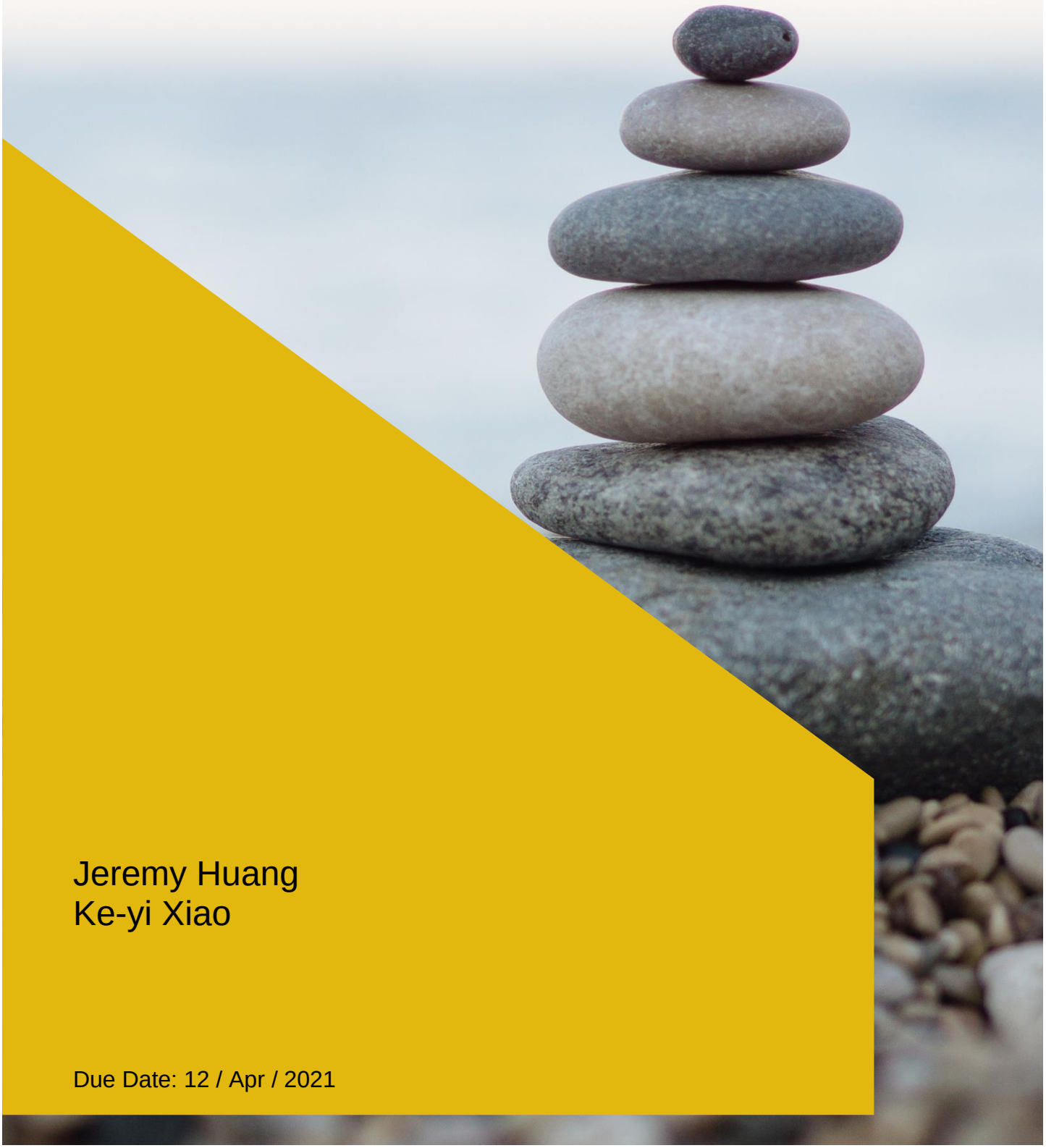
Machine Learning

Project 1 (Comp30027)



Jeremy Huang
Ke-yi Xiao

Due Date: 12 / Apr / 2021

Brief Intro

First, we are given 4 empty functions to code, and we have 2 kinds of data set to be trained, **train.csv & test.csv.**

After reading the specs from canvas, and check the working direction to the Project, we are going to trimmed the data without a instance having all the features(points) are missing value 9999.

We also train the model to be a dictionary and make our data to be easily searched.

Question 1

Code

```
True_Positive = []; False_Positive = []; False_negative = []; Total_Precision = []; Total_Recall = []; Total_F_Score = []

for name in labels:
    the_info = evaluations[name]
    # Collect all the performance data
    True_Positive.append(the_info['TFPN'][0]); False_Positive.append(the_info['TFPN'][1]); False_negative.append(the_info['TFPN'][2])
    Total_Precision.append(the_info['precision']); Total_Recall.append(the_info['recall']); Total_F_Score.append(the_info['F_score'])

TP_Sum = sum(True_Positive); FP_Sum = sum(False_Positive); FN_Sum = sum(False_negative)

# Micro Average Values
micro_average_precision = TP_Sum / (TP_Sum + FP_Sum)
micro_average_recall = TP_Sum / (TP_Sum + FN_Sum)
micro_average_F_score = (2 * micro_average_precision * micro_average_recall) / ((1 * micro_average_precision) + micro_average_recall)

# Macro Average Values
macro_average_precision = sum(Total_Precision) / len(labels)
macro_average_recall = sum(Total_Recall) / len(labels)
macro_average_F_score = sum(Total_F_Score) / len(labels)

print("Micro_average_precision: {}\nMicro_average_recall:    {}\nMicro_average_F_score:   {}\n----------------------------------------\nMacro_average_precision: {}
\nMacro_average_recall:    {}\nMacro_average_F_score:   {}\n".format(micro_average_precision, micro_average_recall, micro_average_F_score, macro_average_precision, \
    macro_average_recall, macro_average_F_score))
```

Result

```
Micro_average_precision: 0.8879781420765027
Micro_average_recall:    0.8879781420765027
Micro_average_F_score:   0.8879781420765027
----------------------------------------
```

```
Macro_average_precision: 0.8903779417864419
Macro_average_recall:    0.9026112839276113
Macro_average_F_score:   0.8870969959345972
```

Discussion

Macro-averages and Micro-averages interpretation in different directions as they compute slightly different things. A Macro-average is a straightforward method, which takes the average of the precision and recall of the system in given data. whereas a Micro-average will sum up the individual metrics (TP/TN/FP/FN) for each class, then apply them to get a statistic.

Based on our test result, it shows Macro-average is better than Micro-average, which indicates that the overall performance of data is satisficed (approximately 0.8904).

As we know, one of the advantages of Macro-average is it treats each class equally, therefore the movement as bridge with low precision (0.48717) will not have a great impact on the final result. In contrast, Micro-average is more susceptible to similar data.

Question 2

Code

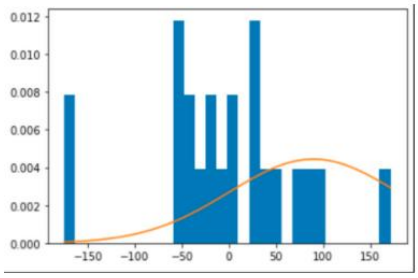
```
for name in labels:
    test_model = trained[name]
    space = []
    for feature in range(len(test_model['mean'])):
        # Get mean and std, then make a table to fit the data
        mu = test_model['mean'][feature]; sigma = test_model['std'][feature]
        each_feature = cleaned.loc[name].iloc[feature]
        matrix = np.linspace(min(each_feature), max(each_feature))

        # Plotting
        plt.hist(each_feature, bins=30, density=True)
        plt.plot(matrix, norm.pdf(matrix, mu, sigma))
        plt.show()
```

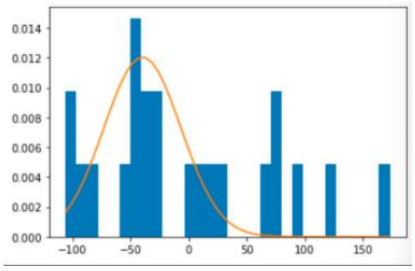
Discussing Case

The Gaussian naive Bates classifier assumes that numeric attributes come from a Gaussian normal distribution, but this assumption is not always true for the numeric attributes in our dataset.

Below are two groups of graphs, which are selected from 220 data images (10 poses, 22 features(points)). The blue bar chart shows the distribution of the training set group, with the yellow curve representing gaussian normal probability density function.



(Figure 2.1)



(Figure 2.2)

As we know, normal distribution is perfectly symmetrical around its peak in the centre. And the mean, median and mode are all the same value.

In Figure 2.1, we normally predict that peak will appear in the range of -50 to 15 from the training set. But the peak from gaussian distribution appears in the range of 70 to 90. Besides, the training set may occur at two peaks at points -50 and 25, which may have a significant impact on predictions as normal distribution only has one peak.

From Figure 2.2, we can clearly see that there is a low density area between 50 and 150 , but in fact, the training set has a high proportion of data in this region. This may lead to wrong judgment, resulting in the final prediction is not satisfactory.

Question 3(1/2)

Code

```
def kdeTrain(labels, cleaned_data):
    """
        This function calculates prior probabilities and return a trained model(a dictionary) with points
    """
    # Get labels and get teh prior for each pose(class)
    prob_class =  [cleaned_data.loc[name].shape[0] for name in labels]
    total = sum(prob_class)
    for i in range(len(prob_class)):
        prob_class[i] = prob_class[i]/total

    # Make a dictionary contains all the features(prior, mean and std) needed to predict
    model = {}; check = dict(zip(labels, prob_class))
    for nam in labels:
        small = cleaned_data.loc[nam].reset_index(drop=True)
        model[nam] = {'prior': check[nam]}; model[nam]['point'] =  small.transpose()

    return model
```

```
def kdePredict(labels, cleaned_data, model):
    '''
        This function predicts classes for new items in a test dataset (re-use the training data as a test set)
    '''
    # Loop through the training data to collect the result after predicting
    prediction = []; label_name = labels.copy()

    for idx in range(cleaned_data.shape[0]):
        name = list(cleaned_data.index.tolist())[idx]
        score = []; k = 5
        for pose in model.keys():
            # Prior probability from each class
            probs = math.log2(model[pose]['prior'])
            # Likelihoods plus the prior from class using log
            for i in range(1, model[pose]['point'].shape[0]):
                each_diff = []; each_pdf = []

                # Get all the difference from test point to each point
                for x_test in model[pose]['point'].loc[i]:
                    x_test = float(x_test)
                    for x_train in model[pose]['point'].loc[i]:
                        x_train = float(x_train)
                        each_diff.append(x_test - x_train)

                # Get all the pdf from each difference
                for diff in each_diff:
                    each_pdf.append(norm.pdf(diff, 0, k))
                probs += math.log2(sum(each_pdf) / model[pose]['point'].shape[0])
            score.append(probs)
        # Collect index ,true name labels and the predicted name
        prediction.append([idx, name, label_name[np.argsort(score)[-2]]])

    return prediction
```

Question 3(2/2)

Results

Discussion

Gaussian naive bayes are based on the assumption that we assume models each as conforming to a Gaussian (normal) distribution. Besides, the attributes of a Gaussian classifier are numeric.

As same as Gaussian classifier, Kernel density estimation (KDE)'s attributes are numeric. The difference is that KDE can come from an arbitrary distribution, as it doesn't assume a general structure (such as gaussian normal distribution) of the whole input space, which indicates an optimal combination of attributes. And it learns probability from data. In addition, the KDE algorithm takes a parameter, bandwidth that affects how "smooth" the resulting curve is in a given set of data.

Furthermore, kernel density estimate (KDE) naive bayes classifier is a lazy learner, as we can use observations directly at prediction time. Naive Bayes with Gaussian is a good opposing example. In that case, the training data is not used anymore as the parameters of conditional normal distributions involved are fitted. Therefore, the prediction is computed solely, which depends on the fitted parameter values and functional form. Moreover, KDE is slow to compute probability at new points because it needs many parameters to represent probability density function.

Question 4