Implementing PEGASOS

Rudrabha Mukhopadhyay (2018801004), Sangeeth Reddy Battu (2018900002)

May 2019

1 Introduction

In general, Support Vector Machines are generally implemented by solving the dual optimization problem using quadratic programming. But this is time consuming and may not be suitable for problems which involves very large amount of data. The method discussed in this report is known as Pegasos which stands for Primal Estimated sub-GrAdient SOlver. This method was introduced in [1]. In this method we implement the vanilla version of Pegasos for binary classification. We later kernelize this implementation and test it with two different types kernels. Finally we also extend the vanilla implementation for multi class classification. Finally we compare the results obtained by our classifier against the results that were obtained by using sklearn's SVM function. [1]

2 Vanilla Algorithm and Implementation

In its native form, the primal objective function of a SVM is in fact an unconstrained empirical loss minimization with a penalty term for the norm of the classifier that is being learned. Formally, given a training set $S = \{(x_i, y_i)\}_{i=1}^{m}$ where $x_i \in \mathbf{R}$ and $y_i \in \{-1, +1\}$. The primal objective function is given in equation 1.

[1] Github Link: `https://github.com/sangeeth93/Pegasos_SVM`

$$\min_{\mathbf{w}} \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{m}\sum_{(\mathbf{x},y)\in S} \ell(\mathbf{w};(\mathbf{x},y)) \ , \qquad (1)$$

where

$$\ell(\mathbf{w};(\mathbf{x},y)) \ = \ \max\{0, 1 - y\,\langle \mathbf{w},\mathbf{x}\rangle\} \ . \qquad (2)$$

In this section we discuss about the pegasos algorithm and check how to implement it. The algorithm receives as input two parameters: T - the number of iterations to perform; k - the number of examples to use for calculating sub-gradients. Initially, we set w_1 to any vector whose norm is at most $1/\lambda$. We modify the objective function given in equation 1 to the following,

$$f(\mathbf{w};A_t) \ = \ \frac{\lambda}{2}\|\mathbf{w}\|^2 + \frac{1}{k}\sum_{(\mathbf{x},y)\in A_t} \ell(\mathbf{w};(\mathbf{x},y)) \ .$$

Next, the vanilla version of the pegasos algorithm is given by,

INPUT: S, λ, T, k
INITIALIZE: Choose $\mathbf{w}_1$ s.t. $\|\mathbf{w}_1\| \leq 1/\sqrt{\lambda}$
FOR $t = 1, 2, \ldots, T$
  Choose $A_t \subseteq S$, where $|A_t| = k$
  Set $A_t^+ = \{(\mathbf{x},y) \in A_t : y\,\langle \mathbf{w}_t,\mathbf{x}\rangle < 1\}$
  Set $\eta_t = \frac{1}{\lambda t}$
  Set $\mathbf{w}_{t+\frac{1}{2}} \ = \ (1 - \eta_t\,\lambda)\mathbf{w}_t + \frac{\eta_t}{k}\sum_{(\mathbf{x},y)\in A_t^+} y\,\mathbf{x}$
  Set $\mathbf{w}_{t+1} = \min\left\{1, \frac{1/\sqrt{\lambda}}{\|\mathbf{w}_{t+\frac{1}{2}}\|}\right\} \mathbf{w}_{t+\frac{1}{2}}$
OUTPUT: $\mathbf{w}_{T+1}$

Here, S is the total number of samples present in the training data. T is the total number of iterations. k is the total number of samples to consider while calculating the sub-gradient. In our implementation, we choose $k = S$. Thus, we use all the samples present in the training set for sub gradient calculation. We, also choose

$\lambda = 1$ in our implementation. This algorithm uses hinge loss as the loss function. Hinge loss is generally defined as for an intended output $t = +1 or t = -1$ the loss is, $J(y) = max(0, 1 - t.y)$

For training, we use Fashion-MNIST dataset [2]. This dataset has 60,000 training images of 10 classes along with 10,000 testing images. We randomly select two classes for our binary classifier and test on them. Hence, the number of training samples for our system at any point reduces to 12,000 and we have 2000 test samples of 2 classes.

In our implementation, we use a publicly available data loader to load the Fashion-MNIST dataset [2]. Figure 1 refers to the function used to load the dataset.

```
def load_mnist(path, kind='train'):
    import os
    import gzip
    import numpy as np

    """Load MNIST data from `path`"""
    labels_path = os.path.join(path,
                               '%s-labels-idx1-ubyte'
                               % kind)
    images_path = os.path.join(path,
                               '%s-images-idx3-ubyte'
                               % kind)

    with open(labels_path, 'rb') as lbpath:
        labels = np.frombuffer(lbpath.read(), dtype=np.uint8,
                               offset=8)

    with open(images_path, 'rb') as imgpath:
        images = np.frombuffer(imgpath.read(), dtype=np.uint8,
                               offset=16).reshape(len(labels), 784)

    return images, labels
```

Figure 1: Code for loading the Fashion-MNIST dataset

We then preprocess the data to re-label the data so that that labels $\in \{+1, -1\}$. The code for this operation is given in figure 2.

[2] https://github.com/zalandoresearch/fashion-mnist

```
def create_train_data(X_train, y_train, labels):
    X_train_binary=[]
    y_train_binary=[]
    for k in range(0,y_train.shape[0]):
        if y_train[k]==labels[0]:
            X_train_binary.append(X_train[k])
            y_train_binary.append(1)
        elif y_train[k]==labels[1]:
            X_train_binary.append(X_train[k])
            y_train_binary.append(-1)
    print("Number of train set samples for binary case :",len(X_train_binary))
    return X_train_binary, y_train_binary
```

Figure 2: Code for re-labeling

We then implement the pegasos algorithm as given in figure 3. This figure also contains the loss function implemented in our code.

```
def loss(w,x,y):
    op = y*np.dot(w,x)
    return op

def pegasos(data,labels,T,lamda=1.0):
    S = len(data)
    w = np.zeros((data[0].shape[0]))
    for k in range(1,T):
        i = random.randrange(S)
        step = 1/(k*lamda)
        if loss(w,data[i],labels[i])<1:
            w1 = ((1-step*lamda)*w)+step*data[i]*labels[i]
        elif loss(w,data[i],labels[i])>=1:
            w1 = ((1-step*lamda)*w)
        tt = (1/np.sqrt(lamda)/np.linalg.norm(w1))
        w1 = min(1,tt)*w1
        w = w1
    return w
```

Figure 3: Code for Pegasos

In this function we take data and labels as input. Labels are required for calculating the loss. We also take the total number of iterations as input along with λ which we set as 1. We then call this function calculate the weight vector. The weight vector returned by this function is then used to take inference on the test set. The code for

this operation is given in figure 4.

```
def fit(X_train_binary, y_train_binary):
    w=pegasos(X_train_binary,y_train_binary,6000)
    return w

def test(X_test_binary, y_test_binary, w):
    correct=0
    for k in range(0,len(y_test_binary)):
        if np.dot(w,X_test_binary[k])<0 and y_test_binary[k]<0:
            correct+=1
        elif np.dot(w,X_test_binary[k])>0 and y_test_binary[k]>0:
            correct+=1
    acc = (correct*1.0/len(y_test_binary))
    return acc

X_train, y_train = load_mnist('../fashionmnist/', kind='train')
X_test, y_test = load_mnist('../fashionmnist/', kind='t10k')

X_train_binary, y_train_binary = create_train_data(X_train, y_train, [1,2])
X_test_binary, y_test_binary = create_test_data(X_test, y_test, [1,2])



w = fit(X_train_binary, y_train_binary)
acc = test(X_test_binary, y_test_binary, w)
#print(correct)
print("Test accuracy :", acc)
```

Figure 4: Code for taking an inference for the binary model

We then move on to the kernelized version of this algorithm in the next section.

3 Kernelizing the SVM

The above mentioned vanilla algorithm can be extended for a kernelized version. The common approach for solving the optimization problem for SVM when kernels are employed is to switch to the dual problem and find the optimal set of dual variables. A different approach is used in [1] Here we, directly minimize the primal problem while still using kernels. The main observation by the authors of the original paper was that if w_1 is initialized with a zero vector then at each iteration $w_t = \sum_{i \in I_t} \alpha_i x_i$, where I_t is a subset of $\{1, 2, ..., m\}$. Hence, we can replace w_t from the previous version with α_t. The full algorithm is given below.

INPUT: S, λ, T

INITIALIZE: Set $\boldsymbol{\alpha}_1 = 0$

FOR $t = 1, 2, \ldots, T$

Choose $i_t \in \{0, \ldots, |S|\}$ uniformly at random.

For all $j \neq i_t$, set $\alpha_{t+1}[j] = \alpha_t[j]$

If $y_{i_t} \frac{1}{\lambda t} \sum_j \alpha_t[j] y_{i_t} K(\mathbf{x}_{i_t}, \mathbf{x}_j) < 1$, then:

Set $\alpha_{t+1}[i_t] = \alpha_t[i_t] + 1$

Else:

Set $\alpha_{t+1}[i_t] = \alpha_t[i_t]$

OUTPUT: $\boldsymbol{\alpha}_{T+1}$

This was also implemented in a similar way as the previous one. We use some parts of the previous codes as well. We also test with two different kernels. We use the Radial Basis Function (RBF) kernel and also a homogeneous polynomial kernel. The RBF kernel is given by the below equation.

$$K(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}'\|^2}{2\sigma^2}\right)$$

Here x and x' are two inputs to the function. This kernel is widely used in kernel SVM's because of the fact that it projects the features to an infinite dimensional space. We also use another kernel called the polynomial kernel. A polynomial kernel of degree d is defined as,

$$K(x, y) = (x^T y + c)^d.$$

Here x and y are input vectors and $c \geq 0$. If $c = 0$, then this kernel is called a polynomial homogeneous kernel. We use such a kernel in our implementation. The kernel function implemented by us given in figure 6.

```python
def kernel_f(k,v1,v2,degree):
    #implementation of kernels
    if k == 'homogenious':
        op = np.dot(v1, v2)
        op=np.power(op,degree)
        return op
    elif k == 'radial':
        v=v1-v2
        op = math.exp(-1 * degree * np.linalg.norm(v) ** 2)
        return op
```

Figure 5: Code for implementing the kernel.

The iterative algorithm is implemented using the code snippet given in figure **??**.

```python
S=len(data)
alpha = np.zeros((S))
w = np.zeros((data[0].shape[0]))
for k in tqdm(range(1,T)):
    it = random.randrange(S)
    sm=0
    for j in range(0,S):
        sm+=alpha[j]*labels[it]*kernel_f(kernel,data[it],data[j],deg)
    if labels[it]*(1/(k*lamda))*sm <1:
        alpha[it] = alpha[it]+1 #Update date of alpha vaules

for k in range(0,S):
    w += alpha[k]*labels[k]*data[k] #Converting alpha's to weights for inference
return w
```

Figure 6: Code for implementing the iterative algorithm

We finally get the w by the equation,

$$w = \sum_{i=1}^{S} \alpha_i . y_i . x_i$$

We take inference using this w and report results later. We also extend the linear SVM to a multi class setting instead of the binary classification task in the next section.

4 Multi Class SVM

SVM in its purest form is a binary class classifier. However we can extend this to a multi class setting using two approaches. The are namely, $1vs1$ and $1vsall$. We take the first approach. We create $\binom{N}{2}$ binary classifiers. We choose $N = 3$, i.e. we have 3 classes in our set up. Hence we have 3 binary classifiers as well. They are, label 1 vs label 2, label 2 vs label 3 and label 1 vs label 3. Therefore, during inference, we use a simple voting among all the classifiers while assigning labels. We post a few code snippets for the same in this section. We use the exact same code given in section 2 to create all the binary classifiers. Figure 7 has the code snippet for predicting the class of a test sample.

```
def predict_label(X_test_binary, w):
    predicted = []
    for k in range(0,len(X_test_binary)):
        pred = np.dot(w,X_test_binary[k])
        if pred < 0:
            predicted.append(-1)
        else:
            predicted.append(1)
    return predicted



def predict_class(predicted_labels, labels):
    for i in range(len(predicted_labels)):
        if predicted_labels[i] == 1:
            predicted_labels[i] = labels[0]
        else:
            predicted_labels[i] = labels[1]
    return predicted_labels
```

Figure 7: Code for taking a prediction which will be used for voting later

We also provide the code for the voting function in figure 8.

```
def voting(lab1, lab2, lab3, labels):
    final_pred = []
    for i in range(len(lab1)):
        temp = [lab1[i], lab2[i], lab3[i]]
        c1 = temp.count(labels[0])
        c2 = temp.count(labels[1])
        c3 = temp.count(labels[2])
        if c1 == 1 and c2 == 1 and c3==1:
            final_pred.append(-1)
        elif c1 == max(c1,c2,c3):
            final_pred.append(labels[0])
        elif c2 == max(c1,c2,c3):
            final_pred.append(labels[1])
        elif c3 == max(c1,c2,c3):
            final_pred.append(labels[2])
    return final_pred
```

Figure 8: Function used for voting

Finally we provide the part of the code which is used to implement the multi class classification using these helper functions in figure 9 and 10.

```
X_train, y_train = load_mnist('../fashionmnist/', kind='train')
X_test, y_test = load_mnist('../fashionmnist/', kind='t10k')


# THIS IS FOR CLASS 1 vs 2
X_train_binary, y_train_binary = create_train_data(X_train, y_train, [1,2])
X_test_binary, y_test_binary = create_test_data(X_test, y_test, [1,2])

test = X_test_binary
test_labels = y_test_binary

test=np.array(test)
test_labels = np.array(test_labels)

w12 = fit(X_train_binary, y_train_binary)


X_train_binary, y_train_binary = create_train_data(X_train, y_train, [2,3])
X_test_binary, y_test_binary = create_test_data(X_test, y_test, [2,3])
test = np.append(test, X_test_binary, axis=0)
test_labels = np.append(test_labels, y_test_binary, axis=0)

w23 = fit(X_train_binary, y_train_binary)


# THIS IS FOR CLASS 1 vs 3
X_train_binary, y_train_binary = create_train_data(X_train, y_train, [1,3])
X_test_binary, y_test_binary = create_test_data(X_test, y_test, [1,3])
test = np.append(test, X_test_binary, axis=0)
```

Figure 9: Code for multi class classification

```
X_train_binary, y_train_binary = create_train_data(X_train, y_train, [2,3])
X_test_binary, y_test_binary = create_test_data(X_test, y_test, [2,3])
test = np.append(test, X_test_binary, axis=0)
test_labels = np.append(test_labels, y_test_binary, axis=0)

w23 = fit(X_train_binary, y_train_binary)


# THIS IS FOR CLASS 1 vs 3
X_train_binary, y_train_binary = create_train_data(X_train, y_train, [1,3])
X_test_binary, y_test_binary = create_test_data(X_test, y_test, [1,3])
test = np.append(test, X_test_binary, axis=0)
test_labels = np.append(test_labels, y_test_binary, axis=0)

w13 = fit(X_train_binary, y_train_binary)



pred12 = predict_label(test, w12)
pred12 = predict_class(pred12, [1,2])


pred23 = predict_label(test,w23)
pred23 = predict_class(pred23, [2,3])

pred13 = predict_label(test, w13)
pred13 = predict_class(pred13, [1,3])

final_pred = voting(pred12, pred23, pred13, [1,2,3])

acc = calc_acc(final_pred, test_labels)
print ("Accuracy: " + str(acc))
```

Figure 10: Code for multi class classification

We report and analyze our results in the next section.

5 Experiments and Results

We test all our models on the test set provided in the Fashion-MNIST dataset. For the binary classifiers we use 2000 test images while for the multi-class classifier we use 3000 test images. We report the best accuracy we could achieve by tuning different hyper-parameters using different models as a summary.

Type of SVM	Accuracy
Vanilla SVM	0.97
SVM using RBF Kernel	0.95
SVM using Polynomial Kernel (degree 5)	0.91
Sklearn SVM	0.72
Multi Class SVM (Test data is different)	0.94

Table 1: Accuracy for the vanilla SVM at different iterations

As we can see from the above results, the vanilla SVM actually achieves the best accuracy among all. We did not do an extensive hyper parameter search for all of these models. Hence, some of these results might actually improve. We can also see that the multi class SVM achieves accuracy close to 94%. We analyze the results for the vanilla SVM. We use different number of iterations to compare the results at different stages for the vanilla SVM in Table 2.

Iteration	Accuracy
100	0.5
500	0.52
1000	0.57
5000	0.94
7000	0.97

Table 2: Accuracy for the vanilla SVM at different iterations

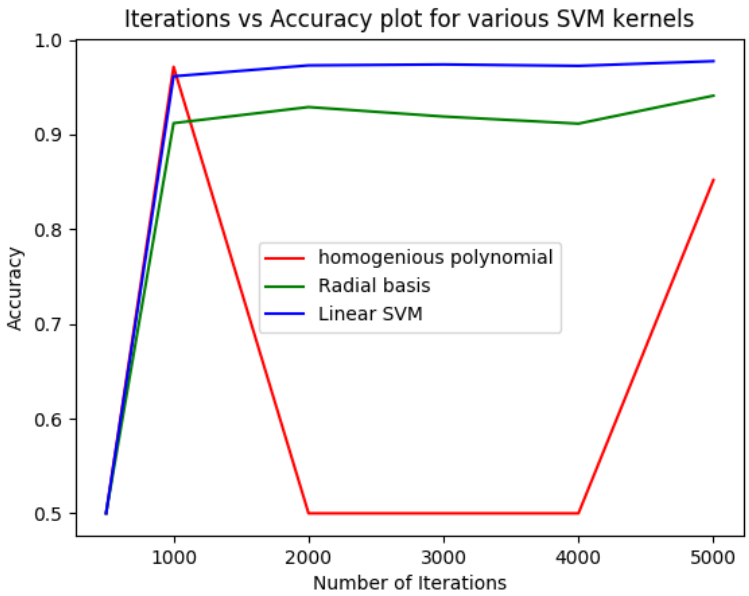

Figure 11: Iteration vs Accuracy graph for different SVMs

Figure 11 contains a comparison of iterations vs accuracy for different types of SVM. As we can see, we get a very high accuracy for all 3 types within 5000 iterations but the polynomial kernel overfits quickly after that, before regaining some accuracy later. Next we first analyze the homogeneous polynomial kernel first. We vary the degree of this kernel along with the iterations and check its accuracy. Figure 12 comparing this.

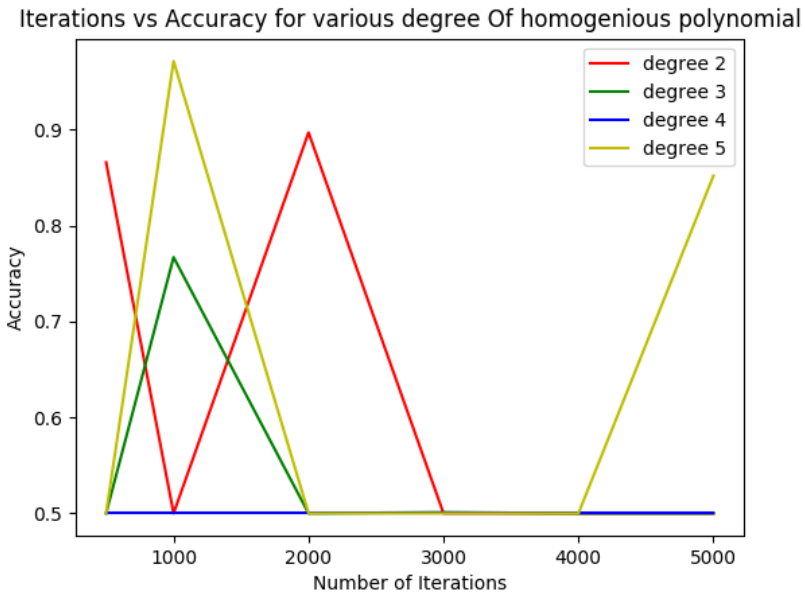

Figure 12: Iteration vs Accuracy graph for SVMs with polynomial kernel with different degrees

As we can see the SVM behaves pretty oddly and we cannot draw a definite inference from these graphs. As we can see, degree 5 seems to be the best performing model among these. Next we check the same statistics for the RBF kernel while varying gamma and iterations together. We show this in figure 13.

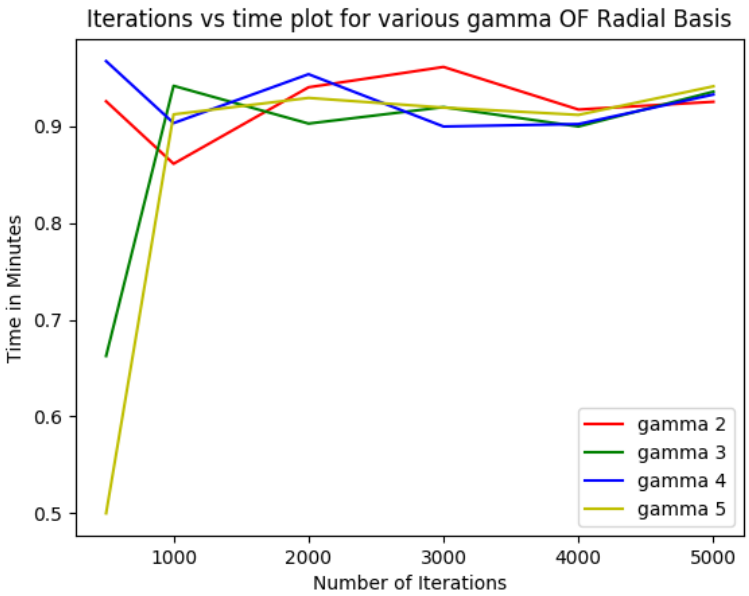

Figure 13: Iteration vs Accuracy graph for SVMs with RBF kernel with different gammas

As we can see from this figure, the accuracy converge to roughly the same spot. Gamma= 2 and gamma = 3 seems to give the best results. Gamma =4 actually converges in lesser iterations but then overfits. We also compare, the time taken by different models for giving the best achievable accuracy in table 3.

Type of SVM	**Time-taken (Approximately in Seconds)**
Vanilla version of SVM	45
SVM with polynomial kernel	4000
SVM with RBF kernel	7000
Sklearn's linear SVM	12000

Table 3: Accuracy for the vanilla SVM at different iterations

From the above table, as we can see, we get a huge time boost using pegasos algorithm. This is an essential advantage which this algorithm provides over the normal quadratic programming based SVMs. We attach screen cast videos for all

these types of SVMs along with this report. Our screen cast videos are present in the github repository given in the footnote at the first page.

6 Conclusion

In this project, we implemented the pegasos algorithm from scratch. We implemented the kernelized version of the same algorithm and tested on different kernels. We also extended the algorithm to a multi class setting. Finally we extensively analyzed and compared our results

. Screen cast video results:

`https://drive.google.com/drive/folders/1EDKXfjsm80JLrMUMEOeTZ5KVNdIfVd9Y?usp=sharing`

Codes are available in:

`https://github.com/sangeeth93/Pegasos_SVM`

References

[1] Shai Shalev-Shwartz, Yoram Singer and Nathan Srebro
"Pegasos: Primal Estimated sub-GrAdient SOlver for SVM," ICML '07 Proceedings of the 24th international conference on Machine learning Pages 807-814

[2] Han Xiao, Kashif Rasul, Roland Vollgraf *"Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms"*