COL 774 – Machine Learning – Assignment 1 – AIZ238140 – T Karthikeyan

1.

Implementation of gradient descent to learn the relationship between $x^{(i)}$, $y^{(i)}$.

We decide to approximate y as a linear function of x

$$h_\theta(x) = \theta_0 + \theta_1 x_1$$

Error metric used for this question

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(y^{(i)} - h_\theta(x^{(i)})\right)^2$$

Gradient of $J(\theta)$

$$\begin{aligned}
\frac{\partial}{\partial \theta_j} J(\theta) &= \frac{\partial}{\partial \theta_j} \frac{1}{2} (h_\theta(x) - y)^2 \\
&= 2 \cdot \frac{1}{2} (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} (h_\theta(x) - y) \\
&= (h_\theta(x) - y) \cdot \frac{\partial}{\partial \theta_j} \left(\sum_{i=0}^{n} \theta_i x_i - y \right) \\
&= (h_\theta(x) - y)\, x_j
\end{aligned}$$

Theta is updated as follows:

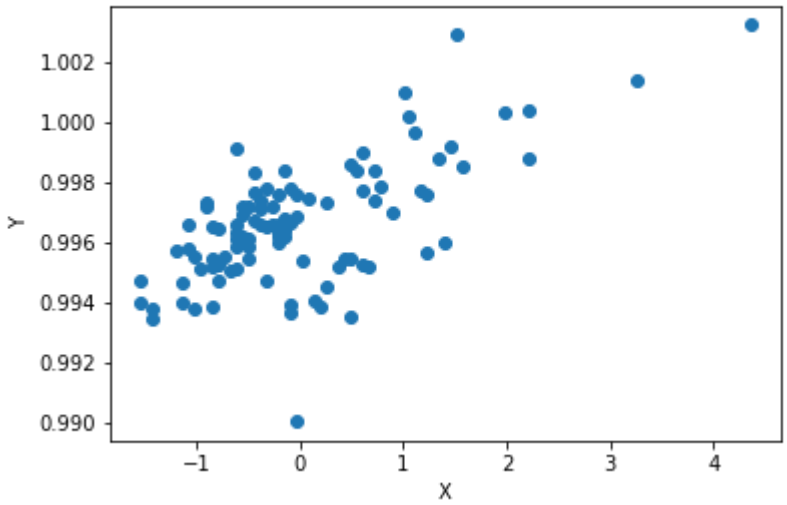
$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Where α is learning rate

Normalization of x is done according to Z-score technique

$$x := \frac{(x - \mu)}{\sigma}$$

After normalizing the values of $x^{(i)}$, I plotted the values of $x^{(i)}$ and $y^{(i)}$, to get an idea



Initialized the theta to zero and added an extra column to incorporate the intercept term

(a)

Learning rate: 0.002

Stopping criteria:

$$\| \nabla_\theta J(\theta) \|_2 \leq \phi$$

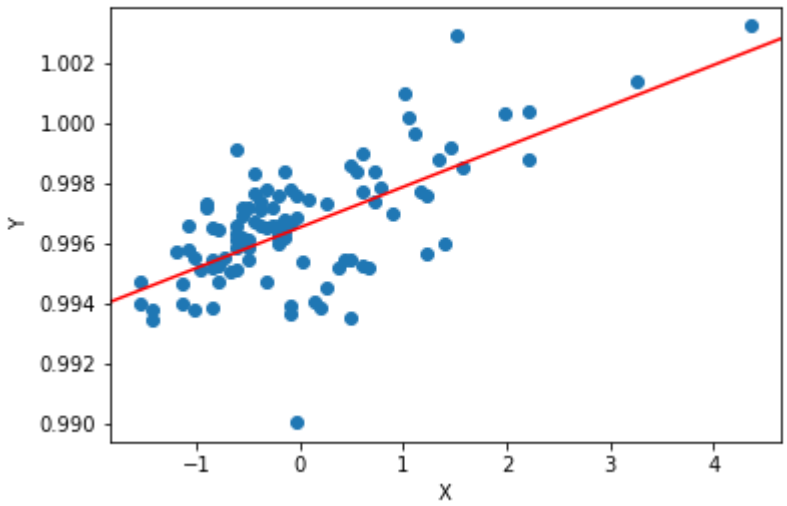
Where is small and positive

Here ϕ = 0.0001

Final set of parameters: $[0.99652013, 0.0013468]$

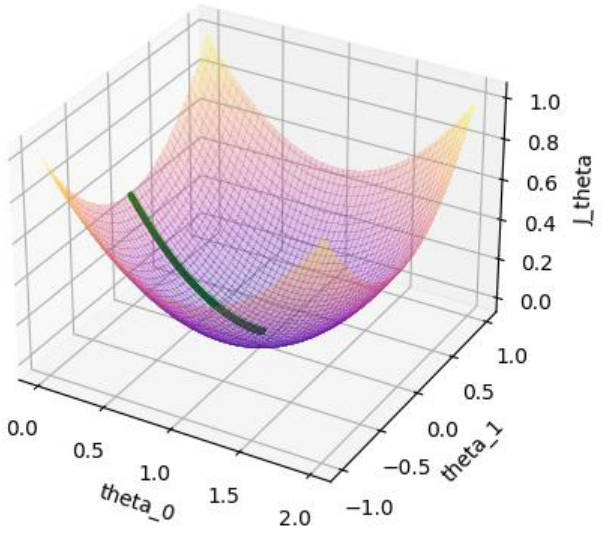
(b)

Plotting the hypothesis line with the estimated parameters.



(c)

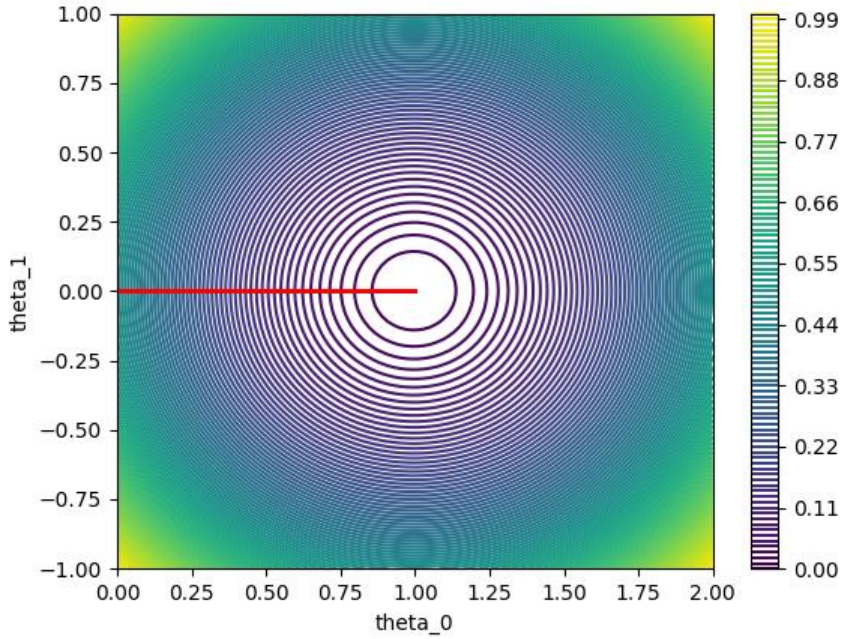
3D Mesh showing the $J(\theta)$



Please find the gif in the submitted zip file

(d)

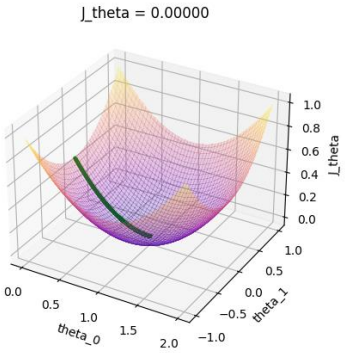
Contour plot

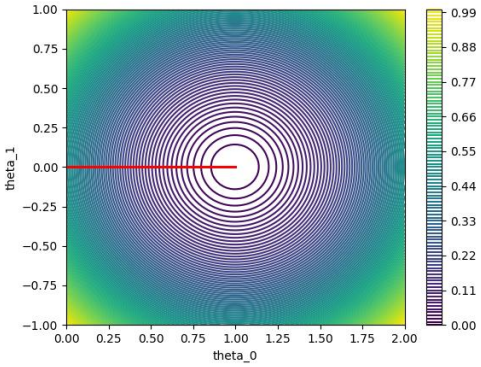


Please find the gif in the submitted zip file

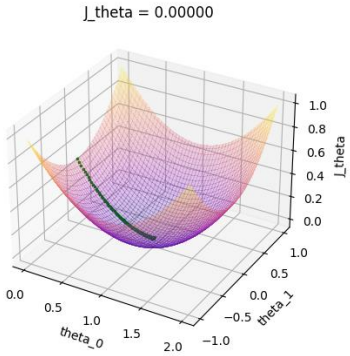
(e)

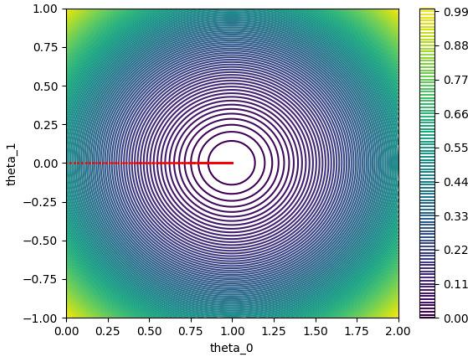
When learning rate was 0.001



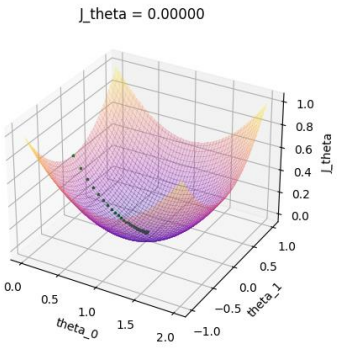


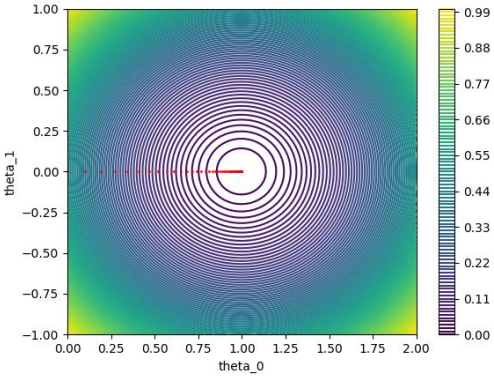
When learning rate was 0.025





When learning rate was 0.1





Observations:

Big steps are taken for large values of learning rate. In the above example, it follows a straight line to reach the minimum loss value but in general, it might take a slight zigzag path to reach minimum loss value

2.

Stochastic Gradient descent (SGD) for various batch sizes

(a)

Sampled 1 million data points and stored it in a text file

```
data = {'X_1':np.random.normal(3, 2, 1000000),
        'X_2':np.random.normal(-1, 2, 1000000),
       'error':np.random.normal(0, np.sqrt(2), 1000000)}

train_df = pd.DataFrame(data)
train_df["Y"] = 3 + (1*train_df["X_1"]) + (2*train_df["X_2"]) + train_df["error"]
train_df.to_csv('q2train.csv', index=False)
```

(b)

Since we are dealing with various batch size, we cannot use the convergence criteria used in batch gradient descent, hence we use the below convergence criteria

Loss used in gradient descent

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(y^{(i)} - h_\theta(x^{(i)})\right)^2$$

We calculate the average of loss over a window size, then in the next iteration we compute the average of loss over a window size.

$$\left(J(\theta)\right)_{avg} = \frac{1}{winsize}\sum^{winsize} L(\theta)$$

We compute

$$\| \left(J(\theta)\right)_{curr\ avg} - \left(J(\theta)\right)_{prev\ avg} \| < \phi$$

If the above condition holds for γ iterations, then the SGD algorithm converges

Hyperparameters used for various batch sizes are shown below

Batch size	winsize	ϕ	γ
1	10	0.005	3
100	10	0.001	3
10000	10	0.0002	3
1000000	10	0.000001	3

Keeping the learning rate η to be 0.001, I got the following values of theta for batch size r = {1,100,10000,1000000}

Batch size	Theta

1	[[3.01976638], [0.99488842], [2.03425438]]
100	[[2.99283814], [1.00084866], [2.00248258]]
10000	[[2.80072196], [1.0437344], [1.98525517]]
1000000	[[2.88963331], [1.02413802], [1.99184302]]

(c)

No, they do not converge to the same parameter values.

	Parameters (theta)	Number of iterations	Number of epochs	Time taken (milli sec)	MSE on test data
Original hypothesis	[[3.0], [1.0], [2.0]]				1.965893
r=1	[[3.01976638], [0.99488842], [2.03425438]]	78831	<1	2190.456	2.078397
r=100	[[2.99283814], [1.00084866], [2.00248258]]	23107	2.3107	723.384	1.965963
r=10000	[[2.80072196], [1.0437344], [1.98525517]]	9665	96.65	7057.081	2.198215
r=1000000	[[2.88963331], [1.02413802], [1.99184302]]	11802	11802	916911.383	2.037023

Observations

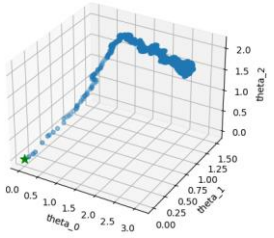
Batch size 100 performs at par with original hypothesis on the test data

Since the theta update happens after going through r items, hence time taken is directly proportional to batch size

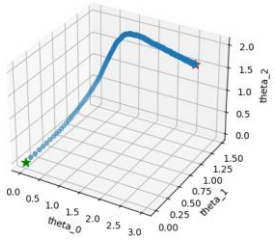
(d)

Plotting the value of theta over the iterations on a 3D plot

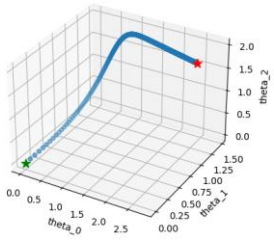
When batch size was 1



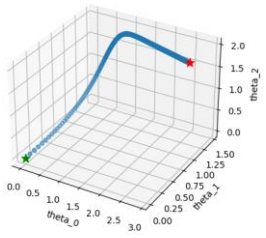
When batch size was 100



When batch size was 10000



When batch size was 1000000



The shape of movement is zigzag for lower batch sizes, and it gets refined as we increase the batch size. This is intuitive in the sense that lower batch size is having a local view and as we keep on increasing it, we have a global view.

3.

(a)

Logistic regression using Newton's method

Log likelihood function for logistic regression

$$L(\theta) = \sum_{i=1}^{m} y^{(i)} \log h_\theta\left(x^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - h_\theta\left(x^{(i)}\right)\right)$$

Maximising $L(\theta)$ is equivalent to minimizing $(-L(\theta))$

After deducing the gradient of above loss function,

$$\frac{\partial L(\theta)}{\partial \theta_j} = (h_\theta(x) - y)x_j$$

$$\nabla_\theta L(\theta) = \left[\frac{\partial L(\theta)}{\partial \theta_0} \frac{\partial L(\theta)}{\partial \theta_1} \frac{\partial L(\theta)}{\partial \theta_2}\right].T$$

Then we deduce the Hessian matrix for the above loss function

$$\frac{\partial^2 L(\theta)}{\partial \theta_i \partial \theta_j} = h_\theta(x)\left(1 - h_\theta(x)\right)x_i x_j$$

$$H_{ij} = \frac{\partial^2 L(\theta)}{\partial \theta_i \partial \theta_j}$$

Theta is initialized to zero and then updated in further iterations according to

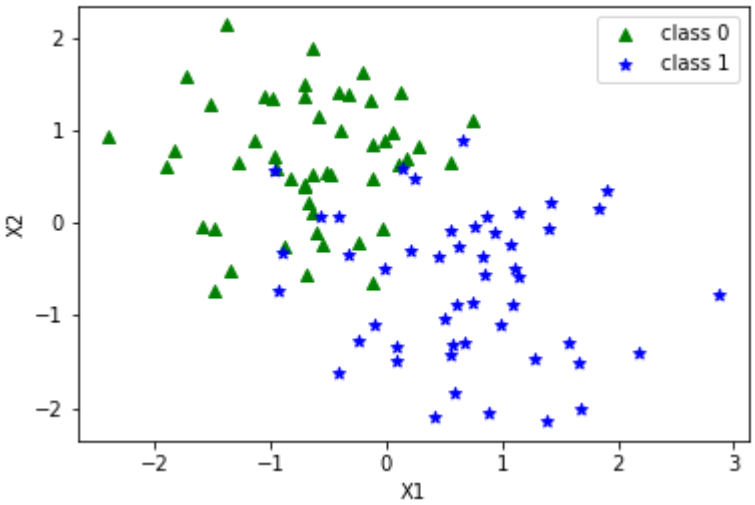
$$\theta := \theta - H^{-1}\nabla_\theta L(\theta)$$

Final coefficients of θ from our fit

[[0.40125316]
 [2.60158832]
 [-2.7393195]]

(b)

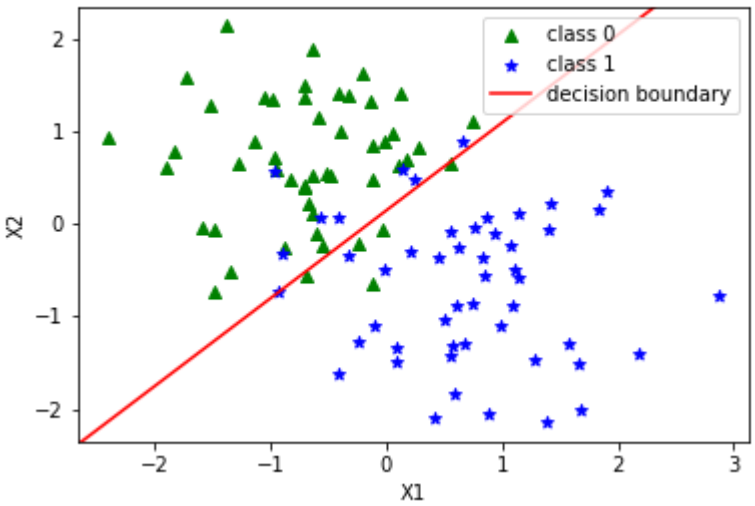
Plotting the training data



Plotting the decision boundary with the help of the theta values obtained

Slope of the decision boundary = $\frac{-\theta_1}{\theta_2}$

Intercept of decision boundary = $\frac{-\theta_0}{\theta_2}$



4.

Gaussian Discriminant Analysis (GDA)

We normalize the value of $x_{(i)}$

Since $y_{(i)}$ is {Alaska, Canada}, I did label encoding to {0,1}

(a)

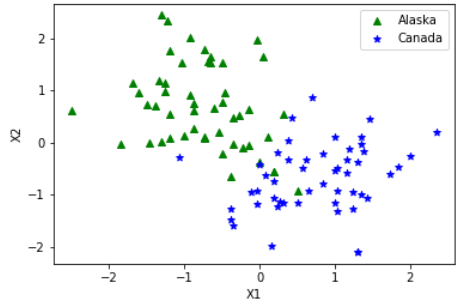
Implemented the GDA using closed form equations.

Assuming the same co-variance matrix

μ_0	[[-0.75150837], [0.68166023]]
μ_1	[[0.75150837], [-0.68166023]]
Σ	[[0.42523517, -0.02224756], [-0.02224756, 0.52533933]]

(b)

Plotted the training data



(c)

Equation of the boundary separating the two regions

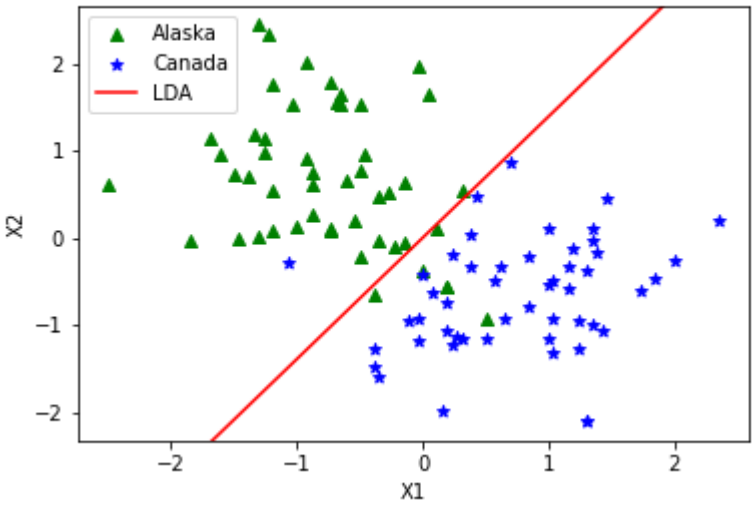
$$Ax + b = 0$$

$$A = \mu_1^T \Sigma^{-1} - \mu_0^T \Sigma^{-1}$$

$$b = \mu_0^T \Sigma^{-1}\mu_0 - \mu_1^T \Sigma^{-1}\mu_1 + \log\left(\frac{\phi}{1-\phi}\right)$$

$$\phi = prob\left(y^{(i)} = 1\right)$$

$$1 - \phi = prob\left(y^{(i)} = 0\right)$$



(d)

Implementing GDA when co-variance matrices are different

μ_0	[[-0.75150837], [0.68166023]]
μ_1	[[0.75150837], [-0.68166023]]
Σ_0	[[0.37777389, -0.15331651], [-0.15331651, 0.6412598]]
Σ_1	[[0.47269646, 0.10882139], [0.10882139, 0.40941887]]

(e)

Equation of the boundary separating the two regions

$$x^T Ax + Bx + c = 0$$

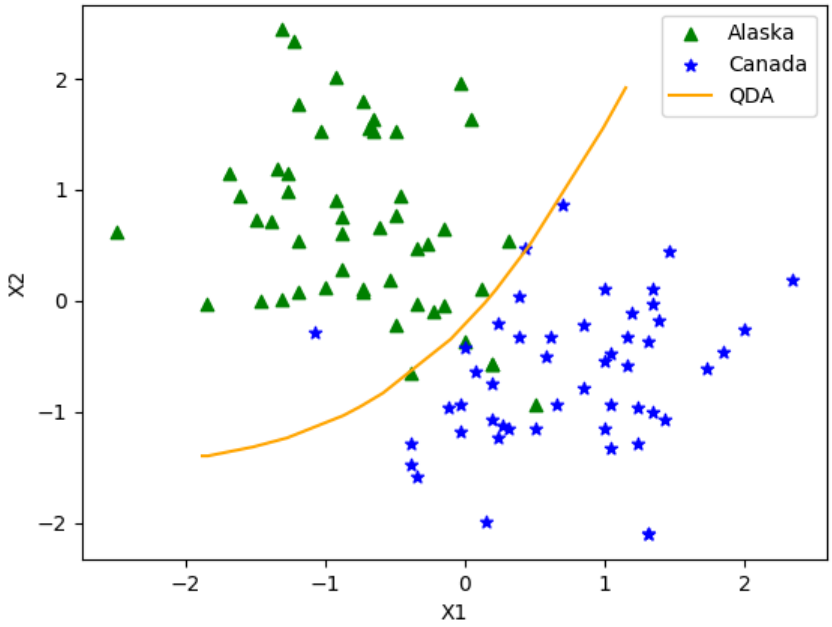
$$A = \Sigma_0^{-1} - \Sigma_1^{-1}$$

$$B = 2(\mu_1^T \Sigma_1^{-1} - \mu_0^T \Sigma_0^{-1})$$

$$c = \mu_0^T \Sigma_0^{-1} \mu_0 - \mu_1^T \Sigma_1^{-1} \mu_1 + \log \frac{|\Sigma_0|}{|\Sigma_1|} + \log\left(\frac{\phi}{1-\phi}\right)$$

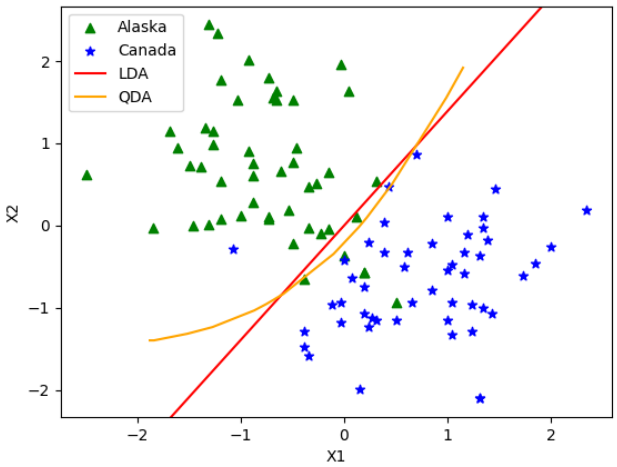
$$\phi = prob\left(y^{(i)} = 1\right)$$

$$1 - \phi = prob\left(y^{(i)} = 0\right)$$



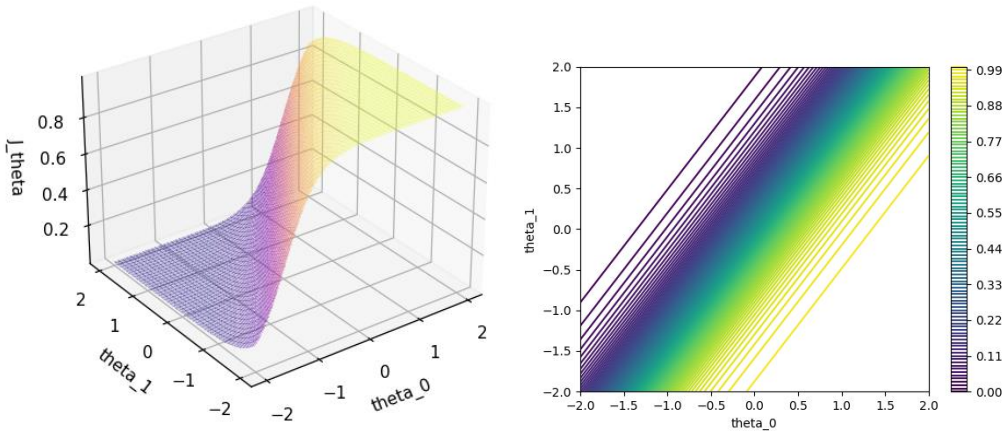
(f)

Combined plot

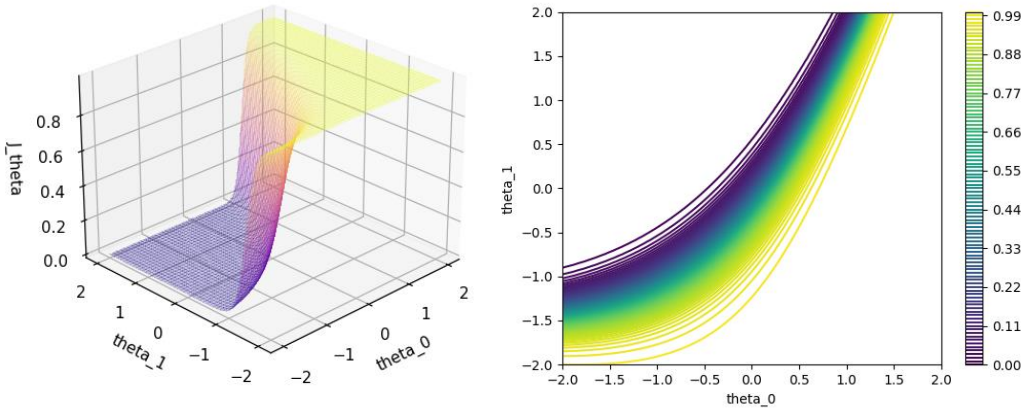


To analyse the linear as well as quadratic boundaries, I plotted the 3D mesh plot and contour plot

When both the co-variance matrices are same



When both the co-variance matrices are different



Observations

QDA has a steeper drop than LDA

It can be observed in the combined plot that QDA has lesser misclassifications of Alaska points

COL 774 Machine Learning Assignment 2

T Karthikeyan (2023AIZ8140)

Q1.

Naïve Bayes Multiclass classification

Tweets in general could have a lot of noise, so I cleaned it up using the following code

```
#Returns a list of words given a string
    @staticmethod
    def initial_preprocess(text, use_stem_remove_stopwords):
        # Remove "\r\n" characters
        temp = text.replace("\r", " ").replace("\n", " ")
        #remove userhandles
        temp = NB_da.remove_pattern(temp, "@\w+")
        #remove hashtags
        temp = NB_da.remove_pattern(temp, "#\w+")
        #remove urls
        temp = re.sub(r'http\S+', ' ', temp.strip())
        #replace diacritic texts
        temp = unidecode(temp)
        # Remove special characters (except alphanumeric characters and spaces)
        temp = re.sub(r'[^\w\s]', ' ', temp.strip())
        #remove full stops and comma
        temp = temp.replace('.', ' ').replace(',', ' ')
        # Remove multiple spaces
        temp = re.sub(r'\s+', ' ', temp.strip())
        #Lowercase a string
        temp = temp.lower()

        # Tokenize
        words = temp.lower().split()
        #remove integer strings
        words = [w for w in words if not w.isdigit()]

        if use_stem_remove_stopwords:
            #Remove stopwords
            english_stopwords = stopwords.words('english')
            words = [w for w in words if w not in english_stopwords]

            #Stemming
            ps = PorterStemmer()
            words = [ps.stem(w) for w in words]

        return words
```

We have M tweets in training examples with labels as [“Negative”, “Neutral”, “Positive”]

For simplicity in computation, we map them to an integer

```
#String to Class
    str2num = {
        'Negative': 0,
        'Neutral': 1,
        'Positive': 2
    }
```

We compute prior of a class using

$\phi_k = \frac{\sum_{i=0}^{m} 1\{y^{(i)}=k\}}{m}$ where k belongs to {0,1,2}

We store all the unique words in the training data and store it in vocabulary

For every word w in the vocabulary, we compute the following probabilities,

$$\phi_{w\,(y=0)} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\left\{x_j^{(i)} = w \wedge y^{(i)} = 0\right\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 0\} n_i + |V|}$$

$$\phi_{w\,(y=1)} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\left\{x_j^{(i)} = w \wedge y^{(i)} = 1\right\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 1\} n_i + |V|}$$

$$\phi_{w\,(y=1)} = \frac{\sum_{i=1}^{m}\sum_{j=1}^{n_i} 1\left\{x_j^{(i)} = w \wedge y^{(i)} = 2\right\} + 1}{\sum_{i=1}^{m} 1\{y^{(i)} = 2\} n_i + |V|}$$

where n_i is number of words in that tweet

Adding 1 in the numerator and |V| in the denominator is to handle words which are not seen in the training data [Laplace correction]

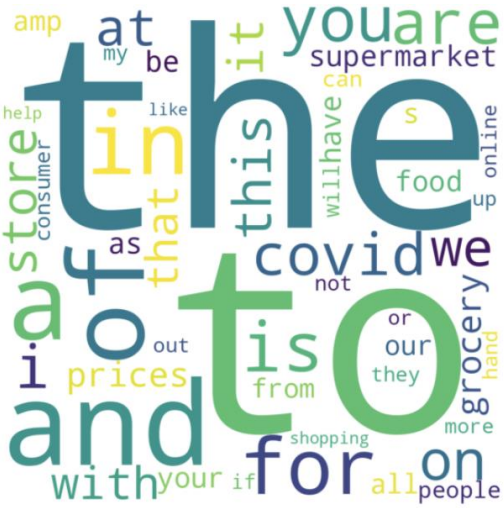
(a)

Naïve Bayes without any stemming or stop words removal

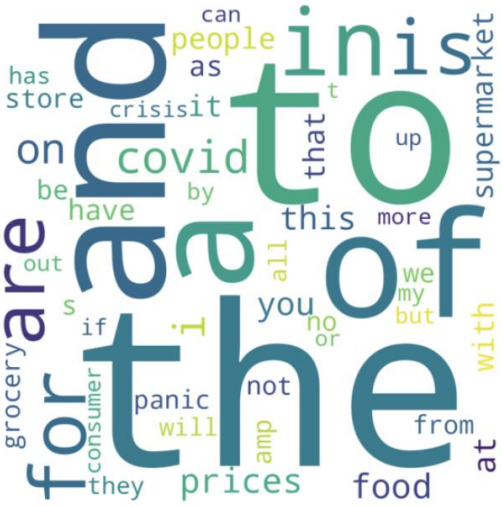
Training accuracy: 59.53676315233467

Test accuracy: 66.0795627087762

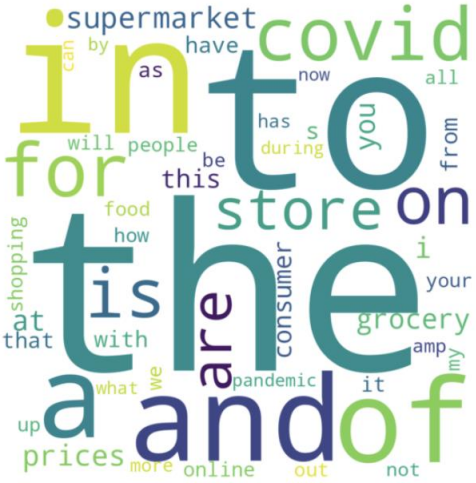
Word cloud for positive tokens



Word cloud for negative tokens



Word cloud for neutral tokens



It could be observed from word cloud that most frequent occurring words are {the, and, to, in, ...} which are common across all the classes.

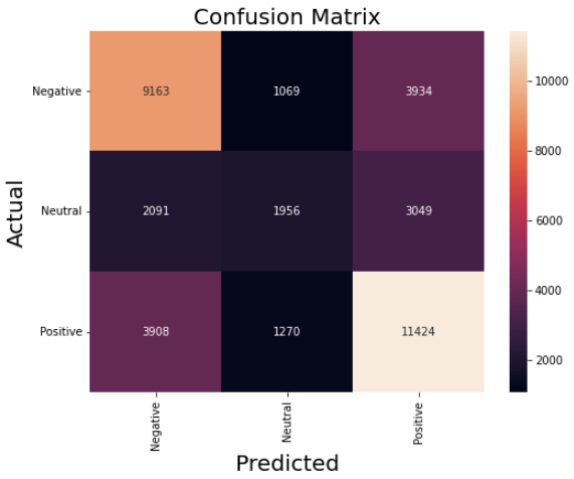
(b)

	Validation accuracy
Random Guessing	32.15912541755238
Always Positive	43.850592165198904
Naïve Bayes	66.0795627087762

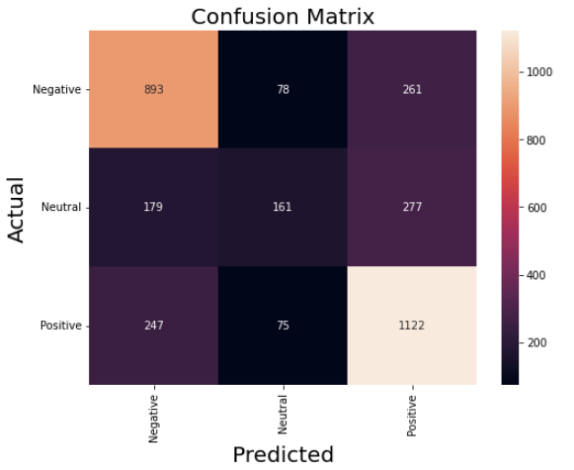
(c)

Naïve Bayes

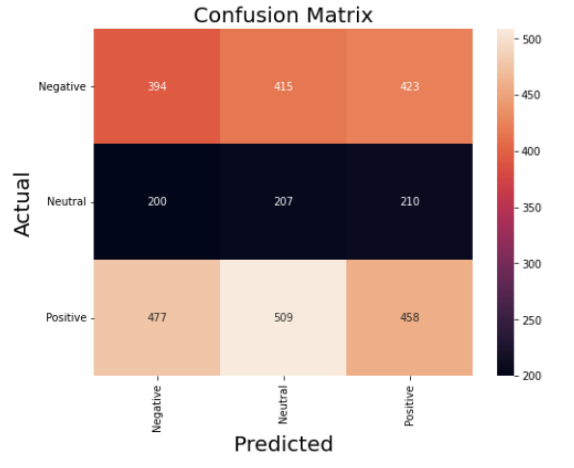
Training data



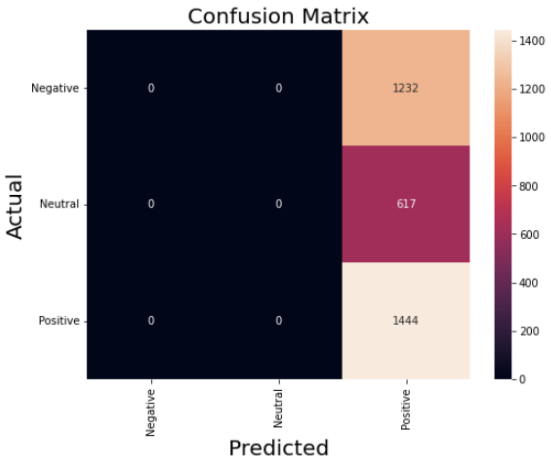
Testing data



Random guessing



Always positive



For every class, “positive” class has the highest value of diagonal entry which is in line with the fact that “positive” class has the most prior probability value

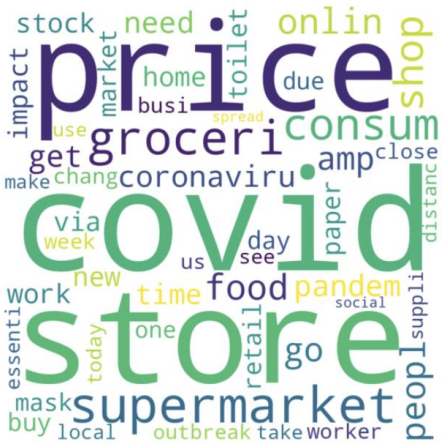
(d)

Stemming and Stop words removal

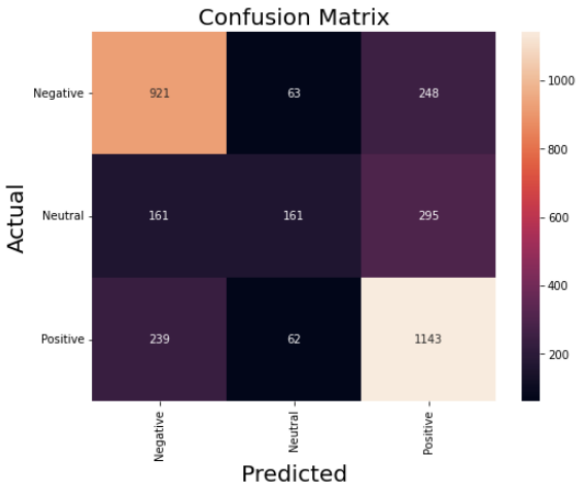
Word cloud for positive tokens

Word cloud for negative tokens

Word cloud for neutral tokens



Confusion matrix



	Validation accuracy
Random Guessing	32.15912541755238
Always Positive	43.850592165198904
Naïve Bayes	66.0795627087762
Naïve Bayes (stemming, stop words removal)	67.56756756756756

There is very little improvement in the validation accuracy even after performing stemming and stop words removal. It could be seen in the word cloud that words like {covid, price, store, groceri, …} occur frequently almost equally in all the classes.

(e)

Bigrams, Trigrams and WordScore

Bigrams - Counting the co-occurrence of two words

Trigrams - Counting the co-occurrence of three words

WordScore

I thought of giving a score to word in the vocabulary based on likeliness of it belonging to class depending on the proportion.

If proportion of positive is thrice of that of negative, then its strong positive

If proportion of negative is thrice of that of positive, then its strong negative

At inference time, I compute the sum of scores of words in test data and assign it to class by thresholding

```
def compute_word_scores(self):
    for word in self.vocabulary:
        # If word occurred atleast once in either positive or negative tweets
        if self.pos_tokens[word] != 0 or self.neg_tokens[word] != 0:
            arr = np.array([self.neg_tokens[word],self.neu_tokens[word],self.pos_tokens[word]])
            #proportion array
            prop_arr = arr / np.sum(arr)
            #temp_type stores the class which have max proportion
            temp_type = np.argmax(arr)

            #We assign scores to word depending on its proportion
            #NEGATIVE CASE
            if temp_type == 0:
                #strong negative
                if prop_arr[0] > 3 * prop_arr[2]:
                    self.score_of_word[word] = -1 * np.exp(abs(prop_arr[0]))
                #weak negative
                else:
                    self.score_of_word[word] = -1 * prop_arr[0]

            #NEUTRAL CASE
            elif temp_type == 1:
                #slightly inclined towards negative
                if prop_arr[0] > 3 * prop_arr[2]:
                    self.score_of_word[word] = -1 * prop_arr[0]
                #slightly inclined towards positive
                elif prop_arr[2] > 3 * prop_arr[0]:
                    self.score_of_word[word] = prop_arr[2]
                else:
                    self.score_of_word[word] = 0

            #POSITIVE CASE
            else:
                #strong positive
                if prop_arr[2] > 3 * prop_arr[0]:
                    self.score_of_word[word] = np.exp(abs(prop_arr[2]))
                #weak positive
                else:
                    self.score_of_word[word] = prop_arr[2]
```

Example

Word “beautiful” occurred 50 times in negative tweets, 150 times in neutral tweets and 200 times in positive tweets

Proportion is [0.125, 0.375, 0.50]

score_of_word[“beautiful”] = $e^{0.50}$

	Training accuracy	Validation accuracy
Unigram	62.750897950559896	67.56756756756756

Bigram	65.96503274878512	66.47433950804738
Trigram	83.19247834354532	65.92772547828727
WordScore	70.94865835622227	65.22927421803826

For this dataset, adding extra features like bigram, trigram only increases the training accuracy whereas testing accuracy remains the same.

WordScore method is highly efficient in terms of computation time.

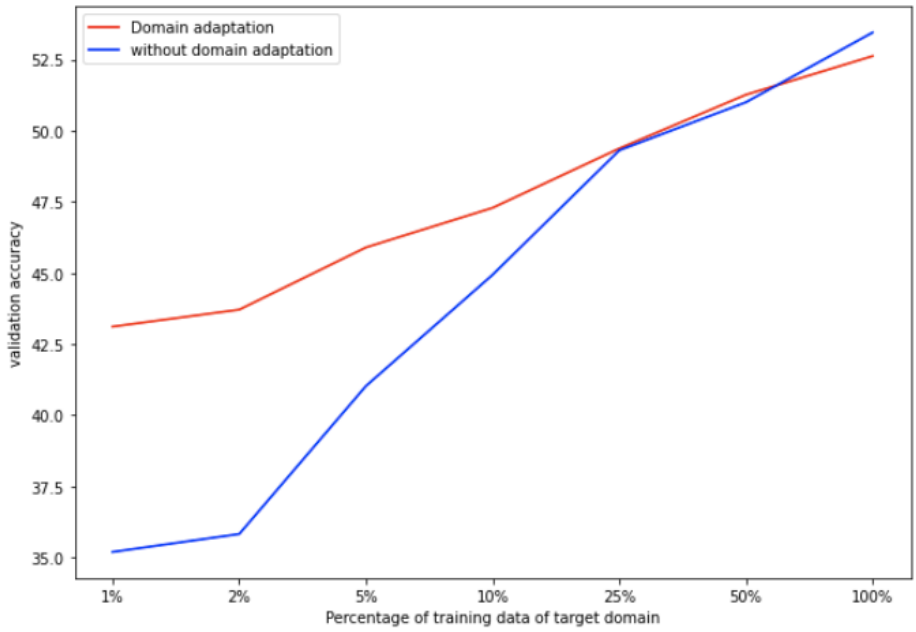
There is a scope of improvement by having a better preprocessing and coming with better features.

(f)

Domain Adaptation

Here we make use of counts of positive tokens, negative tokens and neutral tokens in source domain [corona tweets] and compute validation of target domain

	Validation accuracy without domain adaptation	Validation accuracy with domain adaptation
1% target training data	35.188866799204774	43.10801855533466
2% target training data	35.81842279655401	43.70444002650762
5% target training data	41.020543406229294	45.891318754141814
10% target training data	44.93041749502982	47.28296885354539
25% target training data	49.30417495029821	49.370444002650764
50% target training data	50.99403578528827	51.25911199469848
100% target training data	53.445990722332674	52.61762756792578



It can be observed that for less than 10-15% training data of target domain, domain adaptation is highly effective whereas once we have sufficient training data of target domain, domain adaptation hardly makes any difference in terms of validation accuracy, but it would have generalized well for other domains.

Q2

Image classification based on Intel image classification dataset

First, Let's build a binary classifier between class 0 and class 1 (my entry number is 2023AIZ8140)

Without loss of generality, we label class 0 as –1 and class 1 as 1.

We resize the image (150 x 150 x 3) and flatten it to 1 X 768 feature vector

Dual Problem

$$\min_{\alpha} \frac{1}{2} \sum_{i,j=1}^{m} y^{(i)} y^{(j)} \alpha_i \alpha_j < x^{(i)}, x^{(j)} > \; - \sum_{i=1}^{m} \alpha_i$$

$$s.t.\ \alpha_i \leq C,\ -\alpha_i \leq 0,\ i = 1, \ldots, m$$

$$\sum_{i=1}^{m} \alpha_i \ y^{(i)} = 0$$

CVXOPT package accepts in the following format

$$\min_{x} \frac{1}{2} x^T P x + q^T x$$

$$s.t.\ Gx \leq h$$

$$Ax = b$$

Given a feature mapping, kernel K is given by $K(x, z) = \phi(x)^T \phi(z)$

In case of linear kernel, $K(x, z) = x^T z$

In case of gaussian kernel, $K(x, z) = e^{(-\gamma \|x - z\|^2)}$

(a)

Using CVXOPT package (linear kernel)

Formulating the dual problem and passing it to CVXOPT solver

```
#P
P_x = SVM_cvxopt.linear_kernel(self.X_train)
P_y = np.matmul(self.Y_train, self.Y_train.T)
self.P = matrix(np.multiply(P_x, P_y).astype('float'))

#q
self.q = matrix(-1*np.ones((self.m,1)).astype('float'))

#G
G_u = np.zeros((self.m, self.m))
np.fill_diagonal(G_u, -1)
G_d = np.zeros((self.m, self.m))
np.fill_diagonal(G_d, 1)
self.G = matrix(np.concatenate((G_u, G_d), axis=0).astype('float'))

#h
h_u = np.zeros((self.m,1))
h_d = np.ones((self.m,1))
self.h = matrix(np.concatenate((h_u,h_d), axis=0).astype('float'))

#A
self.A = matrix(self.Y_train.T.astype('float'))

#b
self.b = matrix(np.zeros(1).astype('float'))

# Construct the QP, invoke solver
self.sol = solvers.qp(self.P, self.q, self.G, self.h, self.A, self.b)
```

(i) Number of support vectors = 1400

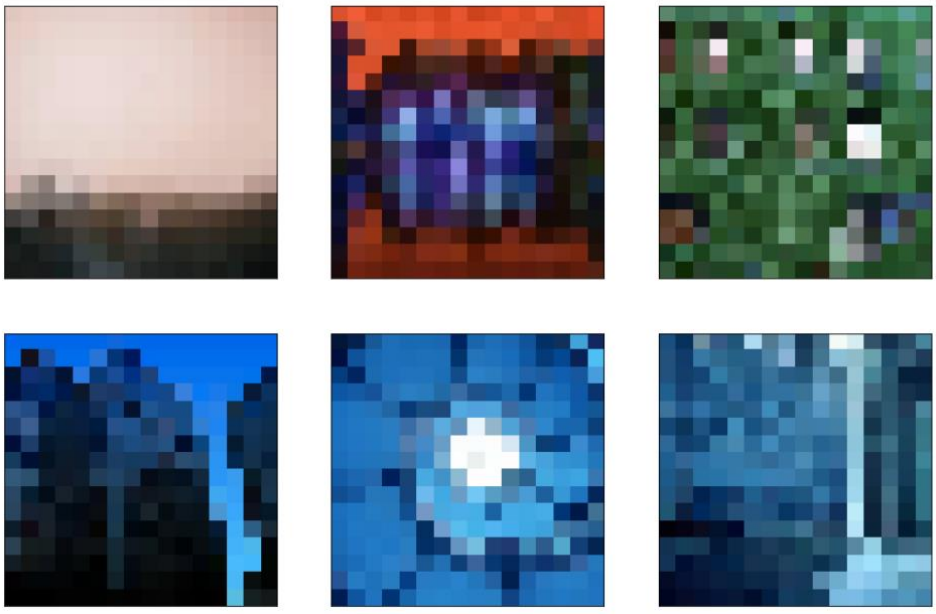
Percentage of training samples which constitute support vectors = 29.41%

Time taken for training: 82261.516ms

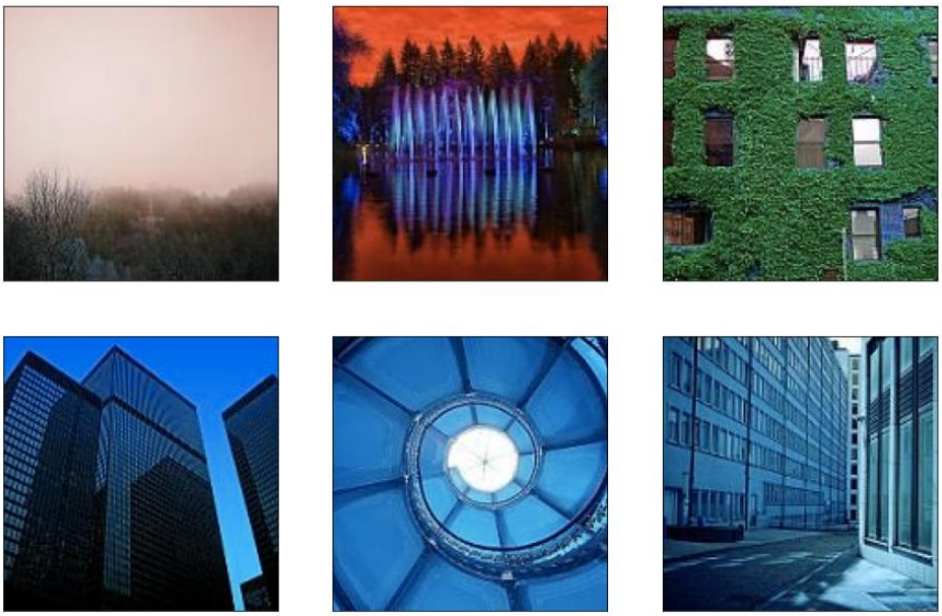
C used here is 1.

(ii) Validation set accuracy = 83.75%

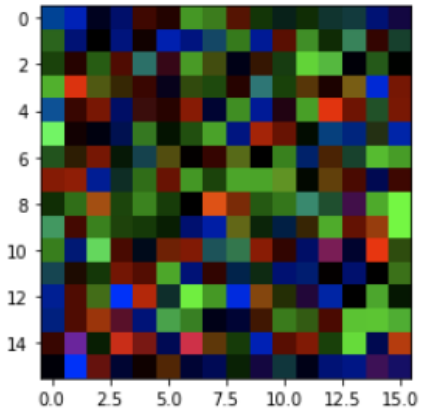
(iii) Support vectors corresponding to top-6% coefficients are reshaped to get images



Their original counterparts are shown below



weight vector is reshaped and plotted below



(b)

Using CVXOPT package (gaussian kernel)

(i)

	#support_vectors	%training_data_are_svs
SVM (Linear kernel)	1400	29.41
SVM (Gaussian kernel)	1957	41.11

Support vectors which are in both are 1240.

Time taken for training: 86239.492ms

(ii)

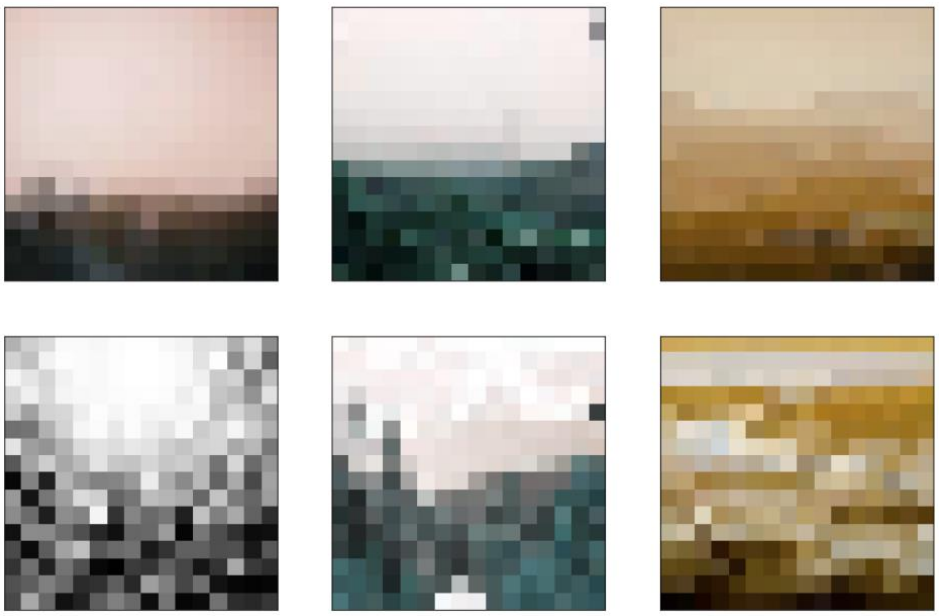
Validation set accuracy: 84.5%

C used here is 1.

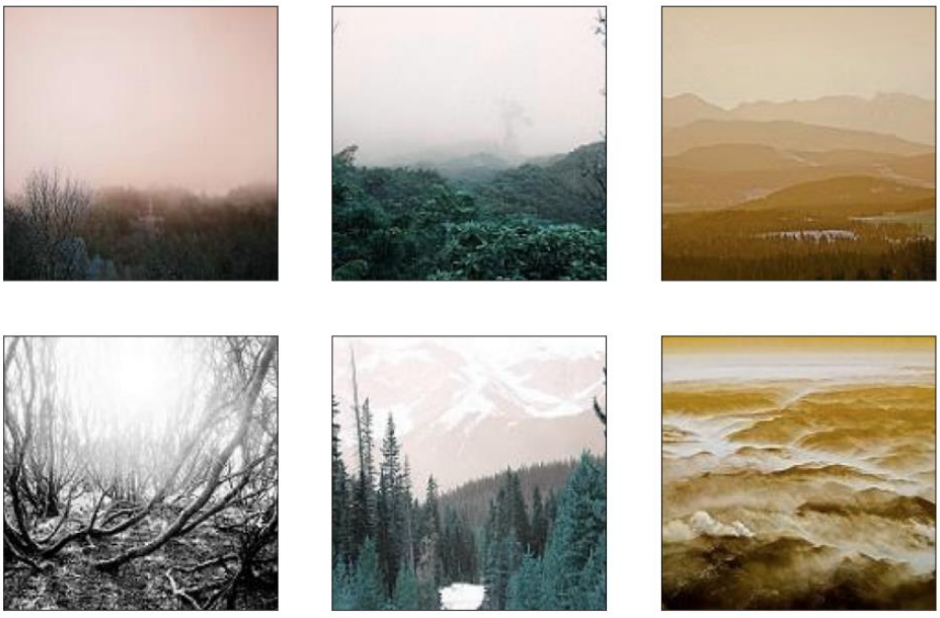
γ used is 0.001

(iii)

Support vectors corresponding to top-6% coefficients are reshaped to get images



Their original counterparts are shown below



Support vectors (here support images) are usually the ones which are closer to the decision boundary. They are relatively difficult images to classify

(iv)

	Validation accuracy
SVM (Linear kernel)	83.75
SVM (Gaussian kernel)	84.5

(c) **Using scikit-learn package (linear kernel and gaussian kernel)**

(i)

Number of support vectors	Using CVXOPT	Using scikit-learn
SVM (Linear kernel)	1400	1397
SVM (Gaussian kernel)	1957	1956

	Number of support vectors match between CVXOPT and scikit-learn
SVM (Linear kernel)	1397
SVM (Gaussian kernel)	1956

(ii)

Comparing weight(w) and bias(b)

For linear case

$\| w_{cvxopt} - w_{scikit} \| = 0.015276561889511025$

$\| b_{cvxopt} - b_{scikit} \| = 0.4846014732641839$

For gaussian case

$\| b_{cvxopt} - b_{scikit} \| = 0.04363545053161211$

(iii)

	Validation accuracy
SVM (Linear kernel)	84
SVM (Gaussian kernel)	84

(iv)

Computational cost (training time) in milliseconds	Using CVXOPT	Using scikit-learn
SVM (Linear kernel)	82261.516ms	4981.047ms
SVM (Gaussian kernel)	86239.492ms	4145.652ms

(d)

Resizing the 150 x 150 x 3 image to 32 x 32 x 3 instead of 16 x 16 x 3 image

Number of support vectors	16 x 16 x 3	32 x 32 x 3
SVM (Linear kernel)	1397	1361
SVM (Gaussian kernel)	1747	1647

	Number of support vectors match between CVXOPT and scikit-learn(16 x 16 x 3)	Number of support vectors match between CVXOPT and scikit-learn(32 x 32 x 3)
SVM (Linear kernel)	1397	882
SVM (Gaussian kernel)	1505	1548

Validation accuracy	16 x 16 x 3	32 x 32 x 3
SVM (Linear kernel)	84	78
SVM (Gaussian kernel)	89	87

Computational cost (training time) in milliseconds	Using scikit-learn (16 x 16 x 3)	Using scikit-learn (32 x 32 x 3)
SVM (Linear kernel)	4981.047ms	18575.003ms
SVM (Gaussian kernel)	3135.113ms	13366.478ms

Observations

-> Number of support vectors slightly decreases

-> Significant drop in overlap of support vectors obtained using CVXOPT for linear kernel case

-> Validation accuracy for gaussian kernel case drops by 2% whereas for linear kernel case it drops by 6%

-> Computation cost (training time) almost triples for higher feature size

Insight

It does not make sense to increase the feature size given the fact that there is no improvement over validation accuracy as well as computation cost

Multiclass classification

I created 15 (6_{C_2}) one vs one models, and during testing whichever gets maximum votes is the predicted value

(a)

Using CVXOPT

Validation set accuracy: 56%

(b)

Using Scikit-learn

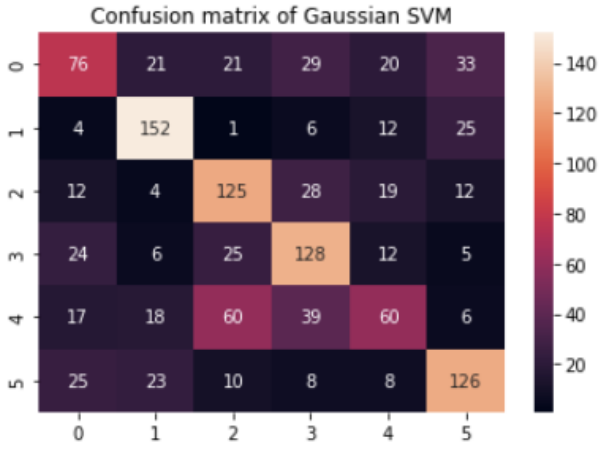
Validation set accuracy: 56%

	Validation accuracy
Using CVXOPT	56
Using Scikit-learn	56

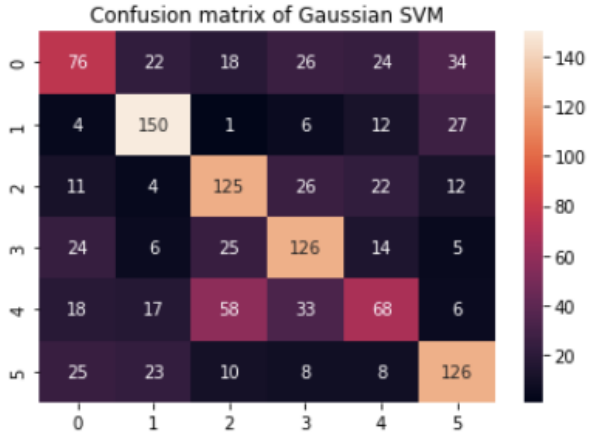
Training using scikit-learn took 71612.690ms. I could not compute the training time using CVXOPT because storing support vectors which constituted around 40% of the training data made the models so heavy that I had to train them in chunks and store them in a file. But roughly I can say that the time took using CVXOPT was far more than the one using scikit-learn.

(c)

Confusion matrix using CVXOPT

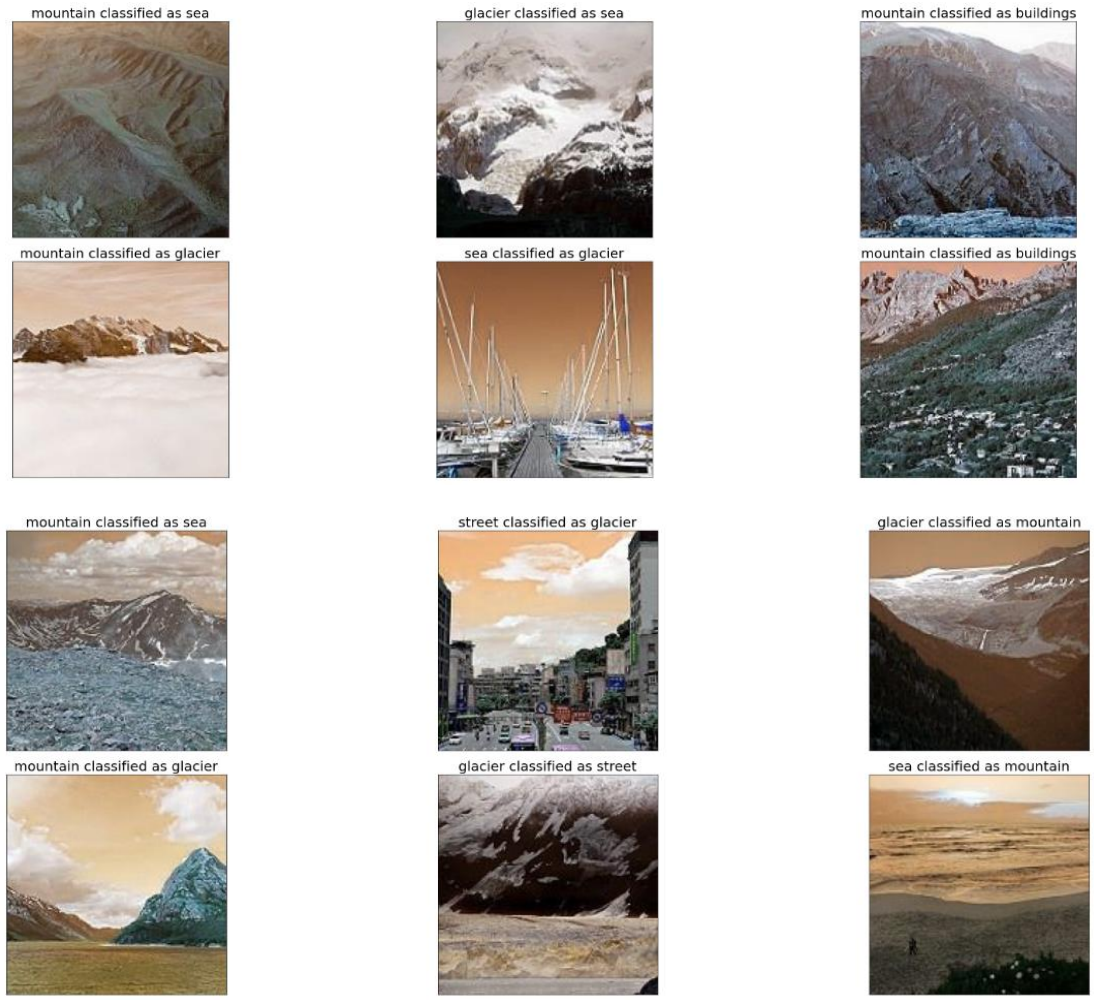


Confusion matrix using scikit-learn



It could be observed that class 0 image is misclassified as class 5 many times. It is reasonable for some buildings to be classified as streets since after all buildings are present in streets.

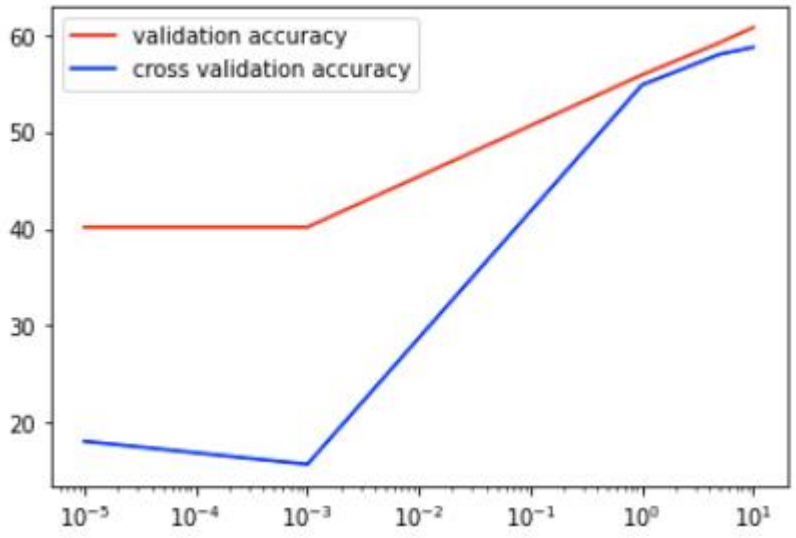
12 misclassified images



(d)

5-Fold Validation

	Validation Accuracy	**Cross validation accuracy(K=5)**
C = 10^-5	40.166666666666664	18.018207282913167
C = 10^-3	40.166666666666664	15.644257703081232
C = 1	55.916666666666664	54.87394957983192
C = 5	59.25	58.060224089635845
C = 10	60.833333333333336	58.78151260504202



As we increase the value of C, both validation and cross validation accuracy increases

C = 10 gives the best validation and cross validation accuracy. It is good to have cross validation accuracy to tune the value of C

COL 774 MACHINE LEARNING | ASSIGNMENT 3

T KARTHIKEYAN | 2023AIZ8140

Q1

Decision Tree and Random Forests

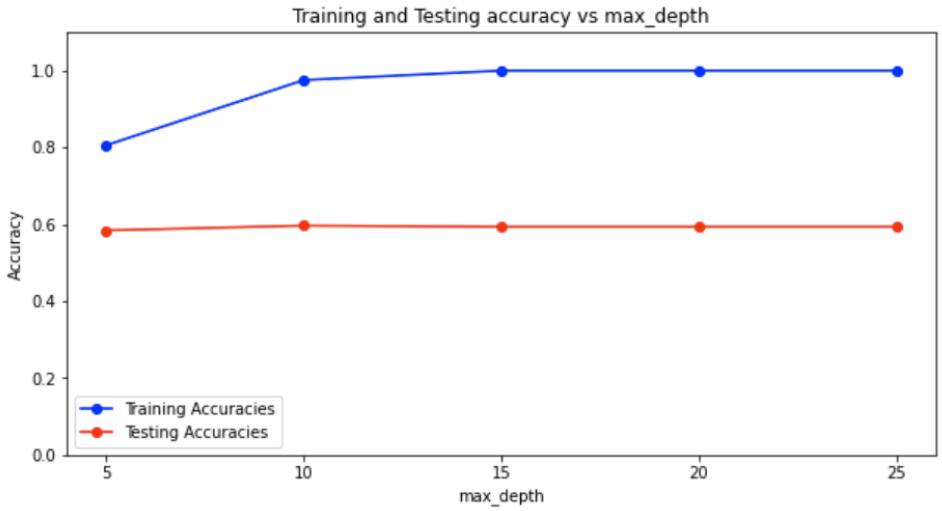
(a)

Decision tree construction

Categorical columns are ordinal encoded (Country: India->0, China->1, US ->2, …)

Experiment with different depths {5,10,15,20,25}

```
Max depth:  5
training accuracy:  0.8058004343937651
testing accuracy:  0.5842812823164426


Max depth:  10
training accuracy:  0.9753417656828925
testing accuracy:  0.5966907962771458


Max depth:  15
training accuracy:  0.9998722371278906
testing accuracy:  0.59358841778697


Max depth:  20
training accuracy:  0.9998722371278906
testing accuracy:  0.59358841778697


Max depth:  25
training accuracy:  0.9998722371278906
testing accuracy:  0.59358841778697
```



Note: depth of the decision tree created in this case is 14, so varying the max_depth beyond 15 is bound to be constant. It could be seen that testing accuracy is around 59% for depth (5,10,15} whereas training accuracy is significantly more for depth {10,15} than depth {5}, so decision tree overfits at depth {10,15}.

Test accuracy for only win prediction

```
[ONLY WIN]testing accuracy:  0.4963805584281282
```

Test accuracy for only loss prediction

```
[ONLY LOSS]testing accuracy:  0.5036194415718718
```

Decision tree performs much better if the baseline was only win/only loss prediction.

(b)

Decision tree construction

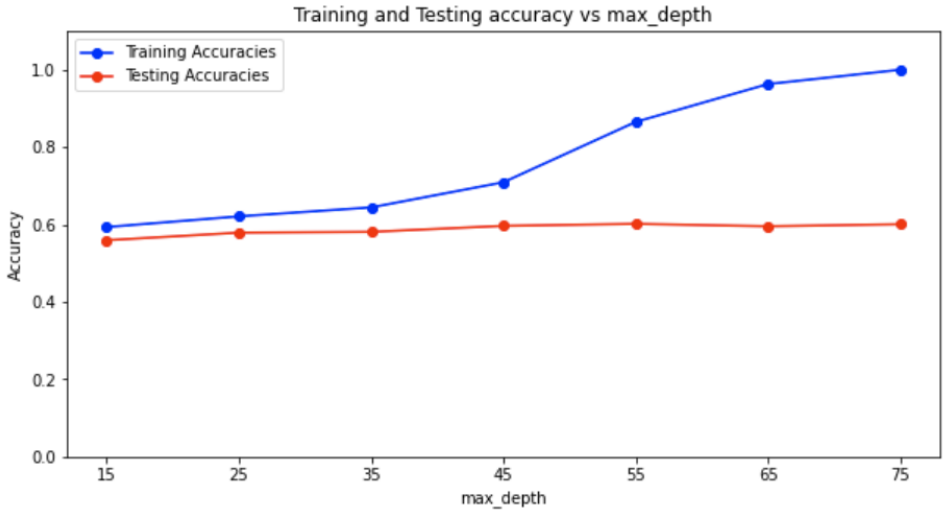
Categorical columns having more than two categories are split into multiple columns which only have two categories (**one-hot encoded)** (Country: {india, china, US) -> [Country_india, Country_china, Country_US, ...)

Experiment with different depths {15,25,35,45}

```
Max depth:  15
training accuracy:  0.5935863038201099
testing accuracy:  0.5594622543950362


Max depth:  25
training accuracy:  0.6211830841957328
testing accuracy:  0.5791106514994829


Max depth:  35
training accuracy:  0.6443081640475278
testing accuracy:  0.5811789038262668


Max depth:  45
training accuracy:  0.7093394659511946
testing accuracy:  0.5966907962771458
```

I tried out few more depths {55,65,75}

```
Max depth:  55
training accuracy:  0.8652101699246199
testing accuracy:  0.6018614270941055


Max depth:  65
training accuracy:  0.9630765299603935
testing accuracy:  0.5956566701137539


Max depth:  75
training accuracy:  1.0
testing accuracy:  0.6008273009307136
```



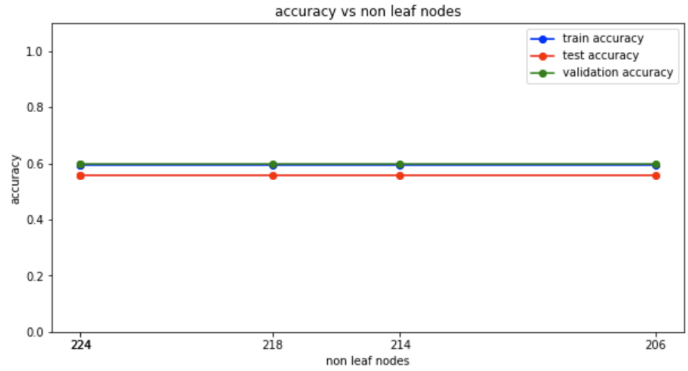
It is noted that depth of the full tree is 73. Testing accuracy in this case for higher depths (>=45) is slightly better than what we achieved in previous part. Since the testing accuracy is not significantly high, it would be computationally efficient to go with previous tree (which only had a depth of 15).

(c)

Post pruning of decision tree

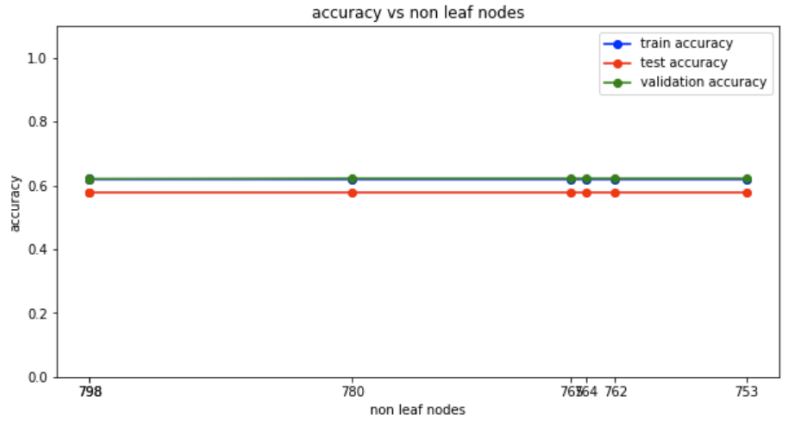
max_depth = 15

```
Final Training accuracy:  0.5934585409480005
Final Validation accuracy:  0.596551724137931
Final Testing accuracy:  0.5594622543950362
```



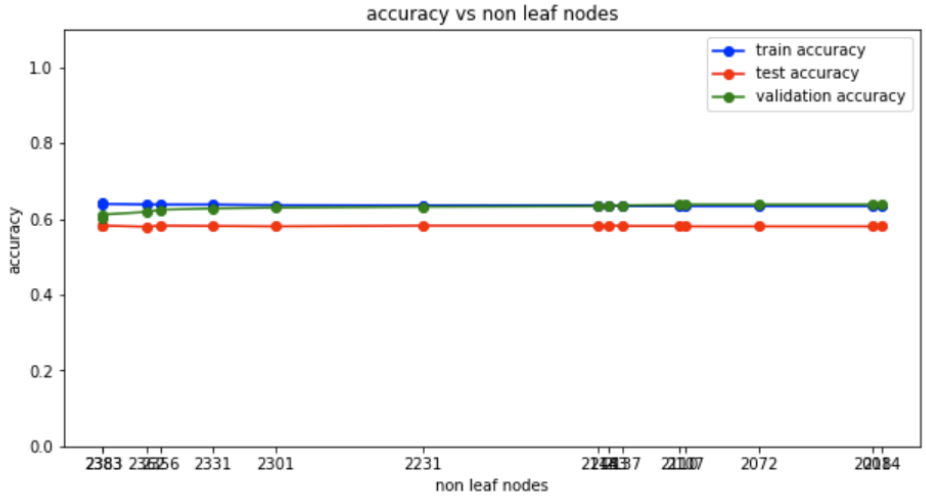
max_depth = 25

```
Final Training accuracy:  0.6191388782419829
Final Validation accuracy:  0.6218390804597701
Final Testing accuracy:  0.578076525336091
```



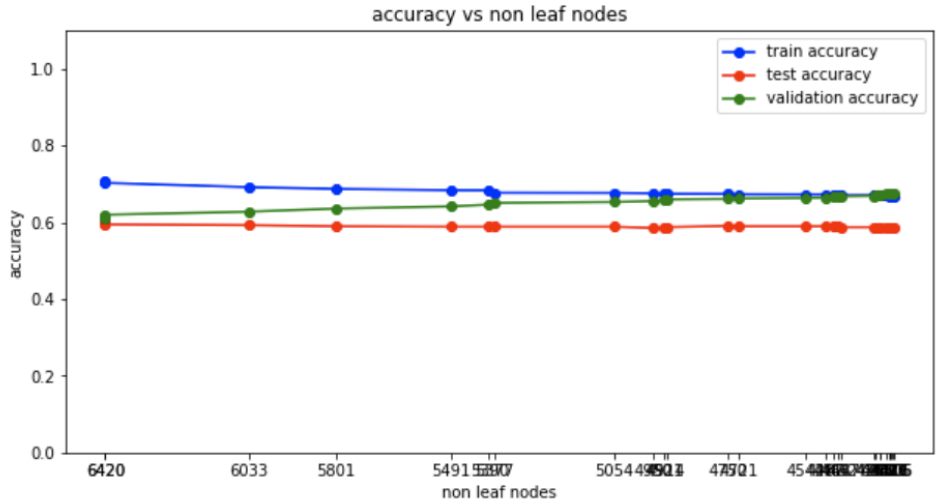
max_depth = 35

```
Final Training accuracy:  0.6331927941740131
Final Validation accuracy:  0.6379310344827587
Final Testing accuracy:  0.5801447776628749
```



max_depth = 45

```
Final Training accuracy:  0.6681998211319791
Final Validation accuracy:  0.6758620689655173
Final Testing accuracy:  0.5853154084798345
```



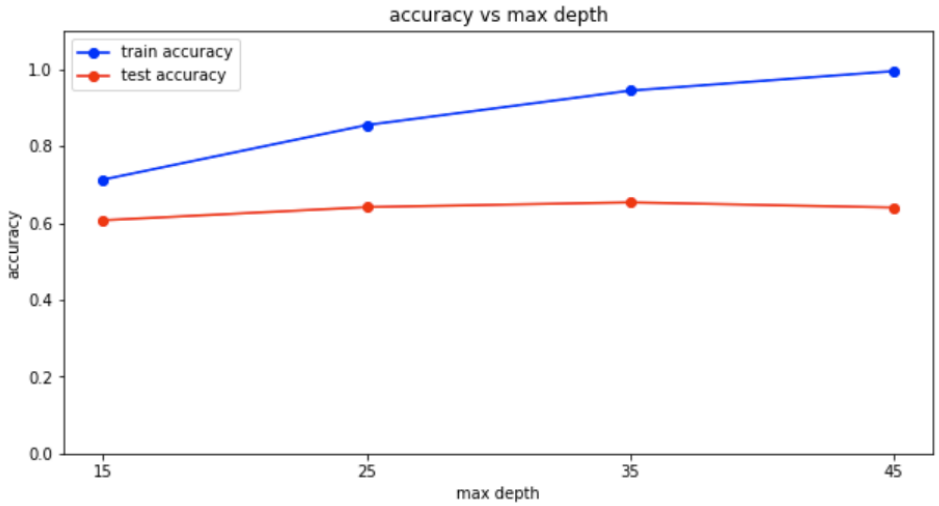
Post pruning have removed some nodes based on validation accuracy, but improving validation accuracy didn't translate to increase in testing accuracy. It is good to note that with reduced number of nodes after post pruning, model's test accuracy doesn't drop much.

(d)

Decision tree using scikit-learn

Experimenting with different max_depths = {15, 25, 35, 45}

```
max_depth used is 15
Training accuracy: 0.7129168263702568
Validation accuracy: 0.5850574712643678
Testing accuracy: 0.6070320579110652

max_depth used is 25
Training accuracy: 0.854733614411652
Validation accuracy: 0.6126436781609196
Testing accuracy: 0.6411582213029989

max_depth used is 35
Training accuracy: 0.9442953877603169
Validation accuracy: 0.6091954022988506
Testing accuracy: 0.6535677352637022

max_depth used is 45
Training accuracy: 0.9950172479877347
Validation accuracy: 0.6275862068965518
Testing accuracy: 0.640124095139607
```



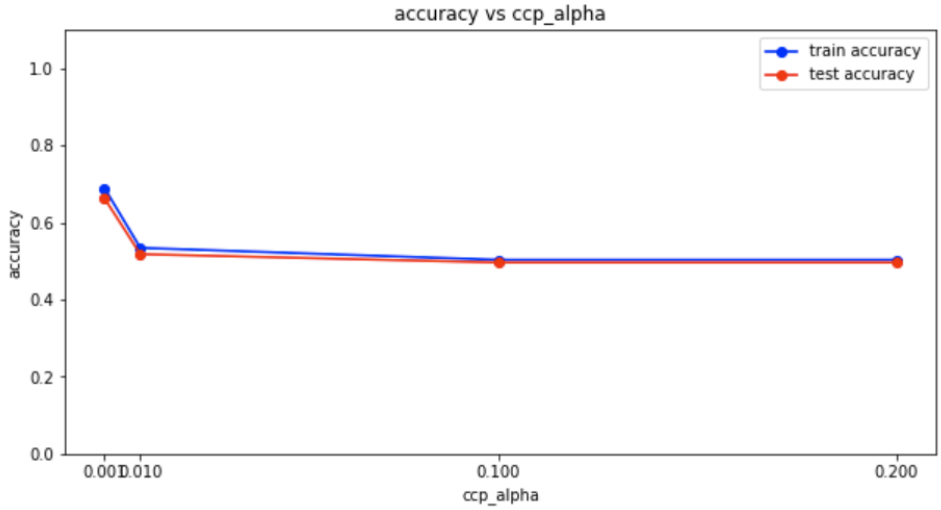
Running the decision tree on the best value of max_depth obtained using validation accuracy

```
clf = tree.DecisionTreeClassifier(criterion="entropy", max_depth=45)
clf = clf.fit(X_train, y_train)
y_train_pred = clf.predict(X_train)
y_test_pred = clf.predict(X_test)
print(f"max_depth used is 45")
print(f"Training accuracy: {accuracy_score(y_train, y_train_pred)}")
print(f"Testing accuracy: {accuracy_score(y_test, y_test_pred)}\n")
```

```
max_depth used is 45
Training accuracy: 0.9951450108598441
Testing accuracy: 0.6349534643226473
```

Experimenting with different ccp_alphas = {0.001, 0.01, 0.1, 0.2)

```
ccp_alpha used is 0.001
Training accuracy: 0.6894084579021337
Validation accuracy: 0.632183908045977
Testing accuracy: 0.6628748707342296

ccp_alpha used is 0.01
Training accuracy: 0.5344320940334739
Validation accuracy: 0.5
Testing accuracy: 0.5180972078593589

ccp_alpha used is 0.1
Training accuracy: 0.5033857161108982
Validation accuracy: 0.4735632183908046
Testing accuracy: 0.4963805584281282

ccp_alpha used is 0.2
Training accuracy: 0.5033857161108982
Validation accuracy: 0.4735632183908046
Testing accuracy: 0.4963805584281282
```



Running the decision tree on the best value of ccp_alpha obtained using validation accuracy

```
clf = tree.DecisionTreeClassifier(criterion="entropy", ccp_alpha=0.001)
clf = clf.fit(X_train, y_train)
y_train_pred = clf.predict(X_train)
y_val_pred = clf.predict(X_val)
y_test_pred = clf.predict(X_test)
print(f"ccp_alpha used is 0.001")
print(f"Training accuracy: {accuracy_score(y_train, y_train_pred)}")
print(f"Testing accuracy: {accuracy_score(y_test, y_test_pred)}\n")
```

```
ccp_alpha used is 0.001
Training accuracy: 0.6894084579021337
Testing accuracy: 0.6628748707342296
```

Observations

Method	Testing Accuracy
Decision Tree (ordinal encoding)	0.5935
Decision Tree (one-hot encoding)	0.6008
Decision Tree (post pruning)	0.5853
Scikit decision tree (with best max_depth)	0.6349
Scikit decision tree (with best ccp_alpha)	0.6628

Decision tree performs well after tuning parameters like ccp_alpha and max_depth.

(e)

Random Forests using scikit learn

```
for n_estimators in [50, 150, 250, 350]:
    for max_features in [0.1, 0.3, 0.5, 0.7, 0.9, 1.0]:
        for min_samples_split in [2, 4, 6, 8, 10]:
            print(f"Parameters->(n_estimators={n_estimators}, max_features={max_features},
min_samples_split={min_samples_split})")
            clf = RandomForestClassifier(n_estimators = n_estimators, max_features=max_features,
min_samples_split=min_samples_split,criterion = "entropy", oob_score=True)
            clf.fit(X_train, y_train.ravel())
            y_val_pred = clf.predict(X_val)
```

```
Optimal parameters based on OOB score:
best_oob_score :0.7258208764533026
best_n_estimators:250
best_max_features:1.0
best_min_samples_split:10
Training accuracy: 0.9814743835441421
Validation accuracy: 0.7045977011494253
Testing accuracy: 0.7249224405377456
OOB(out of bag) accuracy:  0.7199437843362719
```

Grid Search to find the optimal hyperparameters for random forests

```python
#making the instance
model = RandomForestClassifier()

#Hyper Parameters Set
params = {
    'n_estimators': [50, 150, 250, 350],
    'max_features': [0.1, 0.3, 0.5, 0.7, 0.9, 1.0],
    'min_samples_split': [2, 4, 6, 8, 10],
    'criterion' :['entropy'],
    'oob_score': [True],

}

grid = GridSearchCV(estimator=model, param_grid=params, verbose=2)

# Fit the grid search to the data and find best hyperparameters
grid.fit(X_train, y_train.ravel())
```

```
Best parameters:  {'criterion': 'entropy', 'max_features': 0.7, 'min_samples_split': 8, 'n_estimators': 350, 'oob_score': True}
Training accuracy: 0.9892679187428134
Validation accuracy: 0.7034482758620689
Testing accuracy: 0.7269906928645294
OOB(out of bag) accuracy:  0.7191772071036157
```

Method	Training accuracy	Validation accuracy	Testing accuracy
Decision Tree (with post pruning)	0.6681	0.6758	0.5853
Scikit decision tree (with best max_depth)	0.995	0.6275	0.6349
Scikit decision tree (with best ccp_alpha)	0.6894	0.6321	0.6628
Random Forest	0.9892	0.7034	0.7269

Random forest performs significantly better than decision trees in terms of testing accuracy as it is ensemble of decision trees, and it generalizes well.

(f)

Gradient Boosting classifier

```
#making the instance
model = GradientBoostingClassifier()

#Hyper Parameters Set
params = {
    'n_estimators': [50, 100, 150, 200, 250, 300],
    'learning_rate': [0.1, 0.5, 1.0],
    'min_samples_split': [6, 8, 10],
    'ccp_alpha': [0.0, 0.001, 0.005]
}

grid = GridSearchCV(estimator=model, param_grid=params, verbose=2)

# Fit the grid search to the data and find best hyperparameters
grid.fit(X_train, y_train.ravel())
```

```
Best parameters:  {'ccp_alpha': 0.0, 'learning_rate': 0.5, 'min_samples_split': 10, 'n_estimators': 100}
Training accuracy: 0.7926408585665006
Validation accuracy: 0.732183908045977
Testing accuracy: 0.7228541882109617
```

Testing accuracy obtained with Gradient Boosting Classifier was at par with the one obtained with random forests.

Q2

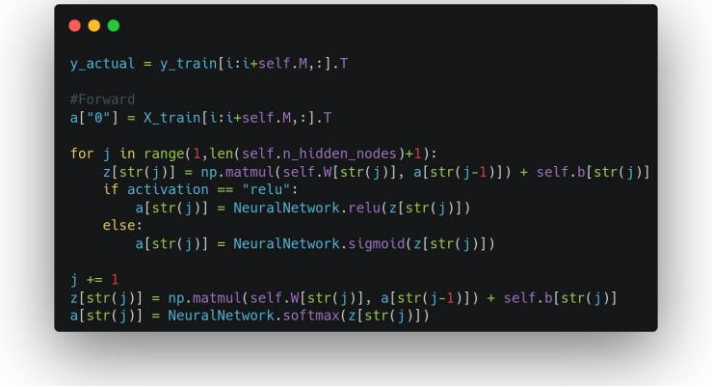
Neural Networks

(a)

Generic Neural network architecture (fully connected) for multi-class classification

Algorithm:

Forward pass and backward pass will be executed for every mini batch and parameters will be updated



```
y_actual = y_train[i:i+self.M,:].T

#Forward
a["0"] = X_train[i:i+self.M,:].T

for j in range(1,len(self.n_hidden_nodes)+1):
    z[str(j)] = np.matmul(self.W[str(j)], a[str(j-1)]) + self.b[str(j)]
    if activation == "relu":
        a[str(j)] = NeuralNetwork.relu(z[str(j)])
    else:
        a[str(j)] = NeuralNetwork.sigmoid(z[str(j)])

j += 1
z[str(j)] = np.matmul(self.W[str(j)], a[str(j-1)]) + self.b[str(j)]
a[str(j)] = NeuralNetwork.softmax(z[str(j)])
```

```
#Backward
del_z[str(j)] = a[str(j)] - y_actual
del_b[str(j)] = np.sum(del_z[str(j)], axis = 1).reshape(-1,1)
del_W[str(j)] = np.matmul(del_z[str(j)], a[str(j-1)].T)

for k in range(j-1,0,-1):
    if activation == "relu":
        del_z[str(k)] = np.matmul(self.W[str(k+1)].T, del_z[str(k+1)])*(NeuralNetwork.relu(z[str(k)],
derivative=True))
    else:
        del_z[str(k)] = np.matmul(self.W[str(k+1)].T, del_z[str(k+1)])*(NeuralNetwork.sigmoid(z[str(k)],
derivative=True))
    del_b[str(k)] = np.sum(del_z[str(k)], axis = 1).reshape(-1,1)
    del_W[str(k)] = np.matmul(del_z[str(k)], a[str(k-1)].T)
```

```
#Update
for l in range(1,len(self.n_hidden_nodes)+2):
    if adaptive_learning == False:
        self.W[str(l)] = self.W[str(l)] - alpha * del_W[str(l)]
        self.b[str(l)] = self.b[str(l)] - alpha * del_b[str(l)]
    else:
        self.W[str(l)] = self.W[str(l)] - (alpha/np.sqrt(epoch)) * del_W[str(l)]
        self.b[str(l)] = self.b[str(l)] - (alpha/np.sqrt(epoch)) * del_b[str(l)]
```

Activation functions used

```
#Sigmoid Function
def sigmoid(x, derivative = False):
    if derivative == False:
        return 1 / (1 + np.exp(-x))
    else:
        return NeuralNetwork.sigmoid(x, derivative = False) * (1 - NeuralNetwork.sigmoid(x, derivative =
False))

#ReLu function
def relu(x, derivative = False):
    if derivative == True:
        return np.where(x > 0, 1, np.where(x < 0, 0, np.random.random_sample()))
    else:
        return np.where(x <= 0, 0, x)

#Softmax fuunction
def softmax(Z):
    return np.exp(Z) / np.sum(np.exp(Z), axis=0)
```

Stochastic Gradient Descent used with mini batch size (M)

Cross entropy loss as the loss function

Sigmoid as the activation function in the hidden layers

#Training samples: 10000

#Test samples: 1000

#epochs = 100

Output:

```
epoch 0
accuracy on train data:  0.2091
softmax loss:  0.00023525478839202835

epoch 10
accuracy on train data:  0.5923
softmax loss:  0.00011264389363753989

epoch 20
accuracy on train data:  0.6572
softmax loss:  5.831073565883006e-05

epoch 30
accuracy on train data:  0.6702
softmax loss:  4.656056662492289e-05

epoch 40
accuracy on train data:  0.6758
softmax loss:  3.9645451044889796e-05

epoch 50
accuracy on train data:  0.6803
softmax loss:  3.4488996828750714e-05

epoch 60
accuracy on train data:  0.6869
softmax loss:  3.0593861550354343e-05

epoch 70
accuracy on train data:  0.6908
softmax loss:  2.8066258449785907e-05

epoch 80
accuracy on train data:  0.6976
softmax loss:  2.7373041902283903e-05

epoch 90
accuracy on train data:  0.7037
softmax loss:  2.8412086233645623e-05

accuracy on test data:  0.714
```

Accuracy: 71.4 %

(b)

Experiment: Varying the number of nodes in the hidden layer

Mini-batch size = 32

Learning rate = 0.01

n = 1024 (16 x 16 x 4)

Hidden layers = [[1], [5], [10], [50], [100]]

r = 5

Stopping criteria: Moving average of softmax loss

Let's take a window size, W = 5 and threshold = 1e-06. We compute the average of softmax loss over W batches. If the difference between present average and the previous average turns out to be less than a threshold, we declare convergence and end the training.

Results:

Hidden layers = [1]

```
accuracy on train data:  0.2091
metrics for train data:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         0
           1       0.00      0.00      0.00         0
           2       0.00      0.00      0.00         0
           3       0.00      0.00      0.00         0
           4       1.00      0.21      0.35     10000

   micro avg       0.21      0.21      0.21     10000
   macro avg       0.20      0.04      0.07     10000
weighted avg       1.00      0.21      0.35     10000
 samples avg       0.21      0.21      0.21     10000

accuracy on test data:  0.187
metrics for test data:
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         0
           1       0.00      0.00      0.00         0
           2       0.00      0.00      0.00         0
           3       0.00      0.00      0.00         0
           4       1.00      0.19      0.32      1000

   micro avg       0.19      0.19      0.19      1000
   macro avg       0.20      0.04      0.06      1000
weighted avg       1.00      0.19      0.32      1000
 samples avg       0.19      0.19      0.19      1000
```

Hidden layers = [5]

accuracy on train data: 0.6977

metrics for train data:

	precision	recall	f1-score	support
0	0.95	0.86	0.90	2176
1	0.64	0.77	0.70	1652
2	0.73	0.56	0.63	2544
3	0.41	0.56	0.47	1447
4	0.77	0.73	0.75	2181
micro avg	0.70	0.70	0.70	10000
macro avg	0.70	0.70	0.69	10000
weighted avg	0.72	0.70	0.70	10000
samples avg	0.70	0.70	0.70	10000

accuracy on test data: 0.689

metrics for test data:

	precision	recall	f1-score	support
0	0.94	0.87	0.91	246
1	0.63	0.76	0.69	163
2	0.72	0.58	0.64	248
3	0.41	0.49	0.45	158
4	0.69	0.70	0.69	185
micro avg	0.69	0.69	0.69	1000
macro avg	0.68	0.68	0.68	1000
weighted avg	0.71	0.69	0.69	1000
samples avg	0.69	0.69	0.69	1000

Hidden layers = [10]

accuracy on train data: 0.7436

metrics for train data:

	precision	recall	f1-score	support
0	0.93	0.95	0.94	1930
1	0.75	0.81	0.78	1852
2	0.67	0.66	0.66	2000
3	0.51	0.59	0.55	1752
4	0.85	0.72	0.78	2466
micro avg	0.74	0.74	0.74	10000
macro avg	0.74	0.74	0.74	10000
weighted avg	0.75	0.74	0.75	10000
samples avg	0.74	0.74	0.74	10000

accuracy on test data: 0.724

metrics for test data:

	precision	recall	f1-score	support
0	0.89	0.96	0.92	212
1	0.73	0.76	0.75	191
2	0.65	0.64	0.65	204
3	0.51	0.55	0.53	175
4	0.80	0.69	0.74	218
micro avg	0.72	0.72	0.72	1000
macro avg	0.72	0.72	0.72	1000
weighted avg	0.73	0.72	0.72	1000
samples avg	0.72	0.72	0.72	1000

Hidden layers = [50]

accuracy on train data: 0.788

metrics for train data:

	precision	recall	f1-score	support
0	0.94	0.98	0.96	1878
1	0.79	0.88	0.84	1783
2	0.64	0.72	0.68	1734
3	0.65	0.62	0.64	2106
4	0.91	0.76	0.83	2499
micro avg	0.79	0.79	0.79	10000
macro avg	0.79	0.79	0.79	10000
weighted avg	0.79	0.79	0.79	10000
samples avg	0.79	0.79	0.79	10000

accuracy on test data: 0.735

metrics for test data:

	precision	recall	f1-score	support
0	0.93	1.00	0.96	215
1	0.76	0.82	0.79	183
2	0.55	0.65	0.60	168
3	0.63	0.52	0.57	229
4	0.76	0.70	0.73	205
micro avg	0.73	0.73	0.73	1000
macro avg	0.73	0.74	0.73	1000
weighted avg	0.73	0.73	0.73	1000
samples avg	0.73	0.73	0.73	1000

Hidden layers = [100]

accuracy on train data: 0.8133

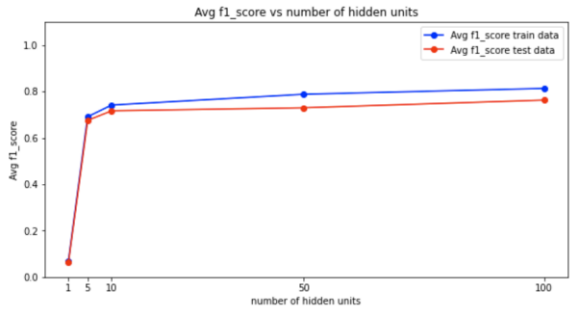
metrics for train data:

	precision	recall	f1-score	support
0	0.98	0.96	0.97	2010
1	0.82	0.92	0.87	1766
2	0.74	0.77	0.75	1883
3	0.64	0.67	0.65	1914
4	0.88	0.76	0.82	2427
micro avg	0.81	0.81	0.81	10000
macro avg	0.81	0.82	0.81	10000
weighted avg	0.82	0.81	0.81	10000
samples avg	0.81	0.81	0.81	10000

accuracy on test data: 0.77

metrics for test data:

	precision	recall	f1-score	support
0	0.97	0.97	0.97	228
1	0.74	0.84	0.79	174
2	0.65	0.68	0.67	191
3	0.63	0.59	0.61	197
4	0.82	0.73	0.78	210
micro avg	0.77	0.77	0.77	1000
macro avg	0.76	0.77	0.76	1000
weighted avg	0.77	0.77	0.77	1000
samples avg	0.77	0.77	0.77	1000



After 10 nodes in the single hidden layer, f1_score saturates. It shows that simply increasing the nodes in the hidden layer doesn't guarantee increase in performance. It could be seen that model haven't learnt anything with just 1 parameter in the single hidden layer.

(c)

Experiment: Trying out different depths with decreasing number of hidden nodes

Mini-batch size = 32

Learning rate = 0.01

n = 1024 (16 x 16 x 4)

Hidden layers = [[512], [512,256], [512,256,128], [512,256,128,64]]

r = 5

Stopping criteria: Moving average of softmax loss (W = 5 and threshold = 1e-06).

Hidden layers = [512]

accuracy on train data: 0.831
metrics for train data:

	precision	recall	f1-score	support
0	0.97	0.99	0.98	1933
1	0.87	0.92	0.89	1863
2	0.70	0.82	0.75	1669
3	0.75	0.66	0.70	2286
4	0.87	0.81	0.84	2249
micro avg	0.83	0.83	0.83	10000
macro avg	0.83	0.84	0.83	10000
weighted avg	0.83	0.83	0.83	10000
samples avg	0.83	0.83	0.83	10000

accuracy on test data: 0.782
metrics for test data:

	precision	recall	f1-score	support
0	0.97	0.99	0.98	223
1	0.81	0.88	0.84	181
2	0.58	0.72	0.64	162
3	0.73	0.56	0.64	241
4	0.80	0.77	0.78	193
micro avg	0.78	0.78	0.78	1000
macro avg	0.78	0.79	0.78	1000
weighted avg	0.78	0.78	0.78	1000
samples avg	0.78	0.78	0.78	1000

Hidden layers = [512,256]

accuracy on train data: 0.8206
metrics for train data:

	precision	recall	f1-score	support
0	0.97	0.99	0.98	1943
1	0.91	0.90	0.90	1997
2	0.71	0.83	0.76	1661
3	0.58	0.68	0.63	1733
4	0.93	0.73	0.82	2666
micro avg	0.82	0.82	0.82	10000
macro avg	0.82	0.82	0.82	10000
weighted avg	0.84	0.82	0.82	10000
samples avg	0.82	0.82	0.82	10000

accuracy on test data: 0.78
metrics for test data:

	precision	recall	f1-score	support
0	0.98	0.99	0.98	226
1	0.88	0.84	0.86	206
2	0.59	0.76	0.66	154
3	0.57	0.57	0.57	186
4	0.85	0.70	0.77	228
micro avg	0.78	0.78	0.78	1000
macro avg	0.77	0.77	0.77	1000
weighted avg	0.79	0.78	0.78	1000
samples avg	0.78	0.78	0.78	1000

Hidden layers = [512,256,128]

accuracy on train data: 0.8138
metrics for train data:

	precision	recall	f1-score	support
0	0.99	0.98	0.99	1996
1	0.88	0.94	0.90	1853
2	0.68	0.80	0.74	1651
3	0.58	0.64	0.61	1809
4	0.94	0.73	0.82	2691
micro avg	0.81	0.81	0.81	10000
macro avg	0.81	0.82	0.81	10000
weighted avg	0.83	0.81	0.82	10000
samples avg	0.81	0.81	0.81	10000

accuracy on test data: 0.778
metrics for test data:

	precision	recall	f1-score	support
0	1.00	0.99	0.99	230
1	0.82	0.89	0.85	182
2	0.58	0.72	0.65	160
3	0.60	0.56	0.58	201
4	0.86	0.70	0.77	227
micro avg	0.78	0.78	0.78	1000
macro avg	0.77	0.77	0.77	1000
weighted avg	0.79	0.78	0.78	1000
samples avg	0.78	0.78	0.78	1000

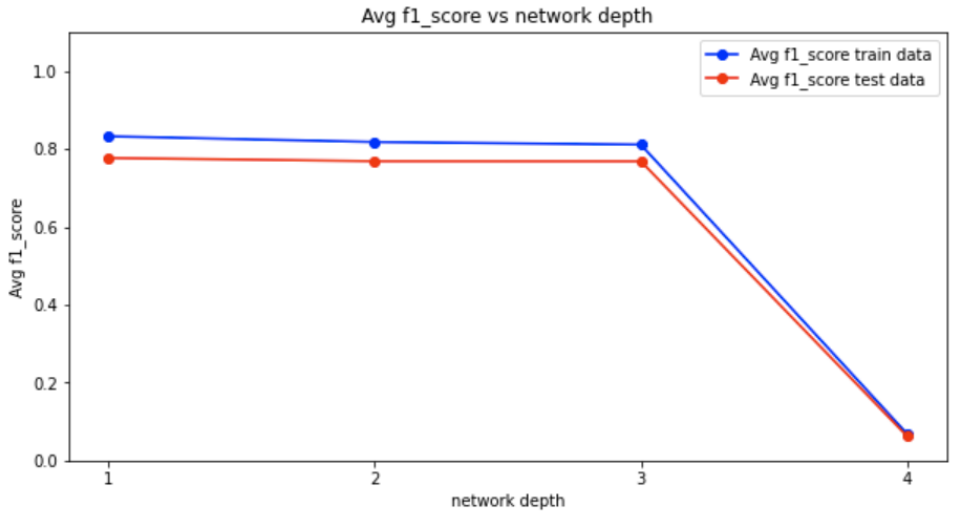
Hidden layers = [512,256,128,64]

accuracy on train data: 0.2091
metrics for train data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.21	0.35	10000
micro avg	0.21	0.21	0.21	10000
macro avg	0.20	0.04	0.07	10000
weighted avg	1.00	0.21	0.35	10000
samples avg	0.21	0.21	0.21	10000

accuracy on test data: 0.187
metrics for test data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.19	0.32	1000
micro avg	0.19	0.19	0.19	1000
macro avg	0.20	0.04	0.06	1000
weighted avg	1.00	0.19	0.32	1000
samples avg	0.19	0.19	0.19	1000



It could be observed that average f1_score is constant almost the same for three depths and then it drops when depth is 4. Adding more layers beyond a limit might make the model learn irregularities. In this case model have only learned to predict label 5.

(d)

Experiment: Trying out different depths with decreasing number of hidden nodes with adaptive learning rate

Mini-batch size = 32

Learning rate, α_0 = 0.01

Learning rate at epoch e = $\frac{\alpha_0}{\sqrt{e}}$

n = 1024 (16 x 16 x 4)

Hidden layers = [[512], [512,256], [512,256,128], [512,256,128,64]]

r = 5

Stopping criteria: Moving average of softmax loss (W = 5 and threshold = 5e-06). Since the parameter will take smaller steps over time, threshold is relaxed little.

Results

Hidden layers = [512]

accuracy on train data: 0.7705

metrics for train data:

	precision	recall	f1-score	support
0	0.97	0.94	0.95	2031
1	0.81	0.83	0.82	1937
2	0.65	0.70	0.68	1809
3	0.57	0.62	0.59	1849
4	0.85	0.75	0.80	2374
micro avg	0.77	0.77	0.77	10000
macro avg	0.77	0.77	0.77	10000
weighted avg	0.78	0.77	0.77	10000
samples avg	0.77	0.77	0.77	10000

accuracy on test data: 0.769

metrics for test data:

	precision	recall	f1-score	support
0	0.97	0.96	0.96	232
1	0.80	0.81	0.81	194
2	0.61	0.69	0.65	176
3	0.63	0.59	0.61	199
4	0.80	0.75	0.77	199
micro avg	0.77	0.77	0.77	1000
macro avg	0.76	0.76	0.76	1000
weighted avg	0.77	0.77	0.77	1000
samples avg	0.77	0.77	0.77	1000

Hidden layers = [512,256]

accuracy on train data: 0.779

metrics for train data:

	precision	recall	f1-score	support
0	0.98	0.96	0.97	1996
1	0.84	0.86	0.85	1951
2	0.66	0.72	0.69	1801
3	0.56	0.61	0.58	1850
4	0.85	0.74	0.79	2402
micro avg	0.78	0.78	0.78	10000
macro avg	0.78	0.78	0.78	10000
weighted avg	0.79	0.78	0.78	10000
samples avg	0.78	0.78	0.78	10000

accuracy on test data: 0.78

metrics for test data:

	precision	recall	f1-score	support
0	0.97	0.98	0.98	228
1	0.83	0.83	0.83	199
2	0.63	0.71	0.67	177
3	0.62	0.60	0.61	193
4	0.80	0.74	0.77	203
micro avg	0.78	0.78	0.78	1000
macro avg	0.77	0.77	0.77	1000
weighted avg	0.78	0.78	0.78	1000
samples avg	0.78	0.78	0.78	1000

Hidden layers = [512,256,128]

accuracy on train data: 0.7838

metrics for train data:

	precision	recall	f1-score	support
0	0.98	0.97	0.98	1991
1	0.85	0.87	0.86	1923
2	0.70	0.72	0.71	1914
3	0.54	0.62	0.58	1762
4	0.85	0.73	0.79	2410
micro avg	0.78	0.78	0.78	10000
macro avg	0.78	0.78	0.78	10000
weighted avg	0.79	0.78	0.79	10000
samples avg	0.78	0.78	0.78	10000

accuracy on test data: 0.783

metrics for test data:

	precision	recall	f1-score	support
0	0.99	0.99	0.99	229
1	0.83	0.85	0.84	194
2	0.68	0.71	0.70	192
3	0.57	0.60	0.58	176
4	0.80	0.72	0.76	209
micro avg	0.78	0.78	0.78	1000
macro avg	0.77	0.77	0.77	1000
weighted avg	0.79	0.78	0.78	1000
samples avg	0.78	0.78	0.78	1000

Hidden layers = [512,256,128,64]

accuracy on train data: 0.2091

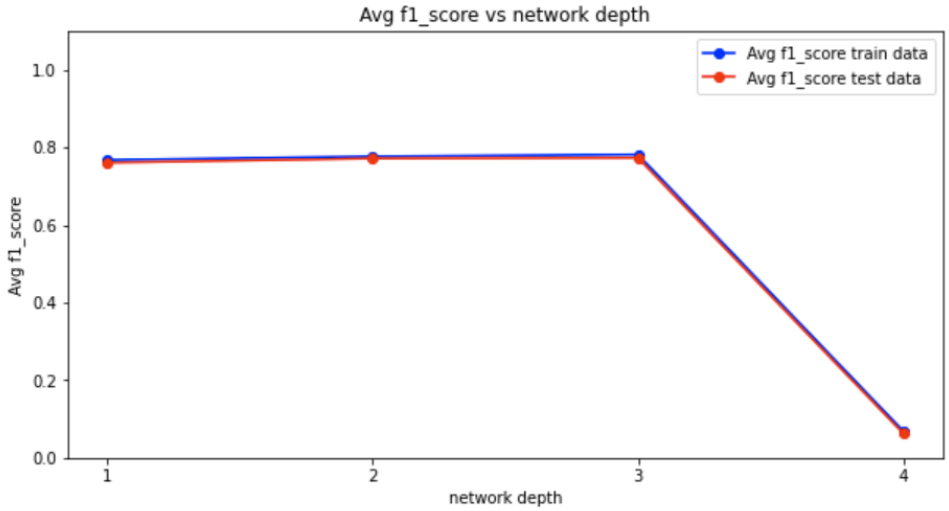
metrics for train data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.21	0.35	10000
micro avg	0.21	0.21	0.21	10000
macro avg	0.20	0.04	0.07	10000
weighted avg	1.00	0.21	0.35	10000
samples avg	0.21	0.21	0.21	10000

accuracy on test data: 0.187

metrics for test data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.19	0.32	1000
micro avg	0.19	0.19	0.19	1000
macro avg	0.20	0.04	0.06	1000
weighted avg	1.00	0.19	0.32	1000
samples avg	0.19	0.19	0.19	1000



It could be observed that average f1_score overlaps for both training and testing data which is an indication that model is generalizing well. Here also we had the same problem with depth=4 as in the previous part. Adaptive learning rate had made the training faster, thereby increased the test accuracy at depth=3.

(e)

Experiment: Trying out different depths with decreasing number of hidden nodes with adaptive learning rate. **Activation function used is relu**

Mini-batch size = 32

Learning rate, α_0 = 0.01

Learning rate at epoch e = $\frac{\alpha_0}{\sqrt{e}}$

n = 1024 (16 x 16 x 4)

Hidden layers = [[512], [512,256], [512,256,128], [512,256,128,64]]

r = 5

Stopping criteria: Moving average of softmax loss (W = 5 and threshold = 5e-06). Since the parameter will take smaller steps over time, threshold is relaxed little.

Results

Hidden layers = [512]

accuracy on train data: 0.2091

metrics for train data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.21	0.35	10000
micro avg	0.21	0.21	0.21	10000
macro avg	0.20	0.04	0.07	10000
weighted avg	1.00	0.21	0.35	10000
samples avg	0.21	0.21	0.21	10000

accuracy on test data: 0.187

metrics for test data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.19	0.32	1000
micro avg	0.19	0.19	0.19	1000
macro avg	0.20	0.04	0.06	1000
weighted avg	1.00	0.19	0.32	1000
samples avg	0.19	0.19	0.19	1000

Hidden layers = [512,256]

accuracy on train data: 0.2091
metrics for train data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.21	0.35	10000
micro avg	0.21	0.21	0.21	10000
macro avg	0.20	0.04	0.07	10000
weighted avg	1.00	0.21	0.35	10000
samples avg	0.21	0.21	0.21	10000

accuracy on test data: 0.187
metrics for test data:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.00	0.00	0.00	0
2	0.00	0.00	0.00	0
3	0.00	0.00	0.00	0
4	1.00	0.19	0.32	1000
micro avg	0.19	0.19	0.19	1000
macro avg	0.20	0.04	0.06	1000
weighted avg	1.00	0.19	0.32	1000
samples avg	0.19	0.19	0.19	1000

Hidden layers = [512,256,128]

accuracy on train data: 0.8045
metrics for train data:

	precision	recall	f1-score	support
0	0.98	0.99	0.99	1966
1	0.87	0.93	0.90	1845
2	0.66	0.80	0.72	1624
3	0.59	0.61	0.60	1932
4	0.91	0.72	0.80	2633
micro avg	0.80	0.80	0.80	10000
macro avg	0.80	0.81	0.80	10000
weighted avg	0.82	0.80	0.81	10000
samples avg	0.80	0.80	0.80	10000

accuracy on test data: 0.762
metrics for test data:

	precision	recall	f1-score	support
0	0.98	0.99	0.98	226
1	0.80	0.88	0.84	179
2	0.57	0.70	0.63	164
3	0.57	0.54	0.55	199
4	0.85	0.69	0.76	232
micro avg	0.76	0.76	0.76	1000
macro avg	0.75	0.76	0.75	1000
weighted avg	0.77	0.76	0.76	1000
samples avg	0.76	0.76	0.76	1000

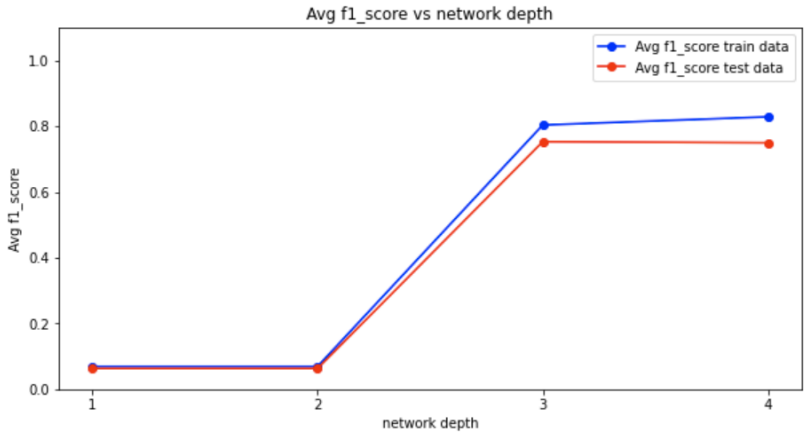
Hidden layers = [512,256,128,64]

accuracy on train data: 0.832
metrics for train data:

	precision	recall	f1-score	support
0	0.99	0.98	0.98	1988
1	0.93	0.91	0.92	2034
2	0.79	0.80	0.80	1932
3	0.57	0.71	0.63	1602
4	0.88	0.76	0.81	2444
micro avg	0.83	0.83	0.83	10000
macro avg	0.83	0.83	0.83	10000
weighted avg	0.85	0.83	0.84	10000
samples avg	0.83	0.83	0.83	10000

accuracy on test data: 0.761
metrics for test data:

	precision	recall	f1-score	support
0	0.97	0.98	0.97	225
1	0.86	0.81	0.83	210
2	0.62	0.68	0.65	182
3	0.52	0.56	0.54	174
4	0.80	0.71	0.75	209
micro avg	0.76	0.76	0.76	1000
macro avg	0.75	0.75	0.75	1000
weighted avg	0.77	0.76	0.76	1000
samples avg	0.76	0.76	0.76	1000



It could be observed that average f1_score is very low for smaller depth and its significant at depths more than 3. There has been a offset b/w training accuracy and testing accuracy which is an indicator of overfitting. Test accuracy with sigmoid activation function was better than that of relu activation function. Sigmoid performs relatively better at this multiclass classification task.

Repeated the above experiment with initial learning rate to be 0.001

Hidden layers = [512]

accuracy on train data: 0.7493

metrics for train data:

	precision	recall	f1-score	support
0	0.95	0.91	0.93	2050
1	0.78	0.77	0.78	1990
2	0.64	0.66	0.65	1904
3	0.54	0.62	0.58	1748
4	0.84	0.76	0.80	2308
micro avg	0.75	0.75	0.75	10000
macro avg	0.75	0.74	0.75	10000
weighted avg	0.76	0.75	0.75	10000
samples avg	0.75	0.75	0.75	10000

accuracy on test data: 0.734

metrics for test data:

	precision	recall	f1-score	support
0	0.94	0.95	0.95	228
1	0.77	0.76	0.76	200
2	0.60	0.63	0.61	189
3	0.57	0.56	0.56	189
4	0.75	0.73	0.74	194
micro avg	0.73	0.73	0.73	1000
macro avg	0.73	0.72	0.73	1000
weighted avg	0.74	0.73	0.73	1000
samples avg	0.73	0.73	0.73	1000

Hidden layers = [512, 256]

accuracy on train data: 0.8419

metrics for train data:

	precision	recall	f1-score	support
0	0.98	0.98	0.98	1956
1	0.88	0.91	0.90	1910
2	0.77	0.81	0.78	1857
3	0.69	0.71	0.70	1963
4	0.89	0.80	0.84	2314
micro avg	0.84	0.84	0.84	10000
macro avg	0.84	0.84	0.84	10000
weighted avg	0.84	0.84	0.84	10000
samples avg	0.84	0.84	0.84	10000

accuracy on test data: 0.803

metrics for test data:

	precision	recall	f1-score	support
0	0.97	1.00	0.98	224
1	0.85	0.86	0.86	196
2	0.69	0.75	0.72	182
3	0.64	0.62	0.63	191
4	0.83	0.75	0.79	207
micro avg	0.80	0.80	0.80	1000
macro avg	0.80	0.80	0.80	1000
weighted avg	0.80	0.80	0.80	1000
samples avg	0.80	0.80	0.80	1000

Hidden layers = [512, 256, 128]

accuracy on train data: 0.8618

metrics for train data:

	precision	recall	f1-score	support
0	0.97	0.99	0.98	1922
1	0.88	0.93	0.91	1872
2	0.82	0.83	0.82	1927
3	0.71	0.76	0.74	1893
4	0.92	0.81	0.86	2386
micro avg	0.86	0.86	0.86	10000
macro avg	0.86	0.86	0.86	10000
weighted avg	0.86	0.86	0.86	10000
samples avg	0.86	0.86	0.86	10000

accuracy on test data: 0.816

metrics for test data:

	precision	recall	f1-score	support
0	0.97	0.99	0.98	224
1	0.83	0.89	0.86	186
2	0.70	0.76	0.73	182
3	0.72	0.64	0.67	211
4	0.83	0.79	0.81	197
micro avg	0.82	0.82	0.82	1000
macro avg	0.81	0.81	0.81	1000
weighted avg	0.81	0.82	0.81	1000
samples avg	0.82	0.82	0.82	1000

Hidden layers = [512, 256, 128, 64]

accuracy on train data: 0.8908

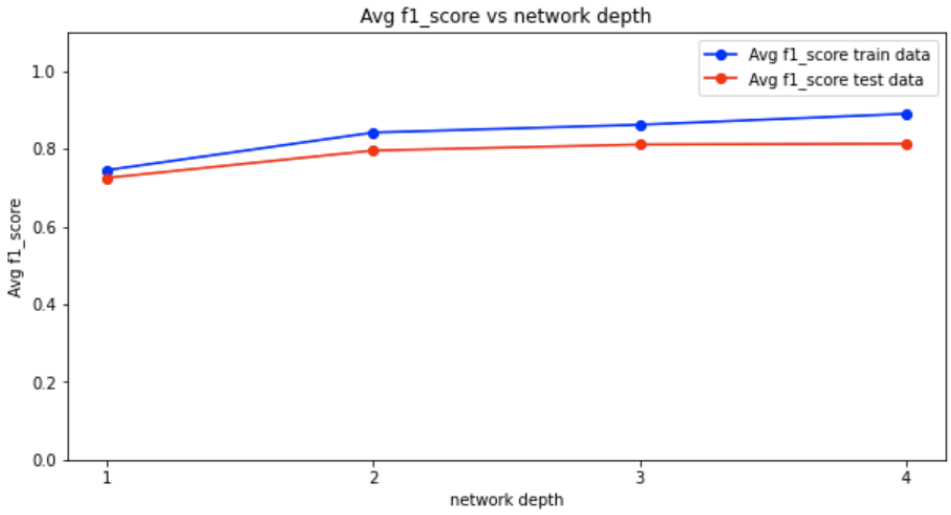
metrics for train data:

	precision	recall	f1-score	support
0	0.99	0.98	0.99	1990
1	0.93	0.95	0.94	1941
2	0.85	0.87	0.86	1903
3	0.74	0.81	0.78	1832
4	0.94	0.84	0.89	2334
micro avg	0.89	0.89	0.89	10000
macro avg	0.89	0.89	0.89	10000
weighted avg	0.89	0.89	0.89	10000
samples avg	0.89	0.89	0.89	10000

accuracy on test data: 0.82

metrics for test data:

	precision	recall	f1-score	support
0	1.00	0.98	0.99	232
1	0.87	0.91	0.89	191
2	0.71	0.80	0.76	177
3	0.67	0.63	0.65	201
4	0.81	0.76	0.78	199
micro avg	0.82	0.82	0.82	1000
macro avg	0.81	0.82	0.81	1000
weighted avg	0.82	0.82	0.82	1000
samples avg	0.82	0.82	0.82	1000



It can be observed that the training increases whereas the testing accuracy saturates which is a classic case of model overfitting. Hence, we can either use the depth=2 model or the depth=3 model.

(f)

Neural Networks using scikit-learn

```
hidden_layers = [[512], [512,256], [512,256,128], [512,256,128,64]]
network_depth = [1,2,3,4]
f1_score_train = []
f1_score_test = []
for hidden_layer in hidden_layers:
    print(f"Hidden layer: {hidden_layer}")

    clf = MLPClassifier(activation="relu", solver="sgd", alpha = 0, batch_size=32,
hidden_layer_sizes=np.array(hidden_layer), learning_rate="invscaling", tol=5e-6, n_iter_no_change=5,
verbose=True, learning_rate_init=0.01).fit(X_train, y_train_onehot)

    y_pred_train = clf.predict(X_train)
    y_pred_test = clf.predict(X_test)

    print("accuracy on train data: ",accuracy_score(y_train_onehot, y_pred_train))
    print("metrics for train data: ")
    get_metric(y_train_onehot, y_pred_train)
    f1_score_train.append(f1_score(y_train_onehot, y_pred_train, average="macro"))

    print("accuracy on test data: ",accuracy_score(y_test_onehot, y_pred_test))
    print("metrics for test data: ")
    get_metric(y_test_onehot, y_pred_test)
    f1_score_test.append(f1_score(y_test_onehot, y_pred_test, average="macro"))

    print("\n")
```

Activation function: relu

Solver: sgd (stochastic gradient descent)

Alpha: 0 (L2 regularization term)

Batch size:32

Learning rate init: 0.01

Learning rate: invscaling

Stopping criteria:

Tol(tolerance for optimization): 5e-6

n_iter_no_change = 5

Hidden layers = [512]

accuracy on train data: 0.3589

metrics for train data:

	precision	recall	f1-score	support
0	0.88	0.87	0.88	1990
1	0.21	0.75	0.33	557
2	0.03	0.58	0.07	117
3	0.00	0.00	0.00	0
4	0.66	0.73	0.69	1890
micro avg	0.36	0.79	0.49	4554
macro avg	0.36	0.59	0.39	4554
weighted avg	0.68	0.79	0.71	4554
samples avg	0.36	0.36	0.36	4554

accuracy on test data: 0.363

metrics for test data:

	precision	recall	f1-score	support
0	0.88	0.89	0.89	226
1	0.20	0.71	0.31	56
2	0.05	0.60	0.08	15
3	0.00	0.00	0.00	0
4	0.60	0.69	0.65	163
micro avg	0.36	0.79	0.50	460
macro avg	0.35	0.58	0.39	460
weighted avg	0.67	0.79	0.71	460
samples avg	0.36	0.36	0.36	460

Hidden layers = [512, 256]

accuracy on train data: 0.5267

metrics for train data:

	precision	recall	f1-score	support
0	0.89	0.88	0.89	2008
1	0.64	0.70	0.67	1806
2	0.35	0.59	0.44	1168
3	0.06	0.56	0.11	227
4	0.68	0.73	0.70	1967
micro avg	0.53	0.73	0.61	7176
macro avg	0.53	0.69	0.56	7176
weighted avg	0.66	0.73	0.68	7176
samples avg	0.53	0.53	0.53	7176

accuracy on test data: 0.521

metrics for test data:

	precision	recall	f1-score	support
0	0.90	0.90	0.90	229
1	0.62	0.69	0.65	178
2	0.34	0.61	0.44	112
3	0.06	0.50	0.11	22
4	0.61	0.70	0.65	164
micro avg	0.52	0.74	0.61	705
macro avg	0.51	0.68	0.55	705
weighted avg	0.65	0.74	0.68	705
samples avg	0.52	0.52	0.52	705

Hidden layers = [512, 256, 128]

accuracy on train data: 0.5642

metrics for train data:

	precision	recall	f1-score	support
0	0.89	0.88	0.88	1985
1	0.67	0.70	0.68	1889
2	0.42	0.59	0.49	1384
3	0.14	0.54	0.23	531
4	0.71	0.71	0.71	2067
micro avg	0.56	0.72	0.63	7856
macro avg	0.56	0.68	0.60	7856
weighted avg	0.65	0.72	0.68	7856
samples avg	0.56	0.56	0.56	7856

accuracy on test data: 0.562

metrics for test data:

	precision	recall	f1-score	support
0	0.90	0.91	0.90	225
1	0.64	0.68	0.66	184
2	0.40	0.61	0.48	130
3	0.18	0.54	0.27	63
4	0.63	0.68	0.66	173
micro avg	0.56	0.73	0.63	775
macro avg	0.55	0.69	0.59	775
weighted avg	0.63	0.73	0.67	775
samples avg	0.56	0.56	0.56	775

Hidden layers = [512, 256, 128, 64]

accuracy on train data: 0.5846

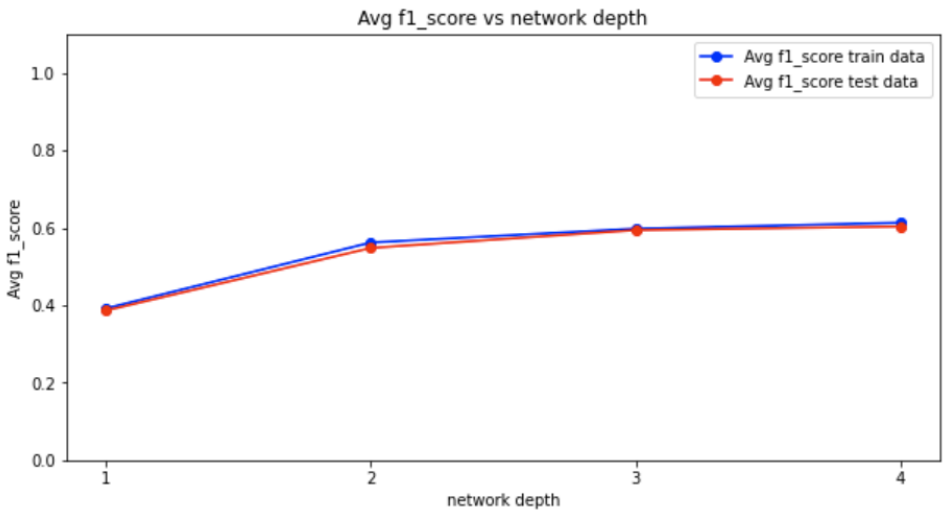
metrics for train data:

	precision	recall	f1-score	support
0	0.89	0.88	0.88	1997
1	0.68	0.69	0.69	1958
2	0.46	0.58	0.51	1568
3	0.18	0.53	0.27	688
4	0.71	0.71	0.71	2064
micro avg	0.58	0.71	0.64	8275
macro avg	0.58	0.68	0.61	8275
weighted avg	0.66	0.71	0.67	8275
samples avg	0.58	0.58	0.58	8275

accuracy on test data: 0.581

metrics for test data:

	precision	recall	f1-score	support
0	0.90	0.90	0.90	228
1	0.66	0.68	0.67	194
2	0.44	0.59	0.51	149
3	0.20	0.45	0.28	85
4	0.64	0.69	0.66	172
micro avg	0.58	0.70	0.64	828
macro avg	0.57	0.66	0.60	828
weighted avg	0.64	0.70	0.66	828
samples avg	0.58	0.58	0.58	828



Part (e) results were better than part (f) results in terms of accuracy and F1 score. It was observed that the scikit implementation of the MLP Classifier takes small steps due to the learning rate as “invscaling” hence taking longer time to converge, thereby reaching the max_iter allowed and stopping before the actual convergence.