 008	0.114 008

objective function
number of iterations
accuracy
f1 Score
training time

Now lets see a Kernalized Pegasos class

In [8]:
```
class Mercer_Pegasos(Pegasos):
    def __init__(self, n_iter=10, lambda1=1, projection=False, bias = False,
kernel = None):
        super().__init__(n_iter, lambda1, projection, bias) # Variables same
as in linear Pegasos
        if not (kernel):    # if not kernal is specified, guassian rbf kernal
is used by default
            kernel = self.rbf
        self.kernel = kernel

    def fit(self, X, Y, verbose=False):
        starttime = time.time()

        self.labelEncoder = preprocessing.LabelEncoder()
        self.labelEncoder.fit(Y)
        Y_labels = 2*self.labelEncoder.transform(Y) - 1
        self.w = np.zeros((X.shape[1]))
        self.alpha = np.zeros((X.shape[0]))
        self.X = X

        for t in range(self.n_iter):
            i_t =  random.randint(0, X.shape[0] - 1)
            x = X[i_t]
            y = Y_labels[i_t]

            eta = 1.0/float(self._lambda * (t+1))

            error = 0
            for j in range(X.shape[0]):
                error += self.alpha[j] * y * (self.kernel(x, X[j]))
            if( (y*(1/self._lambda)* error) < 1):
                self.alpha[i_t]+=1
        if(verbose):
            print("Total time take to train data of size " + str(X.shape[0])
+ " datapoints with " +
             str(X.shape[1]) + " features over " +str(self.n_iter) +  " itera
tions is " + str(time.time() - starttime) + " seconds")

    def predict(self, X, ):
        Ypred = np.zeros((X.shape[0]))
        for i in range(X.shape[0]) :
            wTx = 0
            for j in range(self.X.shape[0]) :
                wTx += self.alpha[j] * self.kernel(X[i], self.X[j])
            Ypred[i] = np.sign(wTx)
        Ypred[Ypred > 0 ] = 1
        Ypred[Ypred <= 0 ] = 0
        return Ypred

    def test(self, X, Y, verbose = False):
        starttime = time.time()

        Ypred = np.zeros((X.shape[0]))
        for i in range(X.shape[0]) :
            wTx = 0
            for j in range(self.X.shape[0]) :
                wTx += self.alpha[j] * self.kernel(X[i], self.X[j])
            Ypred[i] = np.sign(wTx)
        Ypred[Ypred > 0 ] = 1
        Ypred[Ypred <= 0 ] = 0

        Y =  clf.labelEncoder.inverse_transform(Ypred.astype(int))

        accuracy = accuracy_score(Y, testY)
        f1 = f1_score(Y, testY)

        print("Accuracy is : " + str(accuracy))
        print("F1 score is : " + str(f1))
```

Problem with kernels

Major difference in kernalized version of any algorithm is that we dont explicitly have the function mapping. So we cannot store w and multiply it with out feature vector phi(x). becuase we do not have the function phi().

But we have a function K() such that K(x1,x2) = phi(x1) * phi(x2)

Solution

In the linear algorithm w is updates as below.

```
self.w = (1 - eta*self._lambda)*self.w + eta*y*((x))
```

So we take advantage of the fact that in the linear version of the algorithm, w (ie self.w) starting from 0, is always added our featurevector multiplied with some number. Hence w is a linear combination of our feature vectors. So it is sufficient to store the weights (alpha) for each feature vector in the training set, so that we can reconstruct w later by multiplying corresponding alpha and featurevector x.

((alpha(1) * X(1)) + (alpha(2) * X(2)) .....) * X_test

```
                    =  ( alpha(1) * (X(1) * Xtest) ) + ( alpha(2) * (X(2) * Xtest)
) ......)

                    =  ( alpha(1) * (K(X(1),Xtest)) ) + ( alpha(2) * (K(X(2),Xtest)
)  ) ......)
```

Hence it is sufficient to store the alpha array. But this has an obvious disadvantage of too much inference time.

Loading kernels

The kernals are implemented in the file kernels.py

So instead of declaring these fucntions in the following cells you can just uncomment the below line

In [9]:
```
# from kernels import rbf, polynomialkernel, laplacerbfkernel, sigmoidkernel
```

In [10]:
```
def rbf(x1, x2, gamma = 0.5):
    return np.exp(-gamma* np.linalg.norm(x1-x2) * np.linalg.norm(x1-x2))

def polynomialkernel(x1, x2, d =3):
    return np.power(x1.dot(x2) + 1, d)

def laplacerbfkernel(x1, x2, sigma = 0.1):
    return np.exp((-1/sigma) * np.linalg.norm(x1-x2))

def sigmoidkernel(x1, x2, alpha=0.1, c=0):
    return np.tanh(alpha * x1.dot(x2) + c)
```

In [11]:
```
# Load the data. We will be loading a coparitively smaller dataset because ke
rnalized Pegassos is slow
dataset = load_breast_cancer()
X = dataset.data
Y = dataset.target
trainX, testX, trainY, testY = train_test_split(X, Y, test_size=0.2, random_s
tate=42)
```

In [12]:
```
# Call a linear  Pagasos object
clf = Mercer_Pegasos( n_iter=100, kernel=rbf)

# Fit the algortihm.
# Set verbose=True if you want to display fitting time and dataset informatio
n.
clf.fit(trainX, trainY, verbose=True)
```

```
Total time take to train data of size 455 datapoints with 30 features over 10
0 iterations is 0.6901099681854248 seconds
```

In [13]:
```
# Predict on a given testdata
Ypred = clf.predict(testX)
```

In [14]:
```
#Run metrics on the given testdata
# Set verbose=True if you want to display inference time and dataset informat
ion.
clf.test(testX, testY, verbose=True)
```

```
Accuracy is : 0.8859649122807017
F1 score is : 0.906474820143885
Total time take to test on  data of size 114 datapoints with 30 features is 0
.7540173530578613 seconds

/home/anjan/anaconda3/lib/python3.6/site-packages/sklearn/preprocessing/label
.py:151: DeprecationWarning: The truth value of an empty array is ambiguous.
Returning False, but in future this will result in an error. Use `array.size
> 0` to check that an array is not empty.
  if diff:
```

Results

Now lets see the results with different objective kernels

	10		50		100		1000	
Guassianrbf	0.57 0.47	0.100 0.660	0.80 0.83	0.437 0.726	0.89 0.91	0.659 0.678	0.807 0.864	5.975 0.725
Polyno-mial	0.62 0.76	0.022 0.261	0.62 0.76	0.101 0.285	0.62 0.76	0.215 0.263	0.622 0.767	2.078 0.257
Laplacebf	0.82 0.85	0.038 0.471	0.78 0.84	0.230 0.438	0.78 0.84	0.456 0.463	0.699 0.802	4.119 0.530
Sigmoid	0.62 0.76	0.021 0.22	0.62 0.76	0.080 0.225	0.62 0.76	0.185 0.231	0.622 0.767	1.737 0.211

objective function
number of iterations
accuracy
f1 Score
training time
testing time