Pattern recognition and machine learning

G.Sai Harshith

CS22B074

Assignment -3

under supervision of

Dr. Arun Raj Kumar

IIT Madras

Data Collection:

1. We have collect a data from online source of keegle website and It only consists of english words and some numbers.
2. The train data is given in the file “emails.csv” and we have divided them into train and test in the ratio of 4:1.
3. The emails and spam data are stored in the “har” and in 1st column is the text and spam status is in the 2nd column.
4. By, using the library of “from sklearn.feature_extraction.text import CountVectorizer” and using Countvector function this split the data points in the order of dictonary with frequencies in the array.

```
vectoriser = CountVectorizer(stop_words='english')
vectorised_data = vectoriser.fit_transform(har['text'])
vectorised_data = vectorised_data.toarray()
feat=vectoriser.get_feature_names_out()
with open('featurename.txt','w') as file:
    for f in feat:
        file.write(f + '\n')
Y= np.where(har['spam']==1, 1, 0)
```

5. Later, I split the data into X_train, X_test, Y_train, Y_test in the ratio of 4:1.
6. Now, we have to the “Laplace smoothing “ by append the 1’s and 0’s.

```
a_ones = np.ones_like(X_train[0])
X_train= list(X_train)
X_train.append(a_ones)
X_train.append(a_ones)
Y_train= list(Y_train)
Y_train.append(0)
Y_train.append(1)
```

Naive Bayes algorithm[Bernoulli's way]

1. I have used the np.where and updated the frequency as either 0 or 1 which indicates either the language is not present or it's present.
2. Now, we have to calculate the frequency of spam $\hat{p}$ with the variable phi.
   a. $\hat{p} = \sum_{i=1}^{n} y_i/n$
3. Now, probabilities p0 and p1 which shows the probabilities of non-spam and spam.
   a. $\hat{p_i^y} = \sum_{i=1}^{n} I(f_j^i = 1,\ y_i = y) / \sum_{i=1}^{n} I(y_i = y)$
4. I had done it in 2 ways
   a. Direct checking of predictions and non-predictions

```
X_test= np.array(X_test)
Y_test=np.array(Y_test)
pre = np.zeros_like(Y_test)
bin_mat = np.where(X_test>0,1, 0 )
h, w = bin_mat.shape
print(h, w)
print(p1)
acc=0
for i in range(h):
    pro_0=np.prod((p0**bin_mat[i])*((1-p0)**(1-bin_mat[i])))
    pro_1=np.prod((p1**bin_mat[i])*((1-p1)**(1-bin_mat[i])))
    if phi*pro_1>(1-phi)*pro_0:
        pre[i]=1
    else:
        pre[i]=0
temp = np.sum(np.abs(Y_test-pre))
print(temp)
print((1-(temp/Y_train.shape[0]))*100)
```

i. for every test, we have to calculate the p(y=1/x) indicated as pro_0 and p(y=0/x) indicated as pro_2 which is directly proportional phi* p(x/y=1) and (1-phi)*p(x/y=0) using the equations:

```
pro_0=np.prod((p0**bin_mat[i])*((1-p0)**(1-bin_mat
pro_1=np.prod((p1**bin_mat[i])*((1-p1)**(1-bin_mat
```

ii. then check which of the phi* p(x/y=1) and (1-phi)*p(x/y=0) is higher and predict 1 if the phi* p(x/y=1) is higher and 0 otherwise this can be done according to the following loop.

```
if phi*pro_1>(1-phi)*pro_0:
    pre[i]=1
else:
    pre[i]=0
```

iii. and we have to do the difference and find how many inaccurancies in the temp.

iv. Now, I printed the accuracy percentage.

v. I am getting 98.887 for direct checking may be with overflow

```
1146 36996
[0.07727273 0.13545455 0.02454545 ... 0.00090909 0.00090909 0.00272727]
51
98.88791975577846
```

b. Logarithmic:

```
X_test= np.array(X_test)
Y_test=np.array(Y_test)
pre = np.zeros_like(Y_test)
bin_mat = np.where(X_test>0,1, 0 )
h, w = bin_mat.shape
print(h, w)
acc=0
t2= math.log(phi/(1-phi))
t2+=np.sum(np.log((1-p1)/(1-p0)))
print(t2)
print(p0)
print(p1)
t1=np.log((p1*(1-p0))/((p0*(1-p1))), dtype='float64')
print(t1.shape)
for i in range(h):
    if (np.dot(t1, bin_mat[i])+t2)>=0:
        pre[i]=1

temp = np.sum(np.abs(Y_test-pre))
print(temp)
print((1-(temp/Y_train.shape[0]))*100)
```

i. for the same way, check with logarithmic formulae,

ii. $\sum_{i=1}^{d} f_i \; log(\frac{\hat{p}_i^1(1-\hat{p}_i^0)}{\hat{p}_i^0(1-\hat{p}_i^1)})$ + $\sum_{i=1}^{n} log(\frac{1-\hat{p}_i^1}{1-\hat{p}_i^0}) + log(\frac{\pi}{1-\pi})$ if the expression is greater than 0 predict 1 else 0.

iii. These are done w and x and b as constant $w^T x_i + b >= 0$. w is calculated by

```
t1=np.log((p1*(1-p0))/((p0*(1-p1))), dtype='float64')
```

and b can be calculated by

```
t2= math.log(phi/(1-phi))
t2+=np.sum(np.log((1-p1)/(1-p0)))
```

iv. Now, the difference of the predictions are calculated

v. the logarithmic has the accuracy of 99.41 for my logarithm predictions.

```
1146 36996
-10.884282661894884
[0.17785427 0.03614458 0.00143431 ... 0.00028686 0.00086059 0.00028686]
[0.07727273 0.13545455 0.02454545 ... 0.00090909 0.00090909 0.00272727]
(36996,)
27
99.41125163541213
```

5. I am getting 98.887 for direct checking may be with overflow or truncated but the logarithmic has the accuracy of 99.41 for my logarithm predictions.

Logistic Regression:

1. It is one of the discriminative model for classification.
2. The probability given P(y=1/x) = $\frac{1}{1+e^{-w^T x}}$, then after maximizing the log likelihood function we get to maximising max over w on functions $\sum_{i=1}^{n}[(1 - y_i)(-w^T x_i) - log(1 + e^{-w^T xi})]$

a. I had started from w=0 and move towards the direction that increases the function and the gradient = $\sum_{i=1}^{n} x_i(y_i - \frac{1}{1+e^{-w^T xi}})$ and then move towards that direction.

b. We have to go in that direction to maximise upto certain extent which is step size.

c. We have to stop if we cannot reduce any errors or the error's increase or error=0

d. But, there is computation constraint and there might also be some noise so I took max_iterations = 100.

3.
```python
#logistic regression
#gradient accent algorithm:
def sigmoid(z):
    return 1/(1+np.exp(-z))
def logistic_regression(X, y, stepsize,axisx,axisy,  iterations):
    n, d = X.shape
    w= np.zeros(d)
    for i in range(iterations):
        axisx.append(i+1)
        predictions = sigmoid(np.dot(X, w))  #X has the shape nxd and y as nx1
        gradient = np.dot(X.T,(y-predictions))
        w+= stepsize*gradient
        expec= np.matmul(w.T, X_test.T)
        expec= np.where(expec>0.5, 1, 0)
        temp = np.sum(np.abs(Y_test-expec))
        axisy.append(temp)
    return w
```

The provided Python code is implementing a logistic regression model using the gradient ascent optimisation algorithm.

Here's how it works:

1. The `sigmoid` function implements the sigmoid function, which maps any real-valued number to the (0, 1) interval, making it useful for converting an arbitrary-valued score to a probability.
2. In the `logistic_regression` function, it first initialises the weight vector w to zeros.
3. Then, for the specified number of iterations, it performs the following operations:
    - Computes the predicted probabilities of the positive class by taking the dot product of the feature matrix `X` and the weight vector `w`, and feeding this

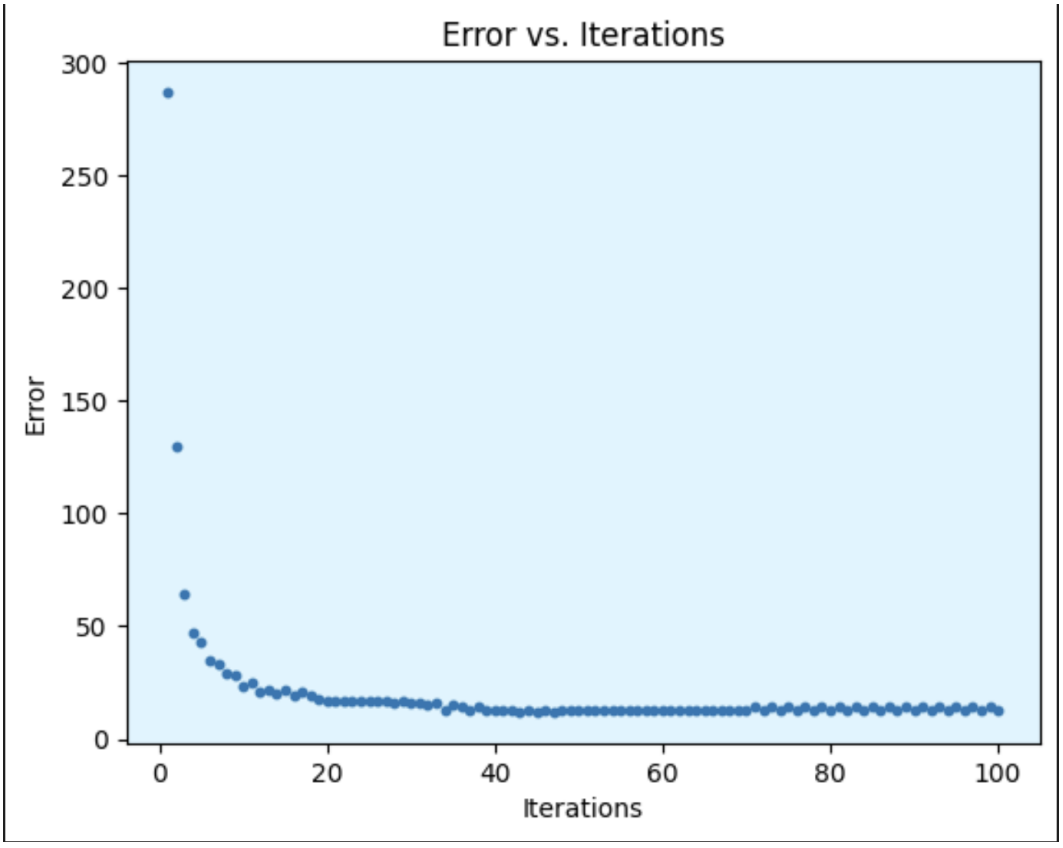
through the sigmoid function.

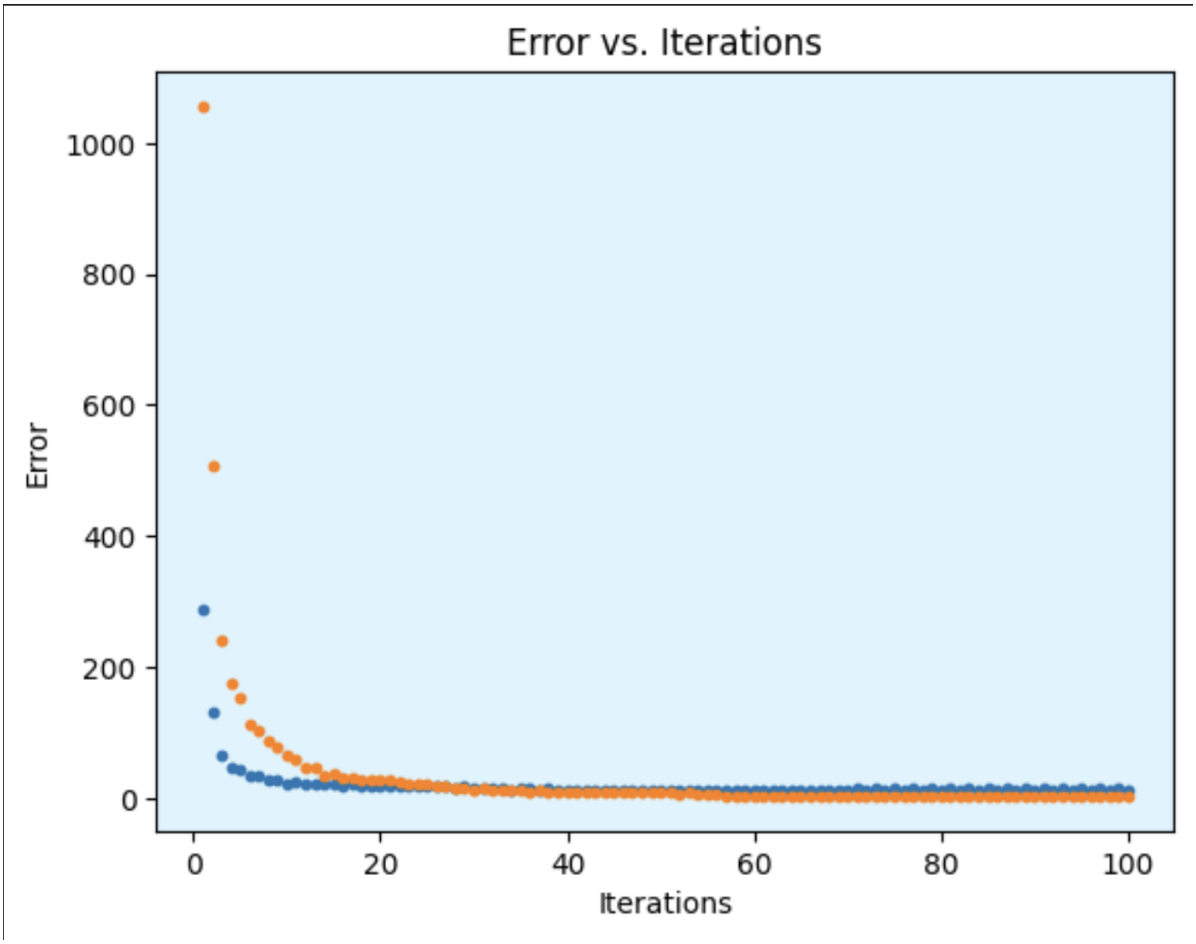
- Computes the gradient of the log-likelihood with respect to the weights. This is done by taking the dot product of the transpose of `X` and the difference between the true class labels `y` and the predictions.
- Updates the weight vector `w` by taking a step in the direction of the gradient. The size of the step is determined by the learning rate `stepsize`.

4. After the specified number of iterations, it returns the weight vector `w` which can be used to make predictions on new data.

   The percentage accuracy for the test data is 99.7165

```
(4586,)
/var/folders/rl/3m72t40d3_18r9rdcry1mc180000gn/T/ipykernel_7867/4219433561.py:4: RuntimeWarning: overflow encountered in exp
  return 1/(1+np.exp(-z))
99.71652856519843
```






```python
#logistic regression
#gradient accent algorithm:
def sigmoid(z):
    return 1/(1+np.exp(-z))
def logistic_regression(X, y, stepsize,axisx,axisy_train,axisy_test,  iterations):
    n, d = X.shape
    w= np.zeros(d)
    for i in range(iterations):
        axisx.append(i+1)
        predictions = sigmoid(np.dot(X, w))  #X has the shape nxd and y as nx1
        gradient = np.dot(X.T,(y-predictions))
        w+= stepsize*gradient
        expec= np.matmul(w.T, X_test.T)
        expec= np.where(expec>0.5, 1, 0)
        temp = np.sum(np.abs(Y_test-expec))
        axisy_test.append(temp)
        expec= np.matmul(w, X_train.T)
        expec= np.where(expec>0.5, 1, 0)
        temp = np.sum(np.abs(Y_train-expec))
        axisy_train.append(temp)
    return w
```

This is the latest update to incorporate the train error. If I am trying to take the errors for train and test. Train has many no. of points so because of that it

takes lot of time, This algorithm takes less than 20s normally, but to draw the graph it became 1m 40s aproximately.

The errors vs iterations on test.

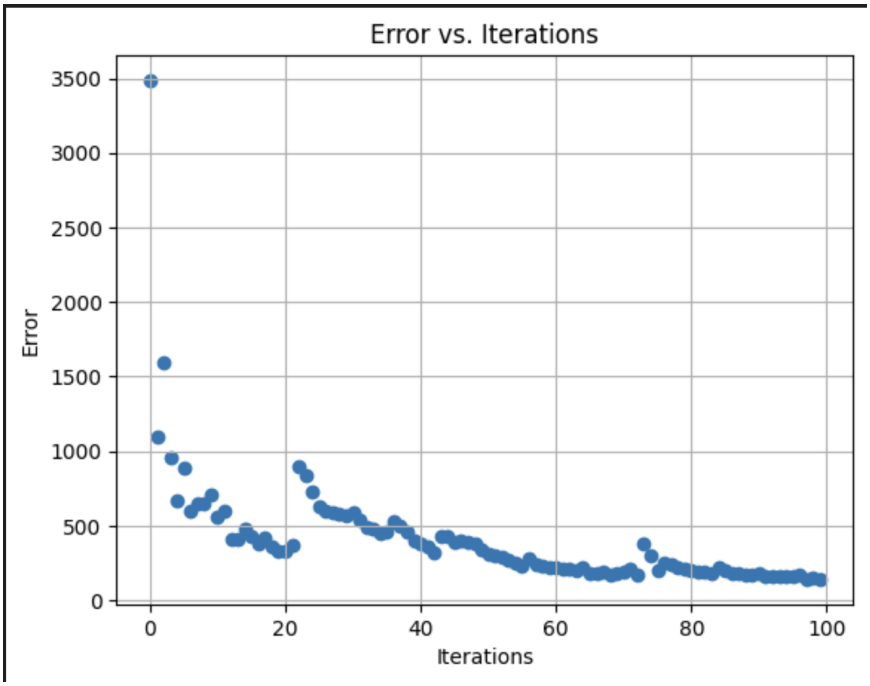
The blue is for test and orange is for train.

Perceptron Algorithm:

1. I wrote 2 algorithms by generally without any linear seperator say gamma=0.
2. I will go from i=0 to i=n where n is no. of points in the train data set.
3. I will calculate the first i starting from i=0 to n which has the false indicator and update $w_{t+1} = w_t + x_i y_i$ and then start over again if there is an updates and the update happens when there are errors.
4.

```
def perceptron(X_train, y_train, max_iter, axisx, axisy):
    w0= np.zeros(X_train.shape[1])
    y_pre= np.zeros(X_train.shape[1])
    error=1
    for i in range(max_iter):
        axisx.append(i)
        if error!=0:
            y_pre= np.matmul(X_train, w0)
            y_pre = np.where(y_pre>=0, 1, -1 )
            y_pre = y_train- y_pre
            ind = np.argmax(y_pre!=0)
            w0 +=y_train[ind]* X_train[ind].T
            error = np.sum(np.abs(y_pre))
            axisy.append(error/2)
        else:
            break
    return w0
```

1. I will start from w=[0, 0, 0, ..., n 0's] and then do the mat mul with X_train, w0 and named the matrix as y_pre.
2. y_pre[i] is effectively $w^T x_i$ and all we want to do is we have to check the sign of this and if the sign≥0, then we have to make y_pre=1 if not -1 .
3. Then we can find the mismatch errors using y_pre = y_train- y_pre if they are not equal to 0 then there is error.
4. The possible errors or 0, -2, +2 the first element we get+2/-2 is the first error, using the we update w.
5. No. of error = np.sum(np.abs(y_pre))/2 since each time there is error it shows +2/-2 but we get only error and I plotted the graph between errors vs iterations.
6. In the above code axisx and axisy lists are used to store iteration vs error.



7. The no. of errors gradually decreases and almost remains same.
8. The percentage accuracy is 98.125.

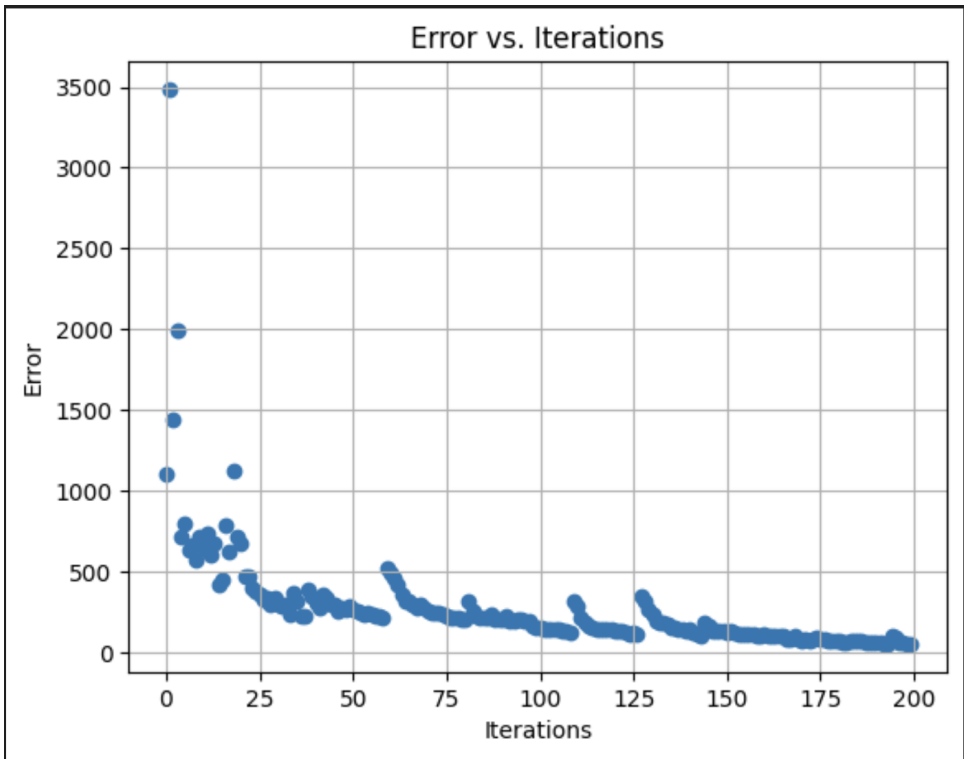
```
43.0
98.12472743131269
```

Perceptron algorithm with gamma linear separation:

1. Here, there is a problem with the implementation we don't what the margin is some times if we put the gamma as too large all the points are inside the two lines and we cannot predict them at all.
2. So, to avoid some things I took only one inequality and treating it into bias.

```python
#perceptron with margin:
def perceptron_with_margin(X_train, y_train, max_iter,gamma, axisx, axisy):
    w0= np.zeros(X_train.shape[1])
    y_pre= np.zeros(X_train.shape[1])
    error=1
    for i in range(max_iter):
        axisx.append(i)
        if error!=0:
            y_pre= np.matmul(X_train, w0)
            y_pre = np.where(y_pre>=gamma, 1, -1 )
            y_pre = y_train- y_pre
            ind = np.argmax(y_pre!=0)
            w0 +=y_train[ind]* X_train[ind].T
            error = np.sum(np.abs(y_pre))
            axisy.append(error/2)
        else:
            break
    return w0
```

3. I took the gamma = 1



4. The percentage accuracy is 98.561 if I am using gamma=1.

```
33.0
98.5608373310074
```

Support Vector Machines:

1. I have take the inbuilt function instead of building it from the scratch.
2. The library used are the following:

```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

3.
```
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Assuming you have already split your data into training and testing sets
# X_train and y_train are your training data and labels
# X_test and y_test are your testing data and labels

# Create an SVM classifier
svm_classifier = SVC(kernel='linear')  # You can specify different kernels like 'linear', 'poly', 'rbf', etc.

# Train the SVM classifier
svm_classifier.fit(X_train, Y_train)

# Make predictions on the testing data
y_pred = svm_classifier.predict(X_test)

# Calculate accuracy
accuracy = accuracy_score(Y_test, y_pred)

print("test_Accuracy using svm:", accuracy*100)
```

✓ 38.6s

```
test_Accuracy using svm: 98.95287958115183
```

4. The test accuracy is around 98.95.
5. I have to fit X_train, Y_train into the SVC and I didn't kernelized and used the linear.
6. It helps us to find the direction of w that increases highest gamma margin.

Support Vector Machine (SVM) is a supervised machine learning algorithm that can be used for both classification and regression challenges. In the case of linearly separable data in two dimensions, SVM would seek to find the hyperplane that best separates the classes. The best hyperplane is defined as the one that maximises the margin between the two classes.

The derivation of SVM starts by setting up the problem of finding the maximum margin hyperplane.

1. For a hyperplane defined as $w^T x + b = 0$, where $\w is the normal vector to the hyperplane and b is the bias term, the distance from a point $x_i$to the hyperplane is given by $\frac{y_i(w^T x_i+b)}{||w||}$where y_i is the true label of the point x_i.
2. The margin of a hyperplane with respect to the data points is the minimum distance from the hyperplane to any of the data points. Therefore, to maximise the margin, we need to maximise the minimum value of $y_i(w^T x_i + b)/||w||$.
3. By convention, we choose the hyperplane such that the distance from the closest positive point $y_i = 1$ and the closest negative point (y_i = -1) to the

hyperplane is 1. Therefore, the problem of maximising $y_i(w^T x_i + b)/||w||$ becomes a problem of minimising $||w||^2/2$ the constraint that for each point x_i in the data set, $y_i(w^T x_i + b) \geq 1$.

4. This is a quadratic programming problem, which can be solved using Lagrange multipliers. The Lagrangian is defined as $L = ||w||^2/2 - \sum_i^N \alpha_i[y_i(w^T x_i + b) - 1]$ where α_i are the Lagrange multipliers.
5. We then take derivatives of L with respect to w and b and set them equal to zero, and substitute these back into the Lagrangian to get the dual problem. The dual problem is to maximise $L = \sum_{i=1}^{N} \alpha_i - 0.5(XY\alpha)^T(XY\alpha)$ under the constraints that $\alpha_i \geq 0$ and $\sum_i^N \alpha_i y_i = 0$, .
6. Solving the dual problem gives the α_i's and the optimal hyperplane can be found by calculating $w = \sum_i^N \alpha_i y_i x_i$ = $XY\alpha$ and $b = y_k - w^T x_k$ for any k such that $\alpha_k > 0$.
7. There are few complete slackness cases that should be taken care off and this is how the algorithm is built.

I got to know from google, inbuilt function incorporates bias.

8. The accuracy of this is around 98.95.

This analysis is done in “q1_test.ipynb”.

FINAL PREDICTOR:

```
#testing of Bernoulli's Naive Bayes algorithm!!
test_folder= 'test1'
test_emails = []
num_emails = len(os.listdir(test_folder))
for i in range(num_emails):
    filename = f'email{i}.txt'
    with open(os.path.join(test_folder, filename), "r") as file:
        test_emails.append(file.read())
vectoriser1= vectoriser.transform(test_emails)
vectoriser1 = vectoriser1.toarray()
pre = np.zeros(vectoriser1.shape[0], dtype='int')
bin_mat= np.where(vectoriser1>0,1, 0 )
for i in range(vectoriser1.shape[0]):
    if np.dot(t1_nb, bin_mat[i])+t2_nb>=0:
        pre[i]=1
print(pre)
pre_nb= pre
print(pre_nb)
```

```
test_folder= 'test'
test_emails = []
num_emails = len(os.listdir(test_folder))
for i in range(num_emails):
    filename = f'email{i}.txt'
    with open(os.path.join(test_folder, filename), "r") as file
        test_emails.append(file.read())
vectoriser1= vectoriser.transform(test_emails)
vectoriser1 = vectoriser1.toarray()
```

The code above is to go through the test_folder and do it to vectorise.

```
weights_logreg = logistic_regression(X_train, Y_train, stepsize, iterations)
expec= np.matmul(weights_logreg.T, vectoriser1.T)
expec= np.where(expec>0.5, 1, 0)
print(expec)
```

expec is variable where I stored the prediction of logistic_regression.

```
# Train the SVM classifier
svm_classifier.fit(X_train, Y_train)

# Make predictions on the testing data
y_pred = svm_classifier.predict(vectoriser1)
print(y_pred)
```

svm_classifier predicts the spam and stored it into “y_pred”.

FINAL MODEL:

1. I had made the predictions from the 3 models except perceptron since SVM is actually the upgrade of perceptron.
2. I predicted the spam as 1 if atleast two of the models predicted is correct.
3. Mixing models for all the classifiers:
4. Taking the 3 models into account and analysing the 3 and returning spam if we have more than 3 models saying them as spam
5. expec is from logarithmic regression, pre is from Bernoulli's Naive Bayes and y_pred is from SVC.

Folder to place file:

email{n}.txt has to be given in the folder named ‘file’.

Where to look for the final answers:

Final predictions are written into the file that is created **predicted.txt**

“Feature.txt” is the file that displays the words arranged by the order count vectoriser.