

Project Report

Lab2

Gajam Anurag

Task 1 – Artificial Neural Network

Dataset: CIFAR10

Data Description: This dataset consists of 60,000 images of 10 different classes, 6,000 images per class and 50,000 (5,000 per class in training) and 10,000 (1,000 per class in testing). Here are class names airplane, automobile, bird, cat, deer, dog, frog, horse, ship and truck

Data Analysis and Visualization:

a. Number of entities in training and testing set and number of classes in target variable

```
[ ] print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)

(50000, 32, 32, 3) (10000, 32, 32, 3) (50000, 1) (10000, 1)
```

```
[ ] classes = np.unique(y_train)
    classes

array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
```

b. Number of pixels in the image

There are 50,000 images with pixel dimension of 32x32x3

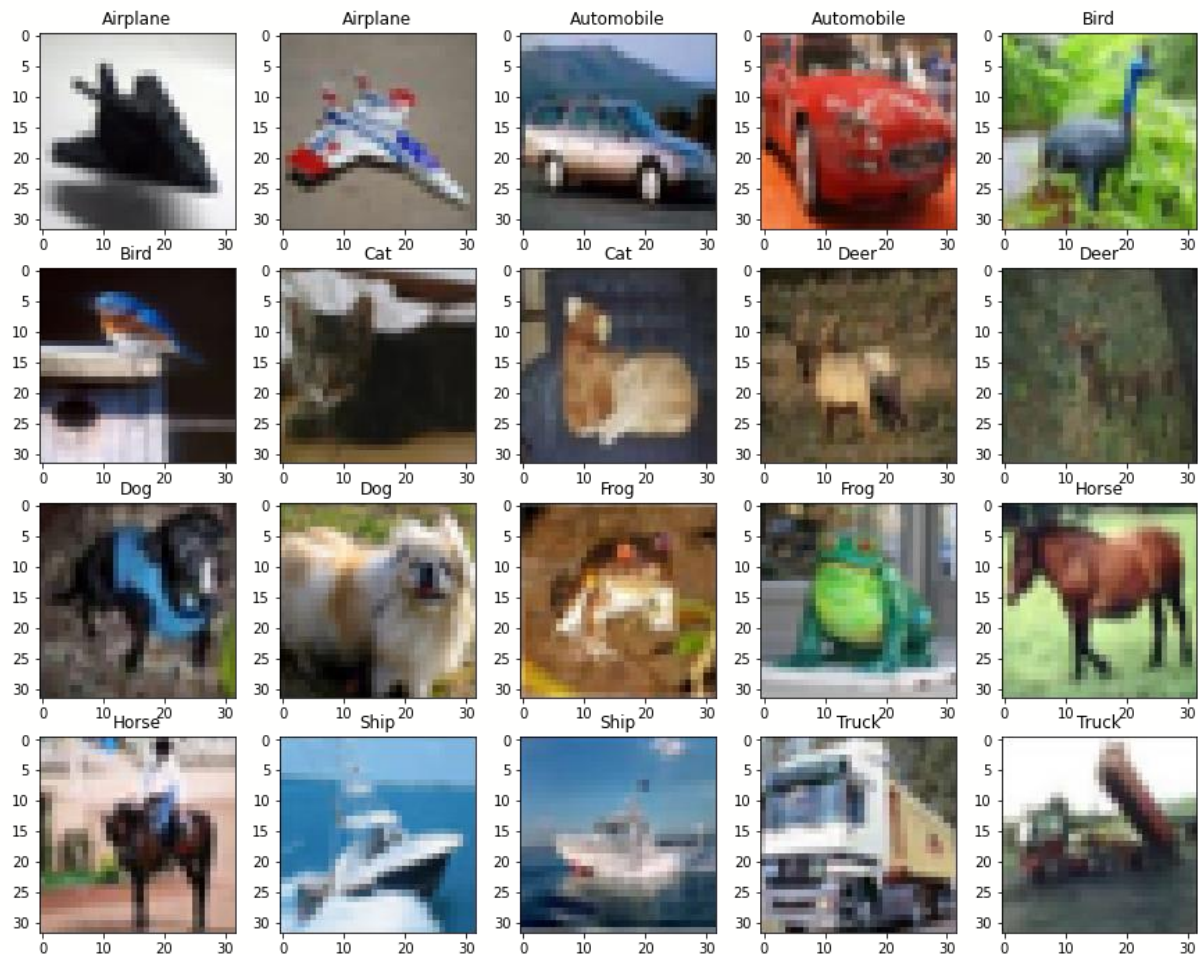
```
[ ] x_train.shape

(50000, 32, 32, 3)
```

c. Number of images in training and test set

train data	In test data
0 : 5000	0 : 1000
1 : 5000	1 : 1000
2 : 5000	2 : 1000
3 : 5000	3 : 1000
4 : 5000	4 : 1000
5 : 5000	5 : 1000
6 : 5000	6 : 1000
7 : 5000	7 : 1000
8 : 5000	8 : 1000
9 : 5000	9 : 1000

d. Two Image from each class



Data Pre-processing:

Here I have one hot encoded the x values using `to_categorical` and converted the datatype to int type. Then I have divided the values with 255 to normalize the values to range 0 to 1. Then I have reshaped the train and test values to single dimension.

```
[ ] y_train = to_categorical(y_train, 10)
    y_test = to_categorical(y_test, 10)

    x_train = x_train.astype('float32')
    x_test = x_test.astype('float32')

    x_train /= 255
    x_test /= 255

    x_train = x_train.reshape((x_train.shape[0], 32*32*3))
    x_test = x_test.reshape((x_test.shape[0], 32*32*3))
```

Defining the Layers:

Here I have used 3 layers. 1 input layer, 2 hidden layer and 1 output layer. For input layer, I have given 128 neurons with input shape as (3072,). In the hidden layers, I have used the same number of neurons 128 and for output I have used 10 neurons as it is multiclassification problem with 10 different classes.

And also, I have tried with different activation functions and different regularization techniques. Below are the some that I have tried.

Model1: without any regularization and relu & softmax as activation functions

```
] ## without dropout
model1 = Sequential(Dense(128, input_shape = (3072,)), activation='relu'))
model1.add(Dense(128, activation='relu'))
model1.add(Dense(128,activation='relu'))
model1.add(Dense(10,activation='softmax'))
```

Model2: without any regularization and elu & sigmoid as activation functions

```
##without dropout
model2 = Sequential(Dense(128, input_shape = (3072,)), activation='elu'))
model2.add(Dense(128, activation='elu'))
model2.add(Dense(128,activation='elu'))
model2.add(Dense(10,activation='sigmoid'))
```

Model3: With dropout and relu & sigmoid activation functions

```
[ ] ## with dropout
model3 = Sequential(Dense(128, input_shape = (3072,)), activation='relu'))
model3.add(Dense(128, activation='relu'))
model3.add(Dropout(0.125))
model3.add(Dense(128,activation='relu'))
model3.add(Dense(10,activation='sigmoid'))
```

Model4: With dropout and elu & softmax activation functions.

```
## with dropout
model4 = Sequential(Dense(128, input_shape = (3072,)), activation='elu'))
model4.add(Dense(128, activation='elu'))
model4.add(Dropout(0.125))
model4.add(Dense(128,activation='elu'))
model4.add(Dropout(0.125))
model4.add(Dense(10,activation='softmax'))
```

Model5: Using l2 regularization and relu & sigmoid activation functions.

```
## with kernel_regularizer l2
model5 = Sequential(Dense(128,input_shape = (3072,), kernel_regularizer='l2', activation = 'relu' ))
model5.add(Dense(128, activation='relu',kernel_regularizer='l2'))
model5.add(Dense(128,activation='relu',kernel_regularizer='l2'))
model5.add(Dense(10,activation='sigmoid'))
```

Model6: Using l2 regularization and elu & softmax activation functions.

```
model6 = Sequential(Dense(128,input_shape = (3072,), kernel_regularizer='l2', activation = 'elu' ))
model6.add(Dense(128, activation='elu',kernel_regularizer='l2'))
model6.add(Dense(128,activation='elu',kernel_regularizer='l2'))
model6.add(Dense(10,activation='softmax'))
```

Model7: Using l1 regularization and relu & sigmoid activation functions

```
## with kernel_regularizer l1
model7 = Sequential(Dense(128,input_shape = (3072,), kernel_regularizer='l1', activation = 'relu' ))
model7.add(Dense(128, activation='relu',kernel_regularizer='l1'))
model7.add(Dense(128,activation='relu',kernel_regularizer='l1'))
model7.add(Dense(10,activation='sigmoid'))
```

Model8: Using l2 regularization and elu & sigmoid activation functions

```
model8 = Sequential(Dense(128,input_shape = (3072,), kernel_regularizer='l1', activation = 'elu' ))
model8.add(Dense(128, activation='elu',kernel_regularizer='l1'))
model8.add(Dense(128,activation='elu',kernel_regularizer='l1'))
model8.add(Dense(10,activation='sigmoid'))
```

Defining f1_score function:

```
▶ ## Creating f1_score function

import keras.backend as K

def get_f1(y_true, y_pred): #taken from old keras source code
    true_positives = K.sum(K.round(K.clip(y_true * y_pred, 0, 1)))
    possible_positives = K.sum(K.round(K.clip(y_true, 0, 1)))
    predicted_positives = K.sum(K.round(K.clip(y_pred, 0, 1)))
    precision = true_positives / (predicted_positives + K.epsilon())
    recall = true_positives / (possible_positives + K.epsilon())
    f1_val = 2*(precision*recall)/(precision+recall+K.epsilon())
    return f1_val
```

Here for every model, I have tried 3 different optimizers. Adam, SGD and Adamax. Out of these optimizers Adamax gave best result for almost all the models but the model4(With dropout and elu & softmax activation functions) gave best result out of all possible outcomes with f1_score 0.48.

On Train Data:

```
model4.compile(optimizer='adamax',loss="categorical_crossentropy", metrics=get_f1)
history12 = model4.fit(x_train,y_train, validation_split=0.25,epochs=20)

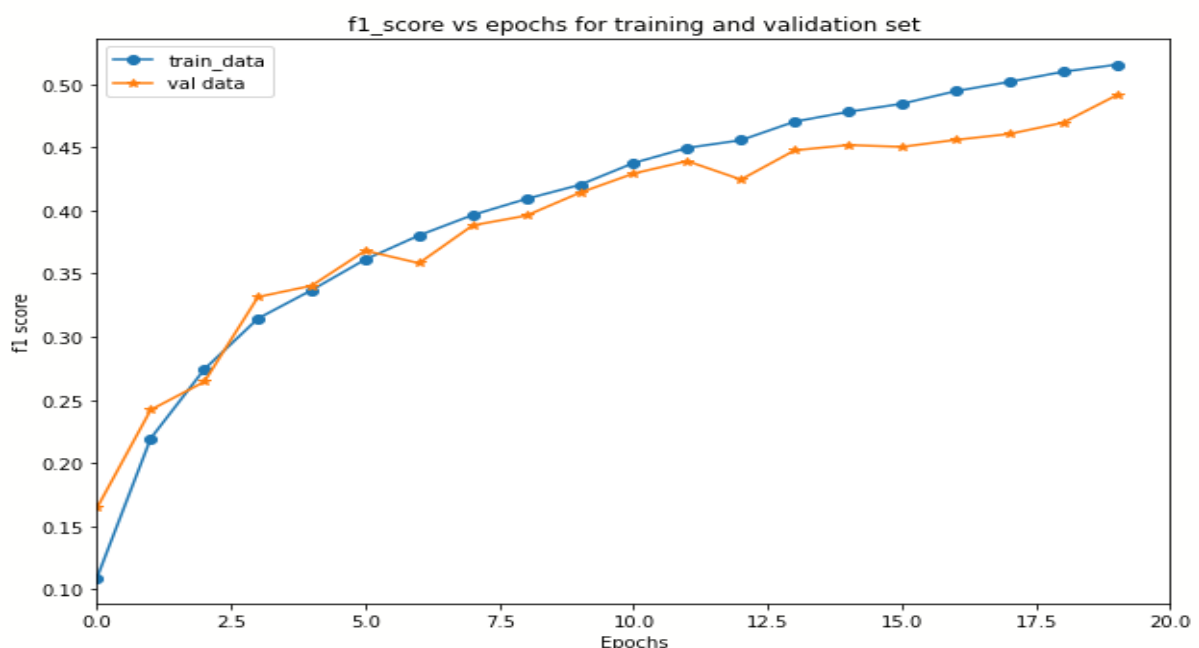
Epoch 1/20
1172/1172 [=====] - 9s 7ms/step - loss: 1.9298 - get_f1: 0.1092 - val_loss: 1.7637 - val_get_f1: 0.1645
Epoch 2/20
1172/1172 [=====] - 7s 6ms/step - loss: 1.7245 - get_f1: 0.2194 - val_loss: 1.6591 - val_get_f1: 0.2421
Epoch 3/20
1172/1172 [=====] - 7s 6ms/step - loss: 1.6417 - get_f1: 0.2742 - val_loss: 1.6089 - val_get_f1: 0.2644
Epoch 4/20
1172/1172 [=====] - 7s 6ms/step - loss: 1.5802 - get_f1: 0.3281 - val_loss: 1.5765 - val_get_f1: 0.3018
Epoch 5/20
1172/1172 [=====] - 7s 6ms/step - loss: 1.5266 - get_f1: 0.3781 - val_loss: 1.5438 - val_get_f1: 0.3454
Epoch 6/20
1172/1172 [=====] - 7s 6ms/step - loss: 1.4746 - get_f1: 0.4245 - val_loss: 1.5170 - val_get_f1: 0.3859
Epoch 7/20
1172/1172 [=====] - 7s 6ms/step - loss: 1.4290 - get_f1: 0.4618 - val_loss: 1.4917 - val_get_f1: 0.4206
Epoch 8/20
1172/1172 [=====] - 8s 7ms/step - loss: 1.3848 - get_f1: 0.4908 - val_loss: 1.4672 - val_get_f1: 0.4495
Epoch 9/20
1172/1172 [=====] - 10s 9ms/step - loss: 1.3469 - get_f1: 0.5154 - val_loss: 1.4432 - val_get_f1: 0.4712
```

On TestData:

```
[ ] test_loss,test_acc=model4.evaluate(x_test,y_test)
print("Test accuracy:",test_acc)

313/313 [=====] - 2s 5ms/step - loss: 1.3469 - get_f1: 0.4894
Test accuracy: 0.48937058448791504
```

Graph of f1_score vs epochs for training and validation set.



Trainable parameters for the model4 which gave best accuracy.

Model: "sequential_3"

Layer (type)	Output Shape	Param #
dense_12 (Dense)	(None, 128)	393344
dense_13 (Dense)	(None, 128)	16512
dropout_1 (Dropout)	(None, 128)	0
dense_14 (Dense)	(None, 128)	16512
dropout_2 (Dropout)	(None, 128)	0
dense_15 (Dense)	(None, 10)	1290
Total params: 427,658		
Trainable params: 427,658		
Non-trainable params: 0		

TASK 2 – NATURAL LANGUAGE PROCESSING

Natural Language Processing or NLP is a field of Artificial Intelligence that gives the machines the ability to read, understand and derive meaning from human languages.

Dataset: Movie Reviews

```
[57] sndata.head()
```

	review	sentiment
0	One of the other reviewers has mentioned that ...	positive
1	A wonderful little production. The...	positive
2	I thought this was a wonderful way to spend ti...	positive
3	Basically there's a family where a little boy ...	negative
4	Petter Mattei's "Love in the Time of Money" is...	positive



Data Pre-processing:

1. Removing Html tags

These html tags do not have any value to the text. So, here I have used **BeautifulSoup** to remove all the html tags from the sentences

```
## removing html tags first
soup = BeautifulSoup(sentence, "html.parser")
sentence = soup.get_text()
```

2. Removing all special characters


```
## removing all the special characters
sen_wo_sc = [x for x in sentence if x not in string.punctuation]
sen_wo_sc = ''.join(sen_wo_sc)
```

3. Converting text into lower case


```
## converting the text into lower case
sen_wo_sc = sen_wo_sc.lower()
```

After performing the above steps, we get the resultant DataFrame as shown in below

```
[60] sndata.head()
```



	review	sentiment
0	one of the other reviewers has mentioned that ...	positive
1	a wonderful little production the filming tech...	positive
2	i thought this was a wonderful way to spend ti...	positive
3	basically theres a family where a little boy j...	negative
4	petter matteis love in the time of money is a ...	positive




4. Removing URL's


Here I have used regular expression to remove the url's from the string

```
[62] ## Removing urls
def remove_urls(text):
    url_pattern = re.compile(r'https?://\S+|www\.\S+')
    return url_pattern.sub(r'', text)
```

```
[63] sndata['review'] = sndata['review'].apply(remove_urls)
sndata.head()
```



	review	sentiment
0	one of the other reviewers has mentioned that ...	positive
1	a wonderful little production the filming tech...	positive
2	i thought this was a wonderful way to spend ti...	positive
3	basically theres a family where a little boy j...	negative
4	petter matteis love in the time of money is a ...	positive



5. Performing text lemmatization:

Lemmatization converts all the words to their root words without missing its vocabulary. Here I have used WordNetLemmatization to perform the task.


```
[64] ## performing text lemmatization
```

```
def text_lemmatization(sentence):  
    lemmatizer = WordNetLemmatizer()  
    text = [lemmatizer.lemmatize(x) for x in sentence.split()]  
    text = ' '.join(text)  
    return text  
  
sndata['review'] = sndata['review'].apply(text_lemmatization)
```

```
[65] sndata.head()
```



	review	sentiment
0	one of the other reviewer ha mentioned that af...	positive
1	a wonderful little production the filming tech...	positive
2	i thought this wa a wonderful way to spend tim...	positive
3	basically there a family where a little boy ja...	negative
4	petter matteis love in the time of money is a ...	positive



6. Removing the stop words:

Final task is to remove all the stop words from the sentence. This is done to give more focus on importance information. Here I have used inbuilt corpus.stopwords to get all the stopwords and get removed from the sentence.

```
### removing the stop words  
tokenizer=ToktokTokenizer()  
stopword_list=nlk.corpus.stopwords.words('english')  
def remove_stop_words(sentence):  
  
    tokens = tokenizer.tokenize(sentence)  
    tokens = [token.strip() for token in tokens]  
  
    filtered_token = [token for token in tokens if token not in stopwords_list]  
  
    filtered_text = ' '.join(filtered_token)  
  
    return filtered_text  
  
sndata['review'] = sndata['review'].apply(remove_stop_words)
```

Vectorization: Here I have performed 2 Vectorization. CountVectorizer and TfidfVectorizer. And for each Vectorization I have performed 3 algorithms.

Declaring the CountVectorizer:

```
[ ] vectorizer = CountVectorizer()  
    x_train = vectorizer.fit_transform(x_train)  
    x_test = vectorizer.transform(x_test)
```

- a. With **LogisticRegression** using CountVectorizer I got 88.68% Accuracy.
- b. With **RandomForestClassifier** using CountVectorizer, I got 85.86% Accuracy
- c. With **XGBoost** using CountVectorizer, I got 80.05% Accuracy

Declaring the TfidfVectorizer:

```
tfidf=TfidfVectorizer(max_features=5000)  
  
x_train=tfidf.fit_transform(x_train)  
x_test = tfidf.transform(x_test)
```

- a. With **LogisticRegression** using TfidfVectorizer I got 88.4% Accuracy.
- b. With **RandomForestClassifier** using TfidfVectorizer, I got 84.3% Accuracy
- c. With **XGBoost** using TfidfVectorizer, I got 80.22% Accuracy

Of all the 6 different types, LogisticRegression using CountVectorizer gave best accuracy.

```
[ ] lr = LogisticRegression()  
    lr.fit(x_train, y_train)  
  
y_pred_test = lr.predict(x_test)  
print(classification_report(y_test,y_pred_test))  
print('accuracy_Score:',accuracy_score(y_test,y_pred_test))
```

	precision	recall	f1-score	support
negative	0.89	0.88	0.89	4998
positive	0.88	0.89	0.89	5002
accuracy			0.89	10000
macro avg	0.89	0.89	0.89	10000
weighted avg	0.89	0.89	0.89	10000
accuracy_Score:	0.8868			

TASK 3 – RECOMMENDER SYSTEM

DATASET: Movielens

Loading the dataset and merging Movies and Ratings DataFrames

```
[ ] r_cols = ['user_id', 'movie_id', 'rating', 'unix_timestamp']
ratings = pd.read_csv('/content/drive/MyDrive/ratings.csv', sep='::', names=r_cols)

m_cols = ['movie_id', 'movie_title', 'genres']
movies = pd.read_csv('/content/drive/MyDrive/movies.csv', encoding='latin-1', sep='::', names=m_cols)
```

```
[ ] movies_fd = movies.merge(ratings, on='movie_id', how='inner')
```

```
[ ] movies_fd.head()
```

	movie_id	movie_title	genres	user_id	rating	unix_timestamp
0	1	Toy Story (1995)	Animation Children's Comedy,,	1	5	978824268
1	1	Toy Story (1995)	Animation Children's Comedy,,	6	4	978237008
2	1	Toy Story (1995)	Animation Children's Comedy,,	8	4	978233496
3	1	Toy Story (1995)	Animation Children's Comedy,,	9	5	978225952
4	1	Toy Story (1995)	Animation Children's Comedy,,	10	5	978226474

Creating m x u matrix and normalizing the data

Here we created a m x u matrix where rows are movie ids and columns as user ids. And finally, we have normalized the data, as to adjust for the fact that some people may rate low or high overall or rarely give range of ratings.

```
[43] mat = np.ndarray(shape=(np.max(movies_fd['movie_id'].values), np.max(movies_fd['user_id'].values)), dtype=np.uint8)
mat.shape

(3952, 6040)
```

```
mat[movies_fd['movie_id'].values-1, movies_fd['user_id'].values-1] = movies_fd['rating'].values
norm_mat = mat - np.asarray([(np.mean(mat, 1))]).T
norm_mat

array([[ -7.22682119e+00, -1.22268212e+01, -1.22268212e+01, ...,
        -1.22268212e+01, -1.22268212e+01, -9.22682119e+00],
       [ 9.83908940e+01, -8.60910596e+00, -1.66091060e+01, ...,
        -1.66091060e+01, -1.66091060e+01, -1.66091060e+01],
       [ 5.76718543e+01,  6.71854305e-01, -1.63281457e+01, ...,
        -1.63281457e+01, -1.63281457e+01, -1.63281457e+01],
       ...,
       [ 1.40728477e-02, -9.85927152e-01,  1.40728477e-02, ...,
        1.40728477e-02,  1.40728477e-02,  1.40728477e-02],
       [-7.61589404e-03, -7.61589404e-03, -7.61589404e-03, ...,
        -7.61589404e-03, -7.61589404e-03, -7.61589404e-03],
       [-1.51655629e-01, -1.51655629e-01, -1.51655629e-01, ...,
        -1.51655629e-01, -1.51655629e-01, -1.51655629e-01]])
```

Performing SVD

Here we performed single value decomposition with the normalized matrix and selecting top 50 components

```
[40] ## performing SVD

A = norm_mat.T / np.sqrt(norm_mat.shape[0] - 1)
U, W, V = np.linalg.svd(norm_mat)

print(U.shape, W.shape, V.shape)

(3952, 3952) (3952,) (6040, 6040)
```

Getting top 50 components:

Top50 = V.T[:50]

Defining Cosine Similarity function:

```
## cosine similarity function
def cosine_sim(ratings, movie_id, top_n=10):
    index = movie_id - 1
    movie_row = ratings[index, :]
    magnitude = np.sqrt(np.einsum('ij, ij -> i', ratings, ratings))
    similarity = np.dot(movie_row, ratings.T) / (magnitude[index] * magnitude)
    sort_indexes = np.argsort(-similarity)
    return sort_indexes[:top_n]
```

Defining the function to print the top n recommended movies:

```
# printing the top n movies
def print_similar_movies(data_movie, movie_id, top_indexes):
    print('Recommendations for:', data_movie[data_movie.movie_id == movie_id].movie_title.values[0], '\n')
    for id in top_indexes + 1:
        print(data_movie[data_movie.movie_id == id].movie_title.values[0])
```

Using SVD, recommending movies based on one movie

```
[ ] indexes = cosine_sim(top_50, 42, 10)
    print_similar_movies(movies, 42, indexes)

Recommendations for: Dead Presidents (1995)

Dead Presidents (1995)
Cry, the Beloved Country (1995)
Clueless (1995)
Across the Sea of Time (1995)
Richard III (1995)
Shanghai Triad (Yao a yao dao waipo qiao) (1995)
When Night Is Falling (1995)
Now and Then (1995)
Seven (Se7en) (1995)
Persuasion (1995)
```

Now performing using PCA:

Calculating covariance of normalized matrix

```
[ ] cov_matrix = np.cov(norm_mat)
```

Getting eigen values and eigen vectors

```
[ ] evals, evecs = np.linalg.eig(cov_matrix)
```

Getting top 50 components using eigen values

```
[ ] top_50_eigen = evecs[:, :50]
```

Using Cosine similarity finding 10 closest movies using the 50 components from PCA

```
[ ] top_indexes= cosine_sim(top_50_eigen, 37, 10)  
    print_similar_movies(movies, 37, top_indexes)
```

Recommendations for: Across the Sea of Time (1995)

Across the Sea of Time (1995)
Kingpin (1996)
Tingler, The (1959)
Guy (1996)
Filth and the Fury, The (2000)
Children of the Revolution (1996)
Single Girl, A (La Fille Seule) (1995)
King and I, The (1956)
Wisdom (1986)
Swan Princess, The (1994)

Conclusion:

SVD is better than PCA and has better results. Singular values from SVD are in sorted order, but in the case of PCA, we have to sort the eigenvalues in ascending order

TASK 4 – RANDOM FOREST

1. Create a subsample of dataset using replacement

Here I have used `np.random.choice` to create the subsamples of dataset with replacement.

```
# Create a random subsample from the dataset with replacement
def subsample(dataset, ratio):
    # WRITE YOUR CODE HERE
    # IF NUMBER OF ROWS IN CURRENT DATASET IS N, THE OUTPUT DATASET SHOULD HAVE SAME NUMBER OF ROWS AS THE INPUT DATASET
    # INPUT AND OUTPUT DATATYPE SHOULD BE PANDAS DATAFRAME
    rows = round(len(dataset) * ratio)
    indexes = np.random.choice(len(dataset), size=rows, replace=True)
    return dataset.iloc[indexes]
```

2. Create a subsample with features reduced.

Here also I have used `np.random.choice` for the columns list and used the entire dataset.

```
32] # Create a random selection of n features
def subsample2(dataset, ratio):
    # WRITE YOUR CODE HERE
    # THE OUTPUT DATASET SHOULD HAVE SAME NUMBER OF ROWS AS THE INPUT DATASET
    # IF NUMBER OF FEATURES IN THE INPUT DATASET IS N, THEN THE NUMBER OF FEATURES IN THE OUTPUT DATASET SHOULD BE N * ratio
    # THE INPUT AND OUTPUT DATATYPE SHOULD BE PANDAS DATAFRAME
    col = dataset.columns
    features = int(len(col) * ratio)
    col1 = np.random.choice(col, size=features, replace=False)
    return col1, dataset[col1]# RETURN 2 ITEMS, 1ST IS THE SELECTED COLUMNS
```

3. Split it into train and test data.

First, I have shuffled the dataset using `sample`. Then calculated the train size and divided the train and test using `iloc`.

```
] # Splitting dataset into train and test
def split_train_test(dataset, test_size):
    # WRITE YOUR CODE HERE
    # DATASET IS THE INPUT DATAFRAME
    # TEST_SIZE IS THE SIZE OF TESTING SET IN FORM OF FRACTION BETWEEN 0 AND 1
    # OUTPUT TWO DATAFRAMES, TRAIN AND TEST DATA
    dataset = dataset.sample(frac=1)
    train_indices = len(dataset) - int(len(dataset)*test_size)
    train_data = dataset.iloc[:train_indices]
    test_data = dataset.iloc[train_indices:]

    return train_data, test_data
```

4. Defining Random Forest

Here I have used step 1 and 2 to create subsample data with reduced features to train and used DecisionTreeClassifier to train each sub trees. This function returns trained trees along with the column names.

```
[65] # Random Forest Algorithm
def random_forest_train(train, n_trees, max_depth, sample_size, n_features_ratio):
    # WRITE YOUR CODE HERE
    # TRAIN IS THE TRAINING DATASET THAT YOU GET FROM THE OUTPUT OF YOUR TRAIN TEST SPLIT
    # N_TREES IS NUMBER OF TREES IN THE FOREST
    # MAX_DEPTH IS THE MAX_DEPTH OF EACH TREE IN THE FOREST
    # SAMPLE_SIZE IS THE RATIO OF ROWS TO BE SENT TO EACH TREE (VALUE BETWEEN 0 AND 1)
    # N_FEATURES_RATIO IS THE RATIO OF FEATURES TO BE SENT TO EACH TREE
    # USE SKLEARN TREES TO SIMPLIFY THE EXERCISE
    result_obj = []
    for i in range(n_trees):
        data_sample = subsample(train, sample_size)
        col, data_sample2 = subsample2(data_sample.iloc[:, :-1], n_features_ratio)
        dtc = DecisionTreeClassifier(max_depth=max_depth, random_state=100)
        dtc1 = dtc.fit(data_sample[col], data_sample['target'])
        result_obj.append([dtc1, col])
    return result_obj# A LIST OF [SKLEARN TREE OBJECT, [COLS USED IN THAT TREE]] (SIZE OF OUTPUT LIST WILL BE (N_TREES*2))
```

5. Defining prediction function:

This function returns the final class prediction of each row and all trees' predictions of each row. For final Class, I have predicted each row and stored the count of the target value and took the majority voting in the target list. The same process is done for each row of the dataset. And for all tree's predictions I have directly append the values to list for each tree.

```
[66] # Make a prediction with a list of bagged trees
def random_forest_predict(test, trees):
    # WRITE YOUR CODE HERE
    # TEST IS THE TESTING DATASET THAT IS THE OUTPUT OF YOUR TRAIN TEST SPLIT FUNCTION
    # TREES ARE THE TRAINED TREES (OUTPUT OF THE RANDOM_FOREST_TRAIN FUNCTION)
    # RETURN TWO THINGS, 1 IS FINAL CLASS PREDICTIONS FOR EACH ROW IN TEST DATA, 2 IS THE P
    sub_pred = []
    complete_pred = []
    target = [0,0,0]
    k = 0
    for i in range(len(trees)):
        sub_pred.append([])
    for i in range(len(test)):
        for j in trees:
            x = test[j[1]]
            x = x.iloc[i].values.reshape(1,-1)
            pred1 = j[0].predict(x)
            target[pred1[0]] += 1
            sub_pred[k].append(pred1[0])
            k += 1
        complete_pred.append(target.index(np.max(target)))
        target = [0,0,0]
        k = 0

    return complete_pred, sub_pred#predictions, all_preds
```

Results:

10 different decision trees:

```
[221] trees
```

```
[[DecisionTreeClassifier(max_depth=3, random_state=100),
  array([2, 1, 3], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([2, 3, 1], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([3, 2, 0], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([0, 3, 1], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([0, 2, 3], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([0, 2, 1], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([3, 0, 1], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([3, 0, 2], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([0, 1, 2], dtype=object)],
 [DecisionTreeClassifier(max_depth=3, random_state=100),
  array([2, 3, 1], dtype=object)]]
```

Confusion matrix.

```
[324] confusion_matrix(test.iloc[:, -1], predictions)
```

```
array([[18,  0,  0],
       [ 0, 12,  0],
       [ 0,  2, 13]])
```

Accuracy Score:



```
accuracy_score(test.iloc[:, -1], predictions)
```

```
0.9555555555555556
```