DS_Assignment_3

November 28, 2019

0.0.1 Algorithmic Discrimination [20 points]

This set of assignments will teach you the differences between various fairness measures. Note that this is not a programming assignment.

Assume we have a Binary classifier to promote employees in an organization. The following table presents the Confusion Matrix for female and male employees:

Female	Actual positive	Actual negative
Predicted positve	30	70
Predicted negative	20	105

Male	Actual positive	Actual negative
Predicted positve	100	15
Predicted negative	90	20

1- Calculate the overall accuracy of the promotion classifier.

2- Calculate the accuracy of the promotion system for female and male employees.

3- Evaluate the pormotion system based on three fairness measures: Demographic parity, Equialized odds, and Equiality of opportunity.

4- Is this promotion model fair? if not, suggest an approach to make it fair. Justify your answer.

Answer 1:
Overall Accuracy = No. of correct predictions/Total No. of predictions
Accuracy = TP + TN / TP + TN + FP + FN
Accuracy = 30+100 + 105+20 / 30+100 + 105+20 + 70+15 + 20+90
Accuracy = 255/450
Accuracy = 56.6%

Answer 2:
Accuracy for Male: 100 + 20 / 100 + 20 + 15 + 90
Accuracy for Male: 120 / 225
Accuracy for Male: 53.3%
Accuracy for Female: 30 + 105 / 30 + 105 + 70 + 20
Accuracy for Female: 135 / 225
Accuracy for Female: 60%

Answer 3:

1. **Demographic Parity**

   Here, it means that the acceptance rate for promotion for applications should be equal for both the genders. The formula is as below. (as male group is the advantageous group here).

$$P(Positive|C = Male)/P(Positive|C = Female) = 1$$
$$P(Prediction = Pos|Male) = 115/225 = 0.51$$
$$P(Prediction = Pos|Female) = 100/225 = 0.44$$

   Here, 0.51/0.44 = 2.2, which is != 1 and clearly shows that the model fails on this metric.

2. **Equalized Odds**

   Unlike demographic parity, this also depends on the target variable. People with positive promotion should have similar classification regardless of their gender.

$$P(Prediction = Pos|actual = Pos, gender = F) = P(Prediction = Pos|actual = Pos, gender = M)$$
$$P(Prediction = Pos|actual = Neg, gender = F) = P(Prediction = Pos|actual = Neg, gender = M)$$

   This can be formalized by measuring the True Positive Rate for y = pos or we can measure the False positive rate(y=neg) of both genders and make sure that they are the same.

   True Positive Rate(TPR) = TP / TP + FN
   TPR(Males): 100 / 100+90 = 0.52
   TPR(Females): 30 / 30+20 = 0.60

   False Positive Rate(FPR) = FP / FP + TN
   FPR(Males): 15 / 15+20 = 0.42
   FPR(Females): 70 / 70+105 = 0.40

   Here it seems like the model is biased towards the female group.

3. **Equality of Opportunity** This is quite similar to above, it means that we should give equal promotion opportunities from qualified applications of both gender groups.

$$P(Prediction = Neg|actual = Pos, gender = F) = P(Prediction = Neg|actual = Pos, gender = M)$$
$$=$$
$$P(Prediction = Pos|actual = Pos, gender = F) = P(Prediction = Pos|actual = Pos, gender = M)$$

   It should be noted that the equality of false negative rates implies the equality of true positive rates so this implies the equality of opportunity.

   True Positive Rate(TPR) = TP / TP + FN
   TPR(Males): 100 / 100+90 = 0.52
   TPR(Females): 30 / 30+20 = 0.60

   False Negative Rate(FNR) = FN / FN + TP
   FNR(Males): 90 / 90+100 = 0.47
   FNR(Females): 20 / 20+30 = 0.40

   Here the model is biased towards one group in one of the metric and towards another group in another metric, and since they are not equal, hence we can safely say that the the models are not fair in both the two metrics.

Answer 4:
Based on the above three fairness metrics, it can be seen that the model is not fair. In some metrics, the male group seems to be at an advantage whereas in some, it favours the female when it comes to predicting the promotion. Even though the accuracy is high for females when compared to males, it is an incorrect metric to evaluate. The above fairness metrics should be used to evaluate the models.

Below approaches can be used to make the model fair.

1. Preprocessing:
We can remove some information correlated to sensitive attributes like gender in our case. Although such information can still be expressed through other variables implicitly, but it can surely help improve the fairness of the model and reduce some bias.

2. Data Repairing:
Other preprocessing techniques like Massaging, Re-weighting and Sampling can be used to correct the data to improve the fairness of the model.

3. Learning Fairness Constraints:
Another idea is to add a constraint or a regularization term to the existing optimization objective. We can convert all the above fairness metric equalities into inequality equations and add these constraints with the existing loss function using Lagrange multiplier. This does add another hyper-parameter to the equation which needs to be tuned according to our fairness standards.

P(pos | male) - P(pos | female) > $\Delta_{fairness}$
where, $\Delta_{fairness}$ <= η and this becomes a hyper-param

4. Post Processing:
The scores from the models can also be altered based on a set threshold to remove biases within the model by changing the posteriors in a manner that it satisfies the fairness constant.

0.0.2 Natural Language Processing [50 points]

This set of assignments will give you experience with a text corpus, Python programming, part-of-speech (PoS) tags, sentiment analysis, and machine learning with scikit-learn.

This assignment shows how you can perform sentiment analysis on reviews using Python and Natural Language Toolkit (NLTK).

Sentiment Analysis means analyzing the sentiment of a given text or document and categorizing the text/document into a specific class or category (like positive and negative). In other words, building a sentiment analysis model classifies any particular text or document as positive or negative. In the simplest form, the classification is done for two classes: positive and negative. However, we can add more classes like neutral, highly positive, highly negative, etc.

Sentiment analysis is an important topic in computational linguistics in which we quantify subjective aspects of language. These aspects can range from biases in social media for marketing, to a spectrum of cognitive behaviours for disease diagnosis.

In this assignment, you learn about labeling data, extracting features, training classifier, and testing the accuracy of the classifier.

For this assignment, we use a reviews dataset as our labeled data which is collected from Yelp, Amazon, and IMDB. We attempted to select sentences that have a clearly positive or negative connotaton, the goal was for no neutral sentences to be selected. The review corpus contains reviews with sentiment polarity classification, where score is either 1 (for positive) or 0 (for negative). You can find the assignment data on the website.

a) (Data processing): First create a list of all reviews and their categories. You can create a list of tuples where the first item is the review and the second item is the sentiment, i.e., '0' or '1'.

```
data_path = "./assignment_3(data)/reviews"
file = open(data_path+"/train.txt")
contents = file.read().splitlines()
train_data = [tuple(x.split("\t")) for x in contents]

file = open(data_path+"/test.txt")
contents = file.read().splitlines()
test_data = [tuple(x.split("\t")) for x in contents]

print(train_data[0], train_data[1])
```

```
('So there is no way for me to plug it in here in the US unless I go by a
converter.', '0') ('Good case, Excellent value.', '1')
```

b) (Tokenization): Extract all the words from the reviews. You can either use the string *split()* method directly or use the following method from NLTK.

Using lower() because nltk.stopwords will be in lowercase, and it is better to use one word for "The" and "the". Moreover, the stemming will also convert everything to lowercase later.

```
def generate_tokens(data):
    from nltk.tokenize import word_tokenize
    X = []
    for x in data:
        try:
            X.append((word_tokenize(x[0].lower()), x[1]))
        except:
            pass
    return X

X_train_tokens = generate_tokens(train_data)
X_test_tokens = generate_tokens(test_data)

print(X_train_tokens[0])
```

```
(['so', 'there', 'is', 'no', 'way', 'for', 'me', 'to', 'plug', 'it', 'in',
'here', 'in', 'the', 'us', 'unless', 'i', 'go', 'by', 'a', 'converter', '.'],
'0')
```

Calculate the number of occurence of each word in the entire corpus and report the 10 most common tokens. You can use the following method from NLTK.
Adding only the train set in the vocabulary to avoid data leakage and fair evaluation of the test set.

```
from nltk.probability import FreqDist

all_word_list = []
for sentences in X_train_tokens:
    for token in sentences[0]:
        all_word_list.append(token)

fdist = FreqDist(all_word_list)
fdist.most_common(10)
```

```
[('.', 2198),
 ('the', 1601),
 (',', 1115),
 ('and', 961),
 ('i', 875),
 ('a', 750),
 ('it', 662),
 ('is', 636),
 ('to', 568),
 ('this', 546)]
```

c) (Stop words removal): Remove all the stop words from the reviews and re-calculate the number of occurence of each word in the entire corpus.

Also removing punctuation because characters like ".", "," had the most frequency and this is a redundant feature.

```
from nltk.corpus import stopwords
stopwords_english = stopwords.words('english')

# Removing both stopwords and punctuations
filtered_word_list = [word for word in all_word_list if word not in␣
 ↪stopwords_english and word not in string.punctuation]

fdist = FreqDist(filtered_word_list)
fdist.most_common(10)
```

```
[("n't", 233),
 ("'s", 216),
 ('good', 186),
 ('great', 169),
 ('movie', 153),
 ('phone', 141),
 ('film', 139),
 ('one', 121),
 ('food', 98),
 ('...', 97)]
```

d) (Stemmization): Normalize the reviews by stemming and re-calculate the number of occurence of each word in the entire corpus. You can use the following function from NLTK.

```
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer('english')
stemmed_filtered_word_list = [stemmer.stem(word) for word in filtered_word_list]

fdist = FreqDist(stemmed_filtered_word_list)
fdist.most_common(10)
```

```
[("n't", 233),
 ("'s", 216),
 ('good', 186),
 ('movi', 182),
 ('great', 170),
 ('film', 164),
 ('phone', 149),
 ('one', 124),
 ('work', 123),
 ('time', 115)]
```

e) (BOW): Create 1-hot encoding and represent each review as Bag of Words (BOW).

```
def preprocess(data):
    '''
    Running Stopwords, Stemming on X_train
    '''
    X = []
    for token_list, label in data:
        new_word_list = [stemmer.stem(word) for word in token_list if word not␣
 ↪in stopwords_english and word not in string.punctuation]
        X.append((new_word_list, label))
    return X

X_train_tokens_processed = preprocess(X_train_tokens)
X_test_tokens_processed = preprocess(X_test_tokens)
```

```
def generate_vectors(data, vocab = stemmed_filtered_word_list):
    '''
    Build vocabulary and generate vectors
    '''
    X_vectors = []
    for sentence, label in data:
        vec = {}
        for token in vocab:
            if token in sentence:
                vec[token] = 1
            else:
                vec[token]= 0
        X_vectors.append((vec, label))
    return X_vectors

train_set = generate_vectors(X_train_tokens_processed)
test_set = generate_vectors(X_test_tokens_processed)
```

```
vocab_length = len(train_set[0][0].items())
vocab_length
```

```
3849
```

train_set is a list of tuple of feature_dictionary and label.

Since the feature dic is of 3849 keys, it is not feasible to print it here.

Its basically like "I":0, "was":1, "like":1..

f) (Train a Classifier): Use reviews in the train folder as "train-set" and use reviews in the test folder as "test-set". Use Naive Bayes Classifer to train your sentiment predictor. You can use the following code for this purpose.

```
from nltk import NaiveBayesClassifier
from nltk import classify


train_set = generate_vectors(X_train_tokens_processed)
test_set = generate_vectors(X_test_tokens_processed)

classifier = NaiveBayesClassifier.train(train_set)

accuracy = classify.accuracy(classifier, test_set)
print("Accuracy: ", accuracy)
```

```
Accuracy:  0.8
```

```
The train_set and test_set have been prepared using the generate_vectors method␣
 ↪mentioned in the previous question along with building the vocabulary. Since we␣
 ↪used stemming and filtered stop words and punctuation, we are getting a good␣
 ↪accuracy of 0.80 with unigram tokens.
```

g) (N-gram): Extract Bigram features from the reviews and re-train the model with Naive Bayes classifier on the train-set and report accuracy on the test-set. You can use the following method from NLTK to extract bigrams.

```
def get_bigrams(word_tokens, n=2):
    n_grams = ngrams(word_tokens, n)
    return [ ' '.join(grams) for grams in n_grams]

vocab_bigram = get_bigrams(stemmed_filtered_word_list)
X_train_bigram = [(get_bigrams(x[0]),x[1]) for x in X_train_tokens_processed]
X_test_bigram = [(get_bigrams(x[0]),x[1]) for x in X_test_tokens_processed]

train_set = generate_vectors(X_train_bigram, vocab = vocab_bigram)
test_set = generate_vectors(X_test_bigram, vocab = vocab_bigram)
vocab_length = len(train_set[0][0].items())
vocab_length
```

14604

```
classifier = NaiveBayesClassifier.train(train_set)
accuracy = classify.accuracy(classifier, test_set)
print("Accuracy: ", accuracy)
```

```
Accuracy:  0.628
I am getting 0.628 accuracy if I use the existing filtered tokens(stop words,␣
 ↪stemming, punctuation removed) to create bigrams.

I also tried creating bigrams from the scratch without removing any stop words or␣
 ↪stemming. It seemed to outperform(accuracy:0.764) the ones with the stopwords/
 ↪stemming(accuracy: 0.628). I think it is fair to say that for atleast bigrams,␣
 ↪the model without any pre-processing seems to outperform the ones with␣
 ↪preprocessed inputs atleast for our dataset.
```

```
vocab_bigram = get_bigrams(all_word_list)
X_train_bigram = [(get_bigrams(x[0]),x[1]) for x in X_train_tokens]
X_test_bigram = [(get_bigrams(x[0]),x[1]) for x in X_test_tokens]

train_set = generate_vectors(X_train_bigram, vocab = vocab_bigram)
test_set = generate_vectors(X_test_bigram, vocab = vocab_bigram)
vocab_length = len(train_set[0][0].items())
vocab_length
```

20207

```
classifier = NaiveBayesClassifier.train(train_set)
accuracy = classify.accuracy(classifier, test_set)
print("Accuracy: ", accuracy)
```

```
Accuracy: 0.764
```

h) (Combined features): Represent each review based on the combination of 2000 most frequent unigram and bi-grams. Re-train the Naive Bayes Classifier and report the accuracy.

```
def combine_tokens(data1, data2):
    final_list = []
    for x,y in zip(data1, data2):
        a = []
        for tokens_x, tokens_y in zip(x[0],y[0]):
            a.append(tokens_x)
            a.append(tokens_y)
        final_list.append((a, x[1]))
    return final_list
```

```
X_train_combined = combine_tokens(X_train_tokens_processed, X_train_bigram)
X_test_combined = combine_tokens(X_test_tokens_processed, X_test_bigram)
X_train_combined[0]
```

```
(['way',
  'way plug',
  'plug',
  'plug us',
  'us',
  'us unless',
  'unless',
  'unless go',
  'go',
  'go convert'],
 '0')
```

```
## Combining unigrams and bigrams for creating combined vocab
fdist = FreqDist(stemmed_filtered_word_list+vocab_bigram)
combined_vocab_most_common = [x[0] for x in fdist.most_common(2000)]

train_set = generate_vectors(X_train_combined, vocab =␣
 ↪combined_vocab_most_common)
test_set = generate_vectors(X_test_combined, vocab = combined_vocab_most_common)

vocab_length = len(train_set[0][0].items())
vocab_length
```

```
2000
```

```
classifier = NaiveBayesClassifier.train(train_set)

accuracy = classify.accuracy(classifier, test_set)
print("Accuracy: ", accuracy)
```

```
Accuracy:  0.786
```

0.0.3 Dense representation with SVD [Bonus point: 10 points]

i) Generate a co-occurance matrix with window size = 2. Represent each revew with a dense vector extracted from SVD where dimention = 300. Re-train the model and report the accuracy on the test-set.

```
def create_cooccurrence_matrix(sentences, window_size=2):
    vocabulary = {}; data,row,col = [], [], []

    for sentence in sentences:
        for pos, token in enumerate(sentence):
            i = vocabulary.setdefault(token, len(vocabulary))
            start = max(0, pos-window_size)
            end = min(len(sentence), pos+window_size+1)
            for pos2 in range(start, end):
                if pos2 != pos:
                    j = vocabulary.setdefault(sentence[pos2], len(vocabulary))
                    data.append(1.); row.append(i); col.append(j)

    cooccurrence_matrix_sparse = scipy.sparse.coo_matrix((data, (row, col)))
    return vocabulary, cooccurrence_matrix_sparse
```

[741]:
```
vocab_svd,coo_matrix = create_cooccurrence_matrix(x[0] for x in␣
↪X_train_tokens_processed)
coo_matrix.shape
```

```
(3769, 3769)
```

```
import scipy
u,s,v = scipy.sparse.linalg.svds(coo_matrix, k = 300)
e_matrix = u*s
u.shape,s.shape,v.shape
```

```
((3769, 300), (300,), (300, 3769))
```

```
def get_vectors_from_svd(data, matrix):
    a = []; y = []
    for sentences,label in data:
        sent_vec = 0
        for token in sentences:
            try:
                index = vocab_svd[token]
                vector_300 = matrix[index]
                sent_vec += vector_300
            except:
                pass
        a.append(sent_vec/len(sentences))
        y.append(label)
    return a,y
```

«next page»

```
train_set, train_set_y = get_vectors_from_svd(X_train_tokens_processed, e_matrix)
test_set, test_set_y = get_vectors_from_svd(X_test_tokens_processed, e_matrix)
```

Although it has its own constraints, but to convert sentences into vectors, it is reasonable to take the average of all word vectors to make a decent representation of the sentence(review). Hence, embedding for each review is calculated by taking the mean of word embedding.

Since the vectors from SVD have negative values,it is not feasible to use them on Naive Bayes unless scaled between 0-1 but then it defeates the purpose, hence using a different classifier below to train the vectors.

We got descent accuracy by using mean of word embedding but it can be further improved by using sequence language modelling techniques on these vectors.

```
from sklearn.ensemble import RandomForestClassifier
classifier = RandomForestClassifier(n_estimators=100)
classifier.fit(train_set, train_set_y)

y_pred = classifier.predict(test_set)
from sklearn.metrics import accuracy_score
accuracy_score(y_pred, test_set_y)
```

```
Accuracy  0.746
```

0.0.4 Word Embeddings [Bonus point: 25 points]

Download pretrained word embeddings from Google News dataset. The model includes embeddings for 3 million words and phrases. Words with frequency below 5 were discarded. Download the model from the link below: https://code.google.com/archive/p/word2vec/

You can load the model using gensim:

```
import gensim.models
model = gensim.models.KeyedVectors.load_word2vec_format('./
 ↪GoogleNews-vectors-negative300.bin', binary=True)
```

```
## Trained on 3 million vocab
vocab_word2vec = list(model.vocab)
len(vocab_word2vec)
```

```
3000000
```

j) Represent the 5 most positive informative words with word2vec (dimension = 300) and SVD.

k) Represent the 5 most negative informative words with word2vec (dimension = 300) and SVD.

Representing the answers for (j) and (k) in the vector forms. Showing 5 vectors for these 5 words for both word2vec and SVD due to space constraints.

```
classifier.show_most_informative_features(10)
```

```
Most Informative Features
                   excel = 1                1 : 0      =     34.3 : 1.0
                     bad = 1                0 : 1      =     21.4 : 1.0
                  terribl = 1               0 : 1      =     20.6 : 1.0
                   worst = 1                0 : 1      =     20.6 : 1.0
                  perfect = 1               1 : 0      =     19.4 : 1.0
                   great = 1                1 : 0      =     18.6 : 1.0
                   money = 1                0 : 1      =     17.3 : 1.0
                    love = 1                1 : 0      =     16.0 : 1.0
                   happi = 1                1 : 0      =     16.0 : 1.0
                      aw = 1                0 : 1      =     13.4 : 1.0
```

```
# Due to stemming in our vocab for svd, few words were modified which are not␣
 ↪present in word2Vec
# like happy -> happi, terrible --> terribl

positive_words_5_word2vec = ['excel', 'perfect', 'great', 'love', 'happy']
negative_words_5_word2vec = ['bad', 'terrible', 'worst', 'money', 'aw']

positive_words_5_svd = ['excel', 'perfect', 'great', 'love', 'happi']
negative_words_5_svd = ['bad', 'terribl', 'worst', 'money', 'aw']
```

```
## Extracting word2vec and SVD vectors for most positive and negative words
X_positive_svd, X_negative_svd = [], []
for x in positive_words_5_svd:
    index = vocab_svd[x]
    X_positive_svd.append(e_matrix[index])
for x in negative_words_5_svd:
    index = vocab_svd[x]
    X_negative_svd.append(e_matrix[index])
X_positive_word2vec = model[positive_words_5_word2vec]
X_negative_word2vec = model[negative_words_5_word2vec]
```

5 most positive informative words with word2vec (dimension = 300) and SVD.

```
df = pd.DataFrame(X_positive_word2vec, index=["w2v_"+x for x in␣
 ↪positive_words_5_word2vec])
df.append(pd.DataFrame(X_positive_svd, index=["svd_"+x for x in␣
 ↪positive_words_5_svd]))
```

```
                0         1         2         3         4         5    \
w2v_excel   -0.186523  0.102051 -0.183594 -0.054443  0.069336  0.096680
w2v_perfect  0.038818 -0.066406 -0.094238 -0.017334  0.132812  0.021973
w2v_great    0.071777  0.208008 -0.028442  0.178711  0.132812 -0.099609
w2v_love     0.103027 -0.152344  0.025879  0.165039 -0.165039  0.066895
w2v_happy   -0.000519  0.160156  0.001610  0.025391  0.099121 -0.085938
svd_excel   -0.011079 -0.000138  0.028625 -0.099014  0.099481 -0.101077
svd_perfect -0.159524  0.312121  0.089989  0.191863 -0.190684  0.137882
svd_great   -0.019641 -0.060399 -0.125312 -0.035521 -0.109890  0.022214
svd_love    -0.137001  0.000175 -0.158545  0.092419 -0.225847  0.083340
svd_happi   -0.333605 -0.051509  0.040898 -0.022789  0.009607 -0.158109
[10 rows x 300 columns]
```

5 most negative informative words with word2vec (dimention = 300) and SVD.

```
df = pd.DataFrame(X_negative_word2vec, index=["w2v_"+x for x in␣
 ↪negative_words_5_word2vec])
df.append(pd.DataFrame(X_negative_svd, index=["svd_"+x for x in␣
 ↪negative_words_5_svd]))
```

```
                0         1         2         3         4         5    \
w2v_bad      0.062988  0.124512  0.113281  0.073242  0.038818  0.079102
w2v_terrible 0.164062  0.192383  0.092285  0.130859  0.075195  0.110352
w2v_worst    0.166016 -0.082031  0.451172  0.122559  0.134766  0.133789
w2v_money    0.158203  0.051270  0.066406  0.210938  0.035156 -0.004669
w2v_aw      -0.114258  0.246094  0.308594  0.335938 -0.137695 -0.121094
svd_bad     -0.138618 -0.117430  0.024735 -0.071512 -0.035568  0.170250
svd_terribl -0.318785  0.190642  0.069203  0.049941 -0.283691  0.194047
svd_worst    0.150274 -0.077653 -0.185413  0.258991  0.259603  0.064842
svd_money    0.055670  0.024516 -0.026653 -0.017566 -0.053925 -0.002983
svd_aw       0.129899 -0.072033 -0.021429 -0.009618 -0.103464  0.060940
[10 rows x 300 columns]
```

l) Calculate the dot product between the most frequent positve word and the most frequent negative word represented with SVD and word2Vec. Compare the results.

```
most_frequent_positive_word = "great"
most_frequent_negative_word = "bad"

most_positive_word = "excel"
most_negative_word = "bad"

from numpy.linalg import norm

a = model[most_positive_word]
b = model[most_negative_word]
c = e_matrix[vocab_svd[most_positive_word]]
d = e_matrix[vocab_svd[most_negative_word]]

dot_product_word2vec = np.dot(a, b)
dot_product_svd = np.dot(c, d)

cosine_word2vec = np.dot(a,b)/(norm(a)*norm(b))
cosine_svd = np.dot(c,d)/(norm(c)*norm(d))

print("Dot product of word2vec: ", dot_product_word2vec)
print("Dot product of SVD: ", dot_product_svd)
print("Cosine similarity of word2vec: ", cosine_word2vec)
print("Cosine similarity of SVD: ", cosine_svd)
```

```
Dot product of word2vec:  0.67587197
Dot product of SVD:  123.99355647151627

Cosine similarity of word2vec:  0.09426507
Cosine similarity of SVD:  0.2500198498415385
```

Since dot product is magnitude sensitive as it depends on the occurrence count of a word. It might have a large or small dot product with another word. To compare and normalize all vectors we divide it by the norm of two vectors, also called as Cosine Similarity.

As we can see the similarity between a positive and negative word vectors is quite less in both word2vec and SVD. Since these two words are far away in the vector space, hence their dot product/cosine similarity is less. Since Word2Vec is trained on a bigger corpus, it seems to have a smaller indicating these vectors for a positive and a negative word are farther apart as desired.

m) Can one use word2vec for sentiment analysis on the review dataset? Justify your answer.

Answer m:
Yes, we can use word2vec model for sentiment analysis as it converts words to vectors in such a way that similar meaning words appear together and dissimilar word are located far away. The model is trained by either predicting the context around the word (skip-gram) or by predicting the word in a context (CBOW). It can capture both syntactic and semantic relationships and can be used in various natural language tasks including a sentiment analysis.

Infact, pre-trained word2vec model used above seems to be performing better than the SVD model we trained here as similar words are much closer and dissimilar words are farther when compared to our SVD model. This could be attributed to the difference in the vocabulary of both the models. As seen in the previous question, the cosine similarity for a positive and negative word was higher than our SVD model indicating that word2vec could perform better on the task of sentiment analysis.

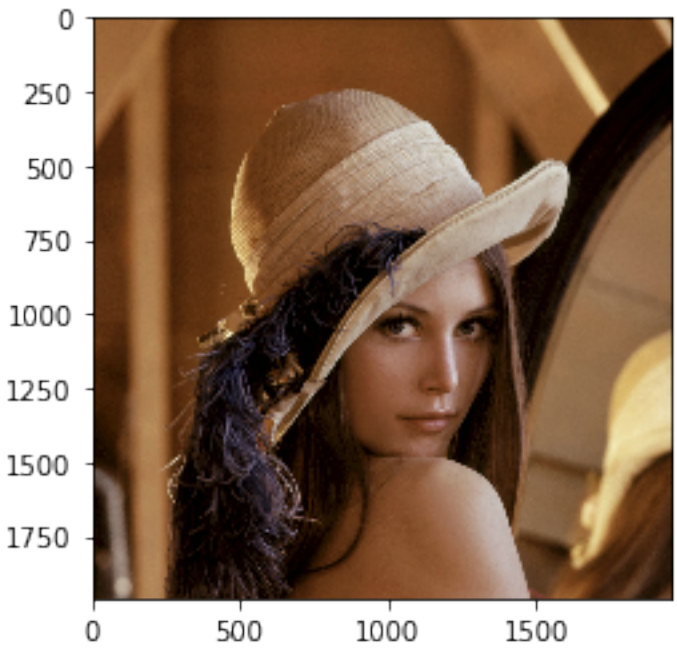
Moreover, I even trained the model using word2vec and I got an accuracy of 0.77 which is better than the accuracy of 0.74 using SVD. In order to mitigate the limitations of bag of words, it is further advisable to use these embedding/vectors in sequence models such as RNN to improve the models performance.

0.0.5 Computer Vision [30 points]

This assignment will give you experience with an image corpus. For most of the questions in this assingment, you need to write a python script.

1- *Read image*: Write a python code to read the image. You can find the image of this assingment on the webpage.

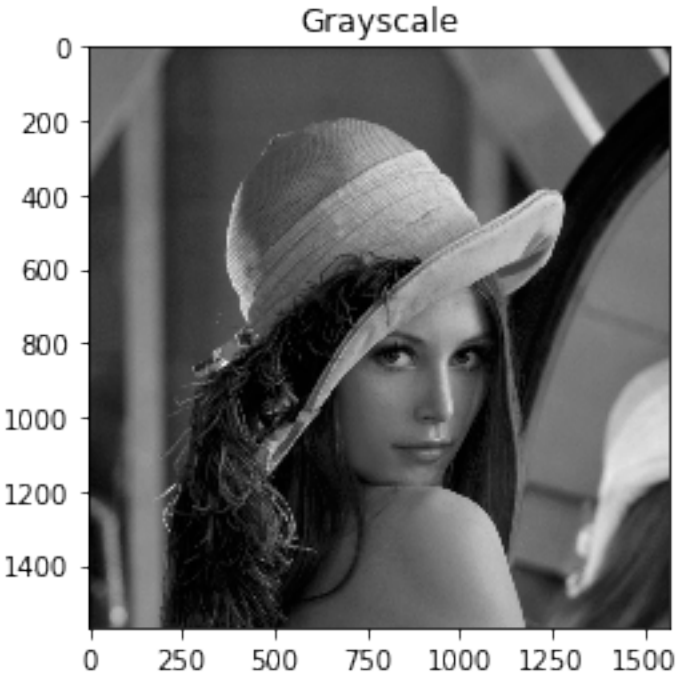
```
import cv2
#write a function to read an image
image = cv2.imread("./assignment_3(data)/lena.jpg")

# for displaying on Matplotlib, converting it from default BGR to RGB
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)
plt.imshow(image)
```



2- *Pre-processing*: data augmentation: Write a python code to resize the image and make it 20% smaller, and save the image as greyscale image.

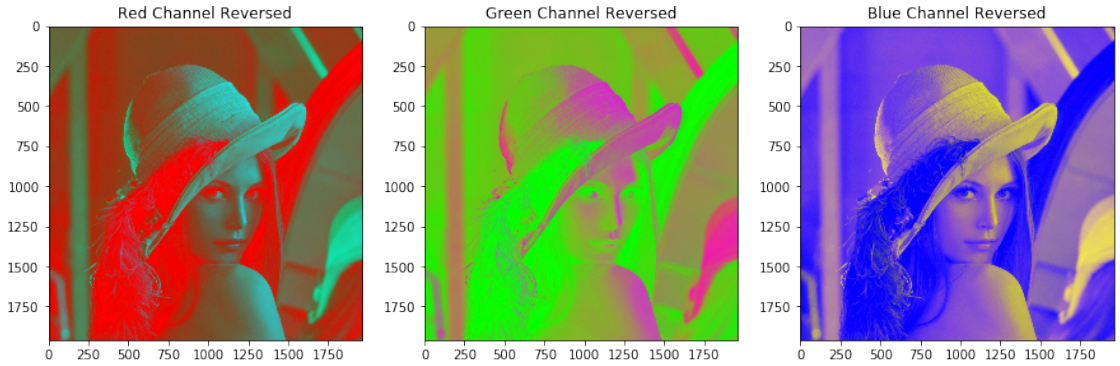
```
# re-size an image
scale_by = 20
width = int(image.shape[1] * ((100 - scale_by) / 100))
height = int(image.shape[0] * ((100 - scale_by) / 100))
resized_image = cv2.resize(image, (width, height), interpolation = cv2.
 ↪INTER_AREA)

print("Original Image Shape: ", image.shape)
print("20% smaller Image Shape: ", resized_image.shape)

# save the processed image in grayscale
gray_image = cv2.cvtColor(resized_image,cv2.COLOR_RGB2GRAY)
cv2.imwrite( "./assignment_3(data)/lena_gray.jpg", gray_image );
plt.imshow(gray_image,'gray');
plt.title("Grayscale");
```

```
Original Image Shape:  (1960, 1960, 3)
20% smaller Image Shape:  (1568, 1568, 3)
```



3- *Filter*: Write a python code that load the image in RGB and generate three images where in each one the colors of one channel is inversed. Show the generated images.

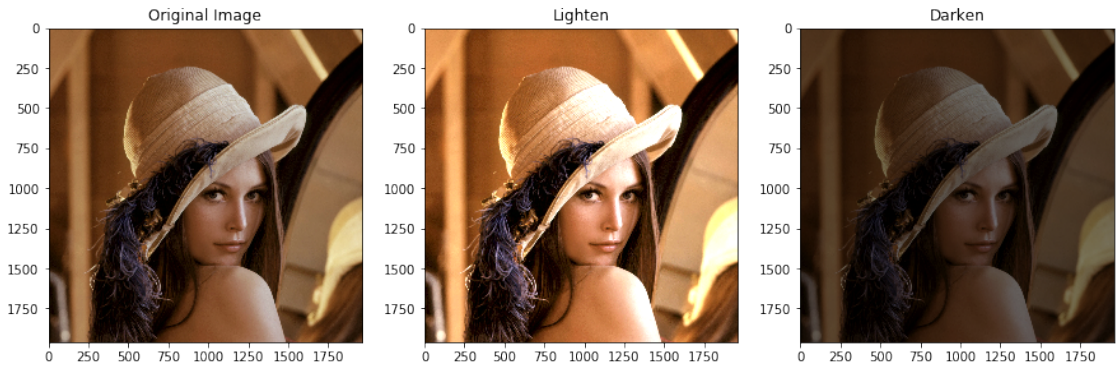
```
# load image in RGB
image = cv2.imread("./assignment_3(data)/lena.jpg", 1)
image = cv2.cvtColor(image,cv2.COLOR_BGR2RGB)

# Generate 3 images where each the color of one channel in each image is inverted
# show the processed images
fig, ax = plt.subplots(nrows = 1, ncols=3, figsize=(15,5))
for c, ax in zip(range(3), ax):
    new_img = np.copy(image)
    new_img[ :, :, c] = 255-image[ :, :, c]
    ax.imshow(new_img)
```



4- Suggest a filter that make the image 50% lighter and 50% darker. Write a python function that applies the filter on the image.

```
# As values close to 0 will be darker and values close to 255 will be lighter.
# Dividing by 255 to scale values between 0-1 float  for matplotlib to print
(to avoid clipping the values myself)


def lightening_filter(img):
    # make the image 50% lighter
    image = img.copy()
    image = image*1.5
    return image/255

def darkening_filter(img):
    # make the image 50% darker
    image = img.copy()
    image = image*0.5
    return image/255
```

```
fig, ax = plt.subplots(nrows = 1, ncols=3, figsize=(15,5))
ax[0].imshow(image);
ax[0].set_title("Original Image");
ax[1].imshow(lightening_filter(image));
ax[1].set_title("Lighten");
ax[2].imshow(darkening_filter(image));
ax[2].set_title("Darken");
```



5- Write two python functions that apply a 3 x 3 median and 3 x 3 mean filter on the image. Mean filter is a simple sliding window that replace the center value with the average of all pixel values in the window. While median filter is a simple sliding window that replace the center value with the Median of all pixel values in the window. Note that the border pixels remain unchanged.

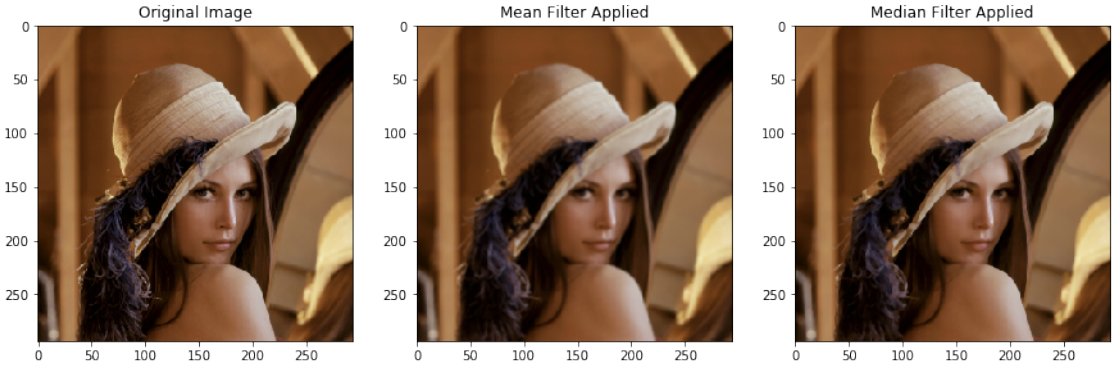
```
## Resizing image to notice blur effects after filters
width = int(image.shape[1] * 0.15)
height = int(image.shape[0] * 0.15)
resized_image = cv2.resize(image, (width, height), interpolation = cv2.
 ↪INTER_AREA)
```

```
def mean_filter(img):
    # Can create a kernel to take mean and convolve this with the image
    mean_kernel = np.ones((3,3),np.float32)/9
    # Can use the internal box Normalized box filter which is the same thing.
    # return cv2.blur(img, (3,3))
    return cv2.filter2D(img,-1,mean_kernel)

def median_filter(img):
    return cv2.medianBlur(img,3)

def mean_filter_implementation(img):
    img_new = np.zeros(img.shape, dtype="uint8")
    for c in range(3):
        im = img[:,:,c]
        for i in range(im.shape[0]-1):
            for j in range(im.shape[1]-1):
                block = im[i:i+3, j:j+3]
                m = np.mean(block)
                img_new[i+1,j+1,c] = m
    return img_new

def median_filter_implementation(img):
    img_new = np.zeros(img.shape, dtype="uint8")
    for c in range(3):
        im = img[:,:,c]
        for i in range(im.shape[0]-1):
            for j in range(im.shape[1]-1):
                block = im[i:i+3, j:j+3]
                m = np.median(block)
                img_new[i+1,j+1,c] = m
    return img_new
```

«next page»

```
fig, ax = plt.subplots(nrows = 1, ncols=3, figsize=(15,5))
ax[0].imshow(resized_image);
ax[0].set_title("Original Image");
ax[1].imshow(mean_filter(resized_image));
ax[1].set_title("Mean Filter");
ax[2].imshow(median_filter(resized_image));
ax[2].set_title("Median Filter");
```

The images with mean and median filters are blurry, which might not be quite visible in the pdf so clearly.



6- A mean filter is a linear filter, but a median filter is not. Why?

Answer 6.
A function F is said to be linear if F(A+B) = F(A) + F(B). Mean filter falls in this area as Mean(A+B) = Mean(A) + Mean(B) whereas, its not true for Median as Median(A+B) != Median(A) + Median(B) as Median filter involves sorting the value of pixes in a batch and replacing it with the middle value..

A median filter can change non-linearly with inputs, for example: a change in the non middle values in the input vector will not change the output of the median. Unless the middle values are changed, the output wont be affected. This produces a non-linear relation between inputs and outputs.

Hence the mean filter is a linear filter, but a median filter is not.