Introduction

DSCI 552 - Machine Learning for Data Science

Homework 3

Matheus Schmitz

USC ID: 5039286453

Imports

In [1]:
```
# tqdm creates a loading bar for loops
# Extremely useful when your loop takes long
!pip install tqdm
```

```
Requirement already satisfied: tqdm in c:\users\matheus\anaconda3\lib\site-pack
ages (4.47.0)
```

In [2]:
```
# Data Manipulation
import numpy as np
import pandas as pd

# Visualization
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Progress bar
from tqdm.notebook import tqdm

# Metrics
from sklearn.metrics import roc_auc_score

# OS
import os
import warnings
warnings.filterwarnings('ignore')
```

In [3]:

```
# Auxilary functions to facilitate plotting classification results
# Based on scikit-learn documentation: https://scikit-learn.org/stable/auto_examp
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pylab
from sklearn.metrics import roc_curve, auc, confusion_matrix, precision_recall_cu

def plot_cm(true_onehot, pred_probs, classes):
    # Dictionaries to store data
    fpr = dict()
    tpr = dict()
    thresholds = dict()
    roc_auc = dict()
    # Coerce inputs to np.array
    true_onehot = np.asarray(true_onehot)
    pred_probs = np.asarray(pred_probs)
    classes = np.asarray(classes)
    N_CLASSES = len(classes)
    # Extract true and predicted labels
    pred_labels = classes[np.argmax(pred_probs, axis=1)]
    true_labels = classes[np.argmax(true_onehot, axis=1)]
    # Get Confusion Matrix and plot
    conf_mat = confusion_matrix(true_labels, pred_labels, labels=classes)
    #plt.axis('equal')
    sns.heatmap(conf_mat, annot=True, cmap='Blues', xticklabels=classes, yticklab
    plt.title('Confusion Matrix', pad = 20, fontweight='bold')
    plt.ylabel('True Emotion', fontsize = 12, labelpad = 10)
    plt.xlabel('Predicted Emotion', fontsize = 12, labelpad = 10)

def plot_roc_multiclass(true_onehot, pred_probs, classes):
    # Dictionaries to store data
    fpr = dict()
    tpr = dict()
    thresholds = dict()
    roc_auc = dict()
    # Coerce inputs to np.array
    true_onehot = np.asarray(true_onehot)
    pred_probs = np.asarray(pred_probs)
    classes = np.asarray(classes)
    N_CLASSES = len(classes)
    # For each class, get the fpr, tpr, thresholds and auc
    for i in range(N_CLASSES):
        fpr[i], tpr[i], thresholds[i] = roc_curve(true_onehot[:, i], pred_probs[:
        roc_auc[i] = auc(fpr[i], tpr[i])
    # Generate multilabel colors with pylab
    class_colors = []
    cm = pylab.get_cmap('nipy_spectral')
    for i in range(len(classes)):
        color = cm(1.*i/N_CLASSES)
        class_colors.append(color)
    # Plot the class-stratified ROCs
    plt.axis('square')
    for i, color in zip(range(N_CLASSES), class_colors):
        plt.plot(fpr[i], tpr[i], color=color, lw=3, label=f'{classes[i]} (area =
    plt.plot([0, 1], [0, 1], 'k--', lw=2, alpha=0.3)
```

```
    plt.xlim([-0.01, 1.01])
    plt.ylim([-0.01, 1.01])
    plt.xlabel('False Positive Rate', fontsize = 12, labelpad = 10)
    plt.ylabel('True Positive Rate', fontsize = 12, labelpad = 10)
    plt.title('Multi-Class ROC Curve', pad = 20, fontweight='bold')
    legend = plt.legend()
    legend._legend_box.align = "right"
    plt.legend(loc="lower right")

def plot_roc_overall(true_onehot, pred_probs, classes):
    # Coerce inputs to np.array
    true_onehot = np.asarray(true_onehot)
    pred_probs = np.asarray(pred_probs)
    classes = np.asarray(classes)
    N_CLASSES = len(classes)
    # Compute global (micro-average) ROC curve and ROC area
    fpr, tpr, thresholds = roc_curve(true_onehot.ravel(), pred_probs.ravel())
    roc_auc = auc(fpr, tpr)
    # Plot the model overall ROC
    plt.axis('square')
    plt.plot(fpr, tpr, label=f'MODEL OVERALL (area = {roc_auc:.2f})', color='deep
    plt.plot([0, 1], [0, 1], 'k--', lw=2, alpha=0.3)
    plt.xlim([-0.01, 1.01])
    plt.ylim([-0.01, 1.01])
    plt.xlabel('False Positive Rate', fontsize = 12, labelpad = 10)
    plt.ylabel('True Positive Rate', fontsize = 12, labelpad = 10)
    plt.title('Overall ROC Curve', pad = 20, fontweight='bold')
    legend = plt.legend()
    legend._legend_box.align = "right"
    plt.legend(loc="lower right")

def plot_prec_recall_curve(true_onehot, pred_probs, classes):
    # Dictionaries to store data
    precision = dict()
    recall = dict()
    thresholds = dict()
    auprc = dict()
    # Coerce inputs to np.array
    true_onehot = np.asarray(true_onehot)
    pred_probs = np.asarray(pred_probs)
    classes = np.asarray(classes)
    N_CLASSES = len(classes)
    # For each class, get the fpr, tpr, thresholds and auc
    for i in range(N_CLASSES):
        precision[i], recall[i], thresholds[i] = precision_recall_curve(true_oneh
        auprc[i] = auc(recall[i], precision[i])
    # Generate multilabel colors with pylab
    class_colors = []
    cm = pylab.get_cmap('nipy_spectral')
    for i in range(len(classes)):
        color = cm(1.*i/N_CLASSES)
        class_colors.append(color)
    # Plot the class-stratified ROCs
    plt.axis('square')
    for i, color in zip(range(N_CLASSES), class_colors):
        plt.plot(recall[i], precision[i], color=color, lw=3, label=f'{classes[i]}
    plt.plot([1, 0], [0, 1], 'k--', lw=2, alpha=0.3)
```

```
    plt.xlim([-0.01, 1.01])
    plt.ylim([-0.01, 1.01])
    plt.xlabel('Recall', fontsize = 12, labelpad = 10)
    plt.ylabel('Precision', fontsize = 12, labelpad = 10)
    plt.title('Precision-Recall Curve', pad = 20, fontweight='bold')
    legend = plt.legend()
    legend._legend_box.align = "left"
    plt.legend(loc="lower left")

def plot_classification_results(true_onehot, pred_probs, classes):
    fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(16,10))
    fig.sca(axs[0][0])
    plot_cm(true_onehot, pred_probs, classes)
    fig.sca(axs[0][1])
    plot_prec_recall_curve(true_onehot, pred_probs, classes)
    fig.sca(axs[1][0])
    plot_roc_overall(true_onehot, pred_probs, classes)
    fig.sca(axs[1][1])
    plot_roc_multiclass(true_onehot, pred_probs, classes)
    fig.tight_layout(h_pad=3, w_pad=-30)
    fig.show()
```

(1) Time Series Classification

1. **Time Series Classification**

   An interesting task in machine learning is classification of time series. In this problem, we will classify the activities of humans based on time series obtained by a Wireless Sensor Network.

(a) Download the dataset

(a) Download the AReM data from: `https://archive.ics.uci.edu/ml/datasets/Activity+Recognition+system+based+on+Multisensor+data+fusion+\%28AReM\%29` . The dataset contains 7 folders that represent seven types of activities. In each folder, there are multiple files each of which represents an instant of a human performing an activity.[1] Each file containis 6 time series collected from activities of the same person, which are called avg_rss12, var_rss12, avg_rss13, var_rss13, vg_rss23, and ar_rss23. There are 88 instances in the dataset, each of which contains 6 time series and each time series has 480 consecutive values.

In [4]:
```
col_names = ['time', 'avg_rss12', 'var_rss12', 'avg_rss13', 'var_rss13', 'avg_rss
```

In [5]:
```
# Get all csv files

all_files = []

for rootname, dirnames, filenames in os.walk('../data'):
    for filename in filenames:
        filepath = os.path.join(rootname, filename)
        filepath = filepath.replace('\\', '/')
        all_files.append(filepath)

len(all_files)
```

Out[5]:
```
88
```

(b) Train-Test Split

(b) Keep datasets 1 and 2 in folders bending1 and bending 2, as well as datasets 1, 2, and 3 in other folders as test data and other datasets as train data.

In [6]:
```
# List train and test datasets
test_datasets = ['../data/bending1/dataset1.csv',
                 '../data/bending1/dataset2.csv',
                 '../data/bending2/dataset1.csv',
                 '../data/bending2/dataset2.csv',
                 '../data/cycling/dataset1.csv',
                 '../data/cycling/dataset2.csv',
                 '../data/cycling/dataset3.csv',
                 '../data/lying/dataset1.csv',
                 '../data/lying/dataset2.csv',
                 '../data/lying/dataset3.csv',
                 '../data/sitting/dataset1.csv',
                 '../data/sitting/dataset2.csv',
                 '../data/sitting/dataset3.csv',
                 '../data/standing/dataset1.csv',
                 '../data/standing/dataset2.csv',
                 '../data/standing/dataset3.csv',
                 '../data/walking/dataset1.csv',
                 '../data/walking/dataset2.csv',
                 '../data/walking/dataset3.csv',]

train_datasets = [fpath for fpath in all_files if fpath not in test_datasets]

print(f'Tally of test datasets: {len(test_datasets)}')
print(f'Tally of train datasets: {len(train_datasets)}')
```

```
Tally of test datasets: 19
Tally of train datasets: 69
```

Some CSVs are separated by commas "," while others are separated by spaces " ", so pandas has to be told to consider two separation criteria. This can be done with `sep=r'\s+|,'` , where `\s+` means to use one or more spaces, and `,` means to use a comma as separator.

In [7]:
```
df_test = pd.DataFrame()

for file in test_datasets:
    tmp = pd.read_csv(file, skiprows=5, header=None, sep=r'\s+|,')
    df_test = df_test.append(tmp, ignore_index=True)

df_test.columns = col_names
df_test.shape
```

Out[7]:
```
(9120, 7)
```

In [8]:
```
df_test.tail(3)
```

Out[8]:

	time	avg_rss12	var_rss12	avg_rss13	var_rss13	avg_rss23	var_rss23
9117	119250	33.00	7.35	14.60	3.14	13.00	5.70
9118	119500	31.67	1.25	11.00	6.16	19.25	2.17
9119	119750	30.75	10.21	11.75	1.09	18.50	3.20

In [9]:
```
df_train = pd.DataFrame()

for file in train_datasets:
    tmp = pd.read_csv(file, skiprows=5, header=None, sep=r'\s+|,')
    df_train = df_train.append(tmp, ignore_index=True)

df_train.columns = col_names
df_train.shape
```

Out[9]:
```
(33119, 7)
```

One of the datasets has one fewer row.

"../data/sitting/dataset8.csv" shape is (479, 7).

The missing row is between lines 59 and 60. The time skips form 13250ms to 13750ms, skipping the expected 13500 measurement.

In [10]:
```
df_train.tail(3)
```

Out[10]:

	time	avg_rss12	var_rss12	avg_rss13	var_rss13	avg_rss23	var_rss23
33116	119250	37.80	7.68	14.20	2.48	17.25	0.83
33117	119500	33.75	1.30	15.75	5.21	16.50	2.69
33118	119750	32.67	3.09	18.67	0.47	14.00	3.16

(c) Feature Extraction

(c) Feature Extraction

Classification of time series usually needs extracting features from them. In this problem, we focus on time-domain features.

(i) Research Time-Domain Features

i. Research what types of time-domain features are usually used in time series classification and list them (examples are minimum, maximum, mean, etc).

The most simple features which can be extracted from the data are those regarding it's distribution, known as descriptive statistics. Examples are: **mean, median, standard deviation, minimum and maximum, quartiles or percentiles, as well as skewness and kurtosis.**

Some slightly more advances features are those employed by ARIMA models and their evolutions, such as SARIMAX, which employes: Seasonality (S); Autoregression (AR); Integration (I); Moving Average (MA); Exogenous Variables (X).

There are three main classes of features for time series analysis:

- **Date Time Features:** Refers to features the the individual observations;
- **Lag Features:** Refers to features comparing changes from an observation regarding previous ones;
- **Window Features:** Refers to features obtained from considering multiple observations at one, using a sliding time window.

Wikipedia has a comprehensive listing with further references:

https://en.wikipedia.org/wiki/Time_series#Measures (https://en.wikipedia.org/wiki/Time_series#Measures)

Another source which balances comprehensiveness and accessibility is the paper which Facebook published together with it's Prophet algorithm for time series forecasting:

https://peerj.com/preprints/3190.pdf (https://peerj.com/preprints/3190.pdf)

(ii) Extract the Time-Domain Features

ii. Extract the time-domain features minimum, maximum, mean, median, standard deviation, first quartile, and third quartile for all of the 6 time series in each instance. You are free to normalize/standardize features or use them directly.[2]

Your new dataset will look like this:

Instance	$\min_1$	$\max_1$	mean_1	median_1 ⋯	1st quart_6	3rd quart_6
1						
2						
3						
⋮	⋮	⋮	⋮	⋮ ⋯	⋮	⋮
88						

where, for example, 1st quart_6, means the first quartile of the sixth time series in each of the 88 instances.

In [11]:
```
table_cols = ['min1', 'max1', 'mean1', 'median1', 'std1', 'firstq1', 'thirdq1',
              'min2', 'max2', 'mean2', 'median2', 'std2', 'firstq2', 'thirdq2',
              'min3', 'max3', 'mean3', 'median3', 'std3', 'firstq3', 'thirdq3',
              'min4', 'max4', 'mean4', 'median4', 'std4', 'firstq4', 'thirdq4',
              'min5', 'max5', 'mean5', 'median5', 'std5', 'firstq5', 'thirdq5',
              'min6', 'max6', 'mean6', 'median6', 'std6', 'firstq6', 'thirdq6']
len(table_cols)
```

Out[11]:
```
42
```

In [12]:
```
# Create a master list containing all features for all csv's, then convert it to
# The list will follow the order of 'table_cols', listing all attributes of the f

master_list = []

# Loop through all CSVs
for file in all_files:

    # Read the CSV and drop the 'time' column
    tmp = pd.read_csv(file, skiprows=5, header=None, sep=r'\s+|,')
    tmp.columns = col_names
    tmp.drop('time', inplace=True, axis=1)

    # Extract the features from each variable in the CSV
    ft_min = pd.Series(tmp.min().values)
    ft_max = pd.Series(tmp.max().values)
    ft_mean = pd.Series(tmp.mean().values)
    ft_median = pd.Series(tmp.median().values)
    ft_std = pd.Series(tmp.std().values)
    ft_firstq = pd.Series(tmp.quantile(0.25).values)
    ft_thirdq = pd.Series(tmp.quantile(0.75).values)

    # Put all the features in a dataframe
    # Each of the features (the 'ft_' pd.Series from above) will be a row, the co
    df_tmp = pd.DataFrame(data=[ft_min, ft_max, ft_mean, ft_median, ft_std, ft_fi

    # The reason for converting to a dataframe of this shape is so that the .melt
    # this allows the sequence to be converted...
    # from: the .min() of all variables, followed by the .max() of all varibles,
    # to: all features from variable 1, then all features from variable 2, then f
    # The resulting pd.Series is now a 42-item long list ordered by CSV variable,
    df_tmp = df_tmp.melt()

    # Melt creates two columns, one named 'varible' which has in each row the nam
    # The other column is 'value' which has the value associated with the previou
    # Keeping only the series of values, ordered by CSV variable
    df_tmp = pd.Series(df_tmp['value'])

    # Then append the data already in the correct order to the master_list which
    master_list.append(df_tmp)

# Now that the loop is over, create the final dataframe from the master_list
df_features = pd.DataFrame(data=master_list)
df_features.reset_index(inplace=True, drop=True)
df_features.columns = table_cols
df_features.shape
```

Out[12]: (88, 42)

In [13]:
```
df_features
```

Out[13]:

	min1	max1	mean1	median1	std1	firstq1	thirdq1	min2	max2	mean2	...	s
0	37.25	45.00	40.624792	40.500	1.476967	39.25	42.0000	0.0	1.30	0.358604	...	2.188
1	38.00	45.67	42.812812	42.500	1.435550	42.00	43.6700	0.0	1.22	0.372438	...	1.995
2	35.00	47.40	43.954500	44.330	1.558835	43.00	45.0000	0.0	1.70	0.426250	...	1.999
3	33.00	47.75	42.179813	43.500	3.670666	39.15	45.0000	0.0	3.00	0.696042	...	3.849
4	33.00	45.75	41.678063	41.750	2.243490	41.33	42.7500	0.0	2.83	0.535979	...	2.411
...	...	...	...	...	...	...	...	...	...	...	...	
83	20.75	46.25	34.763333	35.290	4.742208	31.67	38.2500	0.0	12.68	4.223792	...	3.174
84	21.50	51.00	34.935813	35.500	4.645944	32.00	38.0625	0.0	12.21	4.115750	...	3.192
85	18.33	47.67	34.333042	34.750	4.948770	31.25	38.0000	0.0	12.48	4.396958	...	3.000
86	18.33	45.75	34.599875	35.125	4.731790	31.50	38.0000	0.0	15.37	4.398833	...	2.905
87	15.50	43.67	34.225875	34.750	4.441798	31.25	37.2500	0.0	17.24	4.354500	...	2.992

88 rows × 42 columns

(iii) Build a 90% Bootsrap Confidence Interval for the Standard Deviation of each Feature

iii. Estimate the standard deviation of each of the time-domain features you extracted from the data. Then, use Python's bootstrapped or any other method to build a 90% bootsrap confidence interval for the standard deviation of each feature.

In [14]:

```
%%time
# Calculate the standard deviation for each feature from the dataset
df_column_stds = pd.DataFrame(df_features.std(), columns=['Standard Deviation'])

# Create a dataframe to store all bootstrapped std samples
bootstrap_df = pd.DataFrame()

# Run 1000 loops bootstrapping data
for i in range(0, 1000):

    # Generate a bootstrapped dataset and get it's standard deviation
    bs_std = df_features.sample(frac=1, replace=True).std()

    # Then append it to bootstrap_df as a column
    bootstrap_df = pd.concat([bootstrap_df, bs_std], axis=1)

# After finishing the boostraping loop, get the the 0.05 and 0.95 quantiles (0.95
lower_bound = bootstrap_df.quantile(q=0.05, axis=1)
upper_bound = bootstrap_df.quantile(q=0.95, axis=1)

# Then add the upper and lower bounds to the dataframe with the standard deviatio
df_std_ci = pd.concat([df_column_stds, lower_bound, upper_bound], axis=1)
```

```
Wall time: 1.52 s
```

In [15]:
```
df_std_ci
```

Out[15]:

	Standard Deviation	0.05	0.95
min1	9.569975	8.251400	10.787523
max1	4.394362	3.342743	5.303965
mean1	5.335718	4.700680	5.932338
median1	5.440054	4.770535	6.005406
std1	1.772153	1.565155	1.936148
firstq1	6.153590	5.576668	6.644539
thirdq1	5.138925	4.340240	5.889960
min2	0.000000	0.000000	0.000000
max2	5.062729	4.611788	5.399607
mean2	1.574164	1.394433	1.702126
median2	1.412244	1.238076	1.538067
std2	0.884105	0.802211	0.941845
firstq2	0.946386	0.827918	1.030050
thirdq2	2.125266	1.895560	2.286159
min3	2.956462	2.765112	3.099546
max3	4.875137	4.171398	5.467024
mean3	4.008380	3.405868	4.459528
median3	4.036396	3.418142	4.504938
std3	0.946710	0.759068	1.113457
firstq3	4.220658	3.597023	4.666014
thirdq3	4.171628	3.525404	4.664036
min4	0.000000	0.000000	0.000000
max4	2.183625	1.974012	2.353316
mean4	1.166114	1.074192	1.223736
median4	1.145586	1.053069	1.202796
std4	0.458242	0.420782	0.485593
firstq4	0.843620	0.771277	0.893230
thirdq4	1.552504	1.430158	1.629951
min5	6.124001	4.465545	7.445365
max5	5.741238	4.737664	6.511223
mean5	5.675593	4.416105	6.648944
median5	5.813782	4.502835	6.818501
std5	1.024898	0.817497	1.228245
firstq5	6.096465	4.803599	7.141807

	Standard Deviation	0.05	0.95
thirdq5	5.531720	4.337268	6.408084
min6	0.045838	0.000000	0.078476
max6	2.518921	2.244562	2.762226
mean6	1.154812	1.060639	1.217206
median6	1.086474	0.995200	1.149008
std6	0.517617	0.478323	0.545247
firstq6	0.758584	0.689229	0.806400
thirdq6	1.523599	1.401017	1.600415

(iv) Select the Three Most Important Features

> iv. Use your judgement to select the three most important time-domain features (one option may be min, mean, and max).

min, **max** and **mean** seem like a good choice for representing the data, so I'll take the advice and go with those, as they do seem to together portray a good image of the data distribution.

In [16]:
```
top_features = ['min1', 'max1', 'mean1',
                'min2', 'max2', 'mean2',
                'min6', 'max6', 'mean6']
```

(d) Binary Classification Using Logistic Regression

(i) Scatterplot of Bending x Non-Bending Activities

> i. Assume that you want to use the training set to classify bending from other activities, i.e. you have a binary classification problem. Depict scatter plots of the features you specified in 1(c)iv extracted from time series 1, 2, and 6 of each instance, and use color to distinguish bending vs. other activities. (See p. 129 of the textbook).[4]

In [17]:
```
# This will require extracting the labels from each class
# Use list comprehension to go through all_files (a list with file paths)
# And use the folder in the file path to set the label
labels = [all_files[i].split('/')[2] for i in range(len(all_files))]

# Then create a new column on the dataframe containing the labels
df_features['label'] = labels
```

In [18]:
```
# Create a binary indicator to define if the label is for bending (1) or for some
labels_bending = [1 if 'bending' in label else 0 for label in labels]
df_features['is_bending'] = labels_bending
labels_binary_unique = ['non-bending', 'bending']
```

In [19]:
```
# Since we are at it, also attach the filepath to each sample, just in case it be
# Attach the filepatchs to each row
df_features['fpath'] = all_files
df_features.head(3)
```

Out[19]:

	min1	max1	mean1	median1	std1	firstq1	thirdq1	min2	max2	mean2	...	min6	
0	37.25	45.00	40.624792	40.50	1.476967	39.25	42.00	0.0	1.30	0.358604	...	0.0	
1	38.00	45.67	42.812812	42.50	1.435550	42.00	43.67	0.0	1.22	0.372438	...	0.0	
2	35.00	47.40	43.954500	44.33	1.558835	43.00	45.00	0.0	1.70	0.426250	...	0.0	

3 rows × 45 columns

In [20]:
```
# Train-test split

# Create a new df_test, overwriting the old one from (b)
df_test = df_features[df_features['fpath'].isin(test_datasets)]
print(f'Test dataset shape: {df_test.shape}')

# Create a new df_train, overwriting the old one from (b)
df_train = df_features[~df_features['fpath'].isin(test_datasets)]
print(f'Train dataset shape: {df_train.shape}')
```

```
Test dataset shape: (19, 45)
Train dataset shape: (69, 45)
```

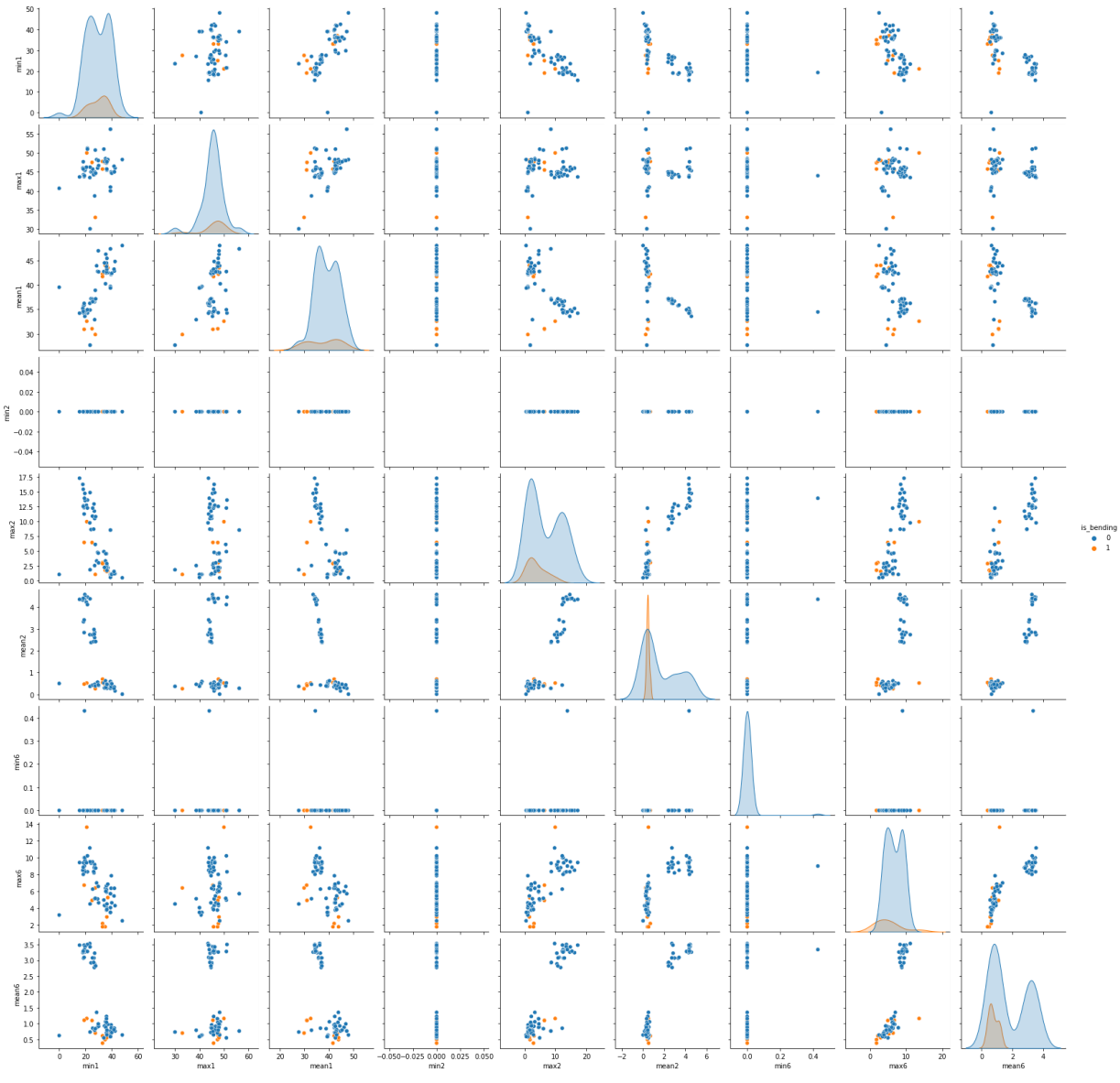
In [21]:
```
# Create a training dataset with only the top features and the is_bending label
df_train_top_f = df_train[top_features]
df_train_top_f['is_bending'] = df_train['is_bending']
df_train_top_f.head(3)
```

Out[21]:

	min1	max1	mean1	min2	max2	mean2	min6	max6	mean6	is_bending
2	35.0	47.40	43.954500	0.0	1.70	0.426250	0.0	1.79	0.493292	1
3	33.0	47.75	42.179813	0.0	3.00	0.696042	0.0	2.18	0.613521	1
4	33.0	45.75	41.678063	0.0	2.83	0.535979	0.0	1.79	0.383292	1

In [22]:
```
sns.pairplot(data=df_train_top_f, hue='is_bending')
```

Out[22]:
```
<seaborn.axisgrid.PairGrid at 0x25b37df9f48>
```



(ii) Break each time series in your training set into two

ii. Break each time series in your training set into two (approximately) equal length time series. Now instead of 6 time series for each of the training instances, you have 12 time series for each training instance. Repeat the experiment in 1(d)i, i.e depict scatter plots of the features extracted from both parts of the time series 1,2, and 12. Do you see any considerable difference in the results with those of 1(d)i?

In [23]:
```
# We'll have 12 columns, so need to create appropriate headings
table_cols2 = ['min01', 'max01', 'mean01', 'median01', 'std01', 'firstq01', 'thir
               'min02', 'max02', 'mean02', 'median02', 'std02', 'firstq02', 'thir
               'min03', 'max03', 'mean03', 'median03', 'std03', 'firstq03', 'thir
               'min04', 'max04', 'mean04', 'median04', 'std04', 'firstq04', 'thir
               'min05', 'max05', 'mean05', 'median05', 'std05', 'firstq05', 'thir
               'min06', 'max06', 'mean06', 'median06', 'std06', 'firstq06', 'thir
               'min07', 'max07', 'mean07', 'median07', 'std07', 'firstq07', 'thir
               'min08', 'max08', 'mean08', 'median08', 'std08', 'firstq08', 'thir
               'min09', 'max09', 'mean09', 'median09', 'std09', 'firstq09', 'thir
               'min10', 'max10', 'mean10', 'median10', 'std10', 'firstq10', 'thir
               'min11', 'max11', 'mean11', 'median11', 'std11', 'firstq11', 'thir
               'min12', 'max12', 'mean12', 'median12', 'std12', 'firstq12', 'thir
len(table_cols2)
```

Out[23]: 84

In [24]:

```
# Redoing all the previous procedure, now with changes to take the inner (tmp) da

# Create a master list containing all features for all csv's, then convert it to
# The list already follows the order of 'table_cols', listing all attributes of t

master_list2 = []

# Loop through all CSVs
for file in all_files:

    # Read the CSV
    tmp = pd.read_csv(file, skiprows=5, header=None, sep=r'\s+|,')

    # Drop the first column with time data
    tmp.drop(0, axis=1, inplace=True)

    # Take the halfway point to split the dataframe
    cutpoint = len(tmp)//2

    # Make one dataframe from the first half, and one from the second half
    pt1 = tmp[:cutpoint].reset_index(drop=True)
    pt2 = tmp[cutpoint:].reset_index(drop=True)

    # Put them together as a dataframe with twice as many columns and half as mar
    tmp = pd.concat(objs=[pt1, pt2], axis=1, ignore_index=True)

    # Extract the features from each variable in the dataframe
    ft_min = pd.Series(tmp.min().values)
    ft_max = pd.Series(tmp.max().values)
    ft_mean = pd.Series(tmp.mean().values)
    ft_median = pd.Series(tmp.median().values)
    ft_std = pd.Series(tmp.std().values)
    ft_firstq = pd.Series(tmp.quantile(0.25).values)
    ft_thirdq = pd.Series(tmp.quantile(0.75).values)

    # Put all the features in a dataframe
    # Each of the features (the 'ft_' pd.Series from above) will be a row, the co
    df_tmp = pd.DataFrame(data=[ft_min, ft_max, ft_mean, ft_median, ft_std, ft_fi

    # The reason for converting to a dataframe of this hape is so that the .melt(
    # this allows the sequence to be converted...
    # from: the .min() of all variables, followed by the .max() of all varibles,
    # to: all features from variable 1, then all features from variable 2, then f
    # The resulting pd.Series is now a 42-item long list ordered by CSV variable,
    df_tmp = df_tmp.melt()
    df_tmp = pd.Series(df_tmp['value'])

    # Then append the data already in the correct order to the master_list which
    master_list2.append(df_tmp)

# Now that the loop is over, create the final dataframe from the master_list
df_features2 = pd.DataFrame(data=master_list2)
df_features2.reset_index(inplace=True, drop=True)
df_features2.columns = table_cols2
df_features2.shape
```

Out[24]: (88, 84)

In [25]:
```
df_features2['label'] = df_features['label']
df_features2['is_bending'] = df_features['is_bending']
df_features2['fpath'] = df_features['fpath']
df_features2
```

Out[25]:

	min01	max01	mean01	median01	std01	firstq01	thirdq01	min02	max02	mean02	...
0	38.00	42.33	40.946958	41.250	1.102963	40.3100	42.0000	0.0	1.22	0.375667	...
1	41.75	44.25	42.643292	42.500	0.641849	42.0000	43.3300	0.0	0.94	0.355000	...
2	36.50	46.50	44.057167	44.500	1.556971	43.2500	45.0000	0.0	1.50	0.381042	...
3	33.75	47.75	43.278875	45.000	3.473355	42.0000	45.2500	0.0	3.00	0.673292	...
4	33.00	45.75	41.621208	42.330	3.118644	39.6525	44.2500	0.0	2.83	0.623083	...
...	...	...	...	...	...	...	...	...	...	...	...
83	22.33	46.00	34.966250	35.500	4.733014	32.0000	38.7500	0.0	12.68	4.207958	...
84	21.50	45.67	34.737042	35.500	4.276717	32.0000	37.7500	0.0	12.21	4.217333	...
85	18.33	45.00	33.886458	34.375	5.143776	30.4575	37.5000	0.0	12.44	4.332208	...
86	18.33	44.00	34.836458	35.750	4.869092	32.0000	38.6900	0.0	12.38	4.257750	...
87	23.00	42.75	34.329333	34.710	4.549837	31.3100	37.5425	0.0	14.50	4.315917	...

88 rows × 87 columns

In [26]:
```
# Train-test split

# Create a new df_test, overwriting the old one from (b)
df_test2 = df_features2[df_features2['fpath'].isin(test_datasets)]
print(f'Test dataset shape: {df_test2.shape}')

# Create a new df_train, overwriting the old one from (b)
df_train2 = df_features2[~df_features2['fpath'].isin(test_datasets)]
print(f'Train dataset shape: {df_train2.shape}')
```

```
Test dataset shape: (19, 87)
Train dataset shape: (69, 87)
```

In [27]:
```
top_features2 = ['min01', 'max01', 'mean01',
                 'min02', 'max02', 'mean02',
                 'min12', 'max12', 'mean12']
```

In [28]:
```
# Create a training dataset with only the top features and the is_bending label
df_train_top_f2 = df_train2[top_features2]
df_train_top_f2['is_bending'] = df_train2['is_bending']
df_train_top_f2
```

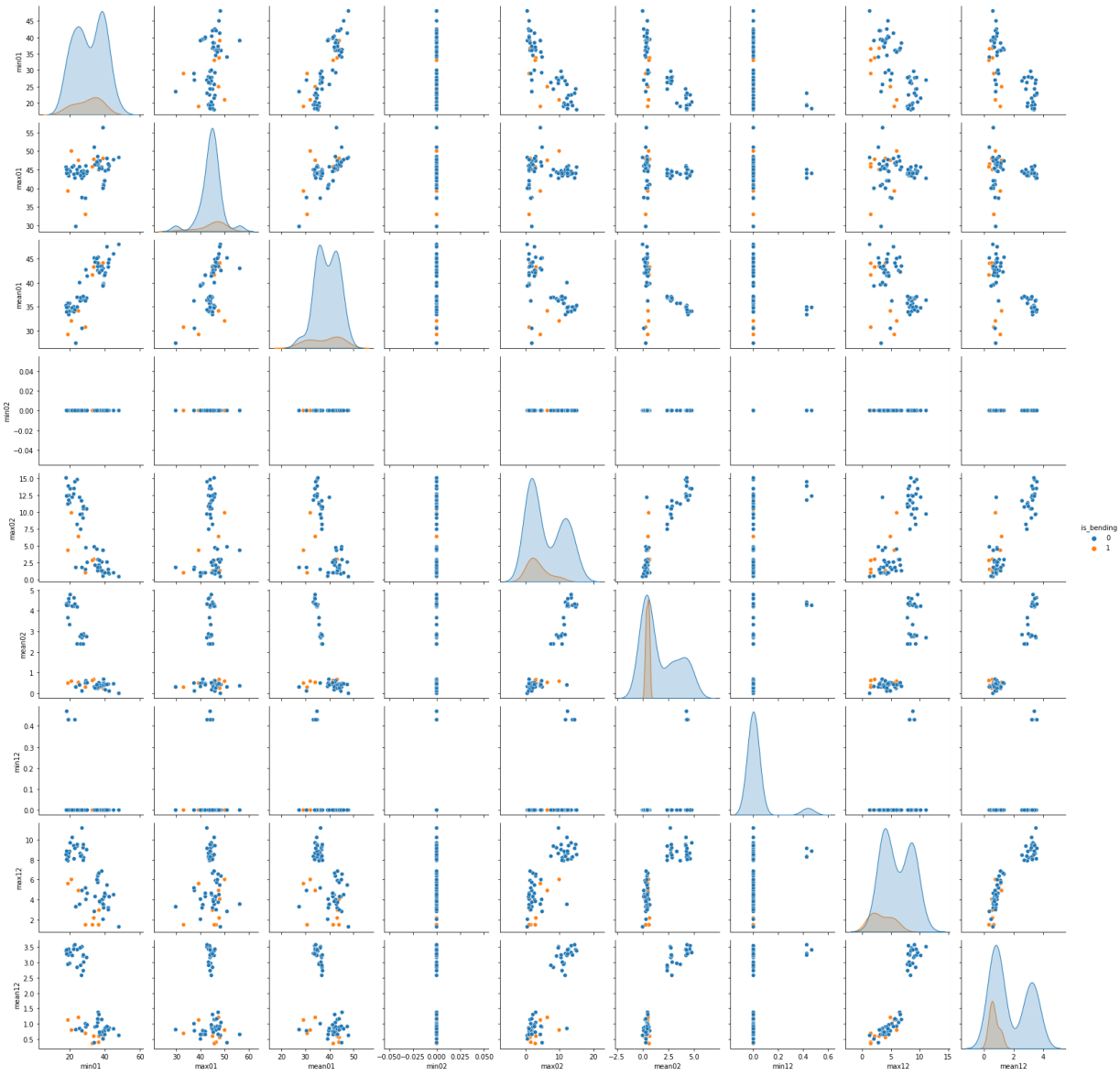
Out[28]:

	min01	max01	mean01	min02	max02	mean02	min12	max12	mean12	is_bending
2	36.50	46.50	44.057167	0.0	1.50	0.381042	0.00	1.50	0.388333	1
3	33.75	47.75	43.278875	0.0	3.00	0.673292	0.00	2.18	0.586083	1
4	33.00	45.75	41.621208	0.0	2.83	0.623083	0.00	1.50	0.347500	1
5	39.00	48.00	44.117042	0.0	1.30	0.250042	0.00	4.06	0.543875	1
6	36.67	45.00	43.486208	0.0	1.30	0.378667	0.00	2.96	0.585750	1
...	...	...	...	...	...	...	...	...	...	...
83	22.33	46.00	34.966250	0.0	12.68	4.207958	0.00	9.39	3.430208	0
84	21.50	45.67	34.737042	0.0	12.21	4.217333	0.00	10.21	3.225917	0
85	18.33	45.00	33.886458	0.0	12.44	4.332208	0.00	8.01	3.274750	0
86	18.33	44.00	34.836458	0.0	12.38	4.257750	0.47	8.84	3.409458	0
87	23.00	42.75	34.329333	0.0	14.50	4.315917	0.43	9.12	3.577833	0

69 rows × 10 columns

In [29]:
```
sns.pairplot(data=df_train_top_f2, hue='is_bending')
```

Out[29]:
```
<seaborn.axisgrid.PairGrid at 0x25b3cc5ba08>
```



The pairplots from **(d) i** and **(d) ii** are highly similar. They do not have considerable differences between them.

(iii) Break the Time Series into l ∈ {1,2,...,20} Time Series & Find the Best Features for each l

iii. Break each time series in your training set into $l \in \{1, 2, \ldots, 20\}$ time series of approximately equal length and use logistic regression[5] to solve the binary classification problem, using time-domain features. Remember that breaking each of the time series does not change the number of instances. It only changes the number of features for each instance. Calculate the p-values for your logistic regression parameters in each model corresponding to each value of l and refit a logistic regression model using your pruned set of features.[6] Alternatively, you can use backward selection using sklearn.feature_selection or glm in R. Use 5-fold cross-validation to determine the best value of the pair (l, p), where p is the number of features used in recursive feature elimination. Explain what the right way and the wrong way are to perform cross-validation in this problem.[7] Obviously, use the right way! Also, you may encounter the problem of class imbalance, which may make some of your folds not having any instances of the rare class. In such a case, you can use *stratified cross validation*. Research what it means and use it if needed.
In the following, you can see an example of applying Python's Recursive Feature Elimination, which is a backward selection algorithm, to logistic regression.

```
# Recursive Feature Elimination
from sklearn import datasets
from sklearn.feature_selection import RFE
from sklearn.linear_model import LogisticRegression
# load the iris datasets
dataset = datasets.load_iris()
# create a base classifier used to evaluate a subset of attributes
model = LogisticRegression()
# create the RFE model and select 3 attributes
rfe = RFE(model, 3)
rfe = rfe.fit(dataset.data, dataset.target)
# summarize the selection of the attributes
print(rfe.support_)
print(rfe.ranking_)
```

This is complex (at least for me!) so I'll break it down in parts:

- First I'll retrofit my previously used code to create 20 dataframes, one for each l value (I know this will burden the memory and isn't a scalable solution, but should work for this problem, and there is no explicit requirement that the solution os production ready)

- Second ??? (let's do #1 and then assess from there)

In [30]:

```
%%time
# Here we go again!

# dictionary to store ALL my dataframes
master_dict = {}

# Set range of l to loop through
l_range = np.arange(1, 21)

# Will be creating one dataframe per l in [1:20]
for l_split in l_range:

    # Create a master list containing all features for all csv's, then convert it
    # The list already follows the order of 'table_cols', listing all attributes
    master_list2 = []

    # Define the cutpoints
    cutpoints = np.arange(0, l_split) + 1

    # Get the number of chunks, from which the chunk size (truncation) will be ob
    n_chunks = max(cutpoints)

    # Loop through all CSVs (done one time for each dataframe to be created)
    for file in all_files:

        # Read the CSV
        tmp = pd.read_csv(file, skiprows=5, header=None, sep=r'\s+|,')

        # Drop the first column with time data
        tmp.drop(0, axis=1, inplace=True)

        # Set the truncation length for splitting the dataframe parts
        truncation = len(tmp)//n_chunks

        # Temporary dataframe to store all parts from the csv file as they are be
        inner_df = pd.DataFrame()

        # For each cutpoint take a chunk/truncation of the dataframe and add it a
        for cutpt in cutpoints:

            # Make a dataframe from the truncation range
            pt1 = tmp[(cutpt-1)*truncation:cutpt*truncation].reset_index(drop=Tru

            # append it to the temporary inner_df (adding columns)
            inner_df = pd.concat(objs=[inner_df, pt1], axis=1, ignore_index=True)

        # Once all columns have been created
        # Extract the features from each variable (column)
        ft_min = pd.Series(inner_df.min().values)
        ft_max = pd.Series(inner_df.max().values)
        ft_mean = pd.Series(inner_df.mean().values)
        ft_median = pd.Series(inner_df.median().values)
        ft_std = pd.Series(inner_df.std().values)
        ft_firstq = pd.Series(inner_df.quantile(0.25).values)
        ft_thirdq = pd.Series(inner_df.quantile(0.75).values)
```

```
        # Put all the features in a dataframe
        # Each of the features (the 'ft_' pd.Series from above) will be a row, th
        df_tmp = pd.DataFrame(data=[ft_min, ft_max, ft_mean, ft_median, ft_std, f

        # The reason for converting to a dataframe of this hape is so that the .m
        # this allows the sequence to be converted...
        # from: the .min() of all variables, followed by the .max() of all varibl
        # to: all features from variable 1, then all features from variable 2, th
        # The resulting pd.Series is now a ?-item long list ordered by variable,
        df_tmp = df_tmp.melt()
        df_tmp = pd.Series(df_tmp['value'])

        # Then append the data already in the correct order to the master_list wh
        master_list2.append(df_tmp)

    # Now that the loop is over (for all files for one given l_split), create the
    df_features2 = pd.DataFrame(data=master_list2)
    df_features2.reset_index(inplace=True, drop=True)
    df_features2['is_bending'] = labels_bending
    master_dict[l_split] = df_features2
```

```
Wall time: 45.6 s
```

There are seven features: min, max, mean, median, std, firstq, thirdq.

There are six variables: avg_rss12, var_rss12, avg_rss13, var_rss13, avg_rss23, var_rss23

The base dataframe has one row per csv file with sample data, resulting in 88 rows (there are 88 CSVs). This never changes across dataframes. The base dataframe has one column for each combination of those (6x7), resulting in 42 columns.

For each increase in **l** the number os variables increases by 6, which means that the number of combinations increase by 42, which means that the number of columns increase by 42.

Hence the first dataframe at master_dict[1] will have 42 columns (42 x 01 = 42)

The last dataframe at master_dict[20] will have 840 columns (42 x 20 = 840)

All dataframes have one more column, which is the 'is_bending' label related to the binary classification problem.

In [31]:
```
print(f'master_dict[1].shape: {master_dict[1].shape}')
print(f'master_dict[20].shape: {master_dict[20].shape}')
```

```
master_dict[1].shape: (88, 43)
master_dict[20].shape: (88, 841)
```

In [32]:
```
# Create arrays of variable length to properly name the dataframes

# Dictionary to store the names for each dataframes
dict_col_names = {}

# The number of variables increase by 6 for each successive dataframe
variable_count = np.arange(6, 121, 6)

# The features remain 7, but multiply the variables
name_list = ['min_', 'max_', 'mean_', 'median_', 'std_', 'firstq_', 'thirdq_']

for idx, var_num in enumerate(variable_count):
    idx += 1 # because master_dict starts at 1
    number_of_variables = np.arange(1, var_num+1)
    list_col_names = []
    for number in number_of_variables:
        for col_n in name_list:
            # The dataframes columns grow by 6*7 per iteration (number_of_variabl
            tmp_name = f'{col_n}{number}'
            list_col_names.append(tmp_name)
    list_col_names.append('is_bending')
    dict_col_names[idx] = list_col_names

print(f'len(dict_col_names[1]: {len(dict_col_names[1])}')
print(f'len(dict_col_names[20]: {len(dict_col_names[20])}')
```

```
len(dict_col_names[1]: 43
len(dict_col_names[20]: 841
```

In [33]:
```
# Now rename all dataframes
for idx in range(1, 21):
    master_dict[idx].columns = dict_col_names[idx]
```

In [34]:
```
# Check result
master_dict[20].head(3)
```

Out[34]:

	min_1	max_1	mean_1	median_1	std_1	firstq_1	thirdq_1	min_2	max_2	mean_2	...
0	39.00	40.67	39.673750	39.50	0.488439	39.4375	39.815	0.00	0.83	0.474583	...
1	41.75	44.25	43.437917	43.50	0.530987	43.4575	43.670	0.43	0.83	0.524583	...
2	40.00	45.00	43.513750	44.04	1.627673	42.1875	45.000	0.00	0.87	0.387500	...

3 rows × 841 columns

In [35]:
```
# Now each dictionary has to be split in a training and testing parts
master_dict_test = {}
master_dict_train = {}

# Loop though each ditionary
for l_split in l_range:

    # First need to append the file patchs to each dictionary, so they can be use
    master_dict[l_split]['fpath'] = all_files

    # Then make the train/test split based on the list test_datasets
    df_test = master_dict[l_split][master_dict[l_split]['fpath'].isin(test_datase
    df_train = master_dict[l_split][~master_dict[l_split]['fpath'].isin(test_data

    # Now drop the file paths from the train and test dataframes, as they aren't
    df_test.drop(['fpath'], axis=1, inplace=True)
    df_train.drop(['fpath'], axis=1, inplace=True)

    # And store them in the new dictionaries
    master_dict_test[l_split] = df_test
    master_dict_train[l_split] = df_train
```

In [36]:
```
# Check if shapes are correct
print(f'master_dict_test[1].shape: {master_dict_test[1].shape}')
print(f'master_dict_test[20].shape: {master_dict_test[20].shape}')
print(f'master_dict_train[1].shape: {master_dict_train[1].shape}')
print(f'master_dict_train[20].shape: {master_dict_train[20].shape}')
```

```
master_dict_test[1].shape: (19, 43)
master_dict_test[20].shape: (19, 841)
master_dict_train[1].shape: (69, 43)
master_dict_train[20].shape: (69, 841)
```

Part #1 is done. All dataframes are prepared for training the Logistic Regression algorithms!

I'll choose to go the route of recursive feature elimination using cross-validation

The ensure the CV won't be biased, the parameters will be kept the same, and it'll be done using only the training dataset, so as avoid data leakage from the test dataset, which would compromise the results.

The wrong approach to cross-validation is to first use the entire dataset to select the predictor variables and then afterwards run the CV with the variables already selected. This is problematic because the variables will have been selected with the model having access to all samples, effectively not having a validation dataset.

The right approach to cross-validation is to first split the data in K-folds, then afterwards select the predictor variables to be used, so that for each CV-fold the validation fold is the one used to select the predictor variables. That is, the right way is to first split the folds, then select the predictors based on the training folds, which means the predictors may vary between the CV iterations.

In [37]:
```
from sklearn.feature_selection import RFECV
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import StratifiedKFold
```

In [38]:
```
# Instance a Recursive Feature Eliminator which uses Cross-Validation
# Set n_jobs to -2 to use parallel processing but leaving 1 core spared, just in
rfecv = RFECV(estimator = LogisticRegression(penalty='none'),
              min_features_to_select=1,
              scoring='accuracy',
              cv=StratifiedKFold(5),
              n_jobs=-1)
```

In [39]:
```
%%time
# Notice, this cell takes ~5 minutes

# Dicts to store the features and accuracies from each l-sized dataframe
dict_features = {}
acc_scores = {}

# Loop through all 'l_splits' in [1:20]
for l_split in tqdm(l_range):

    x = master_dict_train[l_split].iloc[:, :-1]
    y = master_dict_train[l_split].iloc[:, -1]

    rfecv.fit(x, y)

    boolean_mask = rfecv.support_ # This returns an array of boolean indicating w

    # Filter the column names of the df using 'boolean_mask', so as to keep only
    # Those features will be the ones that resulted in the highest accuracy
    # (if there's more than one identical accuracy scores, RFECV picks the one wi
    # And store those features (column names) in a dictionary indexed by 'l_split
    dict_features[l_split] = x.iloc[:, boolean_mask].columns
    # Store all accuracies obtained during recusive feature elimination for the g
    acc_scores[l_split] = rfecv.grid_scores_
```

```
100%                                    20/20 [05:27<00:00, 16.38s/it]


Wall time: 4min 39s
```

From RFECV Documentation: [https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html (https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html)](https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.RFECV.html)

"grid_scores_ : array of shape [n_subsets_of_features]

> The cross-validation scores such that grid_scores_[i] corresponds to the CV score of the i-th subset of features."

Restating it: grid_scores_ at position [0] contains the accuracy using only 1 feature, at position [1] containts the accuracy using 2 features. Implicitly, the 1, 2 or however many features it used to get that accuracy is always the combination of features that results in the best accuracy.

This information can be crossed by looking at dict_features at a given l_split (which will say which features were chosen for that l_split), and then checking the accuracy in acc_scores[l_split] in the item position which matches the number of features shown in dict_features[l_split].

Eg:

dict_features[3] has: ['max_2', 'min_5', 'max_5', 'thirdq_5', 'thirdq_7', 'max_18'], which totals 6 features.

acc_scores[3] has a list of all accuracies, but when we look at the 6th item in that list (which will be index 5), we find that acc_scores[3][5] = 0.9857142857142858

This way, for each dataframe in l_splits, the chosen features and their related accuracy can be found

In [40]:
```
# 'dict_features' contains the features chosen for each value of 'l'
# For example, when l=3, the chosen features were:
dict_features[3]
```

Out[40]:
```
Index(['max_2', 'min_5', 'max_5', 'thirdq_5', 'thirdq_7', 'max_18'], dtype='obj
ect')
```

In [41]:
```
# When l=3 and n_features=6 the score is: (filtering with [5] because indexation
acc_scores[3][5]
```

Out[41]:
```
0.9857142857142858
```

In [42]:
```
# The dictionary returns the feature list with some weird formatting that has Inc
# Use list comprehension to get a clean list of features per 'l'
# Note that unlike the other dictionaries which go [1:20], this list will go [0:1
list_features_clean = [list(dict_features.values())[i].values for i in range(20)]
```

In [43]:
```
# Summarize the number of features chosen per 'l' value
num_features_chosen = {}
for l_split in l_range:
    num_features_chosen[l_split] = len(dict_features[l_split])
```

In [44]:
```
# Now extract the accuracy score obtained from using the chosen features
best_acc_scores = {}

for k,v in num_features_chosen.items():

    # k markes the 'l' in 'l_range'
    # v means the number of features which produced the best score for a given k
    # Since v starts at 1, but the list indexing starts at 0, then position with
    # This mean that eg: with 1 feature the realted accuracy is found in positior
    # In the 'acc_scores' list will contain the accuracy_score assocaited with us
    best_acc_scores[k] = acc_scores[k][v-1]
```

In [45]:
```
# Summarizing it all in a dataframe
summary_df = pd.DataFrame()
summary_df['l_split'] = num_features_chosen.keys()
summary_df['accuracy'] = best_acc_scores.values()
summary_df['num_features_chosen'] = num_features_chosen.values()
summary_df['list_features_chosen'] = list_features_clean
summary_df
```

Out[45]:

	l_split	accuracy	num_features_chosen	list_features_chosen
0	1	1.000000	4	[thirdq_1, min_5, max_5, firstq_5]
1	2	0.985714	14	[max_1, max_2, firstq_3, min_5, max_5, mean_5,...
2	3	0.985714	6	[max_2, min_5, max_5, thirdq_5, thirdq_7, max_18]
3	4	1.000000	18	[max_1, max_2, min_3, min_5, max_5, mean_5, me...
4	5	0.985714	20	[min_3, min_5, max_5, mean_5, median_5, firstq...
5	6	0.985714	20	[min_3, max_5, mean_5, median_5, firstq_5, thi...
6	7	0.985714	23	[min_3, min_5, max_5, mean_5, firstq_5, thirdq...
7	8	0.970330	24	[max_5, mean_5, firstq_5, thirdq_5, max_7, min...
8	9	0.985714	31	[max_5, mean_5, firstq_5, thirdq_5, min_9, max...
9	10	0.971429	20	[max_5, min_11, mean_11, median_11, firstq_11,...
10	11	0.971429	2	[firstq_11, max_43]
11	12	0.985714	4	[min_11, firstq_11, max_37, min_65]
12	13	0.971429	2	[min_11, min_23]
13	14	0.971429	4	[min_11, median_11, max_55, min_77]
14	15	0.985714	33	[max_5, min_11, max_11, mean_11, median_11, fi...
15	16	0.985714	20	[min_11, mean_11, firstq_11, thirdq_11, min_17...
16	17	0.985714	15	[min_11, min_17, mean_17, median_17, firstq_17...
17	18	1.000000	6	[min_17, mean_17, firstq_17, max_61, max_73, m...
18	19	0.985714	4	[min_11, firstq_17, min_35, max_73]
19	20	1.000000	22	[max_5, min_11, min_17, mean_17, median_17, fi...

(iv) Confusion Matrix & ROC-AUC

iv. Report the confusion matrix and show the ROC and AUC for your classifier on train data. Report the parameters of your logistic regression β_i's as well as the p-values associated with them.

I'll chose to use the model which had the best training accuracy, and since multiple models had a training accuracy of 1, I'll choose to select the one with the least engineered features, which happend to be when l = 1.

The dataset associated with this model can be found in master_dict_train[1].

In [46]:
```
# Getting the l_split associated with the best accuracy
selected_split_idx = np.argmax(summary_df['accuracy'])
selected_split_num = summary_df['l_split'][selected_split_idx]
selected_split_num
```

Out[46]:
```
1
```

In [47]:
```
# Showing the model to be used
summary_df[summary_df.l_split==selected_split_num]
```

Out[47]:

	l_split	accuracy	num_features_chosen	list_features_chosen
0	1	1.0	4	[thirdq_1, min_5, max_5, firstq_5]

In [48]:
```
# The best set of features for this dataset is:
selected_features = [i for i in summary_df[summary_df.l_split==selected_split_num
selected_features
```

Out[48]:
```
['thirdq_1', 'min_5', 'max_5', 'firstq_5']
```

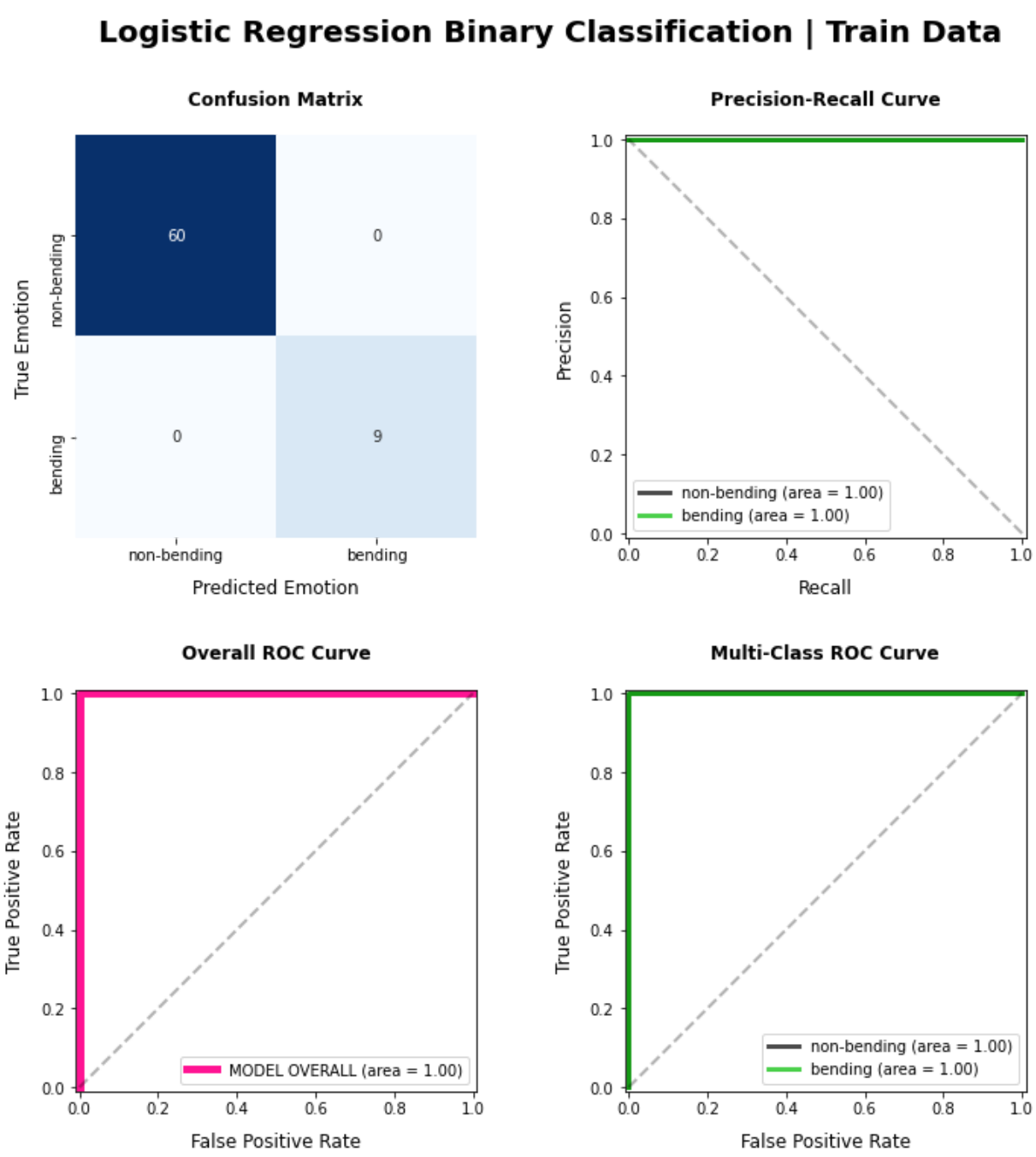
In [49]:
```
# Getting the dataframe associated with the chosen l_split
selected_df_train = master_dict_train[selected_split_num]
```

In [50]:
```
# Logistic regression for the selected dataset with the selected features
model_lr = LogisticRegression(penalty='none')

# Split x and y
x = selected_df_train[selected_features]
y = selected_df_train.iloc[:, -1]

# Train
model_lr.fit(x, y)

# Predict (also on train dataset)
pred_binary = model_lr.predict(x)
pred_probs = model_lr.predict_proba(x)
pred_logp = model_lr.predict_log_proba(x)

# One-hot encode y
true_onehot = [[1,0] if i==0 else [0,1] for i in y]

# ROC-AUC
roc_auc = roc_auc_score(true_onehot, pred_probs, average='micro')
```

In [51]:
```
print(f'The train dataset accuracy is: {model_lr.score(x, y)}')
print(f'The ROC AUC is {roc_auc}')
```

```
The train dataset accuracy is: 1.0
The ROC AUC is 1.0
```

In [52]:
```
classes = ['non-bending', 'bending']
plot_classification_results(true_onehot, pred_probs, labels_binary_unique)
plt.suptitle('Logistic Regression Binary Classification | Train Data', y=1.05, fo
plt.show()
```



In [53]:
```
# In order to get the regression betas and p-values need to use the logistic regr
# As scikit-learn doesn't have those retrievals implemented
import statsmodels.api as sm
```

In [54]:
```
# Train the statsmodels Logit regression
model_sm = sm.Logit(endog=y, exog=sm.add_constant(x))

# Fit the model
model_sm_fitted = model_sm.fit_regularized()

# Get the parameters names
logit_names = model_sm_fitted.params.index

# Get the params coefficients
logit_params = model_sm_fitted.params.values

# Get the p-values
logit_pvals_ = model_sm_fitted.pvalues.values

# When there's perfect classification, statsmodels will return np.NaN for p-value
logit_pvalues = np.zeros(len(logit_pvals_))
for idx, p in enumerate(logit_pvals_):
    if p > 0:
        logit_pvalues[idx] = p
    else:
        logit_pvalues[idx] = 0
```

```
Optimization terminated successfully    (Exit mode 0)
            Current function value: 9.024642575047663e-11
            Iterations: 39
            Function evaluations: 46
            Gradient evaluations: 39
```

In [55]:
```
# Make a dataframe summarizing the Logistic Regression parameters
summary_logit = pd.DataFrame()
summary_logit['parameter'] = logit_names
summary_logit['coefficient'] = logit_params
summary_logit['p-value'] = logit_pvalues

summary_logit
```

Out[55]:

	parameter	coefficient	p-value
0	const	-3.523199	1.0
1	thirdq_1	-29.350200	0.0
2	min_5	7.335411	0.0
3	max_5	24.516355	0.0
4	firstq_5	26.566349	0.0

The constant (beta0) has a p-value of 1, indicating that it is **not** significant.

All other parameters (betas) have a p-value small enough that it gets rounded to 0.

(v) Test Dataset

v. Test the classifier on the test set. Remember to break the time series in your test set into the same number of time series into which you broke your training set. Remember that the classifier has to be tested using the features extracted from the test set. Compare the accuracy on the test set with the cross-validation accuracy you obtained previously.

When I made all my l_splitted train dataset, I also made the test datasets, so now they only need to be retrieved.

They are stored in master_dict_test, indexed by [l_split]

In [56]:
```
print(f'Model was trained on master_dict_train[{selected_split_num}], of shape {m
print()
print(f'Test set dataframe is master_dict_test[{selected_split_num}], of shape {m
```

```
Model was trained on master_dict_train[1], of shape (69, 43)

Test set dataframe is master_dict_test[1], of shape (19, 43)
```

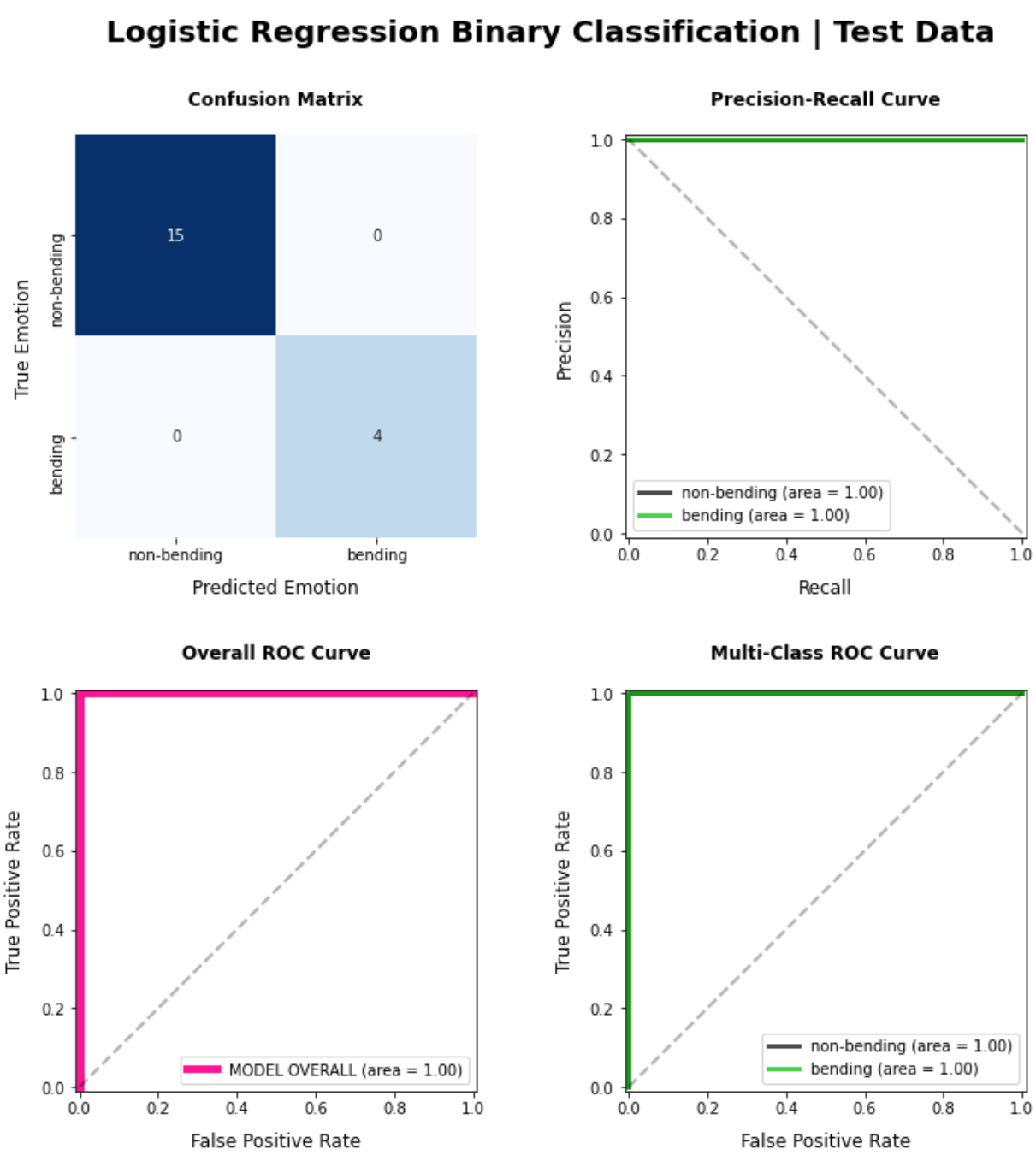
In [57]:
```
# Select the test dataset
selected_df_test = master_dict_test[selected_split_num]
```

In [58]:
```
# For the test dataset:
# Split x and y
x = selected_df_test[selected_features]
y = selected_df_test.iloc[:, -1]
# Predict on the test dataset
pred_binary = model_lr.predict(x)
pred_probs = model_lr.predict_proba(x)
pred_logp = model_lr.predict_log_proba(x)
# One-hot encode y
true_onehot = [[1,0] if i==0 else [0,1] for i in y]
# ROC-AUC
roc_auc = roc_auc_score(true_onehot, pred_probs, average='micro')
```

In [59]:
```
print(f'The test dataset accuracy is: {model_lr.score(x, y)}')
print(f'The ROC AUC is {roc_auc}')
```

```
The test dataset accuracy is: 1.0
The ROC AUC is 1.0
```

In [60]:
```
classes = ['non-bending', 'bending']
plot_classification_results(true_onehot, pred_probs, labels_binary_unique)
plt.suptitle('Logistic Regression Binary Classification | Test Data', y=1.05, for
plt.show()
```



The accuracy for both the cross-validated train dataset as well as the test dataset was the same, with both achieving 100% classification accuracy!

Equally, both train and test achieved a ROC-AUC of 1.0.

(vi) Class Separation & Parameter Instability

vi. Do your classes seem to be well-separated to cause instability in calculating logistic regression parameters?

Yes. From the perfect classification accuracy it's evident that the classes are well-separated, causing instability in calculating the logistic regression parameters.

(vii) Class Balance

vii. From the confusion matrices you obtained, do you see imbalanced classes? If yes, build a logistic regression model based on case-control sampling and adjust its parameters. Report the confusion matrix, ROC, and AUC of the model.

Yes, the classes are imbalanced. On the training daset there are 60 samples for non-bending (class 0) and only 9 for bending (class 1). In the training dataset there are 15 samples for non-bending (class 0) and only 4 for bending (class 1).

In [61]:
```
from sklearn.utils import resample
```

In [62]:
```
# Split the selected_df_train into sub-dataframes for each class, so that case-co
selected_df_train_1 = selected_df_train[selected_df_train.is_bending == 1]
selected_df_train_0 = selected_df_train[selected_df_train.is_bending == 0]
```

In [63]:
```
# Resample the dataframe for class 1 (bending) to get 60 samples
selected_df_train_1_bootstrapped = resample(selected_df_train_1, replace=True, n_
# Make a new train dataset, now with balanced classes
selected_df_train_balanced = pd.concat([selected_df_train_1_bootstrapped, selecte
selected_df_train_balanced.shape
```

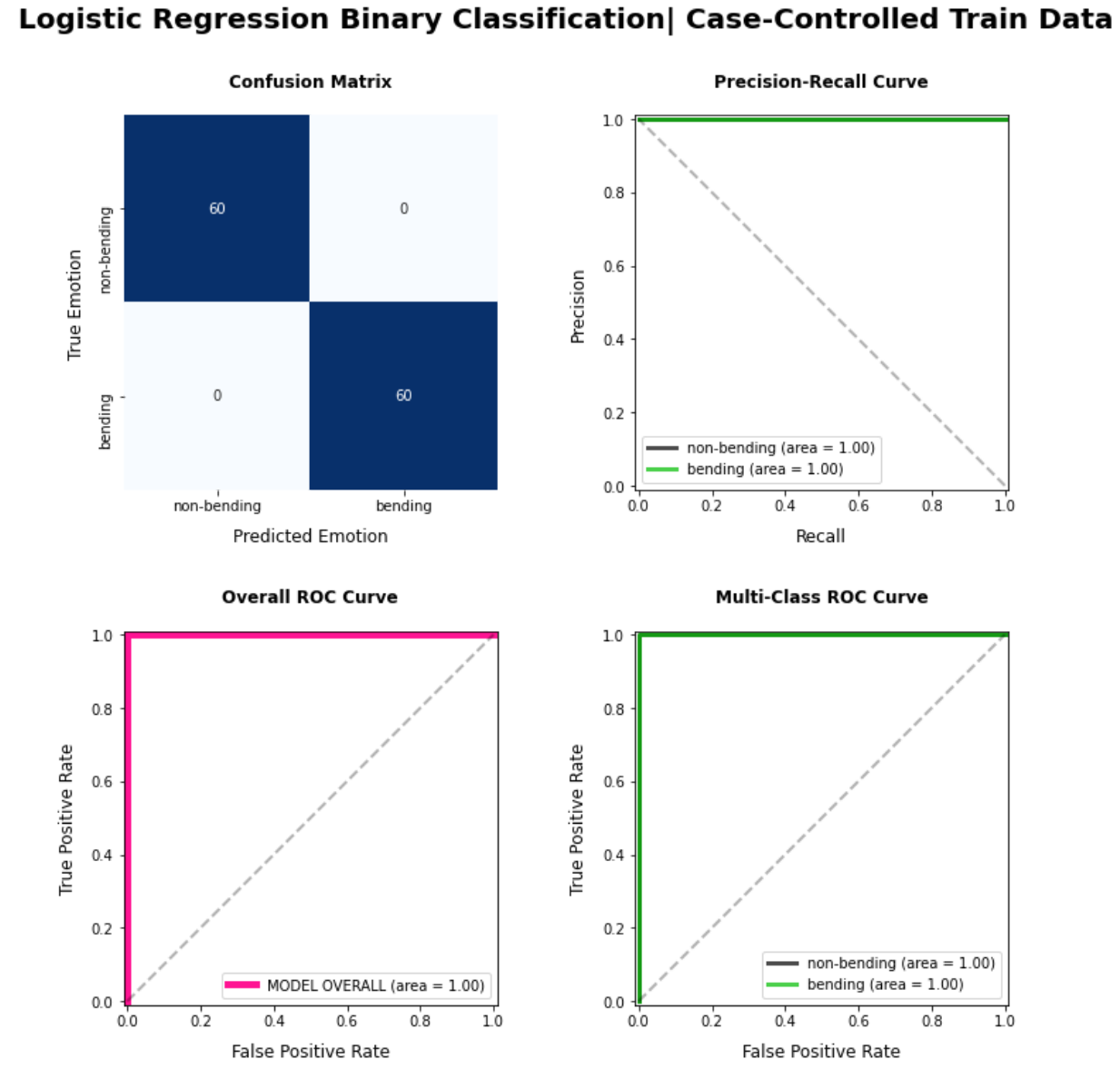
Out[63]:
```
(120, 43)
```

In [64]:
```
# For the balanced dataset:
# Split x and y
x = selected_df_train_balanced[selected_features]
y = selected_df_train_balanced.iloc[:, -1]
# Train
model_lr.fit(x, y)
# Predict on the train dataset
pred_binary = model_lr.predict(x)
pred_probs = model_lr.predict_proba(x)
pred_logp = model_lr.predict_log_proba(x)
# One-hot encode y
true_onehot = [[1,0] if i==0 else [0,1] for i in y]
# ROC-AUC
roc_auc = roc_auc_score(true_onehot, pred_probs, average='micro')
```

In [65]:
```
print(f'The balanced dataset accuracy is: {model_lr.score(x, y)}')
print(f'The ROC AUC is {roc_auc}')
```

```
The balanced dataset accuracy is: 1.0
The ROC AUC is 1.0
```

In [66]:
```
plot_classification_results(true_onehot, pred_probs, labels_binary_unique)
plt.suptitle('Logistic Regression Binary Classification | Case-Controlled Train D
plt.show()
```



(e) L1 Penalization

(e) Binary Classification Using $\mathscr{L}_1$-penalized logistic regression

(i) Best Features for each l ∈ {1,2,...,20} using L1 Penalization

i. Repeat 1(d)iii using $\mathscr{L}_1$-penalized logistic regression,[8] i.e. instead of using p-values for variable selection, use $\mathscr{L}_1$ regularization. Note that in this problem, you have to cross-validate for both l, the number of time series into which you break each of your instances, and λ, the weight of $\mathscr{L}_1$ penalty in your logistic regression objective function (or C, the budget). Packages usually perform cross-validation for λ automatically.[9]

In [67]:
```
from sklearn.linear_model import LogisticRegressionCV
```

In [68]:
```
# Instance the Logistic Regression with 5 CVs, L1 Penalty and 'liblinear'
model_logit_l1 = LogisticRegressionCV(cv=StratifiedKFold(5),
                                      penalty="l1",
                                      solver="liblinear",
                                      scoring='accuracy',
                                      n_jobs=-1,
                                      random_state=1)
```

In [69]:
```
%%time

# Dicts to store the features and accuracies from each l-sized dataframe
dict_features_l1_list = {}
dict_features_l1_num = {}
l1_C_weights = {}
l1_lambda_weights = {}
l1_accuracies = {}

# Loop through all 'l_splits' in [1:20]
for l_split in tqdm(l_range):

    # Split X and Y
    x = master_dict_train[l_split].iloc[:, :-1]
    y = master_dict_train[l_split].iloc[:, -1]

    # Train
    model_logit_l1.fit(x, y)

    # Convert the list of coefficient weights to a boolean list that merely indic
    coefs_bool = []
    for i in model_logit_l1.coef_[0]:
        if i == 0:
            coefs_bool.append(False)
        else:
            coefs_bool.append(True)
    # Add an extra False at the end, to account for the label column
    coefs_bool.append(False)

    # Use the boolean list to get the features selected by the model
    dict_features_l1_list[l_split] = [i for i in master_dict_train[l_split].colum

    # Append the count of chosen features to dict_features_l1_num
    dict_features_l1_num[l_split] = len(dict_features_l1_list[l_split])

    # Get the optimal value of C found by the model, putting it on the L1_C_weigh
    # C = 1/λ
    l1_C_weights[l_split] = model_logit_l1.C_[0]

    # "model_logit_l1.Cs_" is the array with all C values tested on cross-valitic

    # And invert C to get the actual value for the penalty (λ) parameter
    l1_lambda_weights[l_split] = 1/l1_C_weights[l_split]

    # Get the model accuracy
    l1_accuracies[l_split] = model_logit_l1.score(x, y)
```

100% 20/20 [00:05<00:00, 3.65it/s]

```
Wall time: 1.96 s
```

Notes on the C_ parameter

From scikit-learn documentation: [https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html)](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html)

C_ : ndarray of shape (n_classes,) or (n_classes - 1,)

Array of C that maps to the best scores across every class. If refit is set to False, then for each class, the best C is the average of the C's that correspond to the best scores for each fold. C_ is of shape(n_classes,) when the problem is binary.

Each of the values in Cs describes the inverse of regularization strength. Like in support vector machines, smaller values specify stronger regularization.

That is: C = 1/λ

From Kaggle: [https://www.kaggle.com/joparga3/2-tuning-parameters-for-logistic-regression (https://www.kaggle.com/joparga3/2-tuning-parameters-for-logistic-regression)](https://www.kaggle.com/joparga3/2-tuning-parameters-for-logistic-regression)

Lambda (λ) controls the trade-off between allowing the model to increase it's complexity as much as it wants with trying to keep it simple. For example, if λ is very low or 0, the model will have enough power to increase it's complexity (overfit) by assigning big values to the weights for each parameter. If, in the other hand, we increase the value of λ, the model will tend to underfit, as the model will become too simple.

Parameter C will work the other way around. For small values of C, we increase the regularization strength which will create simple models which underfit the data. For big values of C, we low the power of regularization which imples the model is allowed to increase it's complexity, and therefore, overfit the data.

In [70]:
```
# Put the results into a dataframe for visualization
summary_l1 = pd.DataFrame()
summary_l1['l_split'] = l_range
summary_l1['accuracy'] = l1_accuracies.values()
summary_l1['error_rate'] = 1 - summary_l1['accuracy']
summary_l1['l1_C_weights'] = l1_C_weights.values()
summary_l1['l1_lambda_weights'] = l1_lambda_weights.values()
summary_l1['num_features_chosen'] = dict_features_l1_num.values()
summary_l1['list_features_chosen'] = dict_features_l1_list.values()
```

In [71]:
```
summary_l1
```

Out[71]:

	l_split	accuracy	error_rate	l1_C_weights	l1_lambda_weights	num_features_chosen	list_fea
0	1	1.0	0.0	0.359381	2.782559	5	[th
1	2	1.0	0.0	2.782559	0.359381	14	[mea f
2	3	1.0	0.0	2.782559	0.359381	16	[ma th
3	4	1.0	0.0	2.782559	0.359381	19	[ma th
4	5	1.0	0.0	2.782559	0.359381	20	[m mi
5	6	1.0	0.0	21.544347	0.046416	51	mea
6	7	1.0	0.0	2.782559	0.359381	26	[m m
7	8	1.0	0.0	10000.000000	0.000100	287	mea
8	9	1.0	0.0	0.359381	2.782559	10	[max thir
9	10	1.0	0.0	2.782559	0.359381	30	[m m
10	11	1.0	0.0	2.782559	0.359381	29	thirc
11	12	1.0	0.0	2.782559	0.359381	17	[mir r
12	13	1.0	0.0	2.782559	0.359381	35	[m t
13	14	1.0	0.0	0.359381	2.782559	15	[r median
14	15	1.0	0.0	166.810054	0.005995	260	mea
15	16	1.0	0.0	2.782559	0.359381	42	[m m

	l_split	accuracy	error_rate	l1_C_weights	l1_lambda_weights	num_features_chosen	list_fea
16	17	1.0	0.0	166.810054	0.005995	290	mea
17	18	1.0	0.0	0.359381	2.782559	13	[n firs
18	19	1.0	0.0	0.359381	2.782559	17	[m r
19	20	1.0	0.0	2.782559	0.359381	53	[th

Interestingly, using L1 penalization allowed all l_splits to achieve 100% accuracy on the training dataset

(ii) Compare L1 Penalization with Recursive Feature Elimination

ii. Compare the $\mathscr{L}_1$-penalized with variable selection using p-values. Which one performs better? Which one is easier to implement?

In [72]:
```
# Recalling the results table from using Recursive Feature Elimination
summary_df
```

Out[72]:

	l_split	accuracy	num_features_chosen	list_features_chosen
0	1	1.000000	4	[thirdq_1, min_5, max_5, firstq_5]
1	2	0.985714	14	[max_1, max_2, firstq_3, min_5, max_5, mean_5,...
2	3	0.985714	6	[max_2, min_5, max_5, thirdq_5, thirdq_7, max_18]
3	4	1.000000	18	[max_1, max_2, min_3, min_5, max_5, mean_5, me...
4	5	0.985714	20	[min_3, min_5, max_5, mean_5, median_5, firstq...
5	6	0.985714	20	[min_3, max_5, mean_5, median_5, firstq_5, thi...
6	7	0.985714	23	[min_3, min_5, max_5, mean_5, firstq_5, thirdq...
7	8	0.970330	24	[max_5, mean_5, firstq_5, thirdq_5, max_7, min...
8	9	0.985714	31	[max_5, mean_5, firstq_5, thirdq_5, min_9, max...
9	10	0.971429	20	[max_5, min_11, mean_11, median_11, firstq_11,...
10	11	0.971429	2	[firstq_11, max_43]
11	12	0.985714	4	[min_11, firstq_11, max_37, min_65]
12	13	0.971429	2	[min_11, min_23]
13	14	0.971429	4	[min_11, median_11, max_55, min_77]
14	15	0.985714	33	[max_5, min_11, max_11, mean_11, median_11, fi...
15	16	0.985714	20	[min_11, mean_11, firstq_11, thirdq_11, min_17...
16	17	0.985714	15	[min_11, min_17, mean_17, median_17, firstq_17...
17	18	1.000000	6	[min_17, mean_17, firstq_17, max_61, max_73, m...
18	19	0.985714	4	[min_11, firstq_17, min_35, max_73]
19	20	1.000000	22	[max_5, min_11, min_17, mean_17, median_17, fi...

L1-penalized variable selection works better than Recursive Feature Elimination (my chosen approach for 1(d)iii), with L1-penalized being able to achieve 100% for all l ∈ {1,2,...,20} splits of the training dataset, which RFE wasn't able to.

L1-penalized variable selection is also easier to implement, requiring only an extra argument on the LogisticRegression functions, and it also computes faster. Unlike p-value variable selection, this approach does not require refitting the model.

(f) Multi-Class Classification

(f) Multi-class Classification (The Realistic Case)

(i) Find the best l ∈ {1.2.....20}

i. Find the best l in the same way as you found it in 1(e)i to build an $\mathscr{L}_1$-penalized multinomial regression model to classify all activities in your training set.[10] Report your test error. Research how confusion matrices and ROC curves are defined for multiclass classification and show them for this problem if possible.[11]

In [73]:
```
# A list containing multiclass labels was already created previously, under the v
# It was generated from the files path
# First, it need to be converted as to merge 'bending1' and 'bending2' into a sir
labels_new = ['bending' if 'bending' in label else label for label in labels]
labels_multiclass_unique = np.unique(labels_new)
len(labels_new)
```

Out[73]: 88

In [74]:
```
labels_multiclass_unique
```

Out[74]:
```
array(['bending', 'cycling', 'lying', 'sitting', 'standing', 'walking'],
      dtype='<U8')
```

In [75]:
```
# Now retrofit the master_dict, to add multiclass labels, and then split it into
multiclass_dict_test = {}
multiclass_dict_train = {}

# First add the multiclass labels to each dataframe in master_dict
for l_split in l_range:
    master_dict[l_split]['labels_multiclass'] = labels_new

    # Then make the train/test split based on the list test_datasets
    df_test = master_dict[l_split][master_dict[l_split]['fpath'].isin(test_datase
    df_train = master_dict[l_split][~master_dict[l_split]['fpath'].isin(test_data

    # Now drop the unused tag columns from train/test DFs, as they aren't feature
    df_test.drop(['fpath', 'is_bending'], axis=1, inplace=True)
    df_train.drop(['fpath', 'is_bending'], axis=1, inplace=True)

    # And store them in the new dictionaries
    multiclass_dict_test[l_split] = df_test
    multiclass_dict_train[l_split] = df_train
```

In [76]:
```
# Check if shapes are correct
print(f'multiclass_dict_test[1].shape: {multiclass_dict_test[1].shape}')
print(f'multiclass_dict_test[20].shape: {multiclass_dict_test[20].shape}')
print(f'multiclass_dict_train[1].shape: {multiclass_dict_train[1].shape}')
print(f'multiclass_dict_train[20].shape: {multiclass_dict_train[20].shape}')
```

```
multiclass_dict_test[1].shape: (19, 43)
multiclass_dict_test[20].shape: (19, 841)
multiclass_dict_train[1].shape: (69, 43)
multiclass_dict_train[20].shape: (69, 841)
```

In [77]:
```
from sklearn.multiclass import OneVsRestClassifier

# Use OneVsRest to create one Logistic regression per class
ovr_classifier = OneVsRestClassifier(LogisticRegressionCV(cv=StratifiedKFold(5),
                                                           penalty="l1",
                                                           solver="liblinear",
                                                           scoring='accuracy',
                                                           n_jobs=-1,
                                                           random_state=1))
```

In [78]:
```
# One Hot Encoder for y
from sklearn.preprocessing import OneHotEncoder
OHE = OneHotEncoder(sparse=False)
OHE.fit(np.array(labels_new).reshape(-1, 1))
```

Out[78]:
```
OneHotEncoder(sparse=False)
```

In [79]:

```
%%time

# Dicts to store the features and accuracies from each l-sized dataframe
dict_features_l1_list = {}
dict_features_l1_num = {}
l1_C_weights = {}
l1_lambda_weights = {}
l1_accuracies = {}

# Loop through all 'l_splits' in [1:20]
for l_split in tqdm(l_range):

    # Split X and Y
    x = multiclass_dict_train[l_split].iloc[:, :-1]
    y = multiclass_dict_train[l_split].iloc[:, -1]

    # One-Hot-Encode Y
    y = OHE.transform(y.values.reshape(-1, 1))

    # Train
    ovr_classifier.fit(x, y)

    # Convert the list of coefficient weights to a boolean list that merely indic
    coefs_bool = []
    for i in ovr_classifier.coef_[0]:
        if i == 0:
            coefs_bool.append(False)
        else:
            coefs_bool.append(True)
    # Add an extra False at the end, to account for the label column
    coefs_bool.append(False)

    # Use the boolean list to get the features selected by the model
    dict_features_l1_list[l_split] = [i for i in multiclass_dict_train[l_split].c

    # Append the count of chosen features to dict_features_l1_num
    dict_features_l1_num[l_split] = len(dict_features_l1_list[l_split])

    # There is one C_ for each class estimator in ovr_classifier.estimators_[0]
    # Calculating the average C_ for reporting
    avg_c = 0
    for estimator in range(len(ovr_classifier.estimators_)):
        avg_c += ovr_classifier.estimators_[estimator].C_
    avg_c = avg_c/len(ovr_classifier.estimators_)

    # Get the optimal value of C found by the model, putting it on the L1_C_weigh
    # C = 1/λ
    l1_C_weights[l_split] = avg_c[0]

    # "model_logit_l1.Cs_" is the array with all C values tested on cross-valitic

    # And invert C to get the actual value for the penalty (λ) parameter
    l1_lambda_weights[l_split] = 1/l1_C_weights[l_split]

    # Get the model accuracy
    l1_accuracies[l_split] = ovr_classifier.score(x, y)
```

```
100%                                                  20/20 [00:19<00:00, 1.04it/s]

Wall time: 11.5 s
```

In [80]:
```
# Put the results into a dataframe for visualization
summary_multiclass = pd.DataFrame()
summary_multiclass['l_split'] = l_range
summary_multiclass['accuracy'] = l1_accuracies.values()
summary_multiclass['error_rate'] = 1 - summary_multiclass['accuracy']
summary_multiclass['average_C_weights'] = l1_C_weights.values()
summary_multiclass['average_lambda_weights'] = l1_lambda_weights.values()
summary_multiclass['num_features_chosen'] = dict_features_l1_num.values()
summary_multiclass['list_features_chosen'] = dict_features_l1_list.values()
```

In [81]:
```
summary_multiclass
```

Out[81]:

	l_split	accuracy	error_rate	average_C_weights	average_lambda_weights	num_features_choser
0	1	0.652174	0.347826	1670.377218	0.000599	5
1	2	0.826087	0.173913	3340.978559	0.000299	14
2	3	0.652174	0.347826	59.657869	0.016762	16
3	4	1.000000	0.000000	486.643563	0.002055	19
4	5	0.826087	0.173913	271.385302	0.003685	20
5	6	0.652174	0.347826	434.167210	0.002303	51
6	7	1.000000	0.000000	1726.384399	0.000579	26
7	8	0.826087	0.173913	3388.996598	0.000295	287
8	9	0.826087	0.173913	3548.711421	0.000282	10
9	10	0.652174	0.347826	3337.387851	0.000300	30
10	11	0.826087	0.173913	3361.658682	0.000297	29
11	12	0.478261	0.521739	31.856210	0.031391	17
12	13	0.478261	0.521739	28.325382	0.035304	35
13	14	0.652174	0.347826	1670.781081	0.000599	15
14	15	0.652174	0.347826	1722.733811	0.000580	260
15	16	0.478261	0.521739	28.325382	0.035304	42

	l_split	accuracy	error_rate	average_C_weights	average_lambda_weights	num_features_choser
16	17	0.652174	0.347826	298.663338	0.003348	290
17	18	0.478261	0.521739	243.119900	0.004113	13
18	19	0.478261	0.521739	215.378121	0.004643	17
19	20	0.478261	0.521739	31.856210	0.031391	53

In [82]:
```
# Since multiple models achieved an error_rate of 0, I'll choose the l_slit with
idx_best_l = np.argmin(summary_multiclass['error_rate'])
multiclass_chosen_l = summary_multiclass.iloc[idx_best_l]['l_split']
print(f'Will use an l of {multiclass_chosen_l}')
```

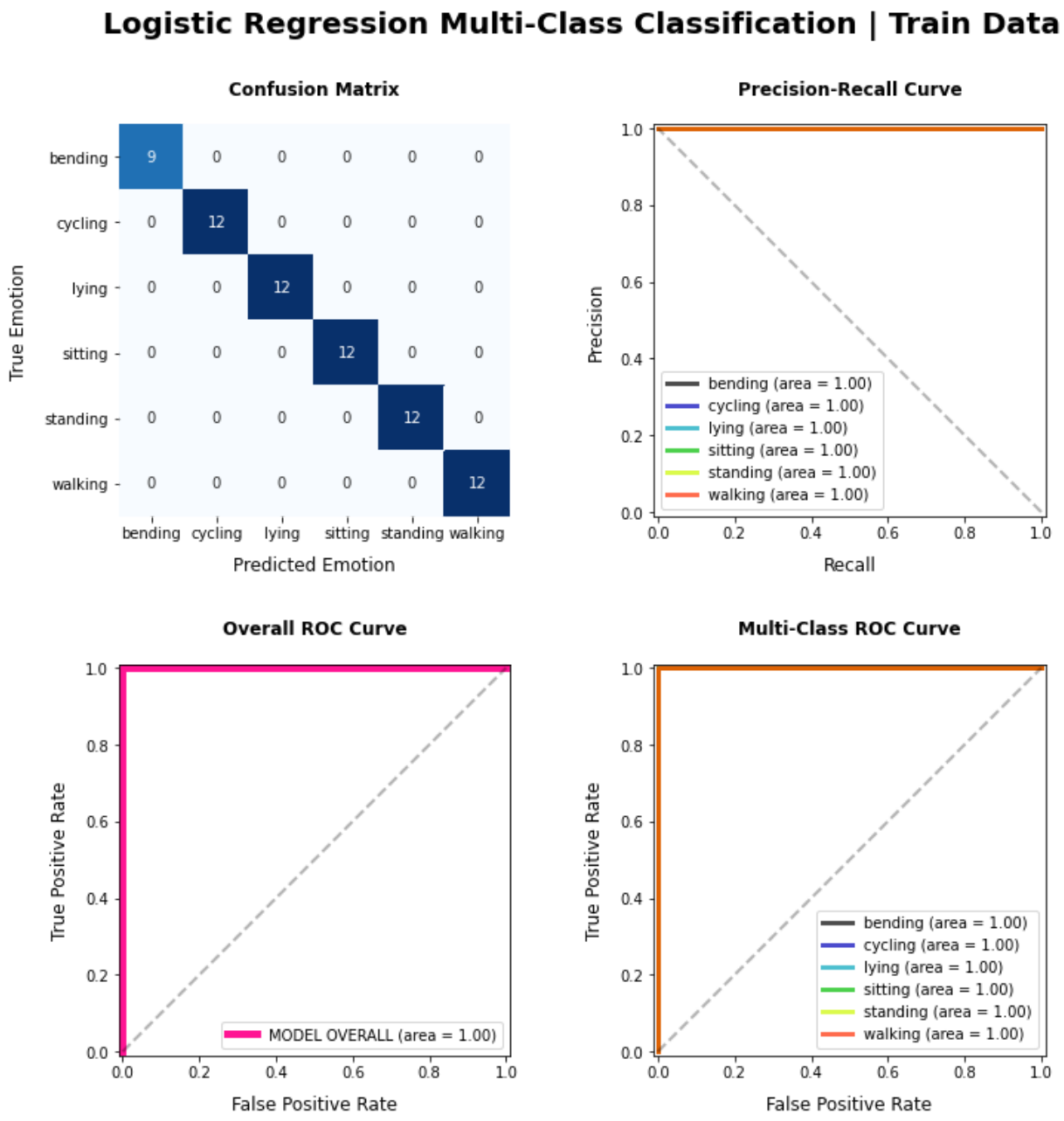
```
Will use an l of 4
```

In [83]:
```
# Now predict with a model using the multiclass_chosen_l

# Split X and Y
x = multiclass_dict_train[multiclass_chosen_l].iloc[:, :-1]
y = multiclass_dict_train[multiclass_chosen_l].iloc[:, -1]

# One-Hot-Encode Y
y = OHE.transform(y.values.reshape(-1, 1))

# Train
ovr_classifier.fit(x, y)

# Predict (also on train dataset)
pred_onehot= ovr_classifier.predict(x)
pred_probs = ovr_classifier.predict_proba(x)

# Set a more obvious name for y
true_onehot = y.copy()

# And finally, retrieve the decoded labels
true_labels = OHE.inverse_transform(true_onehot).flatten()
pred_labels = OHE.inverse_transform(pred_onehot).flatten()

unique_labels = np.unique(true_labels)
```

In [84]:
```
# And get train error
train_acc = ovr_classifier.score(x, y)
train_error = 1 - train_acc
print(f'Train error: {train_error}')
```

```
Train error: 0.0
```

In [85]:
```
plot_classification_results(true_onehot, pred_probs, classes=labels_multiclass_ur
plt.suptitle('Logistic Regression Multi-Class Classification | Train Data', y=1.0
plt.show()
```



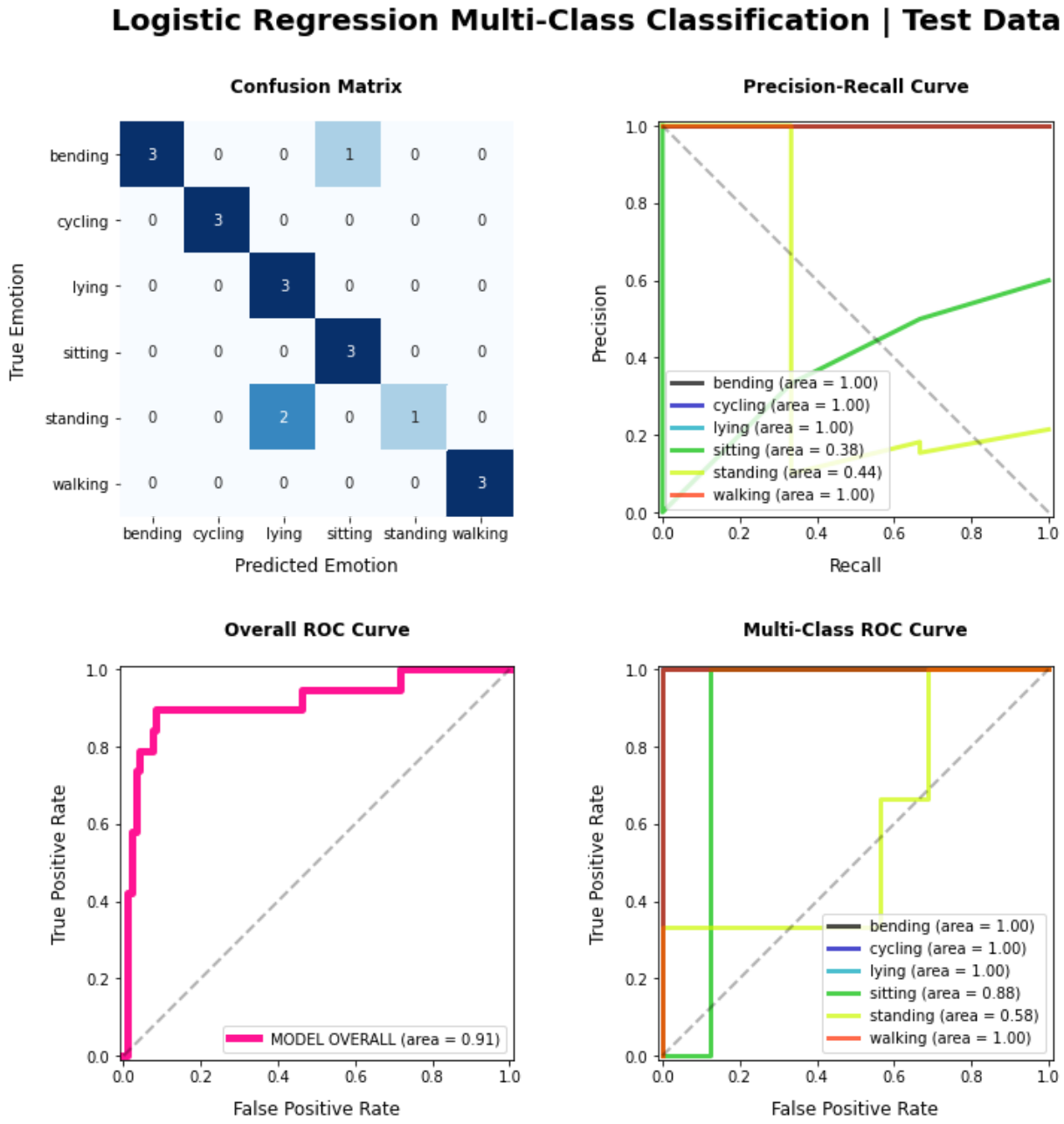
Now repeating everything for the test dataset

In [86]:
```
# Now predict on the test dataset
# Split X and Y
x = multiclass_dict_test[multiclass_chosen_l].iloc[:, :-1]
y = multiclass_dict_test[multiclass_chosen_l].iloc[:, -1]
# One-Hot-Encode Y
y = OHE.transform(y.values.reshape(-1, 1))
# Predict
pred_onehot = ovr_classifier.predict(x)
pred_probs = ovr_classifier.predict_proba(x)
# Set a more obvious name for y
true_onehot = y.copy()
# And finally, retrieve the decoded labels
true_labels = OHE.inverse_transform(true_onehot).flatten()
pred_labels = OHE.inverse_transform(pred_onehot).flatten()
unique_labels = np.unique(true_labels)
```

In [87]:
```
# And get test error
test_acc = ovr_classifier.score(x, y)
test_error = 1 - test_acc
print(f'Test error: {test_error}')
```

```
Test error: 0.368421052631579
```

In [88]:
```
plot_classification_results(true_onehot, pred_probs, classes=labels_multiclass_ur
plt.suptitle('Logistic Regression Multi-Class Classification | Test Data', y=1.05
plt.show()
```



(ii) Naïve Bayes Multiclass Classifier

ii. Repeat 1(f)i using a Naïve Bayes' classifier. Use both Gaussian and Multinomial priors and compare the results.

Gaussian

In [89]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
```

In [90]:

```
%%time

# Instanciate the model
model_GNB = GaussianNB()

# Dicts to store the features and accuracies from each l-sized dataframe
GNB_accuracies_train = {}
GNB_accuracies_test = {}

# Loop through all 'l_splits' in [1:20]
for l_split in l_range:

    # Split X and Y
    x = multiclass_dict_train[l_split].iloc[:, :-1]
    y = multiclass_dict_train[l_split].iloc[:, -1]

    # Cross-Validate and get training error
    GNB_score = cross_val_score(model_GNB, x, y, scoring='accuracy', cv=Stratifie

    # Get the train accuracy
    GNB_accuracies_train[l_split] = GNB_score.mean()

    # To get the test error, need to first need to fit the model on the training
    # cross_val_score does not fit the instantiated model, so the untrained model
    model_GNB.fit(x, y)

    # Change x and y to the test dataset
    x = multiclass_dict_test[l_split].iloc[:, :-1]
    y = multiclass_dict_test[l_split].iloc[:, -1]

    # Get the test accuracy
    GNB_accuracies_test[l_split] = model_GNB.score(x, y)
```

```
Wall time: 2.13 s
```

In [91]:
```
summary_GNB = pd.DataFrame()
summary_GNB['l_split'] = l_range
summary_GNB['accuracy_train_CV'] = GNB_accuracies_train.values()
summary_GNB['error_rate_train_CV'] = 1 - summary_GNB['accuracy_train_CV']
summary_GNB['accuracy_test'] = GNB_accuracies_test.values()
summary_GNB['error_rate_test'] = 1 - summary_GNB['accuracy_test']
summary_GNB
```

Out[91]:

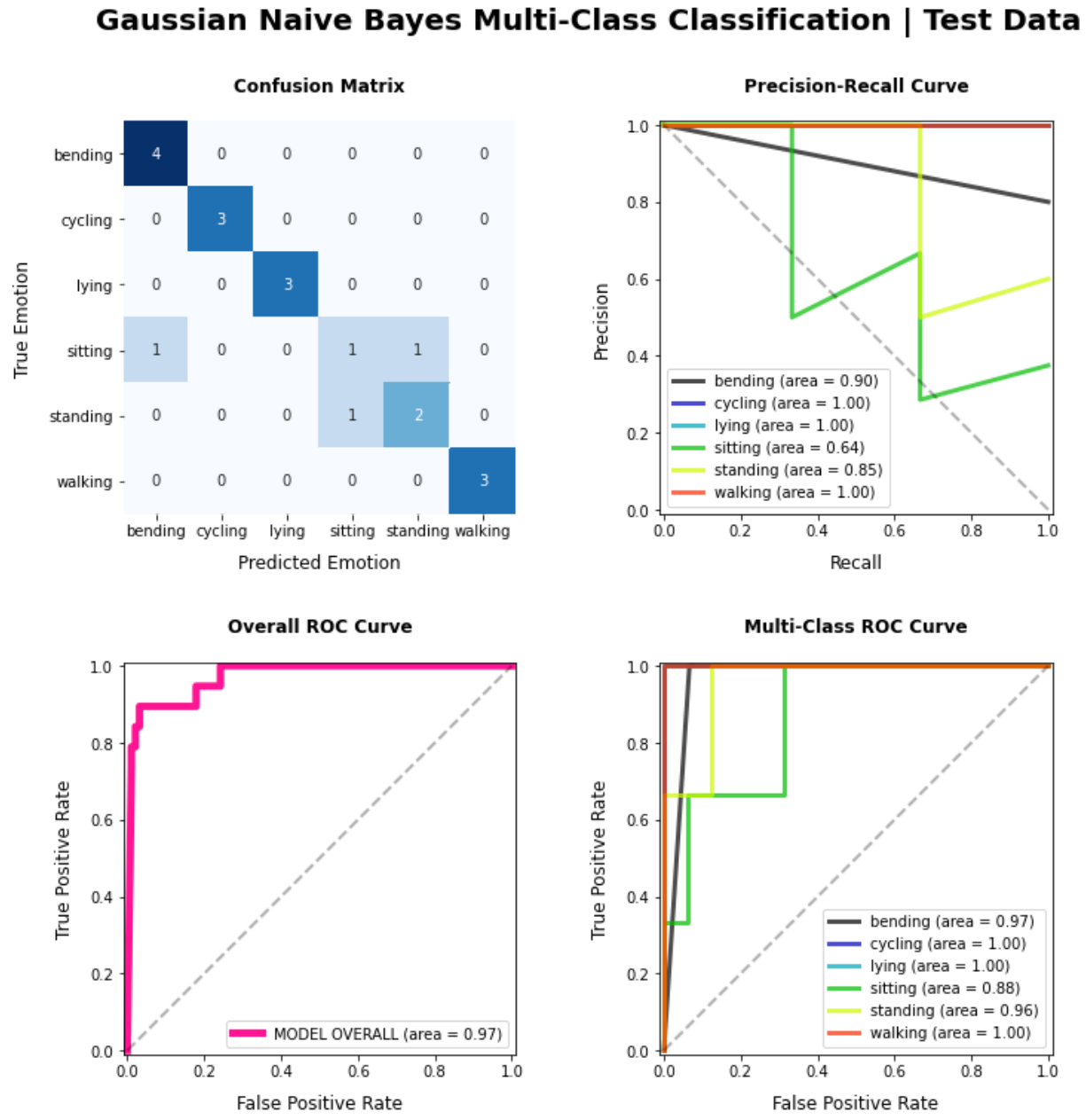
	l_split	accuracy_train_CV	error_rate_train_CV	accuracy_test	error_rate_test
0	1	0.851648	0.148352	0.894737	0.105263
1	2	0.856044	0.143956	0.842105	0.157895
2	3	0.740659	0.259341	0.789474	0.210526
3	4	0.695604	0.304396	0.894737	0.105263
4	5	0.710989	0.289011	0.894737	0.105263
5	6	0.725275	0.274725	0.789474	0.210526
6	7	0.668132	0.331868	0.842105	0.157895
7	8	0.754945	0.245055	0.789474	0.210526
8	9	0.594505	0.405495	0.789474	0.210526
9	10	0.623077	0.376923	0.842105	0.157895
10	11	0.651648	0.348352	0.736842	0.263158
11	12	0.612088	0.387912	0.736842	0.263158
12	13	0.696703	0.303297	0.789474	0.210526
13	14	0.652747	0.347253	0.684211	0.315789
14	15	0.608791	0.391209	0.578947	0.421053
15	16	0.623077	0.376923	0.526316	0.473684
16	17	0.551648	0.448352	0.526316	0.473684
17	18	0.594505	0.405495	0.684211	0.315789
18	19	0.579121	0.420879	0.473684	0.526316
19	20	0.594505	0.405495	0.578947	0.421053

In [92]:
```
# Pick the l-split with lowest train error (because using the test dataset would
GNB_chosen_l_idx = np.argmin(summary_GNB['error_rate_train_CV'])
GNB_chosen_l = int(summary_GNB.iloc[GNB_chosen_l_idx]['l_split'])
GNB_chosen_l
```

Out[92]: 2

In [93]:
```
# With the chosen l, train the model and make predictions to plot ROC-AUC and Cor

# Split X and Y
x = multiclass_dict_train[GNB_chosen_l].iloc[:, :-1]
y = multiclass_dict_train[GNB_chosen_l].iloc[:, -1]
# Train
model_GNB.fit(x, y)
# Change x and y to the test dataset
x = multiclass_dict_test[GNB_chosen_l].iloc[:, :-1]
y = multiclass_dict_test[GNB_chosen_l].iloc[:, -1]
# Get the test accuracy
pred_probs = model_GNB.predict_proba(x)
pred_labels = model_GNB.predict(x)
# Convert y to the necessary formats to plot
true_onehot = OHE.transform(y.values.reshape(-1, 1))
true_labels = y.copy()
unique_labels = np.unique(true_labels)
```

In [94]:
```
plot_classification_results(true_onehot, pred_probs, classes=labels_multiclass_ur
plt.suptitle('Gaussian Naive Bayes Multi-Class Classification | Test Data', y=1.0
plt.show()
```



In [95]:
```
print(f'The selected Gaussian Naïve Bayes model during training achieved a test e
```

```
The selected Gaussian Naïve Bayes model during training achieved a test error r
ate of: 0.1578947368421053
```

It's important to notice that well testing all possible l_splits, the lowest test error was actually achieved by models which hadn't had the lowest training error. That is why the chosen model, which was chosen based on training data, did not have the best performance on testing data.

Multinomial

In [96]:
```
from sklearn.naive_bayes import MultinomialNB
```

In [97]:
```
%%time

# Instantiate the model
model_MNB = MultinomialNB()

# Dicts to store the features and accuracies from each l-sized dataframe
MNB_accuracies_train = {}
MNB_accuracies_test = {}

# Loop through all 'l_splits' in [1:20]
for l_split in l_range:

    # Split X and Y
    x = multiclass_dict_train[l_split].iloc[:, :-1]
    y = multiclass_dict_train[l_split].iloc[:, -1]

    # Cross-Validate
    MNB_score = cross_val_score(model_MNB, x, y, scoring='accuracy', cv=Stratifie

    # Get the train accuracy
    MNB_accuracies_train[l_split] = MNB_score

    # To get the test error, need to first need to fit the model on the training
    # cross_val_score does not fit the instantiated model, so the untrained model
    model_MNB.fit(x, y)

    # Change x and y to the test dataset
    x = multiclass_dict_test[l_split].iloc[:, :-1]
    y = multiclass_dict_test[l_split].iloc[:, -1]

    # Get the test accuracy
    MNB_accuracies_test[l_split] = model_MNB.score(x, y)
```

```
Wall time: 1.58 s
```

In [98]:
```
summary_MNB = pd.DataFrame()
summary_MNB['l_split'] = l_range
summary_MNB['accuracy_train_CV'] = MNB_accuracies_train.values()
summary_MNB['error_rate_train_CV'] = 1 - summary_MNB['accuracy_train_CV']
summary_MNB['accuracy_test'] = MNB_accuracies_test.values()
summary_MNB['error_rate_test'] = 1 - summary_MNB['accuracy_test']
summary_MNB
```

Out[98]:

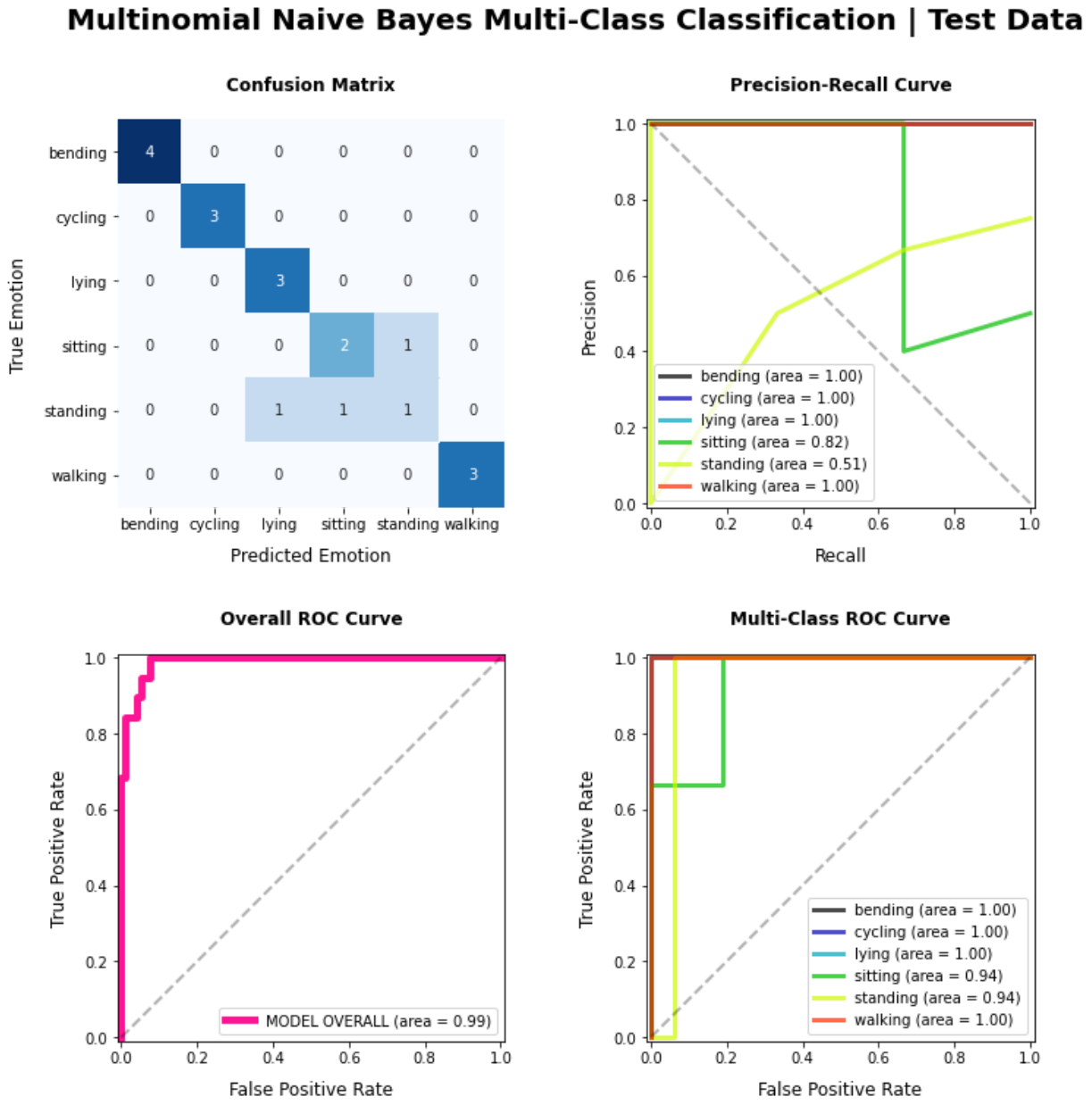
	l_split	accuracy_train_CV	error_rate_train_CV	accuracy_test	error_rate_test
0	1	0.810989	0.189011	0.894737	0.105263
1	2	0.812088	0.187912	0.842105	0.157895
2	3	0.796703	0.203297	0.842105	0.157895
3	4	0.812088	0.187912	0.842105	0.157895
4	5	0.840659	0.159341	0.842105	0.157895
5	6	0.840659	0.159341	0.842105	0.157895
6	7	0.854945	0.145055	0.842105	0.157895
7	8	0.840659	0.159341	0.842105	0.157895
8	9	0.840659	0.159341	0.842105	0.157895
9	10	0.840659	0.159341	0.842105	0.157895
10	11	0.840659	0.159341	0.842105	0.157895
11	12	0.810989	0.189011	0.842105	0.157895
12	13	0.840659	0.159341	0.894737	0.105263
13	14	0.825275	0.174725	0.894737	0.105263
14	15	0.825275	0.174725	0.894737	0.105263
15	16	0.839560	0.160440	0.894737	0.105263
16	17	0.810989	0.189011	0.947368	0.052632
17	18	0.810989	0.189011	0.947368	0.052632
18	19	0.810989	0.189011	0.947368	0.052632
19	20	0.810989	0.189011	0.947368	0.052632

In [99]:
```
# Pick the l-split with lowest train error (because using the test dataset would
MNB_chosen_l_idx = np.argmin(summary_MNB['error_rate_train_CV'])
MNB_chosen_l = int(summary_MNB.iloc[MNB_chosen_l_idx]['l_split'])
MNB_chosen_l
```

Out[99]: 7

In [100]:
```
# With the chosen l, train the model and make predictions to plot ROC-AUC and Cor

# Split X and Y
x = multiclass_dict_train[MNB_chosen_l].iloc[:, :-1]
y = multiclass_dict_train[MNB_chosen_l].iloc[:, -1]
# Train
model_MNB.fit(x, y)
# Change x and y to the test dataset
x = multiclass_dict_test[MNB_chosen_l].iloc[:, :-1]
y = multiclass_dict_test[MNB_chosen_l].iloc[:, -1]
# Get the test accuracy
pred_probs = model_MNB.predict_proba(x)
pred_labels = model_MNB.predict(x)
# Convert y to the necessary formats to plot
true_onehot = OHE.transform(y.values.reshape(-1, 1))
true_labels = y.copy()
unique_labels = np.unique(true_labels)
```

In [101]:
```
plot_classification_results(true_onehot, pred_probs, classes=labels_multiclass_ur
plt.suptitle('Multinomial Naive Bayes Multi-Class Classification | Test Data', y=
plt.show()
```



In [102]:
```
print(f'The selected Multinomial Naïve Bayes model during training achieved a tes
```

```
The selected Multinomial Naïve Bayes model during training achieved a test erro
r rate of: 0.1578947368421053
```

It's important to notice that when testing all possible l_splits, the lowest test error was actually achieved by models which hadn't had the lowest training error. That is why the chosen model, which was chosen based on training data, did not have the best performance on testing data.

(iii) Best Performing Naïve Bayes Method

iii. Which method is better for multi-class classification in this problem?

Both models performed similarly, with both achieving equal error rate on the test dataset. Yet, when considering the roc-auc-curve, which account for the assigned probabilities, the Multinomial Naïve Bayes model performed slightly better.

(2) ISLR 3.7.4

4. I collect a set of data ($n = 100$ observations) containing a single predictor and a quantitative response. I then fit a linear regression model to the data, as well as a separate cubic regression, i.e. $Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \epsilon$.

(a) Linear vs. Cubic Regression on Train Dataset

(a) Suppose that the true relationship between X and Y is linear, i.e. $Y = \beta_0 + \beta_1 X + \epsilon$. Consider the training residual sum of squares (RSS) for the linear regression, and also the training RSS for the cubic regression. Would we expect one to be lower than the other, would we expect them to be the same, or is there not enough information to tell? Justify your answer.

We would expect the cubic regression to perform better, obtaining a lower RSS, since its just an extension of the linear model, but with more flexibility to fit to the data. The best case scenario is an extremely unlikely tie, where $\beta_2=0$ and $\beta_3=0$, turning the cubic model into an effective linear one. But most likely, even for a variable whose true relationship is linear, the β_2 and β_3 parameters will be able to fit to some noise and reduce the train RSS.

(b) Linear vs. Cubic Regression on Test Dataset

(b) Answer (a) using test rather than training RSS.

We would expect the linear model do achieve a lower RSS, mostly because the cubic model would have fitted to noise in the train dataset, and would thus have a higher standard deviation when applied to a test dataset.

(c) Non-Linear Relationship Between X and Y

(c) Suppose that the true relationship between X and Y is not linear, but we don't know how far it is from linear. Consider the training RSS for the linear regression, and also the training RSS for the cubic regression. Would we expect one to be lower than the other, would we expect them to be the same, or is there not enough information to tell? Justify your answer.

Following the same of logic of (a), the cubic regression simply has more flexibility to shape itself to noise in the train dataset, hence regardless of the true relationship between X and Y, for train RSS the cubic model will always outperform a linear model, which is nothing but a less flexible version of the cubic model.

(d) Answer (c) using test rather than training RSS

(d) Answer (c) using test rather than training RSS.

For this scenario there is not enough information to tell, as the answer depends on the true relationship between X and Y.

It could be that the model is mostly linear, with only slight deviations. In this case the linear regression would perform better, as due to its inflexibility it would not fit to noise in the training data, and thus would fit rather well on the test dataset using only a straight regression line.

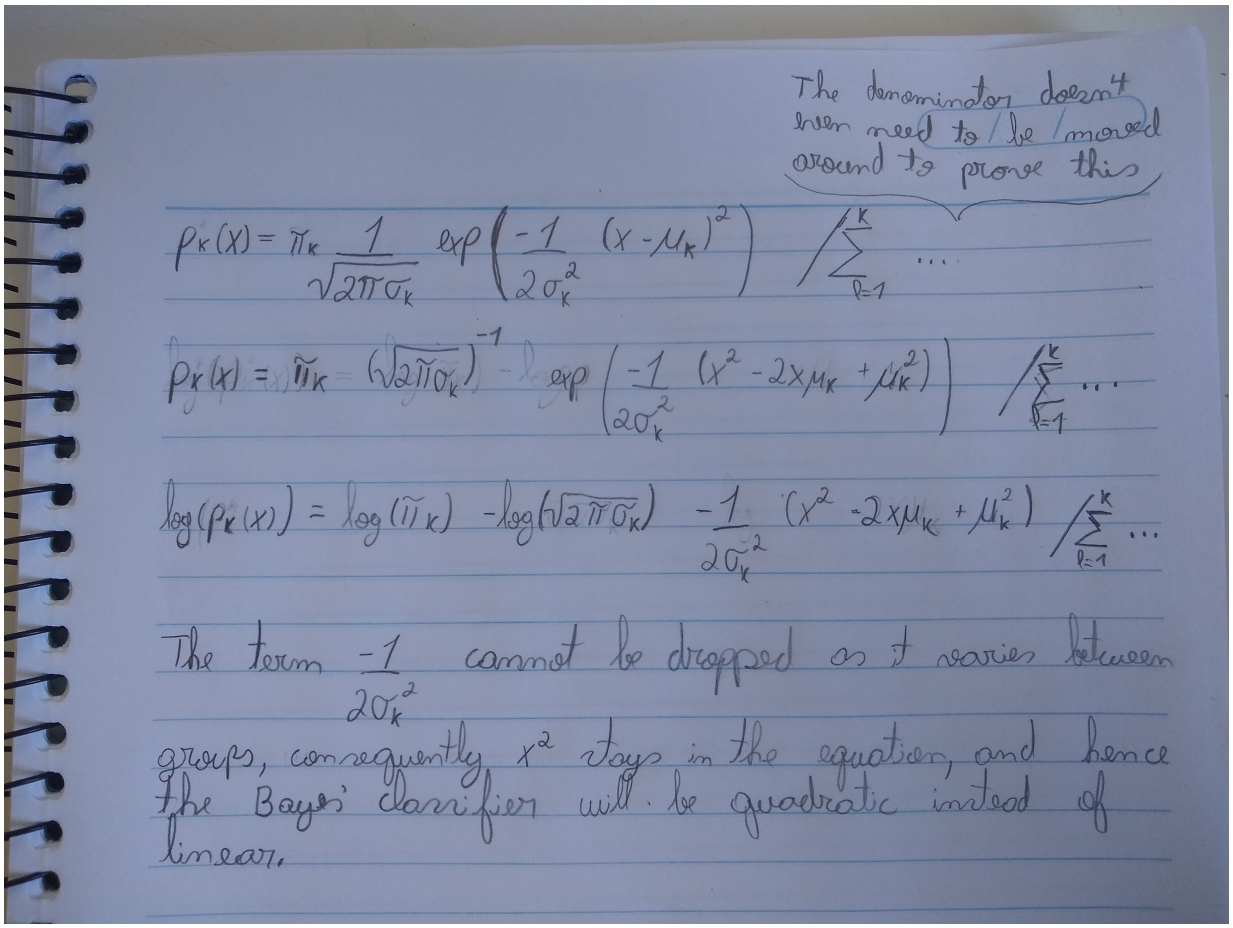
On the other hand, it could be that the variables have indeed a cubic relationship between them, in which case the cubic model would better capture such relationship in the training dataset, and consequently obtain a lower RSS on the test dataset.

(3) ISLR 4.7.3

3. This problem relates to the QDA model, in which the observations within each class are drawn from a normal distribution with a class-specific mean vector and a class specific covariance matrix. We consider the simple case where $p = 1$; i.e. there is only one feature.

   Suppose that we have K classes, and that if an observation belongs to the kth class then X comes from a one-dimensional normal distribution, $X \sim N(\mu_k, \sigma_k^2)$. Recall that the density function for the one-dimensional normal distribution is given in (4.11). Prove that in this case, the Bayes' classifier is *not* linear. Argue that it is in fact quadratic.

   Hint: For this problem, you should follow the arguments laid out in Section 4.4.2, but without making the assumption that $\sigma_1^2 = \ldots = \sigma_K^2$.



If the image doesn't load, it can be found in `./exercices/4.7.3_answer.jpg`

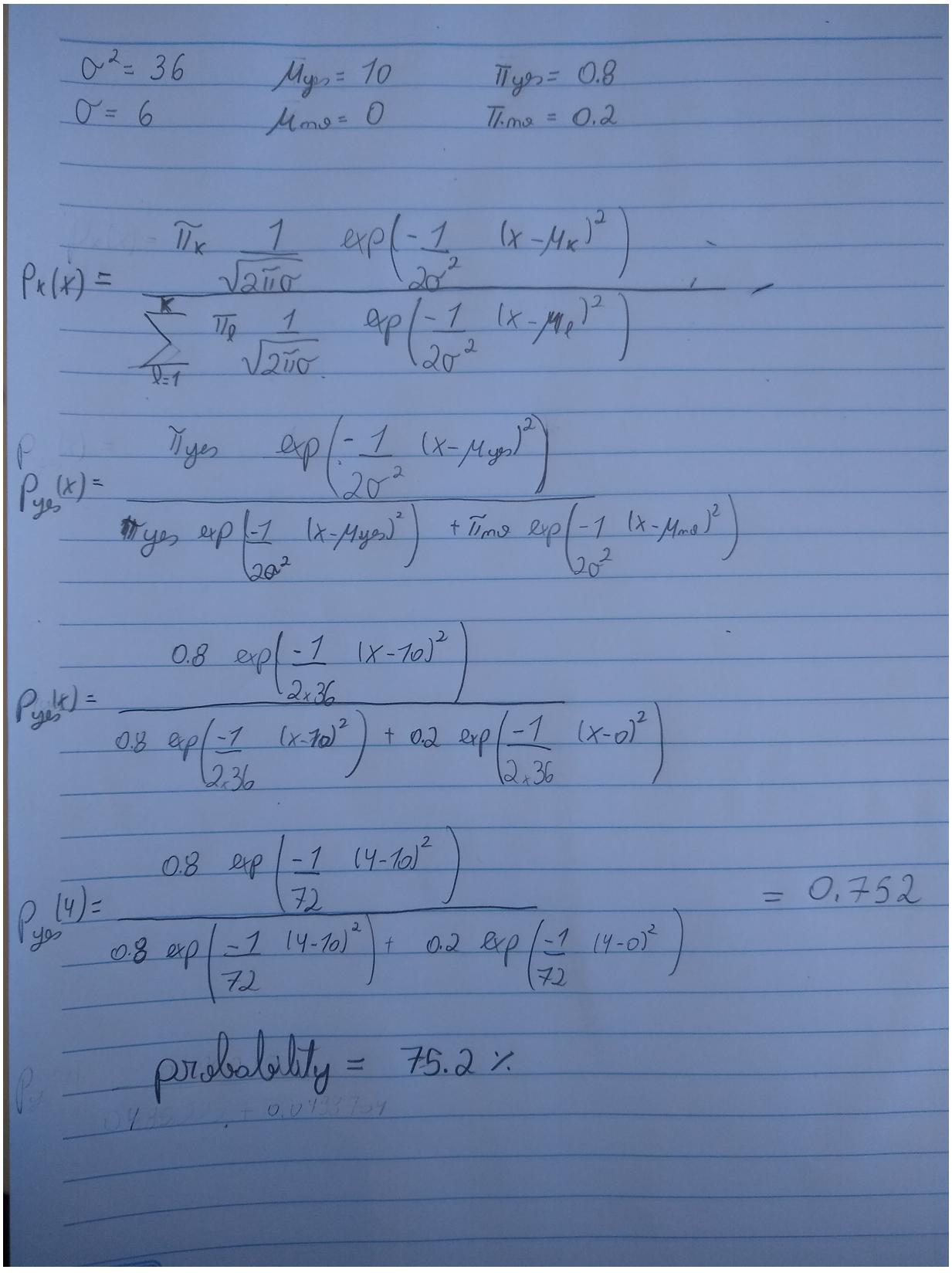
The denominator doesn't even need to be moved around to prove this.

The term `-1/(2 sigmak²)` cannot be dropped as it varies between groups, consequently `x²` stays in the equation, and hence the Bayes' classifier will be quadratic instead of linear.

(4) ISLR 4.7.7

7. Suppose that we wish to predict whether a given stock will issue a dividend this year ("Yes" or "No") based on X, last year's percent profit. We examine a large number of companies and discover that the mean value of X for companies that issued a dividend was $\bar{X} = 10$, while the mean for those that didn't was $\bar{X} = 0$. In addition, the variance of X for these two sets of companies was $\hat{\sigma}^2 = 36$. Finally, 80% of companies issued dividends. Assuming that X follows a normal distribution, predict the probability that a company will issue a dividend this year given that its percentage profit was $X = 4$ last year.

Hint: Recall that the density function for a normal random variable is $f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x-\mu)^2/2\sigma^2}$. *You will need to use Bayes' theorem.*



If the image doesn't load, it can be found in `./exercices/4.7.7_answer.jpg`

The math solution can be checked at: https://www.wolframalpha.com/input/?i=0.8e%5E%28%28-1%2F72%29*6%5E2%29+%2F+%280.8e%5E%28%28-1%2F72%29*6%5E2%29+%2B+0.2e%5E%28%28-1%2F72%29*4%5E2%29%29

(https://www.wolframalpha.com/input/?i=0.8e%5E%28%28-1%2F72%29*6%5E2%29+%2F+%280.8e%5E%28%28-1%2F72%29*6%5E2%29+%2B+0.2e%5E%28%28-1%2F72%29*4%5E2%29%29)

The probability is 75.2%

DSCI 552 - Machine Learning for Data Science

Homework 3

Matheus Schmitz

USC ID: 5039286453