The University of Melbourne, School of Computing and Information Systems

COMP30027 Machine Learning, 2023 Semester 1

Assignment 1: Music genre classification with naive Bayes

Student ID(s): `1174154`

This iPython notebook is a template which you will use for your Assignment 1 submission.

Marking will be applied on the four functions that are defined in this notebook, and to your responses to the questions at the end of this notebook (Submitted in a separate PDF file).

NOTE: YOU SHOULD ADD YOUR RESULTS, DIAGRAMS AND IMAGES FROM YOUR OBSERVATIONS IN THIS FILE TO YOUR REPORT (the PDF file).

You may change the prototypes of these functions, and you may write other functions, according to your requirements. We would appreciate it if the required functions were prominent/easy to find.

Adding proper comments to your code is MANDATORY.

In [ ]:
```
# run this as a start
import math
import random
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from matplotlib.lines import Line2D
from sklearn.metrics import confusion_matrix
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_sc

EPSILON = 0.00000001
BASE = -99999999
```

In [ ]:
```
# This function should prepare the data by reading it from a file and converting
def preprocess(filename):
    # test data classifying
    df = pd.read_csv(filename)

    X = df.iloc[:, 1:-1]
    y = df.iloc[:,-1]
    return X, y, df
```

In [ ]:
```
# This function should calculate prior probabilities and likelihoods from the tr
# them to build a naive Bayes model

def train(X_train, y_train, train_data_df):
    #  X_test, y_test, test_data_df,  = preprocess()

    # calculate prior probability by getting unique labels and
    prior_prob = {}

    unique_labels, counts = np.unique(y_train, return_counts=True)
    n = counts.sum()

    for i in range(len(unique_labels)):
        prior_prob[unique_labels[i]] = (counts[i] / n).round(3)

    # calculate miximum likelihood
    # gives attributes for the trainning dataset
    attribute_lists = X_train.columns
    likelihood_parameters = list()

    # for each label, correspond with a attribute, calculate the mean and sd for
    for label in unique_labels:
        data_with_label = train_data_df[train_data_df['label'] == label]

        for attribute in attribute_lists:
            train_attribute_mean = data_with_label[attribute].mean()
            train_attribute_std = data_with_label[attribute].std()
            likelihood_parameters.append([train_attribute_mean, train_attribute_
    return prior_prob, likelihood_parameters
```

In [ ]:
```
# This function should predict classes for new items in a test dataset

def predict(prior_prob, likelihood_parameters, X_test):
    # prior_prob, likelihood_parameters, X_test, y_test, test_data_df = train()

    # get the labels and attributes for our test dataset
    unique_labels = list(prior_prob.keys())
    attribute_lists = X_test.columns

    choice_list = list()
    for index, row in X_test.iterrows():
        row_len = len(row)

        highest_posterior = BASE
        best_choice = None
        for i in range(len(unique_labels)):
            # take a label and find the prior probability for this label
            label = unique_labels[i]
            prior = prior_prob[label]

            # parameters for each attribute under this label
            correspoding_parameters = likelihood_parameters[int(i*row_len) : int

            posterior = math.log(prior)
            for j in range(len(attribute_lists)):
                # attribute value and parameters for this attribute
                attribute = attribute_lists[j]
                parameters = correspoding_parameters[j]
```

```
                # find likelihood using values invoved
                value = row[attribute]
                mu = parameters[0]
                sd = parameters[1]
                max_likelihood = 1 / (sd * math.sqrt(2 * math.pi)) * math.exp(-(

                # if the max_likelihood is zero, we can use smoothing to deal wi
                if (max_likelihood <= EPSILON):
                    posterior += math.log(1/(row_len+1))
                else:
                    posterior += math.log(max_likelihood)

            if (posterior > highest_posterior):
                highest_posterior = posterior
                best_choice = label

        choice_list.append(best_choice)
    return choice_list
```

In [ ]:

```
# This function should evaliate the prediction performance by comparing your mod
# truth labels

def evaluate(choice_list, y_test):
    # returns the table between the actual and predicted sets
    num = len(np.unique(y_test))
    if (num > 2):
        accuracy = accuracy_score(y_test, choice_list)
        precision = precision_score(y_test, choice_list, average = "weighted")
        recall = recall_score(y_test, choice_list, average = "weighted")
        f1 = f1_score(y_test, choice_list, average = "weighted")
    if (num == 2):
        accuracy = accuracy_score(y_test, choice_list)
        precision = precision_score(y_test, choice_list, pos_label="classical")
        recall = recall_score(y_test, choice_list, pos_label="classical")
        f1 = f1_score(y_test, choice_list, pos_label="classical")

    return accuracy, precision, recall, f1
```

In [ ]:

```
# The following code is used to train the gztan files and get a cross table for

# train_file = 'COMP30027_2023_asst1_data\gztan_train.csv'
# test_file = 'COMP30027_2023_asst1_data\gztan_test.csv'

# X_train, y_train, train_data_df = preprocess(train_file)
# X_test, y_test, test_data_df = preprocess(test_file)

# prior_prob, likelihood_parameters = train(X_train, y_train, train_data_df)

# choice_list = predict(prior_prob, likelihood_parameters, X_test)

# table = evaluate(choice_list, y_test)

# print(table)
```

In [ ]:

```
# this function is used to run all the four basic functions and return accuracy,
def run_all(train_file_name, test_file_name):
    X_train, y_train, train_data_df = preprocess(train_file_name)
    X_test, y_test, test_data_df = preprocess(test_file_name)
```

```
    prior_prob, likelihood_parameters = train(X_train, y_train, train_data_df)

    choice_list = predict(prior_prob, likelihood_parameters, X_test)

    accuracy, precision, recall, f1 = evaluate(choice_list, y_test)

    return accuracy, precision, recall, f1
```

Task 1. Pop vs. classical music classification

NOTE: you may develope codes or functions to help respond to the question here, but your formal answer must be submitted separately as a PDF.

Q1

Compute and report the accuracy, precision, and recall of your model (treat "classical" as the "positive" class).

In [ ]:
```
train_file = 'COMP30027_2023_asst1_data\pop_vs_classical_train.csv'
test_file = 'COMP30027_2023_asst1_data\pop_vs_classical_test.csv'

accuracy, precision, recall, f1 = run_all(train_file, test_file)

print("The accuracy is", accuracy)
print("The precision is", precision)
print("The recall is", recall)
```

```
The accuracy is 0.9767441860465116
The precision is 0.9523809523809523
The recall is 1.0
```

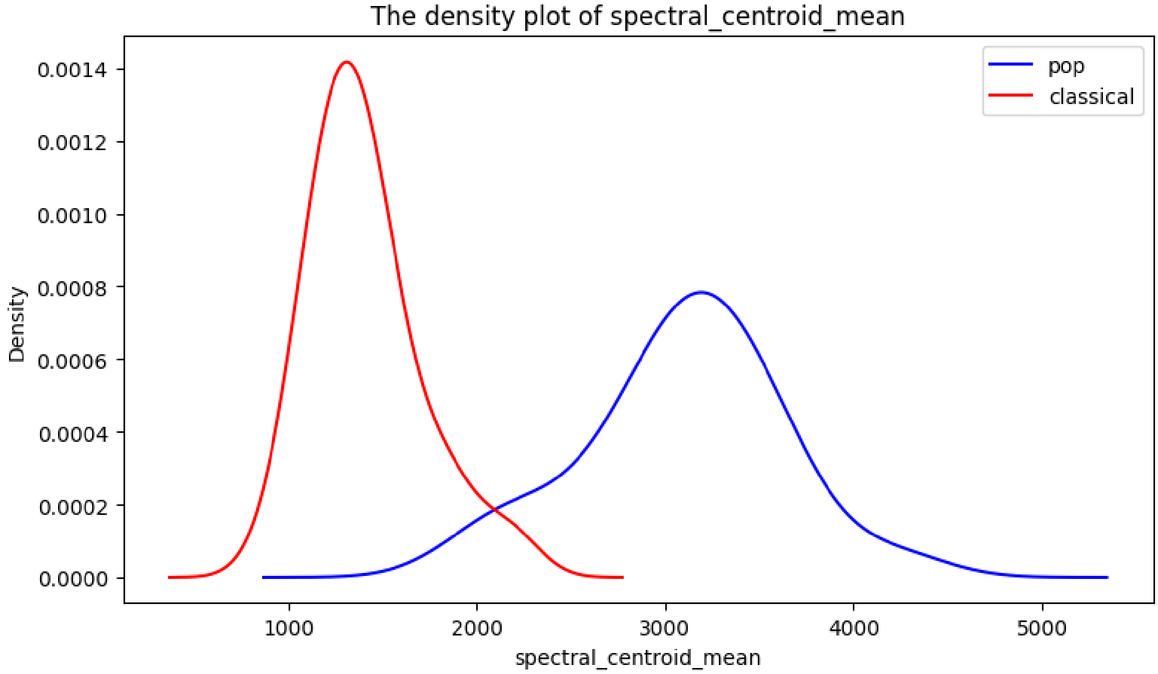
Q2

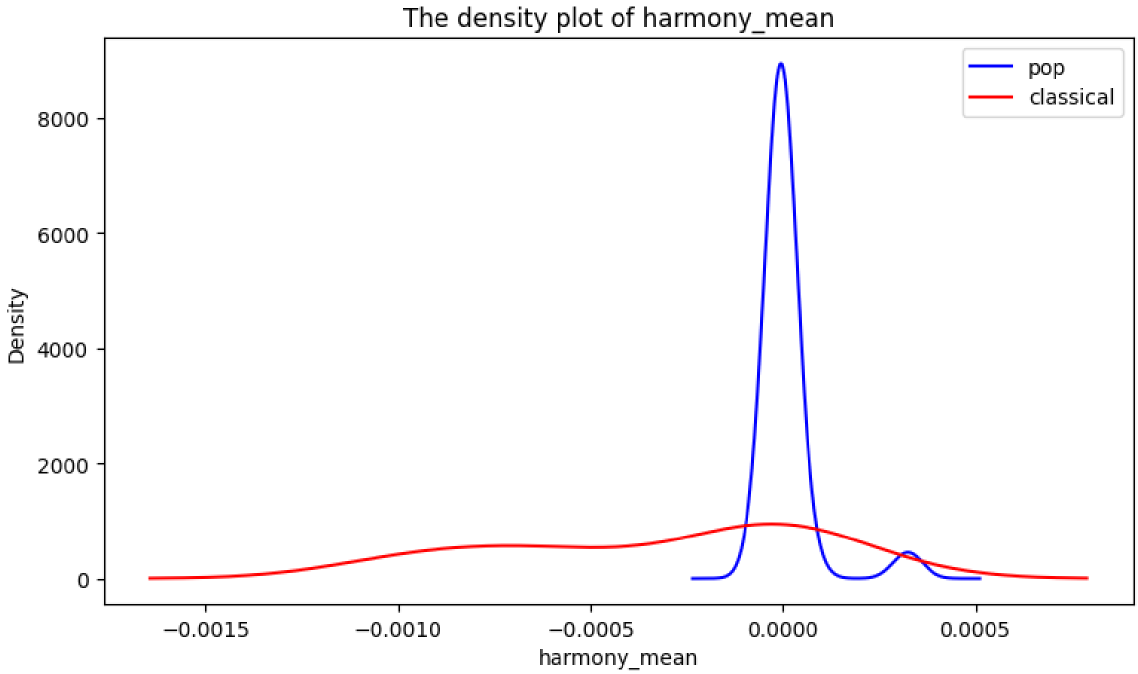
For each of the features X below, plot the probability density functions P(X|Class = pop) and P(X|Class = classical). If you had to classify pop vs. classical music using just one of these three features, which feature would you use and why? Refer to your plots to support your answer.

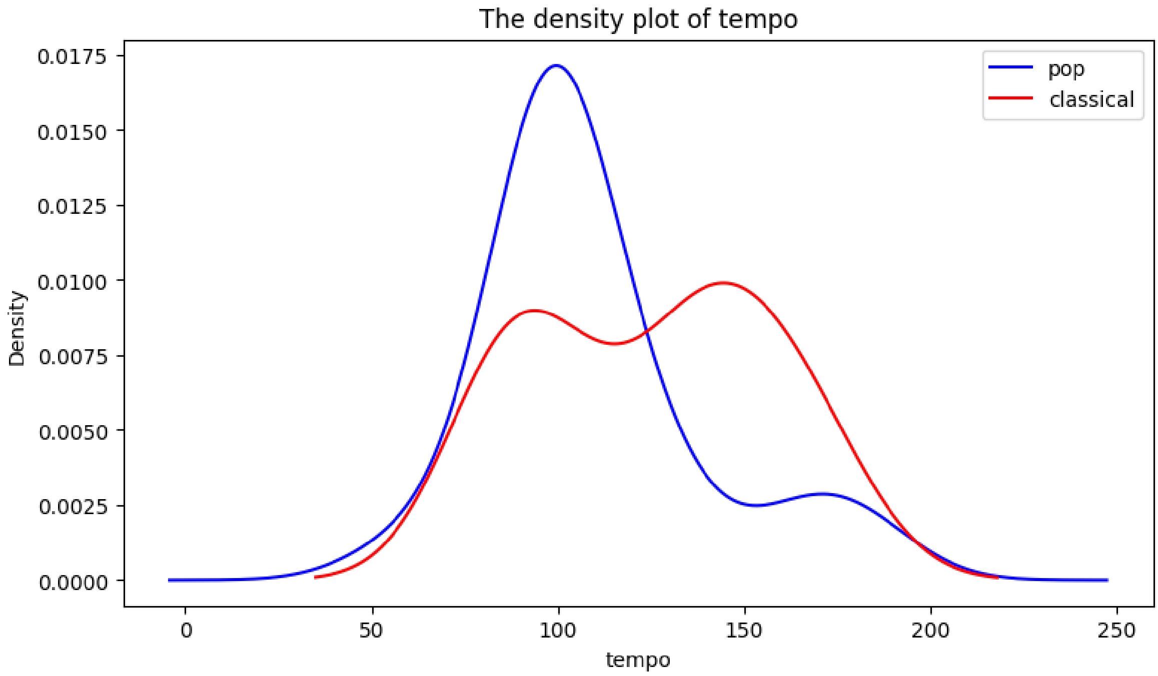
- spectral centroid mean
- harmony mean
- tempo

In [ ]:
```
X_test, y_test, test_data_df = preprocess(test_file)

pop_data = test_data_df[test_data_df['label'] == 'pop']
classical_data = test_data_df[test_data_df['label'] == 'classical']

custom_lines = [Line2D([0], [0], color="blue"),
                Line2D([0], [0], color="red")]

plt.figure(figsize=(8,14))

# create a graph consists three sub density plots by using different attributes
```

```
plt.subplot(311).legend(custom_lines, ['pop', 'classical'])
pop_data.spectral_centroid_mean.plot.density(color='blue')
classical_data.spectral_centroid_mean.plot.density(color='red')
plt.xlabel("spectral_centroid_mean")
plt.title("The density plot of spectral_centroid_mean")

plt.subplot(312).legend(custom_lines, ['pop', 'classical'])
pop_data.harmony_mean.plot.density(color='blue')
classical_data.harmony_mean.plot.density(color='red')
plt.xlabel("harmony_mean")
plt.title("The density plot of harmony_mean")

plt.subplot(313).legend(custom_lines, ['pop', 'classical'])
pop_data.tempo.plot.density(color='blue')
classical_data.tempo.plot.density(color='red')
plt.xlabel("tempo")
plt.title("The density plot of tempo")

plt.tight_layout()
```


The density plot of spectral_centroid_mean
pop
classical
Density
0.0014
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
1000
2000
3000
4000
5000
spectral_centroid_mean


The density plot of harmony_mean
pop
classical
Density
8000
6000
4000
2000
0
−0.0015
−0.0010
−0.0005
0.0000
0.0005
harmony_mean


The density plot of tempo
pop
classical
Density
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0
50
100
150
200
250
tempo

I may choose spectral_centroid_mean to be the attribute X. The reason to choose one of the three attributes is that we want one attribute to best distinguish two labels (pop and classical). We dont want some values of the attribute results into a difficulty for seperating two labels, that is to find the graph which has the minimum area of overlaps between the blue and red curves. By using visual approximation, we will certainly exclude the temp as it appreatly has a big overlapping, and the overlapping area of the spectral_centroid_mean is similar to a triangle (1000 * 0.0002 / 2 = 0.1) while the area of the harmony mean is roughly a rectangle (0.00025 * 1000 = 0.25). We would like to choose the one with the minimum area, that is spectral_centroid_mean.

Task 2. 10-way music genre classification

NOTE: you may develope codes or functions to help respond to the question here, but your formal answer must be submitted separately as a PDF.

Q3

Compare the performance of the full model to a 0R baseline and a one-attribute baseline. The one-attribute baseline should be the best possible naive Bayes model which uses only a prior and a single attribute. In your write-up, explain how you implemented the 0R and one-attribute baselines.

In [ ]:

Q4

Train and test your model with a range of training set sizes by setting up your own train/test splits. With each split, use cross-fold validation so you can report the performance on the entire dataset (1000 items). You may use built-in functions to set up cross-validation splits. In your write-up, evaluate how model performance changes with training set size.

In [ ]:

Q5

Implement a kernel density estimate (KDE) naive Bayes model and compare its performance to your Gaussian naive Bayes model. You may use built-in functions and automatic ("rule of thumb") bandwidth selectors to compute the KDE probabilities, but you should implement the naive Bayes logic yourself. You should give the parameters of the KDE implementation (namely, what bandwidth(s) you used and how they were chosen) in your write-up.

In [ ]:

Q6

Modify your naive Bayes model to handle missing attributes in the test data. Recall from lecture that you can handle missing attributes at test by skipping the missing attributes and computing the posterior probability from the non-missing attributes. Randomly delete some attributes from the provided test set to test how robust your model is to missing data. In your write-up, evaluate how your model's performance changes as the amount of missing data increases.

In [ ]:
```
# This function should predict classes for new items in a test dataset

def predict_missing_value(prior_prob, likelihood_parameters, X_test):
    # prior_prob, likelihood_parameters, X_test, y_test, test_data_df = train()

    # get the labels and attributes for our test dataset
    unique_labels = list(prior_prob.keys())
    attribute_lists = X_test.columns

    choice_list = list()
    for index, row in X_test.iterrows():
        row_len = len(row)

        highest_posterior = BASE
        best_choice = None
        for i in range(len(unique_labels)):
            # take a label and find the prior probability for this label
            label = unique_labels[i]
            prior = prior_prob[label]

            # parameters for each attribute under this label
            correspoding_parameters = likelihood_parameters[int(i*row_len) : int

            posterior = math.log(prior)
            for j in range(len(attribute_lists)):
                # attribute value and parameters for this attribute
                attribute = attribute_lists[j]
                parameters = correspoding_parameters[j]

                # find likelihood using values invoved
                value = row[attribute]
                mu = parameters[0]
                sd = parameters[1]

                if not math.isnan(value):
                    max_likelihood = 1 / (sd * math.sqrt(2 * math.pi)) * math.ex

                    # if the max_likelihood is zero, we can use smoothing to dea
                    if (max_likelihood <= EPSILON):
                        posterior += math.log(1/(row_len+1))
                    else:
                        posterior += math.log(max_likelihood)

            if (posterior > highest_posterior):
                highest_posterior = posterior
                best_choice = label
```

```
        choice_list.append(best_choice)
    return choice_list
```

In [ ]:
```
def delete_value_by_proportion(test_data_df, proportion):
    df_copy = test_data_df.copy()
    for col in df_copy.columns[1:-1]:
        df_copy.loc[df_copy.sample(frac = proportion).index, col] = np.nan

    return df_copy
```

Out[ ]:

col_0 label	blues	classical	country	disco	hiphop	jazz	metal	pop	reggae	rock
blues	3	1	8	1	0	0	5	0	0	1
classical	0	16	1	0	0	2	0	0	0	1
country	1	0	11	2	0	0	1	0	0	1
disco	0	1	2	8	1	1	6	1	1	1
hiphop	0	1	1	2	3	0	6	4	3	1
jazz	1	1	0	2	0	9	2	1	0	2
metal	0	0	0	1	0	0	18	0	0	1
pop	1	2	3	1	0	1	1	14	0	0
reggae	2	1	3	1	0	1	0	0	5	1
rock	1	2	8	4	0	2	8	0	0	2

In [ ]:
```
# this function is used to run all the four basic functions, but can deal with m
def run_all_missing_value(train_file_name, test_file_name, proportion):
    X_train, y_train, train_data_df = preprocess(train_file_name)
    X_test, y_test, test_data_df = preprocess(test_file_name)

    prior_prob, likelihood_parameters = train(X_train, y_train, train_data_df)

    modefied_test = delete_value_by_proportion(test_data_df, proportion)
    modefied_X = modefied_test.iloc[:, 1:-1]

    choice_list = predict_missing_value(prior_prob, likelihood_parameters, modef

    accuracy, precision, recall, f1 = evaluate(choice_list, y_test)

    return accuracy, precision, recall, f1
```

In [ ]:
```
train_file = 'COMP30027_2023_asst1_data\gztan_train.csv'
test_file = 'COMP30027_2023_asst1_data\gztan_test.csv'

# this give 20 random proportions from 0 to 1 and sorted from low to high
proportions = sorted(np.random.uniform(0,1,20))
results = list()

for proportion in proportions:
    accuracy, precision, recall, f1 = run_all_missing_value(train_file, test_fil
    result = [proportion, accuracy, precision, recall, f1]
    results.append(result)
```

```
result_df = pd.DataFrame(results, columns=['proportion', 'accuracy', 'precision'
print(result_df)
```

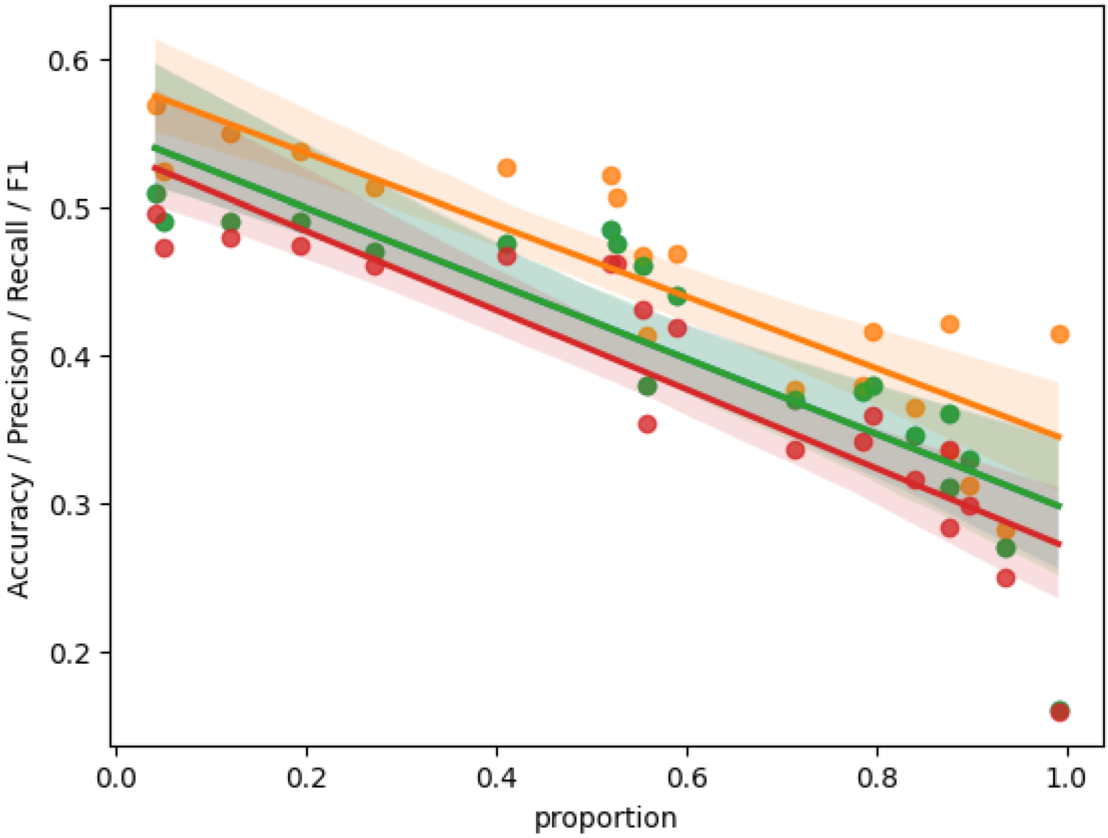
```
    proportion  accuracy  precision  recall        f1
0     0.040865     0.510   0.569590   0.510  0.495475
1     0.049951     0.490   0.524595   0.490  0.473507
2     0.120186     0.490   0.550092   0.490  0.479865
3     0.193746     0.490   0.538104   0.490  0.474130
4     0.271723     0.470   0.513263   0.470  0.460994
5     0.411086     0.475   0.527218   0.475  0.467206
6     0.520939     0.485   0.521695   0.485  0.462429
7     0.525631     0.475   0.506517   0.475  0.462120
8     0.554156     0.460   0.467111   0.460  0.431336
9     0.557445     0.380   0.412656   0.380  0.353915
10    0.588884     0.440   0.468787   0.440  0.418562
11    0.715028     0.370   0.376162   0.370  0.335455
12    0.785039     0.375   0.379405   0.375  0.341246
13    0.796637     0.380   0.415326   0.380  0.358992
14    0.840507     0.345   0.364132   0.345  0.316272
15    0.875768     0.360   0.421106   0.360  0.335452
16    0.876485     0.310   0.335107   0.310  0.283226
17    0.897299     0.330   0.312423   0.330  0.297973
18    0.934827     0.270   0.282404   0.270  0.249711
19    0.992050     0.160   0.414281   0.160  0.158921
```

In [ ]:

```
# plot the points and regression lines for accuracy, precision, recall and f1 on
sns.regplot(x = result_df['proportion'], y = result_df['accuracy'])
sns.regplot(x = result_df['proportion'], y = result_df['precision'])
sns.regplot(x = result_df['proportion'], y = result_df['recall'])
sns.regplot(x = result_df['proportion'], y = result_df['f1'])
plt.ylabel('Accuracy / Precison / Recall / F1')
```

Out[ ]:

```
Text(0, 0.5, 'Accuracy / Precison / Recall / F1')
```


0.6
0.5
0.4
0.3
0.2
Accuracy / Precison / Recall / F1
0.0
0.2
0.4
0.6
0.8
1.0
proportion