- ✓ این گزارش مربوط به چهار بخشِ عملی تمرین سوم است و بخش تئوری از قبل آپلود شده است.
- ✓ در هر یک از بخش‌های عملی، ابتدا کد را شرح می‌دهیم و سپس خروجی‌ها را خدمتتان ارائه خواهیم کرد.

بخش اول – KNN

مراحل پیاده‌سازی بخش اول عملی (KNN) :

گام اول خواندنِ داده‌ها می‌باشد که ما تابعِ load_dataset() را به صورتِ زیر پیاده کردیم. به گونه‌ای که داده‌ها را از فایلِ csv که ورودیِ تابع است خوانده و با توجه به عددی بودنِ آنها، نقاطِ موجود را از string به float تبدیل نماید.

```
7   def load_dataset(filename):
8       with open(filename) as csvfile:
9           lines = csv.reader(csvfile)
10          dataset = list(lines)
11          for x in range(len(dataset)-1):
12              for y in range(19):
13                  dataset[x+1][y+1] = float(dataset[x+1][y+1])
14      dataset.pop(0)
15      return dataset
16
```

در گامِ بعدی پس از تنظیمِ معیار فاصله و تعداد همسگان و نیز با داشتنِ دیتاستِ آموزش و دیتاستِ تست که از تابعِ فوق استخراج نمودیم، این چهار پارامتر را به فانکشنِ خود دادیم. اسم تابع را KNN_algorithm قرار دادیم

```
49
50  def KNN_algorithm(train_set, test_set, k, distanceCretaria):
51      predicted = classification(train_set, test_set, k, distanceCretaria)
52      actual = [row[0] for row in test_set]
53      accuracy = get_accuracy(actual, predicted)
54      return accuracy
55
```

همانطوری که مشاهده می‌کنید، داده‌های پیش‌بینی شده را توسط فانکشن‌هایی که پیاده کرده‌ایم گرفته و صحت را با یک مقایسه‌ی ساده بینِ برچسب‌های predict شده و برچسب‌های واقعی از داده‌های تست بدست می‌آورد که تابع محاسبه‌ی صحت با محاسبه‌ی تعداد پیشبینی‌های درست به صورت زیر است.

```
16
17  def get_accuracy(actual, predicted):
18      truePrediction = 0
19      for i in range(len(actual)):
20          if actual[i] == predicted[i]:
21              truePrediction += 1
22      return truePrediction / float(len(actual))
23
```

همانطوری که در KNN_algorithm مشاهده می‌کنید، تابعی با نامِ classification را صدا کردیم که پیاده‌سازی آن به صورتِ زیر است. در واقع برچسب هر نقطه را که یک سطر از ماتریسِ تست را تشکیل می‌دهد، predict میکند.

```
61
62  def classification(train, test, k, distanceCretaria):
63      predictions = list()
64      for row in test:
65          output = predict_class(train, row, k, distanceCretaria)
66          predictions.append(output)
67      return(predictions)
```

نحوه‌ی پیاده سازی predict_class نیز بدین صورت است که همسایگان را با معیارهای فاصله حساب کرده و با در نظر گرفتنِ بیشترین برچسب مربوط به یک کلاس از بینِ این داده‌های همسایه، آن داده را در کلاسِ مذکور طبقه بندی می‌کند.

```
55
56  def predict_class(train, test_row, k, distanceCretaria):
57      neighborsVector = get_k_nearest_neighbors_of_point(train, test_row, k, distanceCretaria)
58      outputs = [row[0] for row in neighborsVector]
59      prediction = max(set(outputs), key=outputs.count)
60      return prediction
61
```

برای مقایسه شباهت هم با در نظر گرفتنِ ورودی ای که کاربر از معیارِ شباهتِ خود می‌دهد و تعداد همسایگان، آن‌ها را استخراج کرده و به تابعِ فوق می‌دهیم.

```
33  def get_k_nearest_neighbors_of_point(train, test_row, k, distanceCretaria):
34      distances = list()
35      for train_row in train:
36          if distanceCretaria == "euclidean":
37              dist = euclidean_distance_cretaria(test_row, train_row)
38          elif distanceCretaria == "cosine":
39              dist = cosine_distance_cretaria(test_row, train_row)
40          else:
41              print("Distance criteria must be choosen between \"cosine\" and \"euclidean\", Please try again")
42              exit
43          distances.append((train_row, dist))
44      distances.sort(key=lambda tup: tup[1])
45      neighborsVector = list()
46      for i in range(k):
47          neighborsVector.append(distances[i][0])
48      return neighborsVector
```

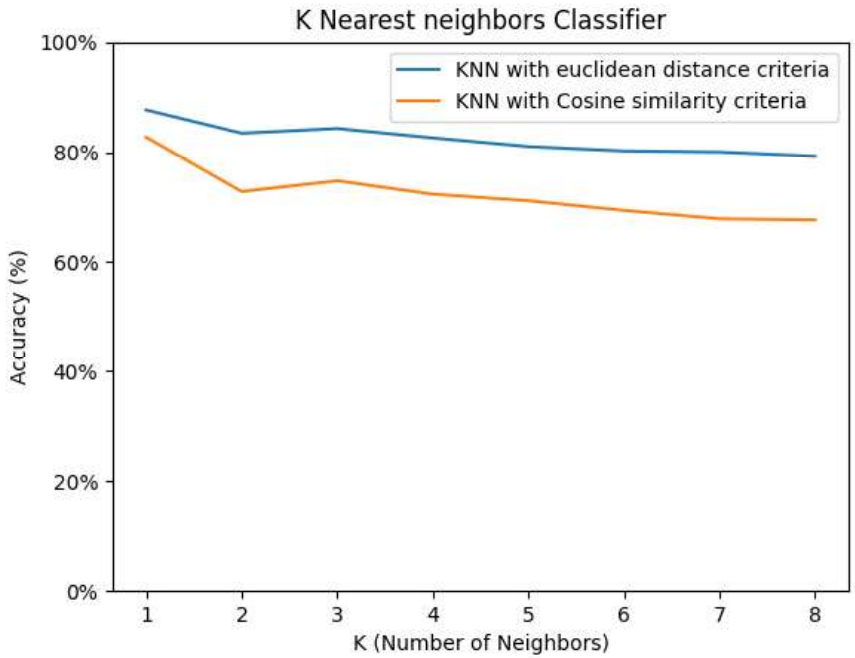
همچنین برای محاسبه‌ی فاصله‌ی اقلیدسی و معیارِ شباهتِ کسینوسی نیز دو تابعِ ساده به صورت زیر نوشتیم که در یکی فاصله‌ی اقلیدسی را با توجه به مقدارِ هر بعد محاسبه می‌کنیم و در دیگری شباهتِ کسینوسی را با استفاده از فورمول ضرب داخلی دو بردار محاسبه نموده‌ایم که به صورت زیر است.

```python
def euclidean_distance_cretaria(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i+1] - row2[i+1])**2
    return math.sqrt(distance)

def cosine_distance_cretaria(row1, row2):
    return 1-(sum([row1[i+1]*row2[i+1] for i in range(len(row1)-1)] )/(math.sqrt(sum([row1[i+1]**2 for i in range(len(row1)-1)]))*math.sqrt(sum([row2[i+1]**2 fo
```

نتایج بخش اولِ عملی (KNN) :

در صورت سوال دقت ذکر شده که بهتر بود با توجه به تعریفی که روبروی آن نوشته شده است، از کلمه‌ی صحت استفاده می‌شد. لذا ما در برچسب‌گذاریِ نمودارهایمان از کلمه‌ی صحت استفاده نمودیم.

نمودار خواسته شده در صورت سوال را برای دو معیارِ فاصله‌ی اقلیدسی و شباهتِ کسینوسی رسم نمودیم که به صورت زیر است.



همچنین صحت‌های موجود در نمودارِ فوق را در خروجیِ ترمینال نیز پرینت کردیم که به صورت زیر می‌باشد. وکتور اول مربوط به معیار فاصله‌ی اقلیدسی است و وکتورِ دوم مربوط به معیارِ شباهتِ کسینوسی.

```
D:\dars\[illegible] [illegible]\HW3_9531701\Codes>python3 part1.py
[0.8766666666666667, 0.8342857142857143, 0.8428571428571429, 0.8257142857142857, 0.81, 0.8019047619047619, 0.8, 0.7923809523809524]
[0.8271428571428572, 0.7280952380952381, 0.7476190476190476, 0.7233333333333334, 0.7114285714285714, 0.6938095238095238, 0.6785714285714286, 0.6766666666666666]

D:\dars\[illegible] [illegible]\HW3_9531701\Codes>
```

که در جدول زیر برای راحتیِ مقایسه مجددا قرار دادیم.

	K=1	K=2	K=3	K=4	K=5	K=6	K=7	K=8
صحت (معیار فاصله اقلیدسی)	0.8766	0.8342	0.8428	0.8257	0.81	0.8019	0.8	0.7923
صحت (معیار شباهت کسینوسی)	0.8271	0.728	0.7476	0.7233	0.7114	0.6938	0.6785	0.6766

همانطور که مشاهده می‌فرمایید، بهترین نتیجه در این مسأله و برای این دیتاست، در $k=1$ و برای معیارِ فاصله‌ی اقلیدسی برآورده شد که صحتِ آن برابر 0.8766 یا 87.66% می‌باشد.

بخش دوم – Decision Tree

مراحل پیاده‌سازی بخش دوم عملی (Decision Tree) :

در بخش دوم نیز مجددا از تابعِ load_dataset() که در بخش قبل پیاده کرده بودیم، برای استخراج داده‌ها و قرار دادنِ آنها در یک لیست استفاده کردیم با این تفاوت که قسمتِ مربوط به float نمودن را حذف نمودیم. دلیلِ اینکار این بود که در اینجا داده‌ها در همه‌جا به صورت کمیتی نیستند و چون ادغامی از کمیت و کیفیت در داده‌ها مشاهده می‌کنیم، مجبوریم که مقادیر را کد کنیم.

برای این هدف از LabelEncoder استفاده می‌کنیم و بدین صورت داده‌ها کد می‌شوند. البته این کدینگ معضلاتی را در پیاده‌سازیِ confusion matrix ایجاد می‌کند که در ادامه روش برطرف کردنِ مشکلِ پیاده‌سازیِ مربوط به آن را خواهم گفت.

```
13  def load_dataset(filename):
14      with open(filename) as csvfile:
15          lines = csv.reader(csvfile)
16          dataset = list(lines)
17      return dataset
```

```
dataset = load_dataset("./datasets/car.csv")
unique_words = getUniqueWords(dataset);

le = preprocessing.LabelEncoder()
le.fit(unique_words)
transformedDataset = [le.transform(dataset[i]) for i in range(len(dataset))]
datas = [x[0:len(transformedDataset[1])-1] for x in transformedDataset]
labels = [x[-1] for x in transformedDataset]
datas, labels = shuffle(datas, labels)
X_train, X_test, y_train, y_test = train_test_split(datas, labels, test_size=0.2)
```

همانطور که بالا مشاهده می‌کنید داده‌ها را پس از انکد کردن، شافل نموده و سپس به تقسیم کننده‌ی test و train داده‌ایم.

حال که داده ها و برچسب‌هایشان آماده است، آن را در یک لوپ و با عمق‌های مختلف به درختِ تصمیم داده‌ایم و داده‌های تست را برحسبِ مدلِ بدست آمده برچسب‌گذاری و در واقع کلاس بندی کرده‌ایم و در آخر هم موارد خواسته شده را پلات کرده و confusion matrix را که در ادامه نحوه‌ی پیاده‌سازی‌اش را خواهم گفت پرینت نمودیم. همچنین طبق خواسته‌ی سوال درختِ تصمیم را هم با دستورِ plot_tree در پوشه ذخیره می‌نماییم.

```
confusion_matrix_vector = list()
accuracy_vector = list()
clf_vector = list()
for i in range(5):
    max_depth = i+1
    clf = tree.DecisionTreeClassifier(max_depth=max_depth)
    clf = clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    confusion_matrix_vector.append(get_confusion_matrix(y_test, y_pred))
    accuracy_vector.append(get_accuracy(y_test, y_pred))
    clf_vector.append(clf)
#print(accuracy_vector)
plt.figure()
plt.plot([x+1 for x in range(len(accuracy_vector))] ,accuracy_vector,'-')
plt.gca().yaxis.set_major_formatter(StrMethodFormatter('{x:,.0%}'))
plt.xlabel("Depth of Tree")
plt.ylabel("Accuracy (%)")
plt.ylim((0,1))
plt.title('Decision Tree Classifier')
plt.show()

maxpos = accuracy_vector.index(max(accuracy_vector))
print("The best depth between these is ", maxpos + 1, "\n")

print("At this depth, confusion matrix is: ")
print(confusion_matrix_vector[maxpos], "\n")

print("At this depth, accuracy is: ")
print(accuracy_vector[maxpos])

fig = plt.figure(figsize=(25,20))
_ = tree.plot_tree(clf_vector[maxpos])
fig.savefig('decisionTree.png')
```

پیاده‌سازی confusion matrix به صورت زیر است.

```
34  def get_confusion_matrix(actual, pred):
35      K = len(np.unique(actual))
36      output = np.zeros((K, K)).astype(int)
37      l = preprocessing.LabelEncoder()
38      l.fit(np.unique(actual))
39      for i in range(len(actual)):
40          x = l.transform([actual[i]])
41          y = l.transform([pred[i]])
42          output[x[0]][y[0]] += 1
43      return output
44
45
```

همانطوری که مشاهده می‌کنید در هر گام برچسب‌های حقیقیِ داده‌های تست و برچسب‌های پیش‌بینی شده را برای ایندکس گذاری استفاده کرده‌ایم به نحوی که سطرها مبینِ برچسب‌های حقیقی و ستونِ ماتریس مبینِ برچسب‌های پیشبینی باشد و لذا ترکیبِ این دو مقایسه‌ای بینِ هر برچسبِ حقیقی با پیش‌بینی شده‌هایش را انجام می‌دهد.

نکته‌ای در اینجا حائز اهمیت است این است که ما دوباره انکدینگ را بر روی برچسب‌ها که خود یکبار انکد شده بودند، انجام دادیم. دلیلِ آن این است که ایندکس‌ها باید بین صفر تا سه (به تعداد برچسب‌ها یعنی چهار تا) باشند ولی در مرتبه‌ی اولی که انکد کردیم، کل لیست را انکد نمودیم و نه فقط برچسب‌ها را، به همین جهت این امکان وجود دارد که کدهای برچسب از سه بیشتر شده و اروِر خارجِ بازه بودنِ ایندکس بخوریم. لذا دوباره برچسب‌ها را انکد کردیم.

تابع دیگر هم get_accuracy بود که در بخش قبل پیاده‌سازیِ آن را توضیح دادم.

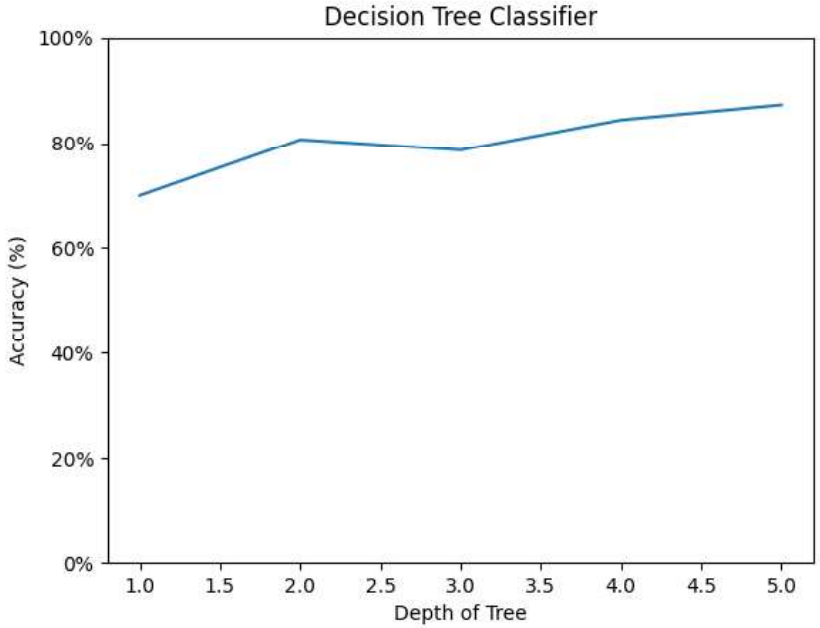
```
27  def get_accuracy(actual, predicted):
28      correct = 0
29      for i in range(len(actual)):
30          if actual[i] == predicted[i]:
31              correct += 1
32      return correct / float(len(actual))
33
```

همچنین برای پیدا کردنِ کلماتِ unique در دیتاست به جهتِ کد کردن نیز یک تابع با نامِ getUniqueWoeds() پیاده نمودم که به صورت زیر می‌باشد.

```
19  def getUniqueWords(dataset):
20      unique_words = list()
21      for x in range(len(dataset)):
22          for y in range(len(dataset[1])):
23              if dataset[x][y] not in unique_words:
24                  unique_words.append(dataset[x][y])
25      return unique_words
26
```

نتایج بخش دومِ عملی (Decision Tree) :

برای آنکه عمق‌های مختلف برای درخت را بررسی کنم، پنج عمق در نظر گرفته و صحتِ کلاسیفایر را بر روی داده‌های تست به ازای این پنج عمق بررسی نمودم که به صورت زیر است.

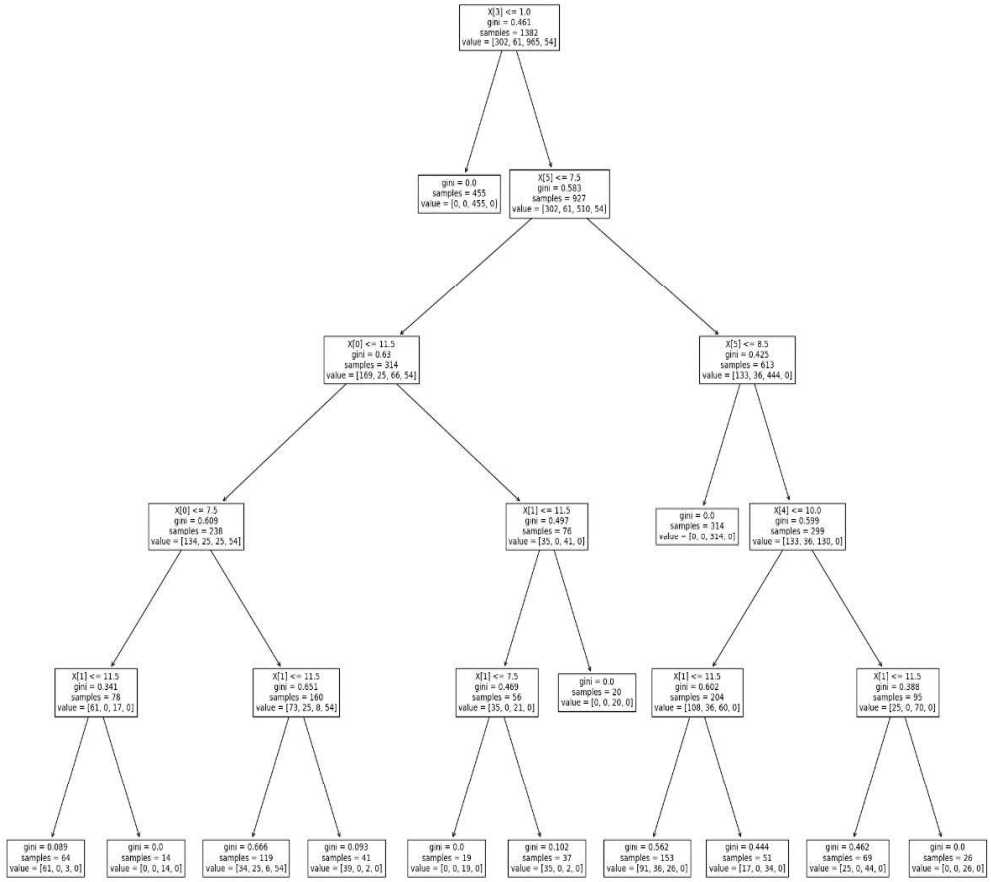


همچنین خروجیِ مربوط به confusion matrix که پیاده‌سازی کردیم به ازای بهترین عمق بین مواردی که تست کردیم به صورت زیر است. و صحتِ آن را نیز پرینت کردیم که خروجی ترمینال به صورتِ زیر می‌باشد.

```
D:\dars\[redacted]\HW3_9531701\Codes>python3 part2.py
The best depth between these is  5

At this depth, confusion matrix is:
[[ 59   0   9  10]
 [ 10   0   0   7]
 [  7   0 234   1]
 [  0   0   0   9]]

At this depth, accuracy is:
0.8728323699421965

D:\dars\[redacted]\HW3_9531701\Codes>
```

در موردِ تحلیلِ خروجیِ confusion matrix باید بگویم که طبیعتاً طبقِ پیاده‌سازی‌ای که خدمتتان عرض نمودم، باید قطر اصلی تعداد عناصری باشند که برچسبشان را کلاسیفایر درست تشخیص داده است. لذا صحتِ گرفته شده را مشاهده می‌کنید که در حدودِ 0.87 درصد به ازای عمقِ ۵ برای درختِ تصمیم است. پس ثییزخطای بوجود آمده در تشخیص باید حدودِ 0.13 باشد.

در آخر نیز که طبق آنچه که ذکر کردیم، درختِ تصمیم را در این حالت با استفاده از دستورِ plot_tree رسم نمودیم.



بخش سوم – Naive Bayes

مراحل پیاده‌سازی بخش سوم عملی (Naïve Bayes) :

در ابتدا مانند بخش‌های قبل، داده‌ها را از فایل استخراج کردیم با ایت تفاوت که بدلیلِ تفاوت در فرمتِ فایل، سطر به سطر از فایل را خوانده و با توجه به دو space و یک tab ای که در نگارشِ فایل نهفته بود، اقدام به جداسازیِ سطر ها از برچسب متناظرشان نمودیم.

```
13
14  def load_dataset():
15      with open("./datasets/IMDB_review_labels.txt", encoding="utf8") as f:
16          text = []
17          label = []
18          for line in f:
19              temp = line.split('  \t')
20              text.append(temp[0])
21              label.append(temp[1])
22
23          return text, label
24
```

سپس عملیاتِ data cleaning را بر روی لیستی از سطرها اعمال کردیم.

```
38
39  def data_cleaning(text):
40      wordnet_lemmatizer = WordNetLemmatizer()
41      stops = stopwords.words('english')
42      nonan = re.compile(r'[^a-zA-Z ]')
43      output = []
44      for i in range(len(text)):
45          sentence = nonan.sub('', text[i])
46          words = word_tokenize(sentence.lower())
47          filtered_words = [w for w in words if not w.isdigit() and not w in stops and not w in string.punctuation]
48          tags = pos_tag(filtered_words)
49          cleaned = ''
50          for word, tag in tags:
51              if tag == 'NN' or tag == 'NNS' or tag == 'VBZ' or tag == 'JJ' or tag == 'RB' or tag == 'NNP' or tag == 'NNPS' or tag == 'RBR':
52                  cleaned = cleaned + wordnet_lemmatizer.lemmatize(word) + ' '
53          output.append(cleaned.strip())
54      return output
55
```

پس از انجامِ عملیاتِ data cleaning، داده‌های خام به داده‌های مناسب برای پردازش تبدیل شده‌اند اما یک گام دیگر لازم است و آن بدست بدست آوردنِ ماتریسِ tf-idf است که در واقع ساختِ مدل با توجه به این ماتریس صورت می‌گیرد.

لذا چیزی که در ابتدا مورد نیاز بود، بدست آوردنِ تعدادی اسنادی بود که کلماتِ موجود در متن را دارا می‌باشند. به همین دلیل تابعِ get_word_frequency() را به صورت زیر پیاده‌سازی نمودیم. در واقع کاری که انجام می‌دهیم این است که با توجه به فاصله‌ی spaceای که بینِ کلمات موجود است (توجه شود که بر روی جملات data cleaning صورت گرفته و بینِ ریشه‌ی کلماتِ هر یک از سندها یک فاصله‌ی space موجود است) ، اقدام به جداسازیِ هر یک کرده و بدین گونه فرکانس آن را در سندها می‌یابیم.

```
def get_word_frequency(text):
    unique_words = {}
    for i in text:
        line = i.split(' ')
        for j in line:
            if j not in unique_words:
                unique_words[j] = 1
            else:
                unique_words[j] = unique_words[j] + 1
    return unique_words
```

حال عملیاتِ بدست آوردنِ ماتریسِ tfidf را توسطِ پیاده‌سازیِ تابعی که اسم آن را calculate_tfidf گذاشته‌ایم، انجام می‌دهیم. نحوه‌ی پیاده‌سازی خیلی ساده و بدین صورت است که تک تکِ سندهای clean شده را بررسی کرده و تکرارِ کلماتش را در سندِ مذکور محاسبه می‌کنیم. سپس با داشتنِ tf و هم idf که از پیاده‌سازیِ تابعِ قبلیمان داشتیم، اقدام به محاسبه‌ی مقدار درایه‌های ماتریسِ tfidf با توجه به فورمولی که در صورتِ سوال ذکر شد، می‌نماییم.

```
def calculate_tfidf(text, unique_words):
    text_list = list()
    for i in text:
        line = i.split(' ')
        word_list = {}
        line_list = list()
        for k in line:
            if k not in word_list:
                word_list[k]=1
            else:
                word_list[k]=word_list[k]+1

        for j in unique_words.keys():
            if j in word_list.keys():
                line_list.append(math.log(1+word_list[j])+math.log(len(text)/unique_words[j]))
            else:
                line_list.append(math.log(len(text)/unique_words[j]))
        text_list.append(line_list)
    return text_list
```

در نهایت نیز نتایج را به صورت bar از نقطه نظرِ صحت و دقتِ کلاسیفایر بر روی داده‌های تست و آموزش (طبق خواسته‌ی سوال) رسم می‌نماییم که در قسمتِ نتایجِ این بخش می‌توانید مشاهده کنید.

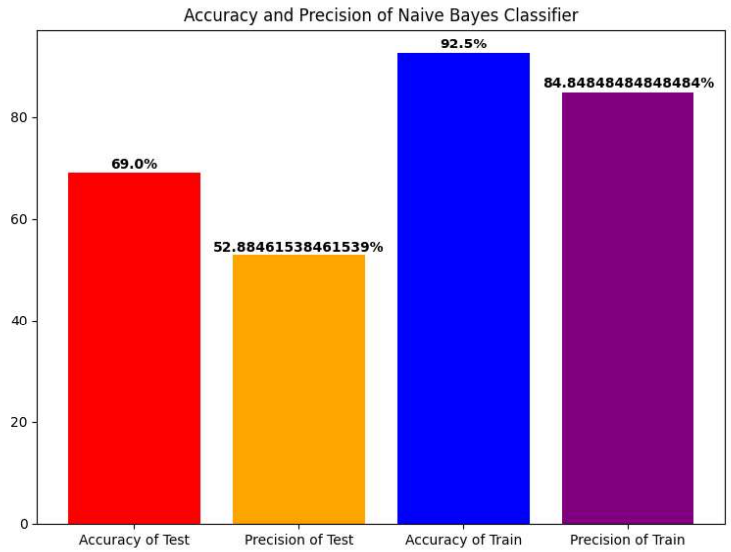
همچنین توجه بفرمایید که طبقِ پیشنهادی که سوال ارائه کرده است، فرضِ گوسی بودنِ فیچرها را در نظر گرفته‌ایم و از Gaussian Naïve Bayes برای ساختِ مدل استفاده نموده‌ایم. که در قطعه کدِ زیر در خطِ 97 این موضوع را مشاهده می‌فرمایید.

این قطعه از کد نیز به صورت زیر است.

```
model = GaussianNB().fit(x_train, y_train)
pred = model.predict(x_test)
accuracy.append(get_accuracy(y_test, pred))
precision.append(get_precision(y_test, pred))

print("Accuracy of naive Bayes classifier for the test data is ", accuracy[0])
print("Precision of naive Bayes classifier for the test is ", precision[0], "\n")

pred = model.predict(x_train)
accuracy.append(get_accuracy(y_train, pred))
precision.append(get_precision(y_train, pred))

print("Accuracy of naive Bayes classifier for the train datas is ", accuracy[1])
print("Precision of naive Bayes classifier for the train datas is ", precision[1], "\n")

plot_list = [accuracy[0], precision[0], accuracy[1], precision[1]]

plt.figure(figsize=(8,8))
colors_list = ['Red','Orange', 'Blue', 'Purple']
graph = plt.bar(['Accuracy of Test','Precision of Test','Accuracy of Train','Precision of Train'], plot_list, color = colors_list)
plt.title('Accuracy and Precision of Naive Bayes Classifier')

i = 0
for p in graph:
    width = p.get_width()
    height = p.get_height()
    x, y = p.get_xy()
    plt.text(x+width/2,
             y+height*1.01,
             str(plot_list[i])+'%',
             ha='center',
             weight='bold')
    i+=1
plt.show()
```

نتایج بخش سومِ عملی (Naïve Bayes) :

نتایج حاصل از دقت و صحت برای ارزیابیِ عملکرد کلاسیفایر بر روی داده‌های تست و آموزش به عنوان داده‌های ارزیابی



همچنین خروجی ترمینال که در نمودار فوق در رسم نمودیم به صورت زیر است.

```
D:\dars\▯▯▯▯ ▯▯▯▯\HW3_9531701\Codes>python3 part3.py
Accuracy of naive Bayes classifier for the test data is  69.0
Precision of naive Bayes classifier for the test is  52.88461538461539

Accuracy of naive Bayes classifier for the train datas is  92.5
Precision of naive Bayes classifier for the train datas is  84.84848484848484
```

با توجه به اینکه صحت برای ارزشیابیِ داده‌های تست 69% است لذا خطا برابرِ 31% خواهد بود و نیز برای داده‌های آموزش به عنوانِ داده‌های ارزشیابی نیز، با توجه به آنکه صحت برابر با 92.5% است. لذا خطا برای آن برابر 7.5٪ می‌باشد.

بخش چهارم – SVM – قسمتِ اول

مراحل پیاده‌سازی بخش چهارم عملی (SVM – قسمت اول) :

لازم به ذکر است که در قسمتِ اول مراحلِ پیاده‌سازی دقیقا مشابه با بخشِ سوم می‌باشد که البته خواسته‌ی سوال برای قسمتِ اولِ این بخش هم همین بود. با این تفاوت که در اینجا کلاسیفایرِ ما SVM است.

طبقِ پیشنهادی که طراح سوال کردند که از SVC استفاده کنیم، ما نیز همین کار را انجام دادیم و در اینجا قطعه کدی که مربوط به ساخت مدل است را ارائه می‌کنیم.

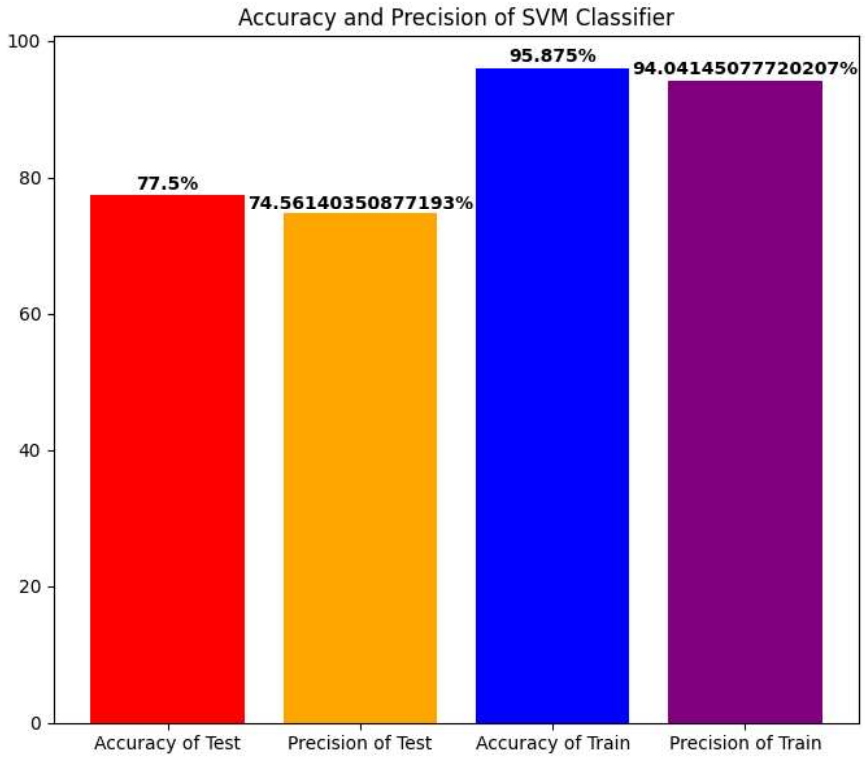
```
96
97    model = SVC(kernel = 'linear').fit(x_train, y_train)
98    pred = model.predict(x_test)
```

به همین سادگی قطع کد بالا را تغییر دادیم. البته پیاده‌سازیِ مربوط به دو تابعِ محاسبه‌ی صحت و نیز محاسبه‌ی دقت به صورت زیر است که با استفاده از تعاریفی که از صحت و دقت داشتیم، صحت را درستیِ پیش‌بینی روی کل داده‌ها و دقت را درستیِ پیش‌بینی روی داده‌های مثبت طبقِ تعریفِ مراجع، پیاده نموده ایم که به صورت زیر است.

```
23
24    def get_accuracy(actual, predicted):
25        truePrediction = 0
26        for i in range(len(actual)):
27            if actual[i] == predicted[i]:
28                truePrediction += 1
29        return 100 * truePrediction / float(len(actual))
30
31    def get_precision(actual, predicted):
32        truePositive = 0
33        for i in range(len(actual)):
34            if actual[i] == predicted[i] and actual[i]=='1':
35                truePositive += 1
36        return 100 * truePositive / float(actual.count('1'))
37
```

نتایج بخش چهارم عملی (SVM – قسمت اول) :

نتایج حاصل از دقت و صحت برای ارزیابیِ عملکرد کلاسیفایر بر روی داده‌های تست و آموزش به عنوان داده‌های ارزیابی



خروجی ترمینال که در نمودار فوق در رسم نمودیم به صورت زیر است.

```
D:\dars\▯▯▯▯ ▯▯▯▯\HW3_9531701\Codes>python3 part4_a.py
Accuracy of SVM Bayes classifier for the test data is  77.5
Precision of SVM classifier for the test is  74.56140350877193

Accuracy of SVM classifier for the train datas is  95.875
Precision of SVM classifier for the train datas is  94.04145077720207
```

با توجه به اینکه صحت برای ارزشیابیِ داده‌های تست 77.5% است لذا خطا برابرِ 22.5% خواهد بود و نیز برای داده‌های آموزش به عنوانِ داده‌های ارزشیابی نیز، با توجه به آنکه صحت برابر با 95.875% است. لذا خطا برای آن برابر 4.125% می‌باشد.

در انتهای قسمتِ اولِ این بخش، سوال از ما خواسته که مقایسه‌ای بینِ نتایج این قسمت و بخشِ قبل انجام دهیم. همانطور که مشاهده می‌کنید هم صحت و هم دقت در SVM از naïve Bayes بیشتر است. اختلافِ صحت بینِ این دو برابر با 8.5% برای داده‌های تست است که عملکردِ بهترِ SVM را نشان می‌دهد. دلیلِ این موضوع می‌تواند این باشد که naïve Bayes داده‌ها را از نقطه نظرِ فیچرهایشان به صورت مستقل در نظر گرفته و مدل را بر مبنای این استقلال می‌سازد ولی این در حالی است که SVM در ساختِ مدل به تعاملاتِ موجود بینِ آنها تا حدودی توجه می‌کند و به نظرم به همین دلیل است که SVM عملکردِ بهتری را از خود نشان می‌دهد.

بخش چهارم – SVM – قسمتِ دوم

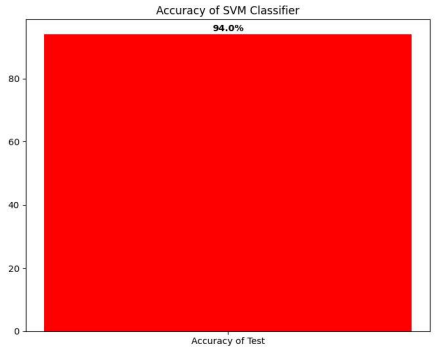
مراحل پیاده‌سازی بخش چهارم عملی (SVM – قسمت دوم) :

در این بخش نیز مشابه آنچه انجام می‌دادیم، در ابتدا داده‌ها را از فایلِ csv با تابعی که پیاده نمودیم می‌خوانیم و پس از تقسیمِ آنها به داده‌های تست و آموزش با نسبتِ گفته شده در مسأله (۱۰ درصد تست و ۹۰ درصد آموزش) به مدل می‌دهیم. طبیعیتا خروجی صحت (تعداد صحیح بودنِ پیش‌بینی‌ها) را هم با تابعی که در بخش‌های قبل پیاده کردیم، محاسبه می‌کنیم.

```
10  def loadDataset(filename):
11      with open(filename) as csvfile:
12          lines = csv.reader(csvfile)
13          dataset = list(lines)
14          for x in range(len(dataset)):
15              for y in range(len(dataset[1])):
16                  dataset[x][y] = float(dataset[x][y])
17      return dataset
18
19  def get_accuracy(actual, predicted):
20      truePrediction = 0
21      for i in range(len(actual)):
22          if actual[i] == predicted[i]:
23              truePrediction += 1
24      return 100 * truePrediction / float(len(actual))
25
```

```
model = SVC(kernel = 'linear').fit(X_train, y_train)
pred = model.predict(X_test)
accuracy = get_accuracy(y_test, pred)
print("Accuracy of SVM classifier for the test datas is ", accuracy)
plot_list = [accuracy]

plt.figure(figsize=(8,8))
colors_list = ['Red']
graph = plt.bar(['Accuracy of Test'], plot_list, color = colors_list)
plt.title('Accuracy of SVM Classifier')

i = 0
for p in graph:
    width = p.get_width()
    height = p.get_height()
    x, y = p.get_xy()
    plt.text(x+width/2,
             y+height*1.01,
             str(plot_list[i])+'%',
             ha='center',
             weight='bold')
    i+=1

fig, ax = plt.subplots()
title = ('SVM (Linear SVC) Classifier ')
X0, X1 = [x[0] for x in X_train], [x[1] for x in X_train]
xx, yy = make_meshgrid(X0, X1)
plot_contours(ax, model, xx, yy, cmap=plt.cm.coolwarm, alpha=0.8)
ax.scatter(X0, X1, c=y_train, cmap=plt.cm.coolwarm, s=20, edgecolors='k')
ax.set_xlabel('x axis')
ax.set_ylabel('y axis')
ax.set_xticks(())
ax.set_yticks(())
ax.set_title(title)
plt.show()
```

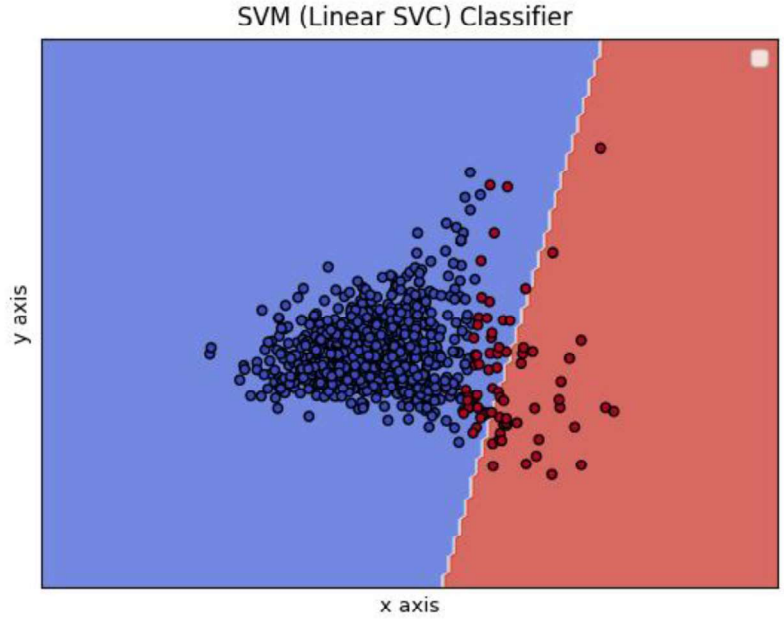
نتایج بخش چهارم عملی (SVM – قسمت دوم) :



همانطور که مشاهده می‌کنید، صحتِ بدست آمده برای داده‌های تست برابر با 94% است یعنی از 100 داده‌ای که به این کلاسیفایر داده‌ایم، 94 عدد پیش‌بینیِ صحیح انجام داده که آن را در ترمینال هم پیرینت نمودیم. این یعنی خطای 6%

```
D:\dars\▯▯▯▯ ▯▯▯▯\HW3_9531701\Codes>python3 part4_b.py
Accuracy of SVM classifier for the test datas is  94.0
No handles with labels found to put in legend.

D:\dars\▯▯▯▯ ▯▯▯▯\HW3_9531701\Codes>
```

در نهایت نیز شکل مربوط به اعمالِ کلاسیفایر به نقاطِ آموزش را نیز به همراه مرز بدست آمده از آنچه مدل اعمال می‌کند را نیز ترسیم نمودیم که به صورت زیر می‌باشد.



با تشکر – بدیعی