<u>**Question** 1</u>

Data and Preprocessing:

- We download the dataset and filter out samples to retain only the samples belonging to the classes ['comp.graphics','sci.med','talk.politics.misc','rec.sport.hockey','sci.space'] as mentioned in the instructions.
- We wrote a custom train test split method as below

```
def train_data_test_data_split(data, split_factor):
    num_train_data = int(split_factor * data.shape[0])
    data_indices = range(data.shape[0])
    data_indices_shuffled = np.random.permutation(data_indices)
    train_data,test_data = data.iloc[data_indices_shuffled[:num_train_data]],data.iloc[data_indices_shuffled[num_train_data:]]
    return train_data, test_data
```

From the code we see that random shuffling as been used to permute the indices to avoid overfitting to the ordering of the samples.

- We preprocess each sentence as follows:
  - First we clean up the meta-data present in each file as they are not very useful in discriminating between the samples to classify them to the appropriate class.

```
text = re.sub(r'(([\sA-Za-z0-9\-]+)?[A|a]rchive-name:[^\n]+\n)', '', text)
text = re.sub(r'(Last-modified:[^\n]+\n)', '', text)
text = re.sub(r'(Version:[^\n]+\n)', '', text)
text = re.sub(r'(XXXMessageID::\s+[^\n]+\n)', '', text)
text = re.sub(r'(XUserAgent:\s+[^\n]+\n)', '', text)
text = re.sub(r'(Message-ID:\s+[^\n]+\n)', '', text)
text = re.sub(r'(X-Newsreader:\s+[^\n]+\n)', '', text)
text = re.sub(r'(References:\s+[^\n]+\n)', '', text)
text = re.sub(r'(Organization:\s+[^\n]+\n)', '', text)
text = re.sub(r'(article:\s+[^\n]+\n)', '', text)
```

- Then we perform other preprocessing steps such as punctuation removal, domain url removal , stop words removal followed by lemmatization. We used nltk wordnet lemmatizer to perform lemmatization

```
def clean_sentence(sentence):
    input_text = sentence.split("Lines:")
    if len(input_text)>1:
        input_text = sentence.split("Lines:")
        split_text = "Lines:"
    else:
        input_text = sentence.split("Date:")
        split_text = "Date:"
    sentence = input_text[1]
    sentence = sentence.lower()
    sentence = sentence.strip()
    sentence = re.sub(re_url, '', sentence)
    sentence = re.sub(re_email, '', sentence)
    sentence = re.sub(f'[{re.escape(string.punctuation)}]', '', sentence)
    sentence = re.sub(r'(\d+)', ' ', sentence)
    sentence = re.sub(r'(\s+)', ' ', sentence)
    sentence = lemmatizer(sentence)
    return sentence
```

-

- As mentioned earlier we save the preprocessed splits and store them in csv files

Feature selection:

We implement the mentioned TF-ICF (a variant of TF-IDF) to help in feature selection

Traditional TF-IDF formulation is

$$TF\text{-}IDF = TF(t) * \log(N/df_t)$$

Where TF(t) is the number of times a term occurs in a document which is usually normalized by number of other terms in the document.

N - is total number of documents, df_t - is the number of documents that contain the term t.

Now TF-ICF is computed at class level than document level.

TF here is the number of occurrences of the terms occurring in documents belonging to a particular class.

ICF(t_ = log(N/CF(t))

Where N is the number of classes and CF(t) is the number of classes containing the particular term.

Then the terms are sorted according to their TF-ICF scores and top-k terms are retained for each class. These terms aid in multinomial naive bayes classification. K is a tunable parameter and is received as input from the user.

```
def derivetficf_and_reduce_features(class_term_freq,term_to_class_dict,unique_labels):
    tf_icf_scores = np.array([dict() for category in unique_labels])
    selected_terms_for_each_class = np.array([dict() for category in unique_labels])
    for index, category in enumerate(unique_labels):
        terms_with_frequencies = class_term_freq[category]
        tf_icf = dict()
        for term in terms_with_frequencies.keys():
            icf = np.log(len(unique_labels)/len(term_to_class_dict[term]))
            tf_icf[term] = terms_with_frequencies[term] * icf
        tf_icf_scores[category] = tf_icf
    for  index, category in enumerate(unique_labels):
        terms_with_frequencies = class_term_freq[category]

        tf_icf_for_class = tf_icf_scores[category]
        terms = tf_icf_for_class.keys()
        tf_icf_scores_with_terms = sorted(list(zip(tf_icf_for_class.values(),terms)),reverse=True)[:top_k]
        top_terms = [value[1] for value in tf_icf_scores_with_terms]
        count_values = [terms_with_frequencies[term] for term in top_terms]
        selected_terms_for_each_class[category] = dict(zip(top_terms,count_values))

    return selected_terms_for_each_class
```

Where class_term_freq contains a mapping from the class to the terms in the class along with the number of occurrences of the terms in each class.

Naive Bayes:

We implement multinomial Naive Bayes. **Multinomial naive bayes** which uses bag of words was implemented. At train time we first tokenize the sentences in the train set and find the count of words for the top selected terms from the feature selection step for each class . Then the word counts were summed across samples to obtain the count of each word in each class (positive and negative). This would help us in calculating the likelihood as

$P(c_i|D) = P(c_i)\ P(x_1|c_i) * P(x_2|c_i)$......

At train time the prior probabilities and the bag of words with word counts for each class are stored, At test time we tokenize the input the same way and compute the posterior by using the bag of words and prior probabilities computed.

```
def train_naive_bayes(labels,unique_classes, bag_of_words):

    prior_prob = np.empty(len(labels))
    vocab = []
    words_in_label = np.empty(len(labels))
    for label_index,label in enumerate(unique_classes):
        prior_prob[label_index] = np.sum(labels == label) / float(len(labels))
        words_in_label[label_index]=np.sum(np.array(list(bag_of_words[label_index].values())))
        vocab += bag_of_words[label_index].keys()
    vocab = np.unique(np.array(vocab))
    likelihood_dividor = np.array([words_in_label[label_index]+len(vocab) for label_index, label in enumerate(uniqu
    lookup_table =[]
    for label_index, label in enumerate(unique_classes):
        lookup_table.append((bag_of_words[label_index],prior_prob[label_index],likelihood_dividor[label_index]))
    return lookup_table
```

```
def test_naive_bayes(sentence, lookup_table):
    likelihood = np.zeros(len(unique_classes))
    words = [str(word) for word in sentence.split(" ")]
    for label_index,index in enumerate(unique_classes):
        for word in words:
            # likelihood = count(word)+1 / vocab + word_counts_in_class
            likelihood[label_index] += np.log((lookup_table[label_index][0].get(word,0)+1) /(float(lookup_table[lab
    posterior = np.empty(len(unique_classes))
    for label_index,label in enumerate(unique_classes):
        posterior[label_index] = np.log(lookup_table[label_index][1])+ likelihood[label_index]
    return unique_classes[np.argmax(posterior)]
```

We try with different values for selecting top k features and results are present in analysis.

Question - 2:

Part - 1:

Preprocessing:

We chose Wiki dataset to analyse a directed graph. Removed the first 4 lines which includes comment about the dataset.

Formulae:

Average in and out degree:
Avg = 1/N * sum(degree)

Degree Centrality measure has been adopted for with the formula is:
in_DCi = in_deg(i)/max_in_deg
out_DCi = out_deg(i)/max_out_deg

Local Clustering Coefficient

Ci = 1/k*(k-1) * sum(j,k) A(ij) A(jk) A(ki)

Where A is the adjacency matrix,
i is the vertex for the clustering coefficient is being calculated
j,k is the pair of vertices that are neighbouring the vertex i
k is the number of neighbour vertices of vertex i

Part 2
We execute the pagerank and hits algorithm. We first form a directed graph using networkx and run the PageRank and HITS algorithms using inbuilt methods.

```
def read_graph():
    fb_graph = nx.read_edgelist("Wiki-Vote.txt",  create_using=nx.DiGraph, nodetype=int)
    print("fb_graph",fb_graph.edges)
    return fb_graph

def plot_graph(graph):
    ig = Graph.TupleList(graph.edges, directed=True)
    plot(ig)

def compute_pagerank(fb_graph):
    pagerank_scores = nx.pagerank(fb_graph,alpha=0.85,tol=0.0001)
    with open("pagerank.json","w") as f:
        json.dump(pagerank_scores,f)
    print("top 5 nodes with pagerank scores are:",sorted([(b, a) for a, b in pagerank_scores.items()], reverse=True)[:5])

def compute_hits(fb_graph):
    hits_scores = nx.hits(fb_graph,tol=0.0001)
    authorities_scores = hits_scores[1]
    hub_scores = hits_scores[0]
    with open("hits_hubs.json","w") as f:
        json.dump(hub_scores,f)
    with open("hits_authorities.json","w") as f:
        json.dump(authorities_scores,f)
    print("top 5 nodes according to authorities scores are:",sorted([(b, a) for a, b in authorities_scores.items()], reverse=Tru
    print("top 5 nodes according to hub scores are:",sorted([(b, a) for a, b in hub_scores.items()], reverse=True)[:5])

if __name__ == "__main__":
    fb_graph = read_graph()
    print("Summary information of the graph is:", nx.info(fb_graph))
    compute_pagerank(fb_graph)
    compute_hits(fb_graph)
    plot_graph(fb_graph)
```

```
Summary information of the graph is: Name:
Type: DiGraph
Number of nodes: 7115
Number of edges: 103689
```

Unlike Pagerank HITS has two notion of pages importance.

1) Authorities: Where certain nodes are valuable as they provide information.
2) Hubs: They point to useful nodes or pages that provide useful information.

So hub scores are calculated according to outgoing links and authorities are computed according ot incoming links

Hubbiness of a page is proportional to sum of authority of it’s successors and authority of a page is sum of hubbiness of it’s predecessors. The values may grow unboundedly and hence has to be normalized. But in pagerank the Pagerank of a node is divided equally to it’s outgoing links an dance needs no normalization. Also pagerank adopts a random walker model. Real surfers may not follow the random walk model.