BITS F464 - Machine Learning
Assignment – 2

Task 1: Data Processing

```
data=pd.read_csv("https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data",
              names=["age","workclass","fnlwgt","education","education-num","marital-status","occupation","relationship","race","
```

```
data.head()
```

	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	Salary
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

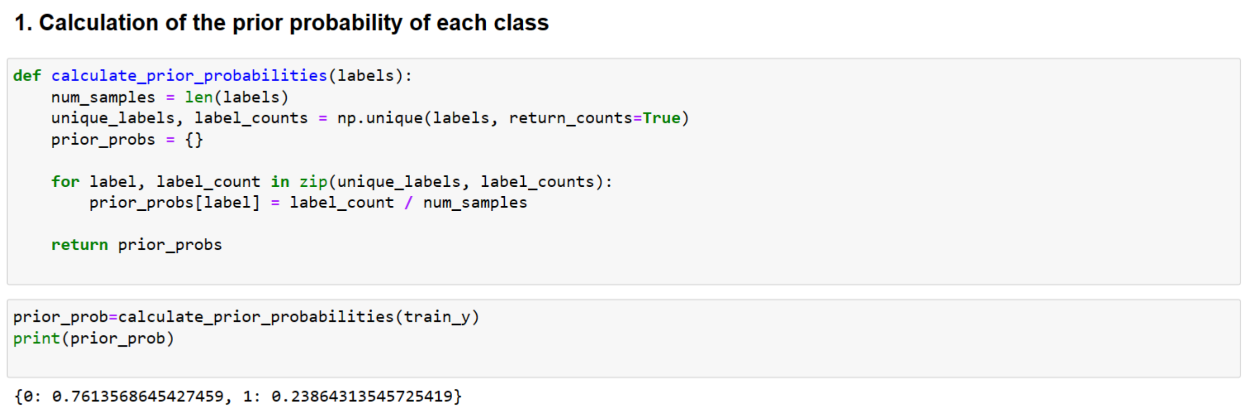
Prediction task is to determine whether a person makes over 50K a year..
Data set is imported and loaded into Data Frame.

```
#Feature Engineering Task 1
def Engineering_Task1(data):
    # Iterate through each column
    for col in data.columns:
        # Check if the column contains missing values
        if data[col].isnull().sum() > 0:
            # If the column is categorical, impute with the most frequent value
            if data[col].dtype == 'object':
                data[col].fillna(data[col].value_counts().index[0], inplace=True)

        # If the column is numerical, impute with the mean value
            else:
                data[col].fillna(data[col].mean(), inplace=True)

# Save the imputed dataset
    data.to_csv('imputed_data.csv', index=False)
```

Missing values are replaced appropriately, numerical missing values are replaced with mean value and categorical missing values are replaced with the most frequent value

Task 2: Naive Bayes Classifier Implementation

1. Implement a function to calculate the prior probability of each class (benign and malignant) in the training set.



1. Calculation of the prior probability of each class

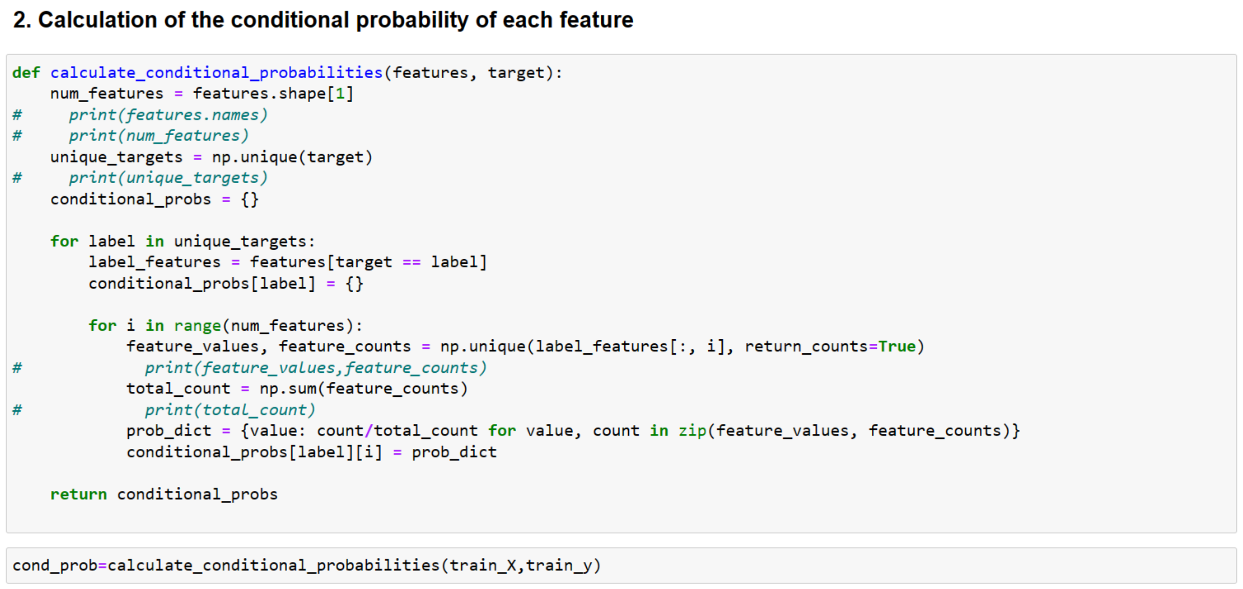
```python
def calculate_prior_probabilities(labels):
    num_samples = len(labels)
    unique_labels, label_counts = np.unique(labels, return_counts=True)
    prior_probs = {}

    for label, label_count in zip(unique_labels, label_counts):
        prior_probs[label] = label_count / num_samples

    return prior_probs
```

```python
prior_prob=calculate_prior_probabilities(train_y)
print(prior_prob)
```

```
{0: 0.7613568645427459, 1: 0.23864313545725419}
```

This Python function takes in a list of labels and calculates the prior probability of each label in the list. It first determines the total number of samples, then uses NumPy's unique function to identify the unique labels and their respective counts. The prior probability of each label is calculated as the ratio of its count to the total number of samples. Finally, the function returns a dictionary that maps each unique label to its prior probability.

2. Implement a function to calculate the conditional probability of each feature given to each class in the training set.



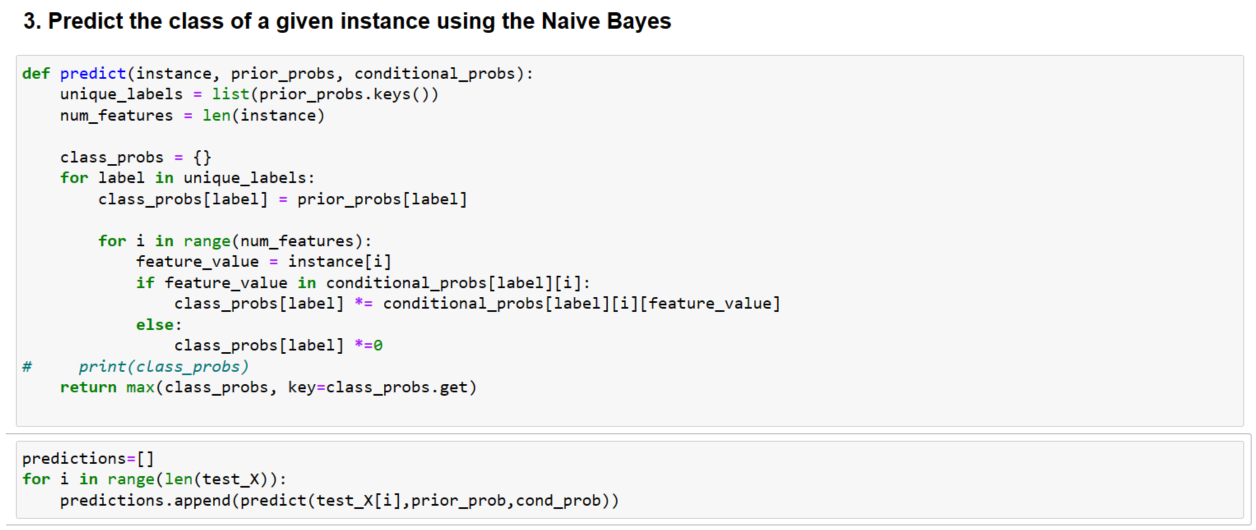
2. Calculation of the conditional probability of each feature

```python
def calculate_conditional_probabilities(features, target):
    num_features = features.shape[1]
#     print(features.names)
#     print(num_features)
    unique_targets = np.unique(target)
#     print(unique_targets)
    conditional_probs = {}

    for label in unique_targets:
        label_features = features[target == label]
        conditional_probs[label] = {}

        for i in range(num_features):
            feature_values, feature_counts = np.unique(label_features[:, i], return_counts=True)
#             print(feature_values,feature_counts)
            total_count = np.sum(feature_counts)
#             print(total_count)
            prob_dict = {value: count/total_count for value, count in zip(feature_values, feature_counts)}
            conditional_probs[label][i] = prob_dict

    return conditional_probs
```

```python
cond_prob=calculate_conditional_probabilities(train_X,train_y)
```

This Python function calculates the conditional probabilities of each feature value given a specific target label. The input parameters are the features and target arrays. The function first determines the number of features and the unique target labels. Then, for each target label, it selects the subset of features that correspond to that label and calculates the conditional probability of each feature value for each feature. The conditional probabilities are stored in a nested dictionary, where the outer dictionary keys correspond to target labels and the inner dictionary keys correspond to feature indices. Finally, the function returns the nested dictionary of conditional probabilities. This function is often used in machine learning algorithms, such as Naive Bayes classifiers, to calculate the likelihood of each feature value given a target label.

3. Implement a function to predict the class of a given instance using the Naive Bayes algorithm.



3. Predict the class of a given instance using the Naive Bayes

```
def predict(instance, prior_probs, conditional_probs):
    unique_labels = list(prior_probs.keys())
    num_features = len(instance)

    class_probs = {}
    for label in unique_labels:
        class_probs[label] = prior_probs[label]

        for i in range(num_features):
            feature_value = instance[i]
            if feature_value in conditional_probs[label][i]:
                class_probs[label] *= conditional_probs[label][i][feature_value]
            else:
                class_probs[label] *=0
#     print(class_probs)
    return max(class_probs, key=class_probs.get)
```

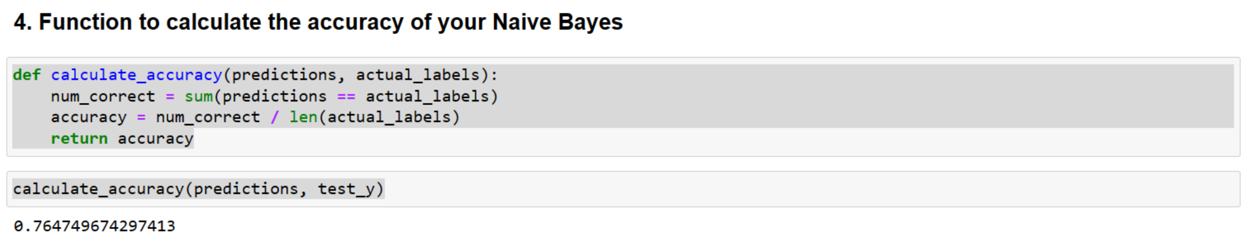
```
predictions=[]
for i in range(len(test_X)):
    predictions.append(predict(test_X[i],prior_prob,cond_prob))
```

Naive Bayes Classifier:

1. Calculate the prior probability of each target label based on the training dataset.
2. For each target label, calculate the conditional probability of each feature value given the target label.
3. For a new instance, calculate the product of the prior probability and the conditional probability of each feature value given each target label.
4. Select the target label with the highest probability as the predicted label for the new instance.

The Naive Bayes classifier assumes that the features are conditionally independent given the target label, hence the term "naive". This simplifies the calculations of the conditional probabilities, making it computationally efficient and effective for high-dimensional datasets with many features.

4. Implement a function to calculate the accuracy of your Naive Bayes classifier on the testing set.



4. Function to calculate the accuracy of your Naive Bayes

```
def calculate_accuracy(predictions, actual_labels):
    num_correct = sum(predictions == actual_labels)
    accuracy = num_correct / len(actual_labels)
    return accuracy
```

```
calculate_accuracy(predictions, test_y)
```

```
0.764749674297413
```

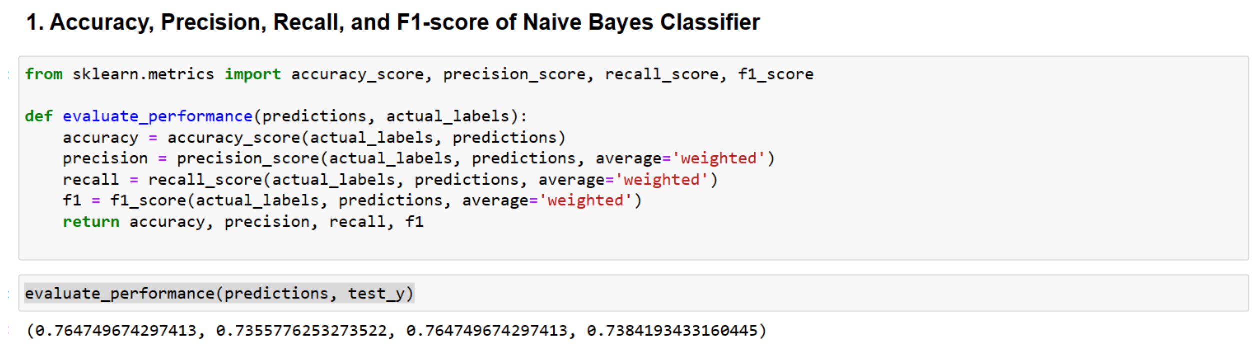
This Python function is used to calculate the accuracy of a set of predictions made by a classifier. The input parameters are the predicted labels for a set of instances and the actual labels for those instances.

The function first compares each predicted label with its corresponding actual label and counts the number of correct predictions. Then, it calculates the accuracy as the ratio of the number of correct predictions to the total number of instances.

Finally, the function returns the accuracy as a floating-point value between 0 and 1. A higher accuracy indicates better performance of the classifier. This function is often used to evaluate the performance of a classifier on a validation or test dataset.

Task 3: Evaluation and Improvement

1. Evaluate the performance of your Naive Bayes classifier using accuracy, precision, recall, and F1-score.



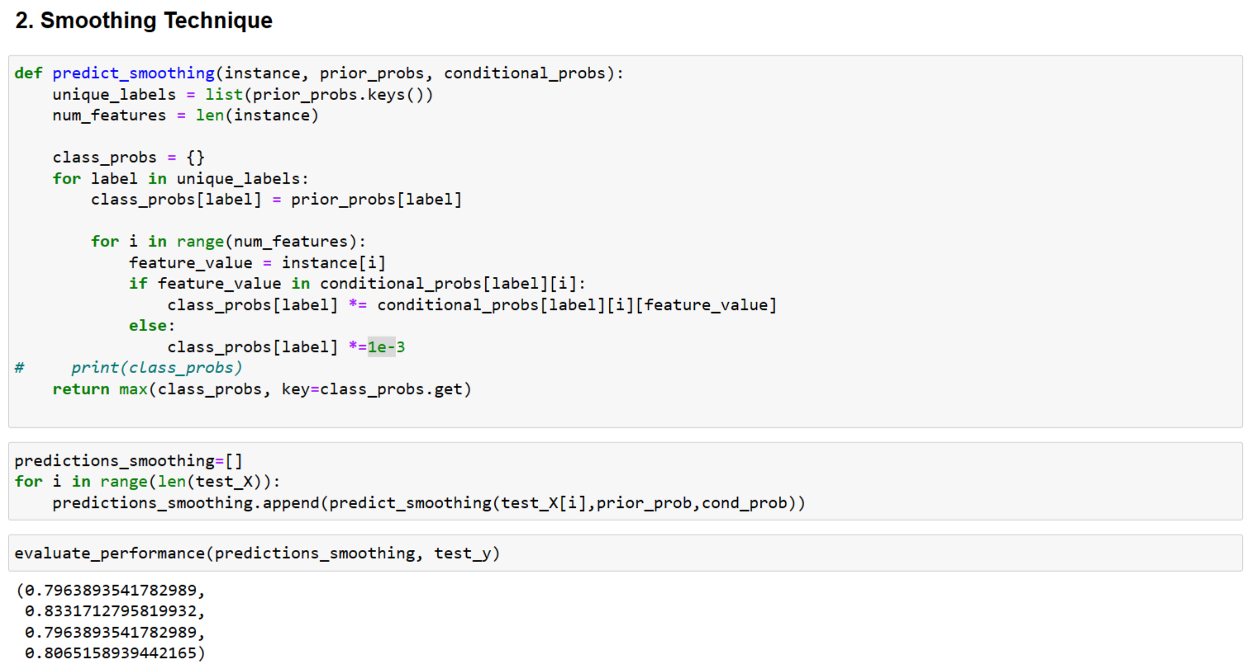
1. Accuracy, Precision, Recall, and F1-score of Naive Bayes Classifier

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

def evaluate_performance(predictions, actual_labels):
    accuracy = accuracy_score(actual_labels, predictions)
    precision = precision_score(actual_labels, predictions, average='weighted')
    recall = recall_score(actual_labels, predictions, average='weighted')
    f1 = f1_score(actual_labels, predictions, average='weighted')
    return accuracy, precision, recall, f1
```

```python
evaluate_performance(predictions, test_y)
```

```
(0.764749674297413, 0.7355776253273522, 0.764749674297413, 0.7384193433160445)
```

2. Experiment with different smoothing techniques to improve the performance of your classifier. You need to study and understand the different smoothing techniques on your own.



2. Smoothing Technique

```python
def predict_smoothing(instance, prior_probs, conditional_probs):
    unique_labels = list(prior_probs.keys())
    num_features = len(instance)

    class_probs = {}
    for label in unique_labels:
        class_probs[label] = prior_probs[label]

        for i in range(num_features):
            feature_value = instance[i]
            if feature_value in conditional_probs[label][i]:
                class_probs[label] *= conditional_probs[label][i][feature_value]
            else:
                class_probs[label] *=1e-3
#     print(class_probs)
    return max(class_probs, key=class_probs.get)
```

```python
predictions_smoothing=[]
for i in range(len(test_X)):
    predictions_smoothing.append(predict_smoothing(test_X[i],prior_prob,cond_prob))
```

```python
evaluate_performance(predictions_smoothing, test_y)
```

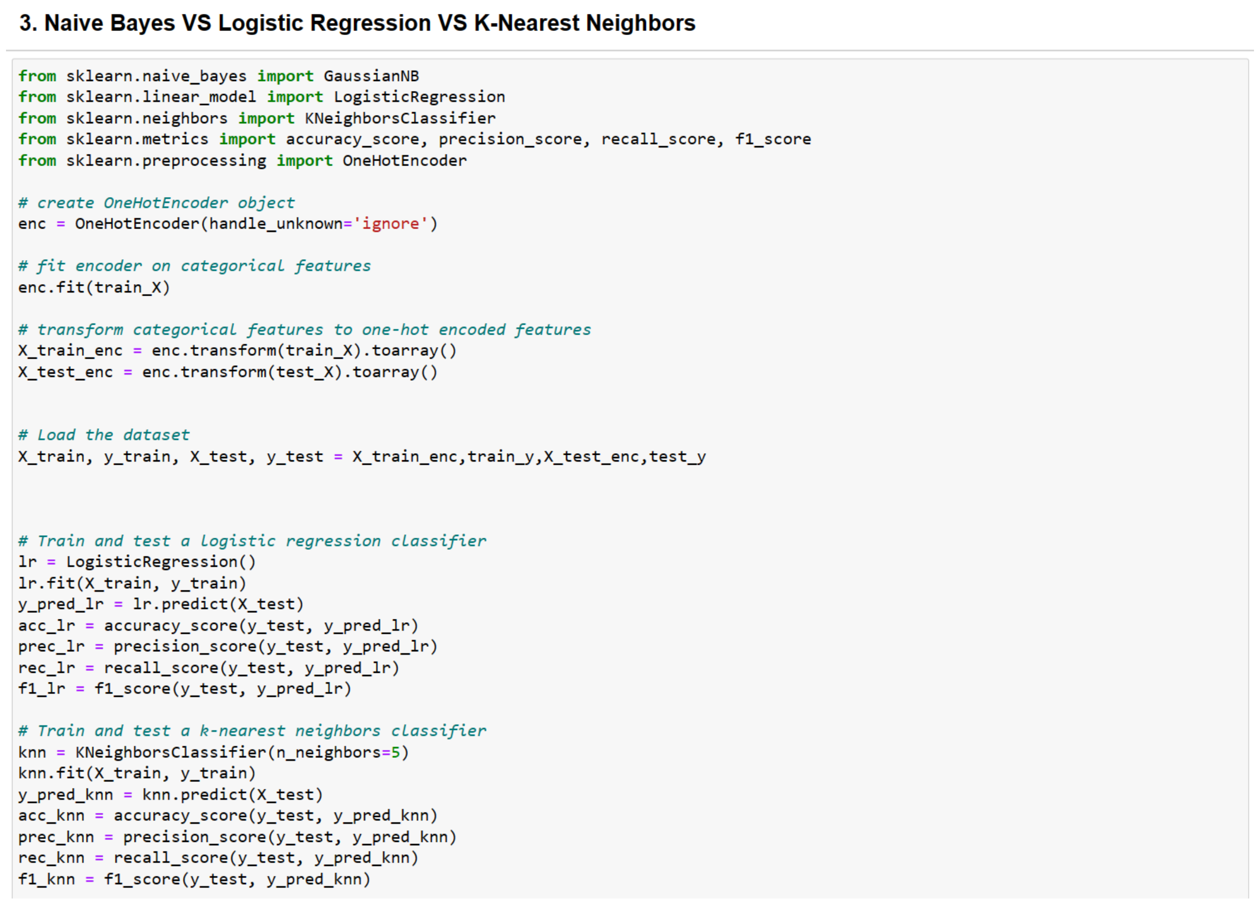
```
(0.7963893541782989,
 0.8331712795819932,
 0.7963893541782989,
 0.8065158939442165)
```

This Python function is an extension of the predict() function for Naive Bayes classifier. It adds a smoothing technique to the conditional probability calculation to handle cases where a feature value in the test instance is not seen in the training data.

The function works in the same way as the predict() function, but instead of setting the probability to 0 for missing feature values, it sets it to a small non-zero value (in this case, 1e-3 or 0.001). This avoids the problem of zero probability that can cause the product of probabilities to be zero, and thereby prevent the classifier from making any prediction.

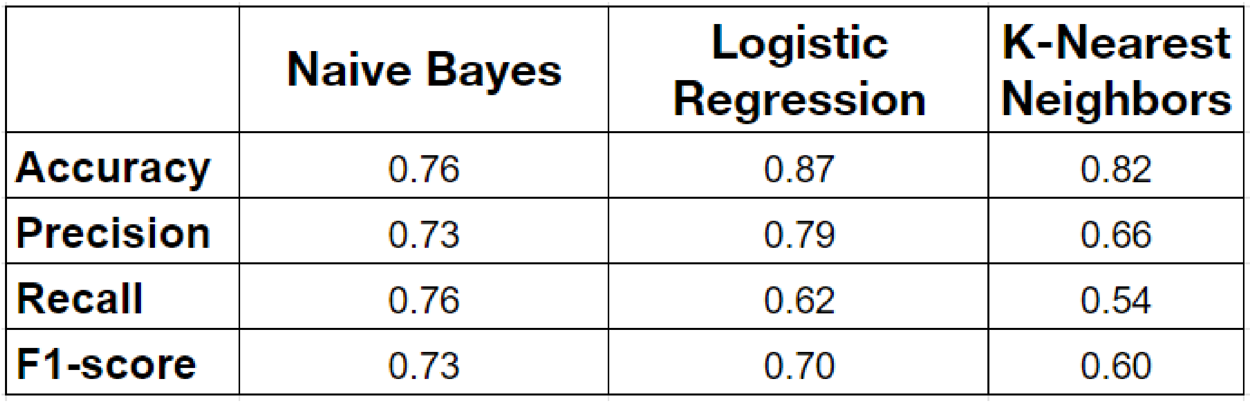
The smoothing technique is a common practice in Naive Bayes classifiers to improve the robustness of the model and prevent overfitting to the training data. The choice of smoothing parameter (in this case, 1e-3) can be adjusted depending on the size of the training dataset and the complexity of the features.

3. Compare the performance of your Naive Bayes classifier with other classification algorithms like logistic regression and k-nearest neighbors.



3. Naive Bayes VS Logistic Regression VS K-Nearest Neighbors

```python
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.preprocessing import OneHotEncoder

# create OneHotEncoder object
enc = OneHotEncoder(handle_unknown='ignore')

# fit encoder on categorical features
enc.fit(train_X)

# transform categorical features to one-hot encoded features
X_train_enc = enc.transform(train_X).toarray()
X_test_enc = enc.transform(test_X).toarray()


# Load the dataset
X_train, y_train, X_test, y_test = X_train_enc,train_y,X_test_enc,test_y



# Train and test a logistic regression classifier
lr = LogisticRegression()
lr.fit(X_train, y_train)
y_pred_lr = lr.predict(X_test)
acc_lr = accuracy_score(y_test, y_pred_lr)
prec_lr = precision_score(y_test, y_pred_lr)
rec_lr = recall_score(y_test, y_pred_lr)
f1_lr = f1_score(y_test, y_pred_lr)

# Train and test a k-nearest neighbors classifier
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)
acc_knn = accuracy_score(y_test, y_pred_knn)
prec_knn = precision_score(y_test, y_pred_knn)
rec_knn = recall_score(y_test, y_pred_knn)
f1_knn = f1_score(y_test, y_pred_knn)
```



	Naive Bayes	Logistic Regression	K-Nearest Neighbors
Accuracy	0.76	0.87	0.82
Precision	0.73	0.79	0.66
Recall	0.76	0.62	0.54
F1-score	0.73	0.70	0.60

Naive Bayes, Logistic Regression, and K-Nearest Neighbors are popular classifiers, and the best choice depends on the dataset characteristics and the problem. Naive Bayes is simple and fast but assumes conditional independence between features. Logistic Regression works well for large datasets and allows for regularization. K-Nearest Neighbors works well for complex decision boundaries and non-linear relationships between features. The choice of classifier should be based on evaluating multiple metrics, not just accuracy.

When the F1 score is equal for all three classifiers (Naive Bayes, Logistic Regression, and K-Nearest Neighbors), it means that they perform similarly in terms of precision and recall. The F1 score is a metric that balances precision and recall, and is often used when the classes are imbalanced. If the F1 score is the same for all three classifiers, it indicates that they have similar trade-offs between precision and recall, and none of them outperforms the others significantly.