In [1]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as Q
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.ensemble import RandomForestClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC

import seaborn as sns

# Set style for better-looking plots
plt.style.use('seaborn-v0_8')
sns.set_palette("husl")
```

Part 1

Team members:

- Marisol Velapatiño
- Ivo Kusijanovic

Abstract

In this project the main-weld segment of power and force signals for 69 welding processes were analyzed. Highly discriminant features for power signals were identified as risePeak, riseSlope, riseDuration, and dipDepth (in section 3.b). Most of the machine learning process was carried out in section 4. Here, the dataset provided from the Ultrasonic welding process was used; a subset of features was chosen using fischer's ratio and classifiers taught in class were used. Finally, it was found that kNN with k=4 paired with the best 5 features classified the dataset with an accuracy of .91.

Part 2

Some parts of this project require a good understanding of the Ultrasonic welding process. Find one or more research papers that explain this process in detail. Then, write a brief description of the Ultrasonic welding process in a single short paragraph, and ensure you correctly reference your data source(s). No AI summary report. Write your own report

Ultrasonic welding has become important in the advent of technologies where lightweight innovations are rising. These can be related but are not limited to the automobile, aviation, and in general transport industries. Composite joints can be made

up of different materials, plastic, paper, stainless steel, aluminum to produce high-end quality lightweight parts. Ultrasonic welding is an industry preferred process thanks to its fast welding time to connect components into a solid state. The ultrasonic welding process is a projected type technique of binding : The process needs a vertical force that makes contact with the components through a horn. These components are placed in an anvil and oscillet at high frequencies in the kHz. The energy used and other parameters are dependent on the material being welded.

Part 3

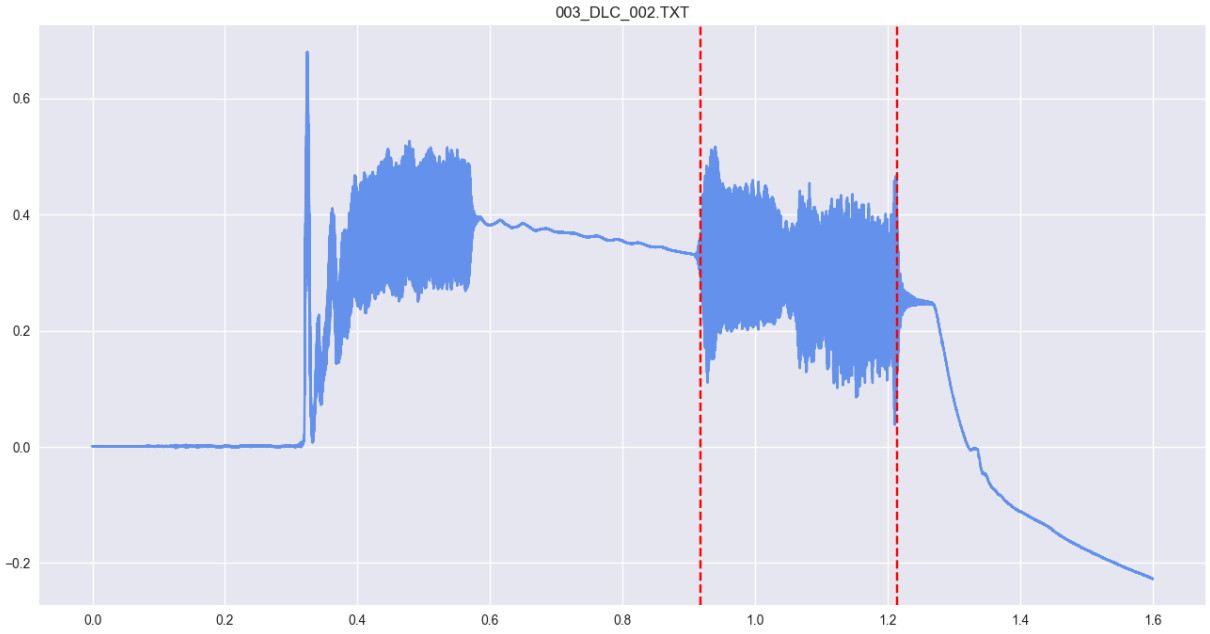
Part 3 Learn to use datasets (30 points): An ultrasonic welding machine was used to perform a set of experiments. A total of sixty-nine welding experiments were conducted across seven pressure levels (psi): 10, 20, 30, 40, 50, 60, and 75. Each condition was repeated ten times. During the welding experiments, two types of signals—power and force—were collected for each sample. Subsequently, a peel test was performed to evaluate the welding quality. Based on the results, the samples were categorized into three classes (as shown in Figure 1): cold, excessive, and good. Dataset: Part3.zip

Part 3.a

Data preprocessing (5 points): As shown in Figures 2 and 3, the power signals consist of a main-weld segment and a post-weld segment, while the force signals include a pre-weld segment, main-weld segment, and post-weld segment.

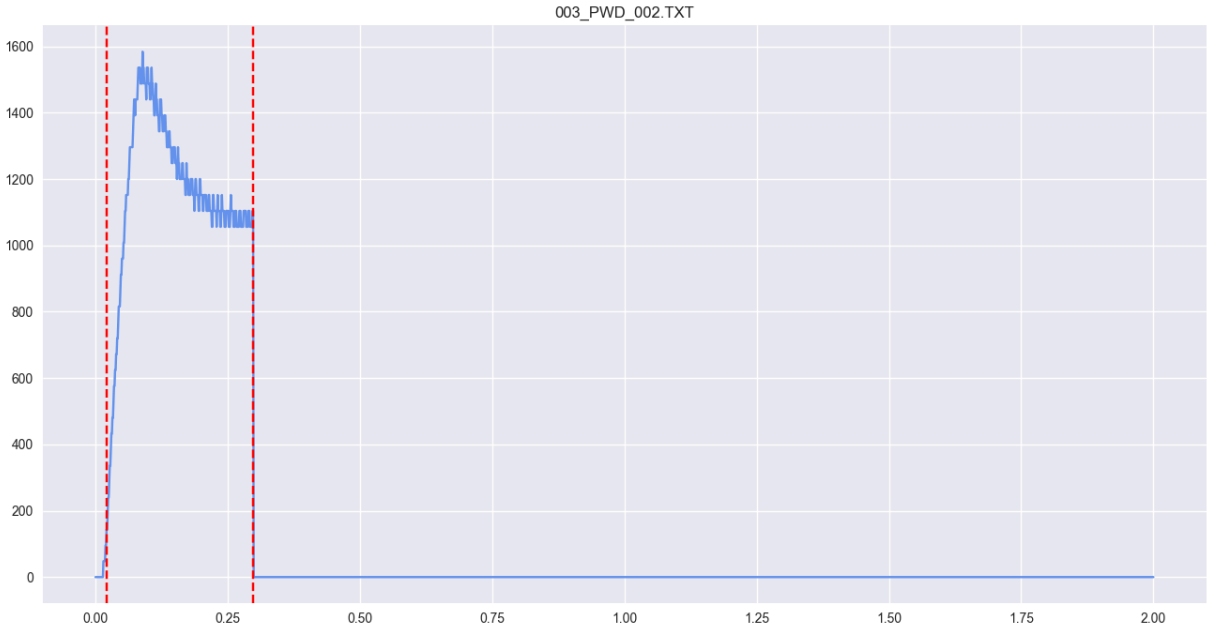
Based on a physical understanding of the welding process, we know that the main-weld segment is the most informative for quality monitoring. Therefore, our analysis will focus solely on this segment. Develop an algorithm to automatically extract the main-weld segment for all welds. In your report, briefly describe the algorithm and demonstrate its effectiveness using one example for each type of signal. Specifically, plot the raw signal and highlight the boundaries of the main-weld segment

In [2]:
```
from part3a import extract_force_signals, extract_force_main_weld_segment

force_signals = extract_force_signals()
experiment, time, force = extract_force_main_weld_segment(*force_signals[0],
```



Extract Force Main Weld Segment: To extract the main weld segment of the force signal consider the following algorithm. Firstly, we start inspecting the signal after the initial power-up noise, seen from 0.3s to 0.6s in the above figure. We denote noise as the absolute difference in force between two consecutive data points. In our case we start at around index 80000, which worked well for all plots. Its a matter of playing around with this initial index if new data that doesn't fit our algorithm is observed. Now, we see that after the initial power-up force decereases stably until a sudden increase in noise. This is what we denote as the start of the main weld segment. To improve consistency we'll use an average window to quantify this increase in noise from the previous window. Conversely, the end of the main weld segment is denoted by a sudden decrease in noise. Using these ideas we create our algorithm.

In [3]:
```
from part3a import extract_power_signals, extract_power_main_weld_segment

power_signals = extract_power_signals()
experiment, time, power = extract_power_main_weld_segment(*power_signals[0],
```



Extract Power Main Weld Segment: To extract the main weld segment of the power signal consider the following algorithm. The main weld segment starts when we see a sudden increase in power. Conversely,the main weld segment ends when we see a sudden decrease in power. Using these ideas we create our algorithm.

Part 3.b

Feature generation (8 points): Create features following the steps below and record all features in a .csv file (part3b.csv).

Each row corresponds to one experiment and each column corresponds to one parameter/feature.

The columns should follow this order: experiment number, quality label, four features from group a, four features from group b, and four features from group c. This table should be included in your report.

- Process output signals **-Group A-** (2 points): Welding pressure, pre-height, height, and height change. The first three were recorded by the welder. Height change is the difference between pre-height and the height.
- Time-domain features from power signals **-Group B-** (3 points): Brainstorm and generate four features. In the report, provide the definitions of all features and a justification of why these features may be helpful.

  We'll be choosing highly discriminative feature that allows us to classify good welds from bad welds:

- **risePeak** : Shows the maximum power reached during the rise and indicates whether target welding power is achieved. If in the expected range, the risePeak indicates that thermal energy coupled efficiently into the metal which likely produces a good weld. Else if it's too low, the risePeak indicates insufficient heat delivery, poor current conduction (due to large surface impurity) and thus a cold weld. If it's too high, the risePeak corresponds to an unstable heat delivery which highly leads to excessive penetration and an excessive weld.
- **riseSlope**: This measures how quickly the system reaches operating power. Usually unstable welds have slower rise slopes. A fast rise indicates proper energy coupling: basically how efficiently the electrical power is delivered by the welding machine into heat energy transferred to the joint. A slow or noisy rise means the connection struggles to form and the power is lost to poor contact or instability.
- **riseDuration**: Indicates the time needed to reach the peak and is complementary to the riseSlope. As explained earlier, a longer rise duration generally correlate with poor contact and energy coupling which can cause a bad weld.
- **dipDepth**: This is the change in height between the risePeak and the ramp/stable power signal. A deep dip after the risePeak likely means that the controller had to pull back to correct and overshoot, indicating a poor heat transfer and bad weld. Conforming welds usually have shallow dip, indicating efficient heat transfer.

- Frequency-domain features from force signals **-Group C-** (3 points): Generate the following features: 1st peak frequency and magnitude, 2nd peak frequency and magnitude.

In [4]:
```
from part3b import extract_features_group_b, extract_features_group_c

name = "part3b.csv"
col_names = ["experiment number", "quality label", "Welding Pressure", "pre-
file  = './data/part3/WeldClassification.xls'

df_raw = pd.read_excel(file, skiprows=6)

zero_data = np.zeros(shape=(len(df_raw['Run No.']),len(col_names)))

df = pd.DataFrame(zero_data, columns=col_names)

## Group A

df[col_names[0]] = df_raw['Run No.']
df[col_names[1]] = df_raw['Class Label']
df[col_names[2]] = df_raw['Pressure [psi]']
df[col_names[3]] = df_raw['PreHeight [mm]']
df[col_names[4]] = df_raw['Height [mm]']
```

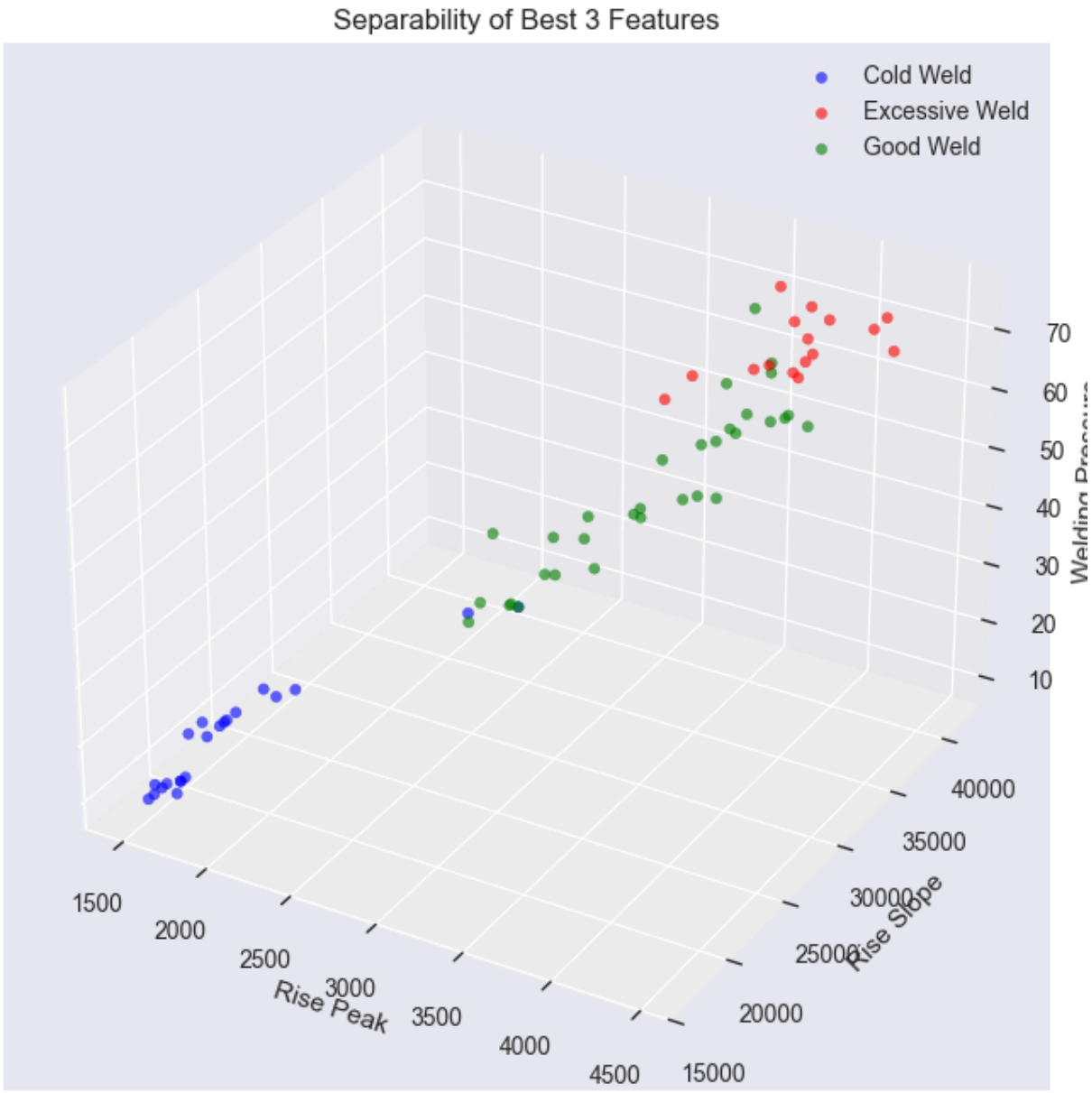
```
df[col_names[5]] = abs(df_raw['Height [mm]'] - df_raw['PreHeight [mm]'])

## Group B

rise_peaks = []
rise_slopes = []
rise_durations = []
dip_depths = []

for power_signal in power_signals:

    experiment, time, power = extract_power_main_weld_segment(*power_signal)
    rise_peak, rise_slope, rise_duration, dip_depth = extract_features_group

    rise_peaks.append(rise_peak)
    rise_slopes.append(rise_slope)
    rise_durations.append(rise_duration)
    dip_depths.append(dip_depth)

df[col_names[6]] = np.array(rise_peaks)
df[col_names[7]] = np.array(rise_slopes)
df[col_names[8]] = np.array(rise_durations)
df[col_names[9]] = np.array(dip_depths)

## Group C

first_peak_freqs = []
first_peak_magns = []
second_peak_freqs = []
second_peak_magns = []

for force_signal in force_signals:

    experiment, time, force = extract_force_main_weld_segment(*force_signal)
    first_peak_freq, first_peak_magn, second_peak_freq, second_peak_magn = e

    first_peak_freqs.append(first_peak_freq)
    first_peak_magns.append(first_peak_magn)
    second_peak_freqs.append(second_peak_freq)
    second_peak_magns.append(second_peak_magn)

df[col_names[10]] = np.array(first_peak_freqs)
df[col_names[11]] = np.array(first_peak_magns)
df[col_names[12]] = np.array(second_peak_freqs)
df[col_names[13]] = np.array(second_peak_magns)
```

Part 3.c

In [5]:
```
print(df.head())
```

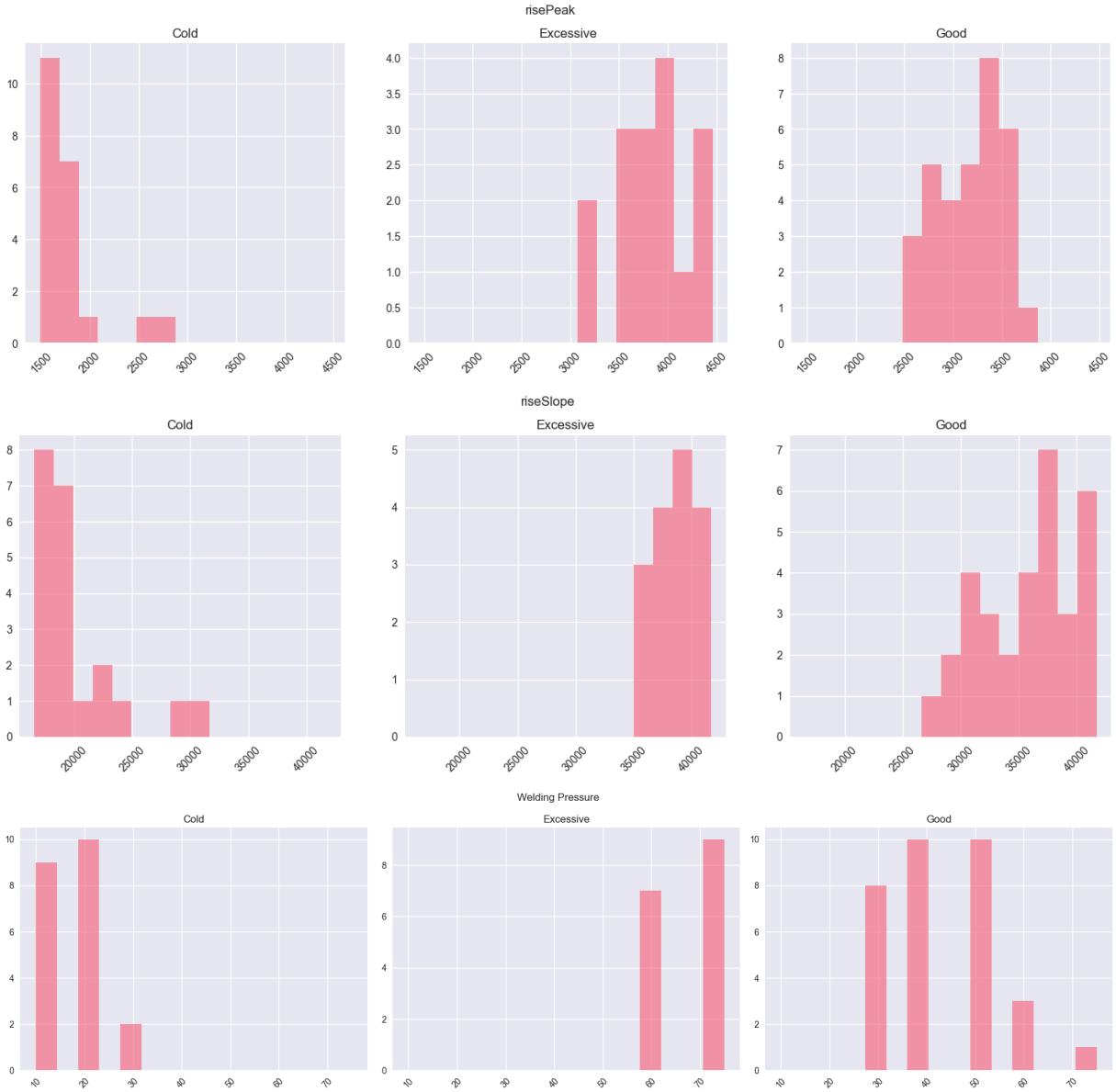
```
   experiment number quality label  Welding Pressure  pre-height  height  \
0                   2             I                10        0.81    0.67
1                   3             I                10        0.83    0.67
2                   4             I                10        0.78    0.63
3                   5             I                10        0.81    0.67
4                   6             I                10        0.83    0.67

   height change  risePeak     riseSlope  riseDuration  dipDepth  \
0           0.14    1536.0  16571.406571         0.062     480.0
1           0.16    1584.0  18461.516462         0.056     432.0
2           0.15    1584.0  18835.423038         0.059     432.0
3           0.14    1488.0  17684.188526         0.054     336.0
4           0.16    1632.0  17560.953610         0.060     480.0

      1peakFreq  1peakMag     2peakFreq  2peakMag
0  19982.462649  0.354729      3.372568  0.153667
1  19989.189554  0.460099  19982.433026  0.152259
2  19982.462649  0.332641  39961.552730  0.165087
3  19975.635342  0.257304  19968.867382  0.189084
4  19948.823272  0.210781      3.366890  0.132414
```

In [6]:
```
from part3c import plot_best_features

best_features = plot_best_features(df)
```



Part 3.d

In [7]:
```
from part3d import plot_class_feature_histograms, plot_shewart_control_chart

plot_class_feature_histograms(df, best_features)
```



The classifier revolves around using a Shewhart Control Chart on good-labeled welds. Then, any data point falling outside the lower and upper control limits would be classified as bad (i.e. cold or excessive).

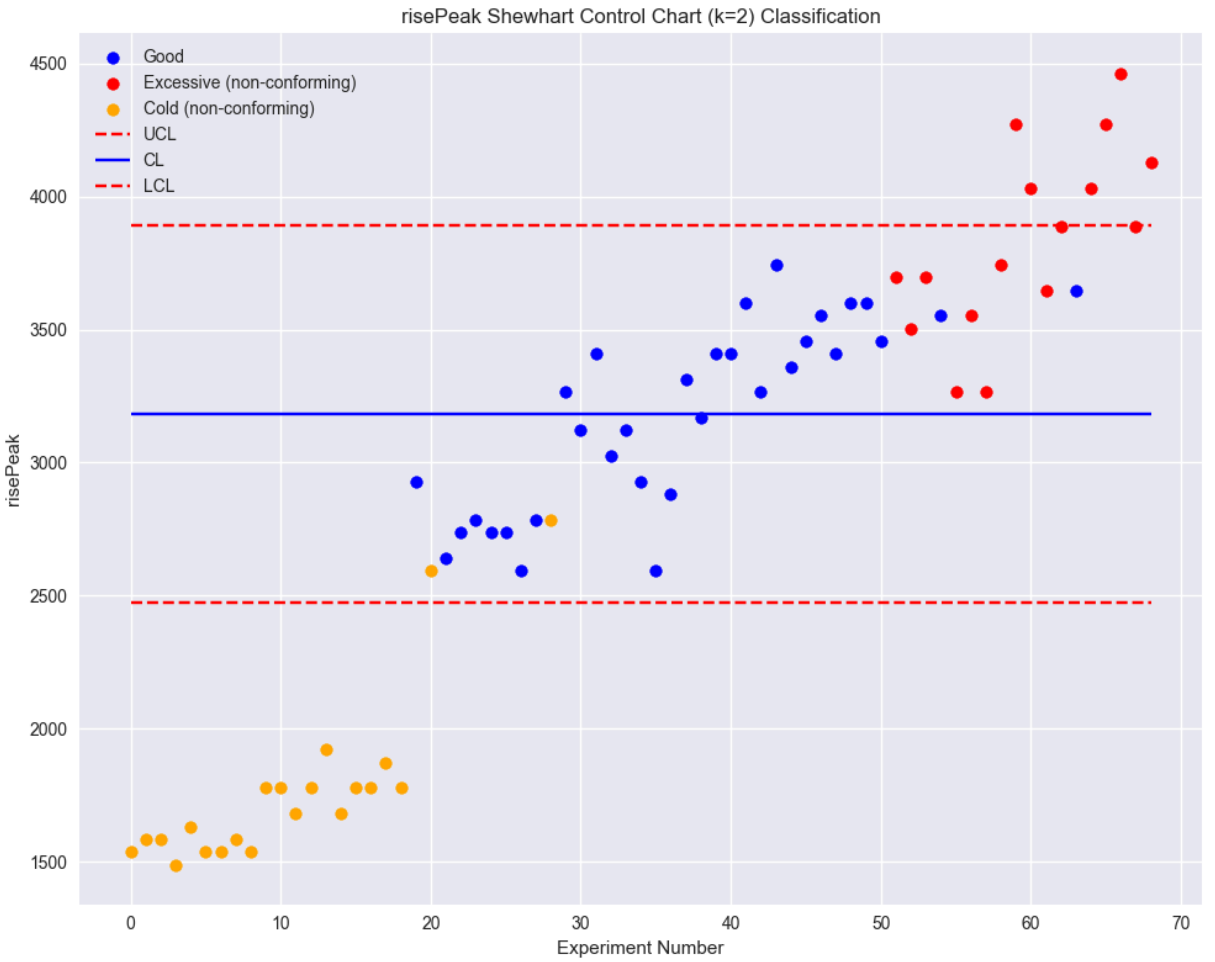
Now the hyper-parameter that we can tune is **k**, the number of standard deviations away from the center line. Although we typically use k=3 as it would enclose around 99.73% of the data, this would likley cause us to underfit our data, causing us to likley classify bad welds as good. Conversely, we can't set k to just fit our train data (i.e. small k) as this would lead to overfitting, causing us to likley classify good welds as bad. Thus, we have to find a balance.

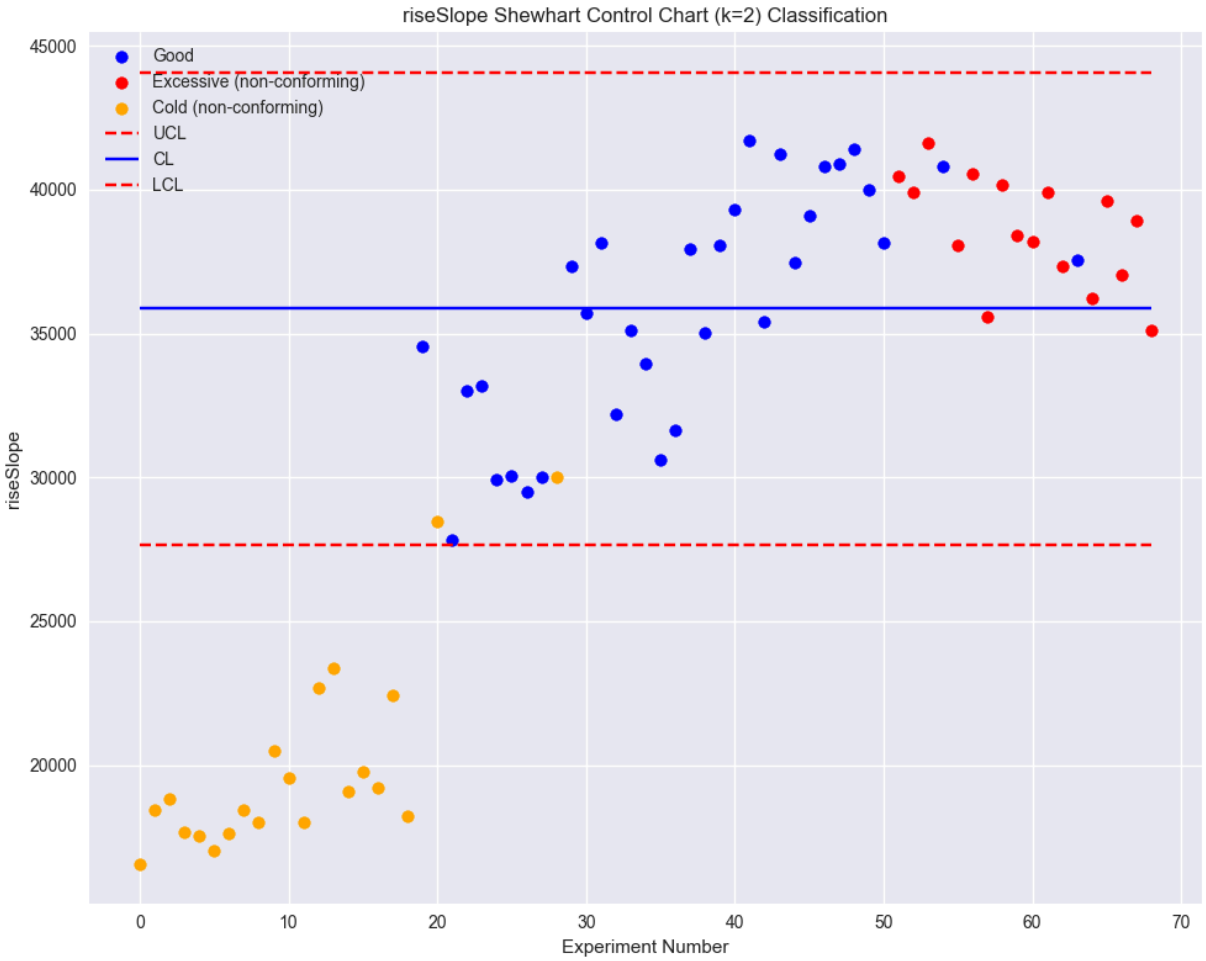
Exploring different ks we saw that setting k=2 provides a sufficiently good classification boundary as to neither overfit or underfit. To provide some more insight, we changed k until there was a decently big margin between the largest good-labeled value and the UCL, and the smallest good-labeled value and the LCL. Moreover, this wasn't the only

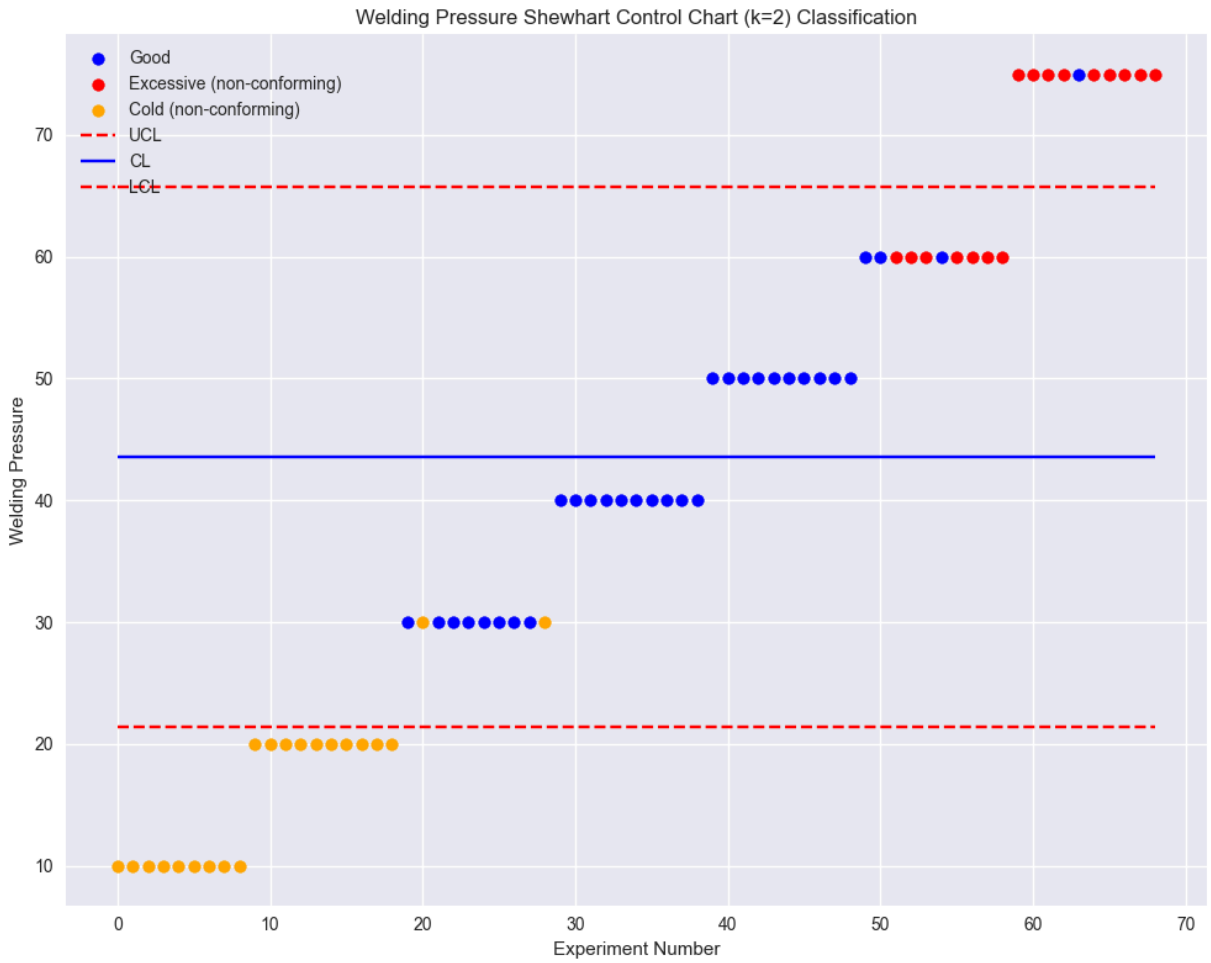
condition as we ignored outlier good-labeled values as seen for the Welding Pressure Shewhart Control Chart classifier.

Just as an aside we more commonly label excessive welds as good in comparison to cold welds, especially for the riseSlope feature where we misclassified all execssive welds as good. Nevertheless, as seen in part (c) the combination of these features in three dimensional space does provide clear separability between the independent classes of cold, good, and excessive, not just good and bad. Thus, it's best to use them in unison.

In [8]:
```
control_limits = plot_shewart_control_chart(df, best_features, k=2)
```




riseSlope Shewhart Control Chart (k=2) Classification
Good
Excessive (non-conforming)
Cold (non-conforming)
UCL
CL
LCL
riseSlope
Experiment Number
45000
40000
35000
30000
25000
20000
0
10
20
30
40
50
60
70



In [9]:
```
misclassification_rates = get_misclassification_rates(df, best_features, cor

for feature, misclassification_rate in zip(best_features, misclassification_
    print(f"{feature} misclassification rate: {misclassification_rate * 100:
```

```
risePeak misclassification rate: 17.39%
riseSlope misclassification rate: 26.09%
Welding Pressure misclassification rate: 14.49%
```

Part 3.e

Based on the analysis above the **power sensor** is more useful for quality monitoring. Firstly, based on the fischer ratio, two of the features extracted from the power signals (risePeak, and riseSlope) provide high inter-class variance. We can clearly see this class seprability, not just from conforming vs non-conforming parts. Secondly, using a simple control limit (k=2) classifier we are able to get a fairly low misclassification rate (17.39%, and 26.09%) which is far better than guessing if the part was conforming or not.

Furthermore, we can extend our analysis by calculating the misclassification rate of the features extracted from the force signal. We find that 3 out of 4 of these misclassification rates hover above 50%, which is worse than just guessing.

Part 4

For this part, use the dataset provided from an Ultrasonic welding process. For your convenience, we have extracted 12 features, which are available in the attached part4.csv file. To avoid any bias from the results of the previous part and to facilitate a data-driven analysis, all features have been normalized and anonymized. This processing will not affect the classification performance.

In [10]:
```
data4 = "./data/part4.csv"
```

Part 4.a

Feature selection (10 points): Select a subset of features to use in your classifiers. Describe the procedure or method you use and explain your reasoning. Note: you may select different subsets of features for each classifier :

Fisher's ratio approach :

We'll use the fisher's ratio to rank fatures based on how well they discriminate Good-Cold, Good-Excessive welds. Similar to the part 3 approach, we'll select the adequate number features from the feature pool based on the following criterion: max (Fisher's ratio 1 + Fisher's ratio 2).

In [11]:
```
from part4a import get_best_features

df_data4 = pd.read_csv(data4)
best_features = get_best_features(df_data4)
```

```
Cumulative sum of ratios: [np.float64(0.15544411626272514), np.float64(0.302
8376106256886), np.float64(0.4489712240961773), np.float64(0.594048847388877
6), np.float64(0.7314874468175232)]
Best 5 features are: Index(['Feature 5', 'Feature 8', 'Feature 9', 'Feature
1', 'Feature 7'], dtype='object')
```

We set an 80% threshold for the cumulative sum of the fisher ratios. This way we capture most of the discriminatory power using a minimum number of features. We get 5 features: 5, 8, 9, 1, 7 in decreasing order of ratio.

Part 4.b

Classification (15 points): Choose three different classifiers (either covered in class or otherwise). Train and test the performance of each model. Use a suitable method or technique to split the dataset into training and testing subsets. Explain your choice of method or technique.

We used K-Fold Cross-Validation to split the dataset as it provides a more reliable estimate of the model performance. This is because we don't rely on a single train-test split to measure the model's performance, rather we average the performance across K distinct train-test splits. This way we reduce randomness and don't worry about the model overperfroming or underperforming from its true ability due to a lucky or unlucky split. Furthermore, using K-Fold Cross Validation we guarantee that almost all (since we use shuffle=True) data points are used in the test set. This way we are more sure that the model's performance doesn't rely on a few data point being present in the train set. This is unlike calling train_test_split K times which doesn't gurantee any of this behavior.

In [12]:
```
from part4b import classify

X = df_data4.iloc[:, 2:]
y = df_data4.iloc[:, 1]

best_3_features = best_features[:3]
```

LDA

5 best features

In [13]:
```
lda = LDA()
_, confusion_matrices_lda = classify(X[best_features], y, lda, "Linear Discr
```

```
Linear Discriminant Analysis Train Accuracy 0.9130434782608695
Average Linear Discriminant Analysis Validation Accuracy: 0.8967032967032967
```

3 best features

In [14]:
```
lda = LDA()
_, best_3_confusion_matrices_lda = classify(X[best_3_features], y, lda, "Lir
```

```
Linear Discriminant Analysis Train Accuracy 0.8695652173913043
Average Linear Discriminant Analysis Validation Accuracy: 0.8538461538461538
```

QDA

5 best features

In [15]:
```
qda = QDA()
_, confusion_matrices_qda = classify(X[best_features], y, qda, "Quadratic Di
```

```
Quadratic Discriminant Analysis Train Accuracy 0.927536231884058
Average Quadratic Discriminant Analysis Validation Accuracy: 0.8417582417582
418
```

3 best features

In [16]:
```
qda = QDA()
_, best_3_confusion_matrices_qda = classify(X[best_3_features], y, qda, "Qua
```

```
Quadratic Discriminant Analysis Train Accuracy 0.8695652173913043
Average Quadratic Discriminant Analysis Validation Accuracy: 0.8681318681318
683
```

SVM with different kernels

5 best features

In [17]:
```
kernels = ["Linear", "RBF", "Poly"]

confusion_matrices_svms = []

for kernel in kernels:

    svm = SVC(kernel=kernel.lower())
    _, confusion_matrices_fold = classify(X[best_features], y, svm, f"{kerne
    confusion_matrices_svms.append(confusion_matrices_fold)
    print()
```

```
Linear SVM Train Accuracy 0.855072463768116
Average Linear SVM Validation Accuracy: 0.853846153846154

RBF SVM Train Accuracy 0.855072463768116
Average RBF SVM Validation Accuracy: 0.8252747252747253

Poly SVM Train Accuracy 0.9130434782608695
Average Poly SVM Validation Accuracy: 0.868131868131868
```

3 best features

In [18]:
```
best_3_confusion_matrices_svms = []

for kernel in kernels:

    svm = SVC(kernel=kernel.lower())
    _, confusion_matrices_fold = classify(X[best_3_features], y, svm, f"{ker
    best_3_confusion_matrices_svms.append(confusion_matrices_fold)
    print()
```

```
Linear SVM Train Accuracy 0.855072463768116
Average Linear SVM Validation Accuracy: 0.8252747252747253

RBF SVM Train Accuracy 0.8695652173913043
Average RBF SVM Validation Accuracy: 0.8252747252747253

Poly SVM Train Accuracy 0.855072463768116
Average Poly SVM Validation Accuracy: 0.8395604395604396
```

KNN with different k's

5 best features

In [19]:
```
ks = [2, 4, 8, 16]

confusion_matrices_knn = []

for k in ks:
    knn = KNeighborsClassifier(n_neighbors=k)
    _, confusion_matrices_fold = classify(X[best_features], y, knn, f"KNN (k
    confusion_matrices_knn.append(confusion_matrices_fold)
    print()
```

```
KNN (k=2) Train Accuracy 0.8985507246376812
Average KNN (k=2) Validation Accuracy: 0.8538461538461538

KNN (k=4) Train Accuracy 0.9130434782608695
Average KNN (k=4) Validation Accuracy: 0.9131868131868133

KNN (k=8) Train Accuracy 0.8840579710144928
Average KNN (k=8) Validation Accuracy: 0.8692307692307691

KNN (k=16) Train Accuracy 0.855072463768116
Average KNN (k=16) Validation Accuracy: 0.853846153846154
```

3 best features

In [20]:
```
ks = [2, 4, 8, 16]

best_3_confusion_matrices_knn = []

for k in ks:
    knn = KNeighborsClassifier(n_neighbors=k)
    _, confusion_matrices_fold = classify(X[best_3_features], y, knn, f"KNN
    best_3_confusion_matrices_knn.append(confusion_matrices_fold)
    print()
```

```
KNN (k=2) Train Accuracy 0.855072463768116
Average KNN (k=2) Validation Accuracy: 0.7670329670329672

KNN (k=4) Train Accuracy 0.8985507246376812
Average KNN (k=4) Validation Accuracy: 0.8692307692307691

KNN (k=8) Train Accuracy 0.8695652173913043
Average KNN (k=8) Validation Accuracy: 0.868131868131868

KNN (k=16) Train Accuracy 0.855072463768116
Average KNN (k=16) Validation Accuracy: 0.853846153846154
```

Random forest with different max depths

5 best features

In [21]:
```
max_depths = [2, 4, 8, 16]

confusion_matrices_random_forest = []
for max_depth in max_depths:
    knn = RandomForestClassifier(max_depth=max_depth, random_state=42)
    _, confusion_matrices_fold = classify(X[best_features], y, knn, f"Random
    confusion_matrices_random_forest.append(confusion_matrices_fold)
    print()
```

```
Random Forest (max_depth=2) Train Accuracy 0.8840579710144928
Average Random Forest (max_depth=2) Validation Accuracy: 0.8538461538461538

Random Forest (max_depth=4) Train Accuracy 0.9420289855072463
Average Random Forest (max_depth=4) Validation Accuracy: 0.8538461538461538

Random Forest (max_depth=8) Train Accuracy 0.9855072463768116
Average Random Forest (max_depth=8) Validation Accuracy: 0.8395604395604396

Random Forest (max_depth=16) Train Accuracy 0.9855072463768116
Average Random Forest (max_depth=16) Validation Accuracy: 0.8395604395604396
```

3 best features

In [22]:
```
max_depths = [2, 4, 8, 16]

best_3_confusion_matrices_random_forest = []
for max_depth in max_depths:
    knn = RandomForestClassifier(max_depth=max_depth, random_state=42)
    _, confusion_matrices_fold = classify(X[best_3_features], y, knn, f"Ranc
    best_3_confusion_matrices_random_forest.append(confusion_matrices_fold)
    print()
```

```
Random Forest (max_depth=2) Train Accuracy 0.8840579710144928
Average Random Forest (max_depth=2) Validation Accuracy: 0.8395604395604396

Random Forest (max_depth=4) Train Accuracy 0.8985507246376812
Average Random Forest (max_depth=4) Validation Accuracy: 0.810989010989011

Random Forest (max_depth=8) Train Accuracy 0.9855072463768116
Average Random Forest (max_depth=8) Validation Accuracy: 0.767032967032967

Random Forest (max_depth=16) Train Accuracy 0.9855072463768116
Average Random Forest (max_depth=16) Validation Accuracy: 0.767032967032967
```

Part 4.c

Performance (10 points): Evaluate the performance of each classifier using confusion matrices and error plots. Based on your analysis, recommend the best combination of feature subset and classifier for this dataset, and justify your recommendation.

In part (b) we trained 5 different classifiers on the best 3 and best 5 features based on the fisher's ratio. We used k-fold cross-validation to have a rough idea of the model's true performance. To enhance out anlysis we'll use confusion matrices C and error plots. Consider the following metric equations

$$\text{Overall Accuracy} = \frac{\sum_i C_{ii}}{\sum_{i,j} C_{ij}}$$

$$\text{Overall Error} = 1 - \text{Overall Accuracy}$$

$$\text{Class Error}_i = 1 - \frac{C_{ii}}{\sum_j C_{ij}}$$

- **LDA analysis**: From the figures we can see that generally the error per class and overall error is higher when using the best 3 features. From the per-class error figure, we see that by choosing the best 3 features the error for folds 2 and 3 in class 2 are significantly higher. The overall error figure reveals that the best 3 features choice increases overall error in fold 2 and 3. From part (b), the average accuracy across folds is higher when using the best 5 features ($\approx .90$).

- **QDA analysis**: From the figures we can see that generally the error per class and overall error is lower when using the best 3 features. In the per-class error figure, the best 3 features option gives higher error for folds 4 and 5 at class 2, and lower error for folds 3 and 2 at class 3. In the overall error figure we see that by choosing 3 best features, the error is lower for folds 2,3,4 but higher for fold 5. From part (b), the average accuracy across folds is higher when using 3 best features ($\approx .87$).

- **SVM with different kernels**:

- From the figures for linear SVM we can see that generally the error per class and overall error are higher for one fold only when using best 3 features. In the per-class error figure, the best 3 features option gives higher error for fold 3 at class 3. In the overall error figure we see that by choosing 3 best features, the error is higher at fold 3 only. From part (b) the average accuracy is higher when using 5 best features ($\approx .85$) .

- From the figures for rbf SVM we can see that generally the error per class and overall error show little to no deviation across feature selection. Indeed, accuracy is the same for both selections $\approx .83$

- From the figures for poly SVM we can see that generally the error-per-class and overall error is higher when using the best 3 features. In the per-class error figure, the best 3 features option gives higher error for folds 2 and 3 at class 2. Similarly, in the overall error figure we see that by choosing 3 best features, the error is higher at folds 2 and 3 only. From part (b) the average accuracy is higher when using 5 best features. ($\approx .87$) .

- **KNN with different ks**: From the figures we can see that generally the error per class and overall error are higher when using best 3 features. However, when we use k=16 this trend denatures as both plots are equal. This makes sense as setting k high usually causes us to underfit, especially when the train-test set is small. From part (b) the average accuracy is the highest for the KNN classifier with k=4 which uses 5 features. Still, the average accuracy tends to be higher when we use 5 features instead of 3 features and lower ks. A intresting finding, however, is that the k=2 and k=4 classifier perform much worse when trained on 3 features in comparison to k=8 and k=16 which see no drop. This is likely due poor separability being present in 3-dimensional space which causes different class neighbors to be closer.

- **Random forest with different max depths**: From the figures we can see that generally the error per class and overall error are higher when using best 3 features. From part (b) the average accuracy is higher for classifier trained on 5 features. Furthermore, the average accuracy tends to decrease as we increase the max_depth hyper-parameter. These trends are likley due to overfitting, which is more prominent for 3 features as the classifier forces itself to make highly specific splits on nodes at each level to reach the target depth, assuming it doesn't no convergence at a previous level.

Based on the analysis, our final recommendation for feature selection is to choose the best 5 features as it generally avoids overfitting for most of the listed classifiers. Our final recommendation for classifier is to choose kNN with k=4 which has accuracy 0.91. kNN has 0 training time and it is good for our small data set.

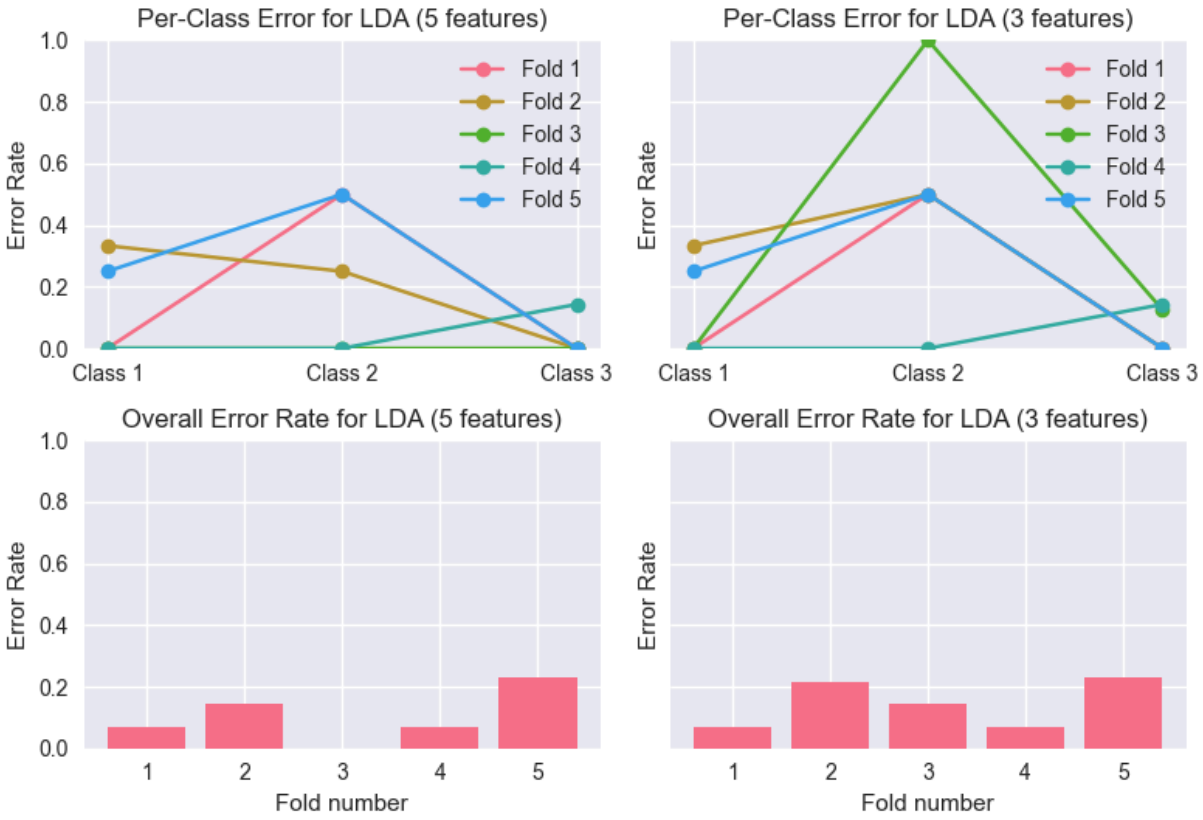
In [23]:
```
from part4c import error_plots
```

LDA

5 and 3 best features

In [24]:
```
print("Confusion matrices for LDA classifier")
error_plots([confusion_matrices_lda, best_3_confusion_matrices_lda], ["LDA (
```

```
Confusion matrices for LDA classifier
LDA (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 1 0]
 [0 0 8]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
LDA (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 1 7]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
```



QDA

5 and 3 best features

In [25]:
```
print("Confusion matrices for QDA classifier")
error_plots([confusion_matrices_qda, best_3_confusion_matrices_qda], ["QDA (
```

```
Confusion matrices for QDA classifier
QDA (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [1 1 6]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 4 0]
 [0 0 5]]
####################
QDA (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 0 8]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
```

SVM with different kernels

5 and 3 best features

In [26]:
```
print("Confusion matrices for SVM classifiers")

for kernel, cm, best_3_cm in zip(kernels, confusion_matrices_svms, best_3_cc
    error_plots([cm, best_3_cm], [f"{kernel} SVM (5 features)", f"{kernel} S
```

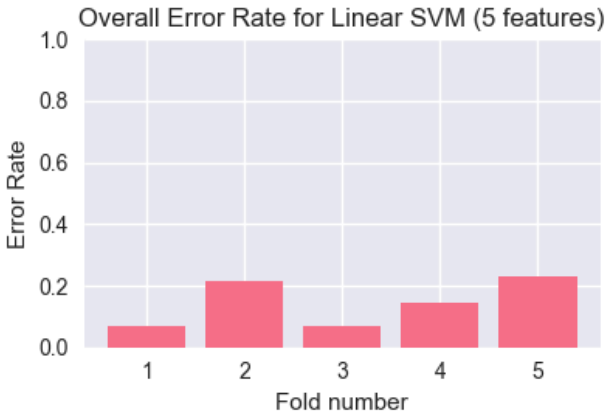
```
Confusion matrices for SVM classifiers
Linear SVM (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 0 8]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
Linear SVM (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
RBF SVM (5 features)
Fold 1
[[7 0 0]
```

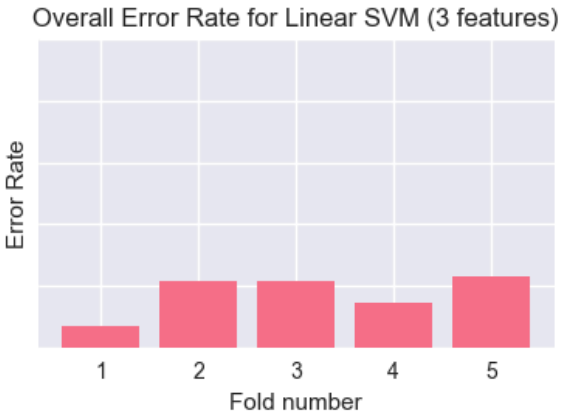
```
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
RBF SVM (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
Poly SVM (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
```

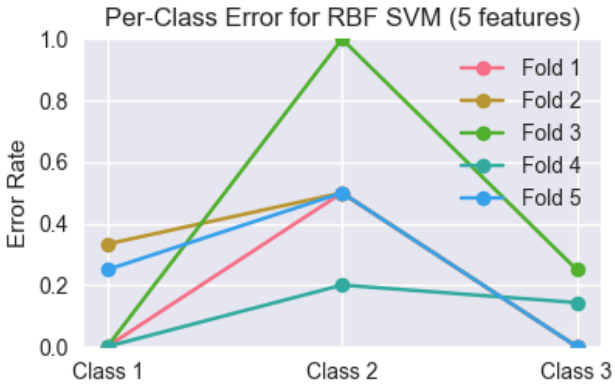
```
[[2 0 1]
 [0 4 0]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 1 0]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
Poly SVM (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
```

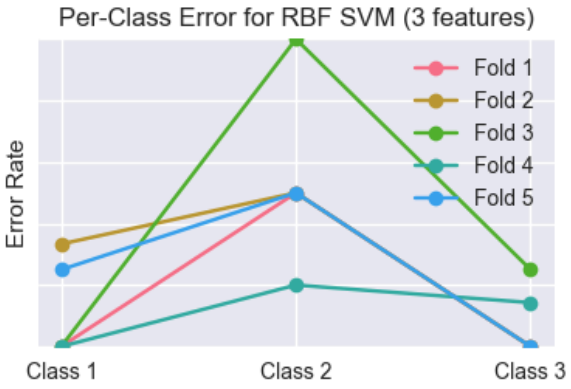

Per-Class Error for Linear SVM (5 features)
Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Class 1
Class 2
Class 3

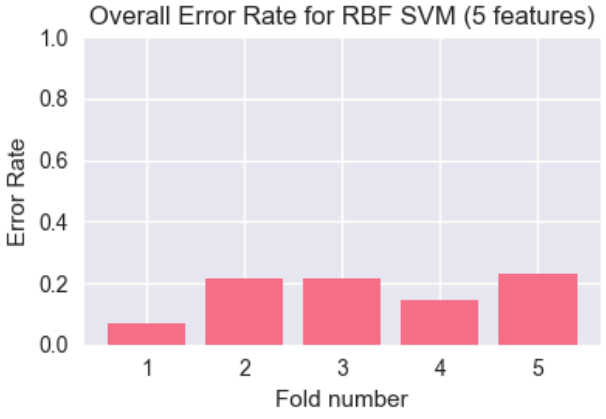
Per-Class Error for Linear SVM (3 features)
Error Rate
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Class 1
Class 2
Class 3

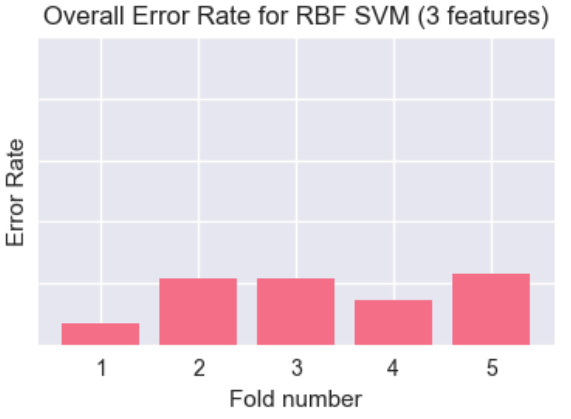

Overall Error Rate for Linear SVM (5 features)
Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
5
Fold number

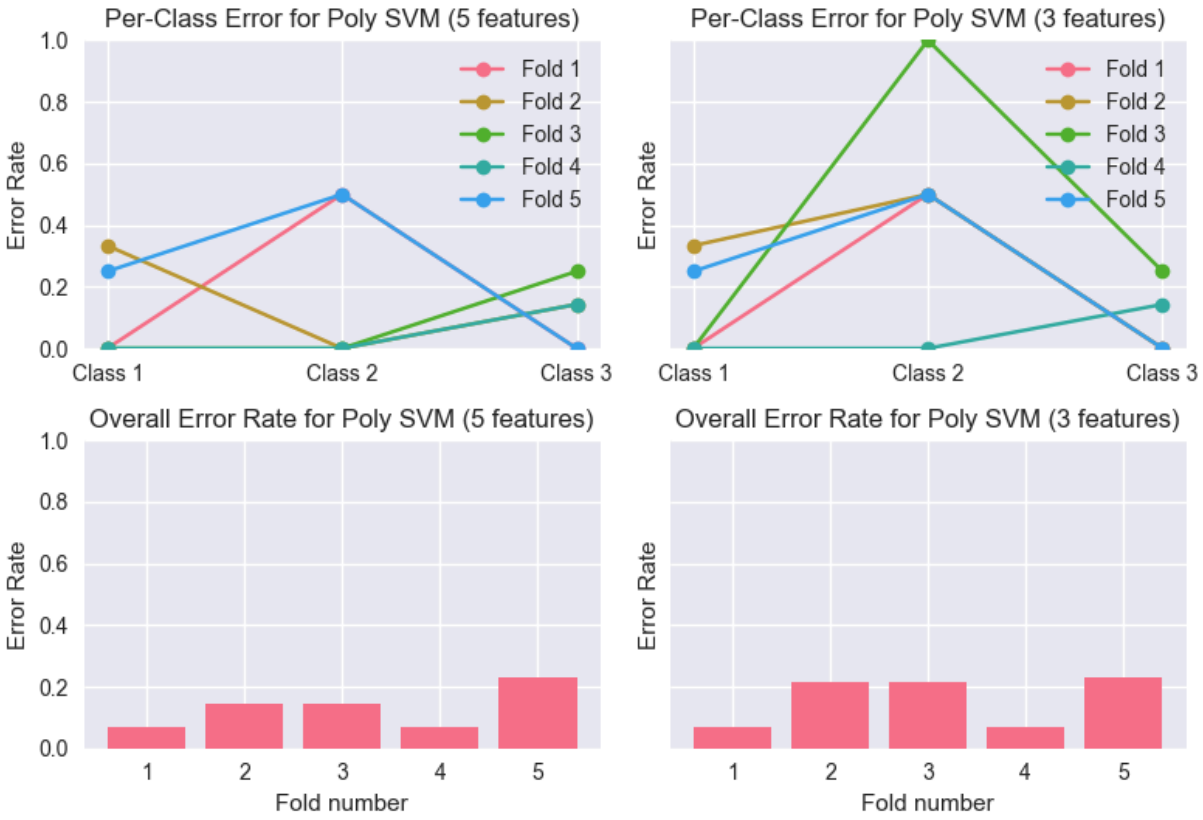

Overall Error Rate for Linear SVM (3 features)
Error Rate
1
2
3
4
5
Fold number


Per-Class Error for RBF SVM (5 features)
Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Class 1
Class 2
Class 3


Per-Class Error for RBF SVM (3 features)
Error Rate
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5
Class 1
Class 2
Class 3


Overall Error Rate for RBF SVM (5 features)
Error Rate
1.0
0.8
0.6
0.4
0.2
0.0
1
2
3
4
5
Fold number


Overall Error Rate for RBF SVM (3 features)
Error Rate
1
2
3
4
5
Fold number



KNN with different k's

5 and 3 best features

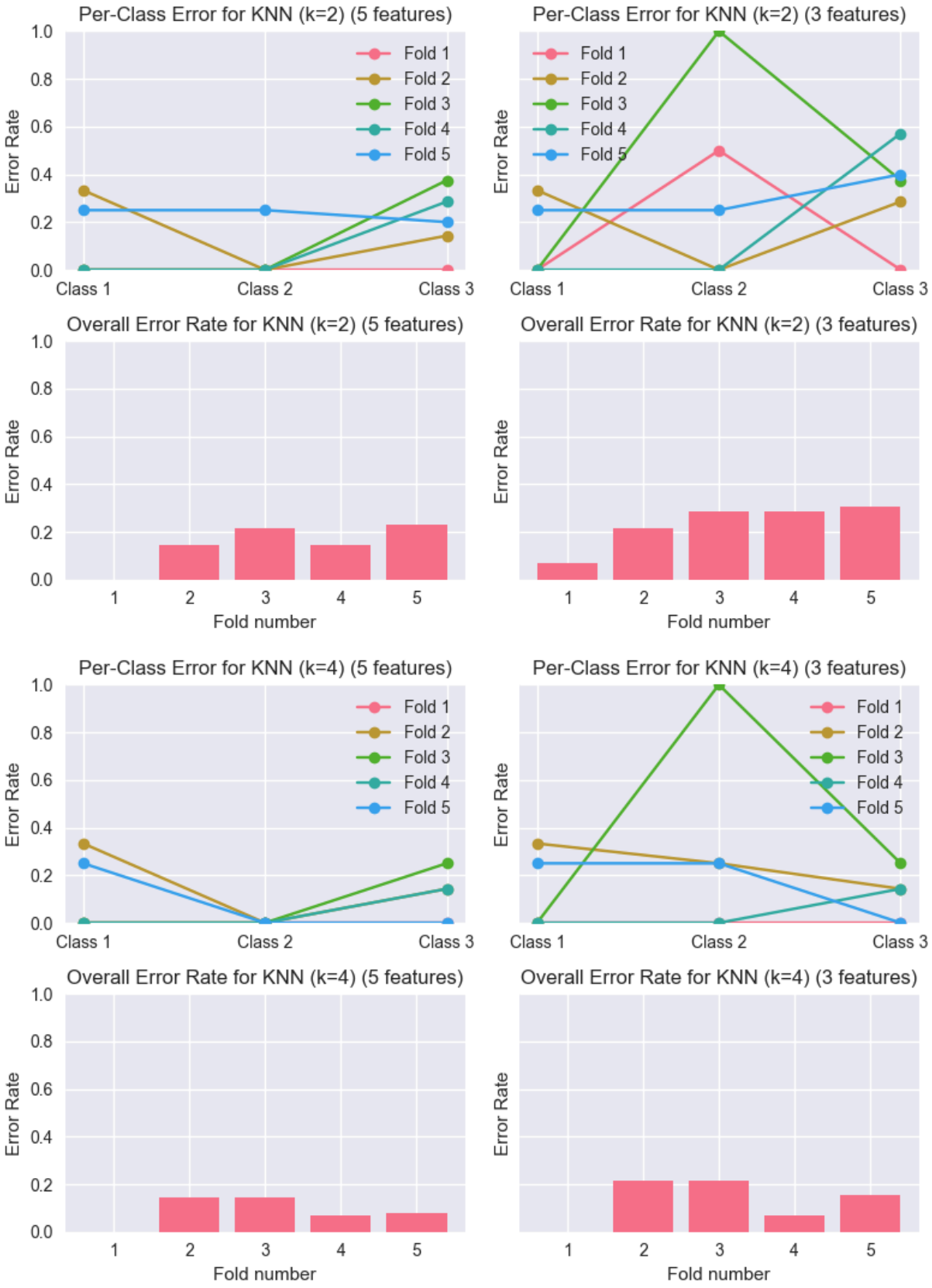
In [27]:
```
print("Confusion matrices for KNN classifiers")

for k, cm, best_3_cm in zip(ks, confusion_matrices_knn, best_3_confusion_mat
    error_plots([cm, best_3_cm], [f"KNN (k={k}) (5 features)", f"KNN (k={k})
```

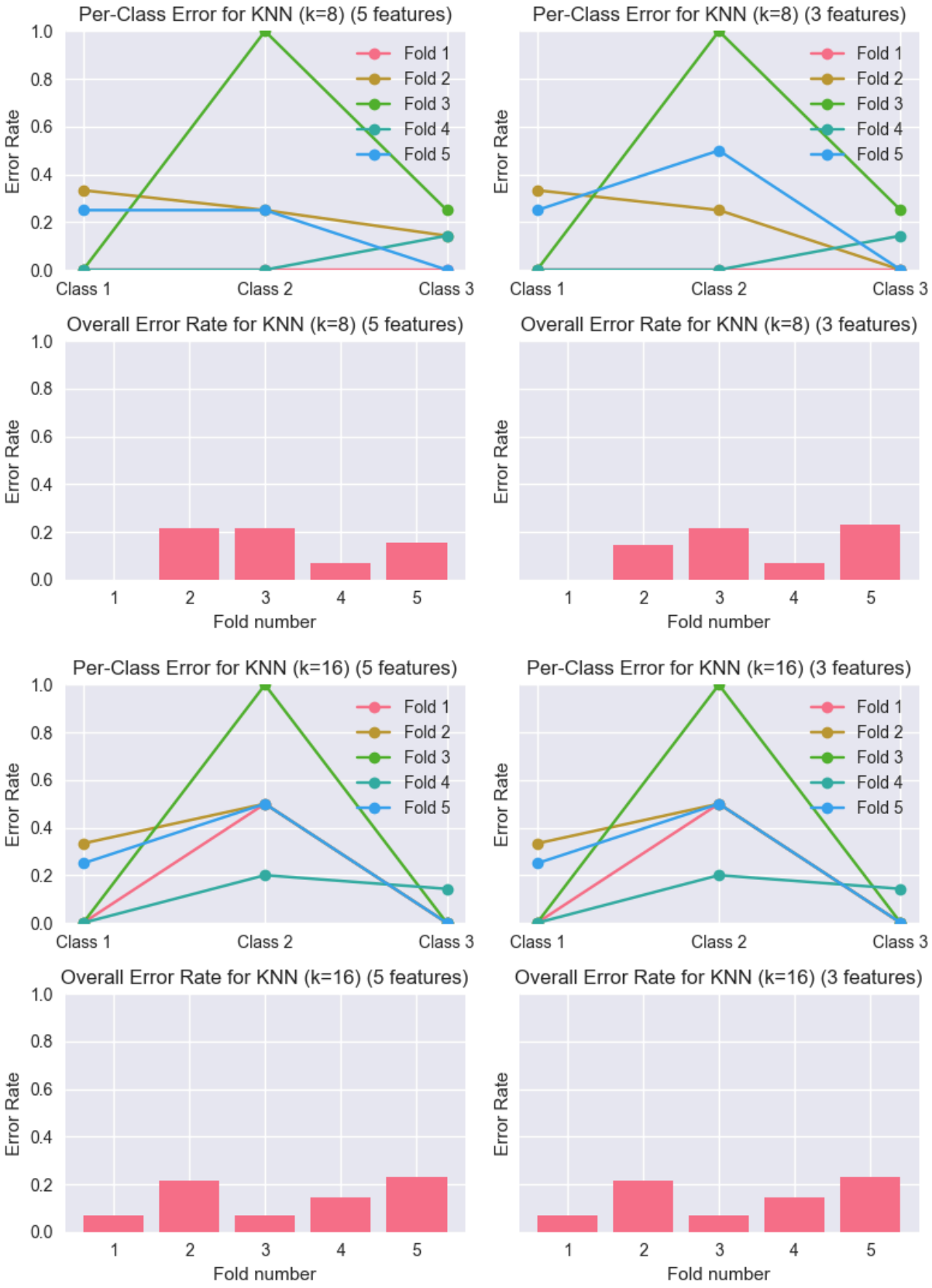
```
Confusion matrices for KNN classifiers
KNN (k=2) (5 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 4 0]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 1 0]
 [1 2 5]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [1 1 5]]
####################
Fold 5
[[3 0 1]
 [0 3 1]
 [1 0 4]]
####################
KNN (k=2) (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 4 0]
 [0 2 5]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [1 2 5]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [2 2 3]]
####################
Fold 5
[[3 0 1]
 [0 3 1]
 [1 1 3]]
####################
KNN (k=4) (5 features)
Fold 1
[[7 0 0]
```

```
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 4 0]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 1 0]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 4 0]
 [0 0 5]]
####################
KNN (k=4) (3 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 3 1]
 [0 0 5]]
####################
KNN (k=8) (5 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
```

```
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 3 1]
 [0 0 5]]
####################
KNN (k=8) (3 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
KNN (k=16) (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
```

```
Fold 3
[[5 0 0]
 [0 0 1]
 [0 0 8]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
KNN (k=16) (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 2 2]
 [0 0 7]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 0 8]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
```


Per-Class Error for KNN (k=2) (5 features)
Per-Class Error for KNN (k=2) (3 features)
Overall Error Rate for KNN (k=2) (5 features)
Overall Error Rate for KNN (k=2) (3 features)
Per-Class Error for KNN (k=4) (5 features)
Per-Class Error for KNN (k=4) (3 features)
Overall Error Rate for KNN (k=4) (5 features)
Overall Error Rate for KNN (k=4) (3 features)
Error Rate
Fold number
Class 1
Class 2
Class 3
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5



Random forest with different max depths

5 and 3 best features

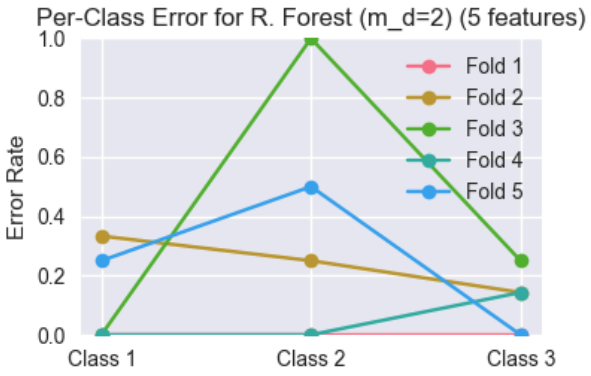
In [28]:
```
print("Confusion matrices for Random Forest classifiers")

for max_depth, cm, best_3_cm in zip(max_depths, confusion_matrices_random_fo
    error_plots([cm, best_3_cm], [f"R. Forest (m_d={max_depth}) (5 features)
```

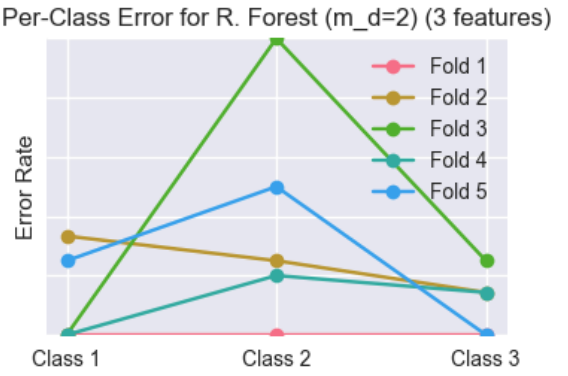
```
Confusion matrices for Random Forest classifiers
R. Forest (m_d=2) (5 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
R. Forest (m_d=2) (3 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
R. Forest (m_d=4) (5 features)
Fold 1
[[7 0 0]
```

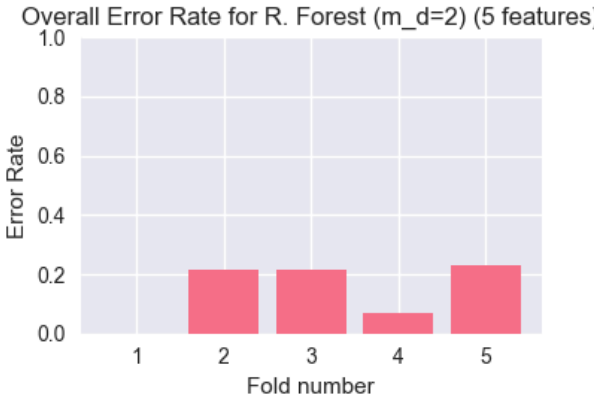
```
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
R. Forest (m_d=4) (3 features)
Fold 1
[[7 0 0]
 [0 2 0]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 1 3]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 4 1]
 [0 1 6]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
R. Forest (m_d=8) (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
```

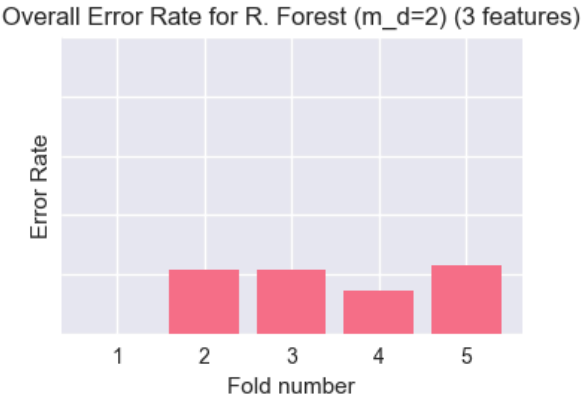
```
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 1 0]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [1 1 5]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
R. Forest (m_d=8) (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 1 3]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [2 1 4]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 1 4]]
####################
R. Forest (m_d=16) (5 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 3 1]
 [0 1 6]]
####################
```

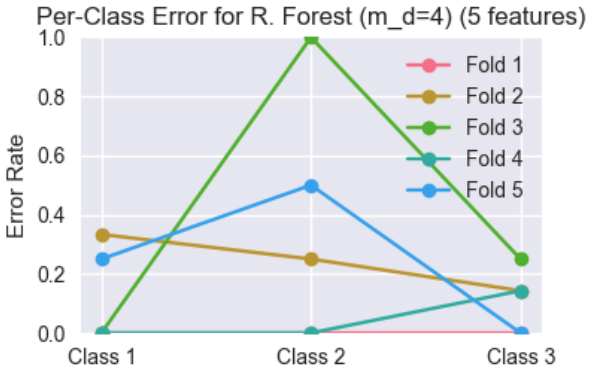
```
Fold 3
[[5 0 0]
 [0 1 0]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [1 1 5]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 0 5]]
####################
R. Forest (m_d=16) (3 features)
Fold 1
[[7 0 0]
 [0 1 1]
 [0 0 5]]
####################
Fold 2
[[2 0 1]
 [0 1 3]
 [0 1 6]]
####################
Fold 3
[[5 0 0]
 [0 0 1]
 [0 2 6]]
####################
Fold 4
[[2 0 0]
 [0 5 0]
 [2 1 4]]
####################
Fold 5
[[3 0 1]
 [0 2 2]
 [0 1 4]]
####################
```


Per-Class Error for R. Forest (m_d=2) (5 features)
Error Rate
Class 1
Class 2
Class 3
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5

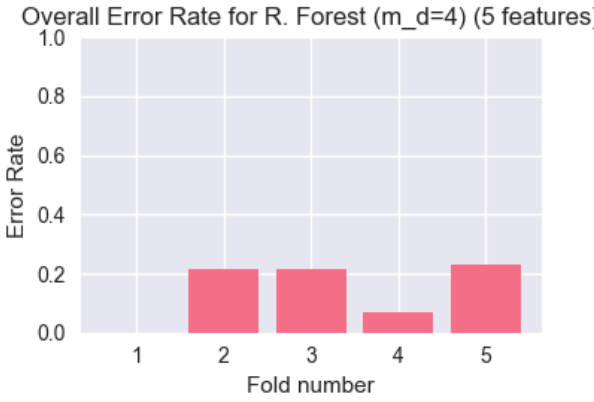

Per-Class Error for R. Forest (m_d=2) (3 features)
Error Rate
Class 1
Class 2
Class 3
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5

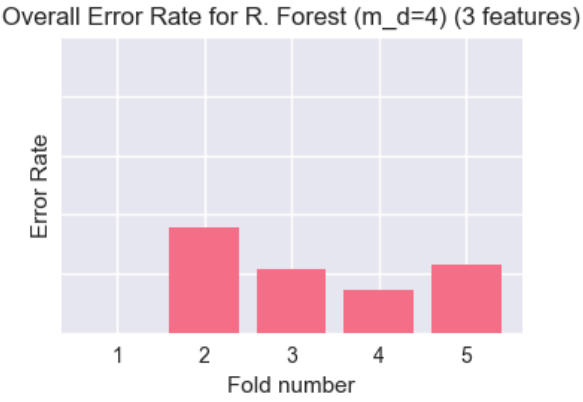

Overall Error Rate for R. Forest (m_d=2) (5 features)
Error Rate
Fold number

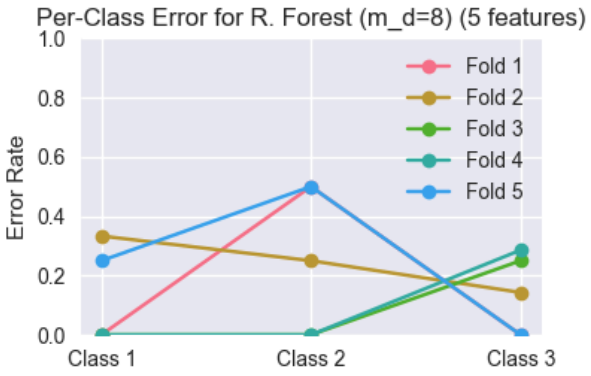

Overall Error Rate for R. Forest (m_d=2) (3 features)
Error Rate
Fold number

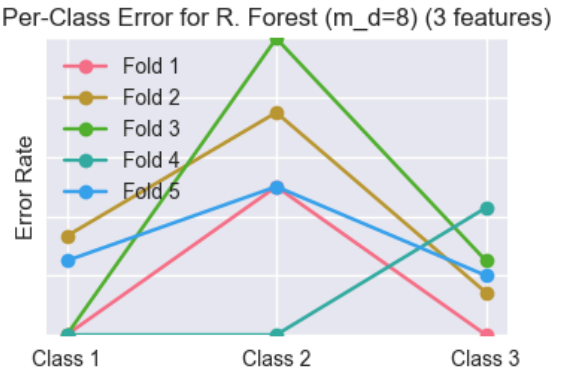

Per-Class Error for R. Forest (m_d=4) (5 features)
Error Rate
Class 1
Class 2
Class 3
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5

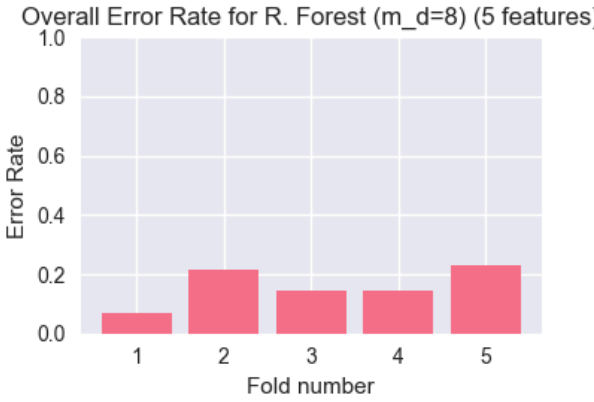

Per-Class Error for R. Forest (m_d=4) (3 features)
Error Rate
Class 1
Class 2
Class 3
Fold 1
Fold 2
Fold 3
Fold 4
Fold 5

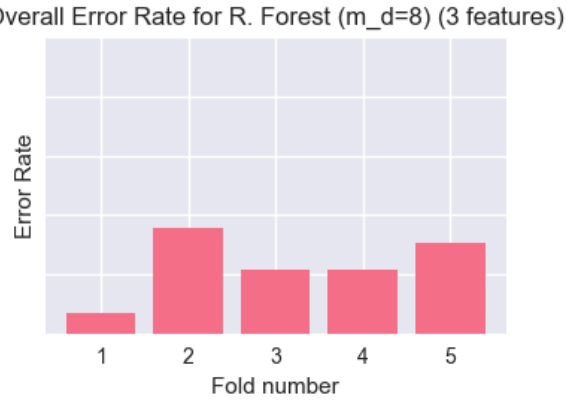

Overall Error Rate for R. Forest (m_d=4) (5 features)
Error Rate
Fold number

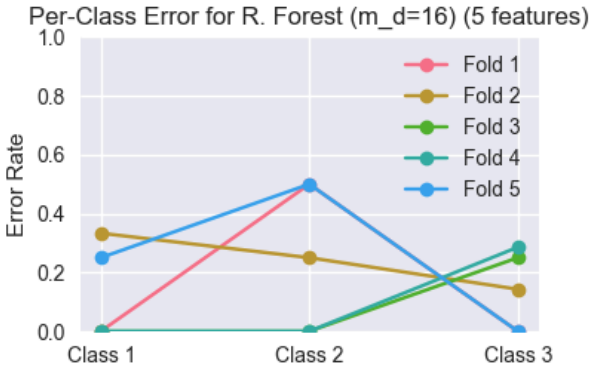

Overall Error Rate for R. Forest (m_d=4) (3 features)
Error Rate
Fold number

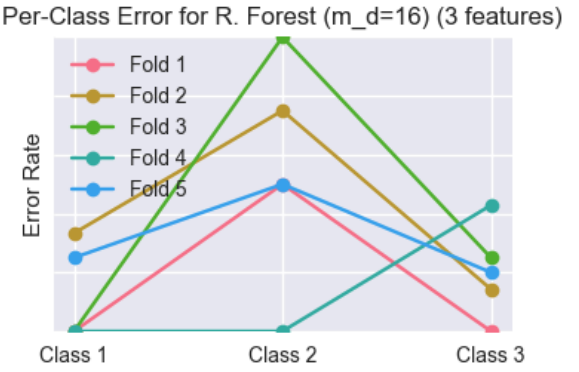


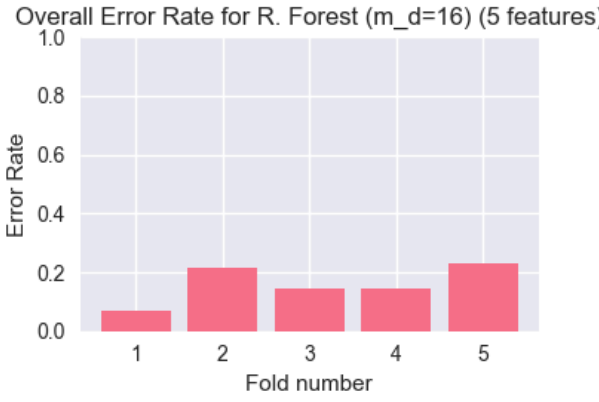


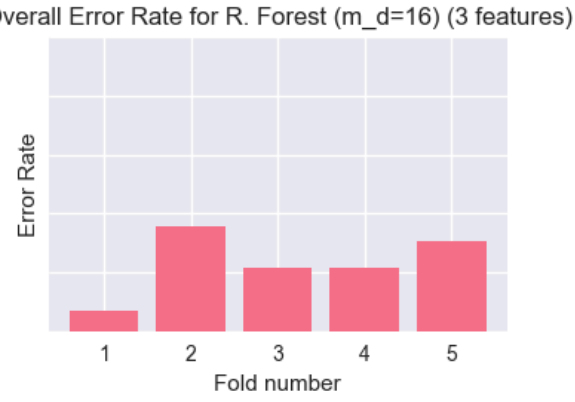












Part 5

Write a short contribution statement for each group member in your report. Clearly outline the specific contributions each person made to the project. This ensures transparency and highlights the collaborative effort of the group.

- **Ivo Kusijanovic (ilk2)**:
  - Part 3: Data processing, feature generation (feature extraction for groups B and C), feature selection, classification, sensor analysis
  - Part 4: Classification (developed framework to get average accuracy for an arbitrary classifier, train KNN and Random Forest), Performance (wrote insights for KNN and Random Forest)
- **Marisol Velapatiño (mv31)**:
  - Part 2: Description of Ultrasonic welding process
  - Part 3: feature generation (feature extraction for groups A, brainstorming of which features to extract for group B)
  - Part 4: Feature Selection, Classification (train LDA, QDA, and SVM), Performance (wrote insights for LDA, QDA, and SVM, and final recommendation)

Appendix

Part 3a

In [29]:
```
import matplotlib.pyplot as plt
import os

base_path = "./data/part3/"

# Force Signals

def extract_force_signals():

    """
    Extracts the time and force signals for each experiment in the Force_Sig

    Returns an array of (experiment name, time array, force array) tuples
    """

    # loop through files within Force_Signals dir

    force_signals = sorted(os.listdir(base_path + "Force_Signals/"))

    signals = []

    for force_signal in force_signals:
```

```
        time = []
        force = []

        with open(base_path + "Force_Signals/" + force_signal, "r") as f:

            lines = f.readlines()

            for line in lines:

                time_, force_ = line.strip().split()

                time.append(float(time_))
                force.append(float(force_))

        signals.append([force_signal, time, force])

    return signals

def extract_force_main_weld_segment(experiment, time, force, plot=False):

    """
    Extract the main weld segment of the force graph for each experiment
    """

    # plt.figure(figsize=(16,8))
    # plt.title(force_signal)
    # plt.plot(time[80000:], force[80000:])
    # plt.show()

    # Based on exploration we know that the main weld for all experiments st

    # We see that leading up to the main weld there is minimal noise.
    # When we reach the main weld we see a spike in noise

    n = len(time)

    window = 50
    noise_threshold_start = 0.025

    ptr = 80000
    while ptr < n:

        # Look into a 20 point window
        diff = 1 / window * sum([abs(force[i + 1] - force[i]) for i in range
        if diff >= noise_threshold_start:
            break

        ptr += window

    main_weld_start_idx = ptr

    # When we neear the end of the main weld we see a decrease in noise

    ptr = main_weld_start_idx + 10000

    noise_threshold_end = 0.035
```

```
    while ptr < n:

        diff = 1 / window * sum([abs(force[i + 1] - force[i]) for i in range
        if diff < noise_threshold_end:
            break

        ptr += window

    main_weld_end_idx = ptr

    if plot:

        plt.figure(figsize=(16,8))
        plt.title(experiment)

        plt.plot(time, force, color='cornflowerblue')
        plt.axvline(x=time[main_weld_start_idx], color='red', linestyle='--'
        plt.axvline(x=time[main_weld_end_idx], color='red', linestyle='--')

        plt.show()

    return (experiment, time[main_weld_start_idx:main_weld_end_idx + 1], for

# Power Signals

def extract_power_signals():

    """
    Extracts the time and power signals for each experiment in the Power_Sig

    Returns an array of (experiment name, time array, power array) tuples
    """

    power_signals = sorted(os.listdir(base_path + "Power_Signals/"))

    signals = []

    for power_signal in power_signals:

        time = []
        power = []

        with open(base_path + "Power_Signals/" + power_signal, "r") as f:

            lines = f.readlines()

            for line in lines:

                time_, power_ = line.strip().split()

                time.append(float(time_))
                power.append(float(power_))

        signals.append([power_signal, time, power])
```

```
    return signals

def extract_power_main_weld_segment(experiment, time, power, plot=False):

    """
    Extract the main weld segment of the power graph
    """

    # plt.figure(figsize=(12,8))
    # plt.title(power_signal)
    # plt.plot(time, power)
    # plt.show()

    # The main weld starts when we observe an increase in power

    n = len(time)

    window = 2
    increase_threshold = 25

    ptr = 0
    while ptr < n:

        increase = 1 / window * sum([power[i + 1] - power[i] for i in range(
        if increase >= increase_threshold:
            break

        ptr += window

    main_weld_start_idx = ptr

    # The main weld ends when we observe a decrease in power

    ptr = main_weld_start_idx + 100

    decrease_threshold = 500

    while ptr < n:

        decrease = 1 / window * sum([power[i] - power[i + 1] for i in range(
        if decrease >= decrease_threshold:
            break

        ptr += window

    main_weld_end_idx = ptr

    if plot:

        plt.figure(figsize=(16,8))
        plt.title(experiment)

        plt.plot(time, power, color='cornflowerblue')
        plt.axvline(x=time[main_weld_start_idx], color='red', linestyle='--'
        plt.axvline(x=time[main_weld_end_idx], color='red', linestyle='--')
```

```
        plt.show()

    return (experiment, time[main_weld_start_idx:main_weld_end_idx + 1], pow
```

Part 3b

In [30]:

```
from sklearn.preprocessing import StandardScaler
from scipy.signal import medfilt, savgol_filter
from scipy.signal import find_peaks
from scipy.fft import fft, fftfreq
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

## Functions for Group B

def denoise_power(power):

  power_med = medfilt(power, kernel_size=7)
  return savgol_filter(power_med, window_length=51, polyorder=3, mode='inter

def extract_features_group_b(time, power):

    """
    Returns the risePeak, riseSlope, riseDuration, and dipDepth of the main
    """

    # Remove noise
    denoised_power = denoise_power(power)

    n = len(time)

    window = 2
    peak_threshold = 5

    ptr = 20 # many experiments start with a small downslope before a rise,
    while ptr < n:

        # peak threshold condition

        diff = 1 / window * sum([abs(denoised_power[i + 1] - denoised_power[

        if diff <= peak_threshold:
            break

        # actual peak condition

        curr_window_avg = 1 / window * sum([denoised_power[i] for i in range
        next_window_avg = 1 / window * sum([denoised_power[i] for i in range

        if next_window_avg < curr_window_avg:
            break

        ptr += 1
```

```
    rise_peak = power[ptr]
    rise_slope = (power[ptr] - power[0]) / time[ptr] - time[0]
    rise_duration = time[ptr] - time[0]

    dip = min(power[ptr:n])

    dip_depth = rise_peak - dip

    # plt.figure(figsize=(16,8))
    # plt.plot(time, denoised_power)
    # plt.hlines(dip, xmin=time[0], xmax=time[-1], color="red")
    # plt.show()

    return rise_peak, rise_slope, rise_duration, dip_depth

## Functions for group C

def extract_features_group_c(time, force):

    time = np.array(time)
    force = np.array(force)

    X = np.column_stack((time, force))

    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X)

    time = X_scaled[:, 0]
    force = X_scaled[:, 1]

    n = len(time)

    sampling_rate = 100000

    fft_values = fft(force) / n
    frequencies = fftfreq(n, 1 / sampling_rate)

    positive_frequencies = frequencies[:n // 2]
    magnitude_spectrum = np.abs(fft_values[:n // 2])

    peaks, _ = find_peaks(magnitude_spectrum)

    sorted_peaks = peaks[np.argsort(magnitude_spectrum[peaks])][::-1]

    first_peak_freq = positive_frequencies[sorted_peaks[0]]
    first_peak_magn = magnitude_spectrum[sorted_peaks[0]]

    second_peak_freq = positive_frequencies[sorted_peaks[1]]
    second_peak_magn = magnitude_spectrum[sorted_peaks[1]]

    return first_peak_freq, first_peak_magn, second_peak_freq, second_peak_m
```

Part 3c

In [31]:
```
import matplotlib.pyplot as plt
import numpy as np

def plot_best_features(df):

    mask_cold = df["quality label"] == "I"
    mask_excessive = df["quality label"] == "II"
    mask_good = df["quality label"] == "III"

    features = df.iloc[:, 2:]
    feature_names = features.columns

    df_cold = features[mask_cold]
    df_excessive = features[mask_excessive]
    df_good = features[mask_good]

    def fisher_ratio(feature_names, df_1, df_2):

        Js = []

        for col_name in feature_names:

            mu_c1 = df_1[col_name].mean()
            std_c1 = df_1[col_name].std()

            mu_c2 = df_2[col_name].mean()
            std_c2 = df_2[col_name].std()

            J = (mu_c1 - mu_c2) ** 2 / (std_c1 ** 2 + std_c2 ** 2)

            Js.append(J)

        return np.array(Js)

    Js_good_cold = fisher_ratio(feature_names, df_good, df_cold)
    Js_good_excessive = fisher_ratio(feature_names, df_good, df_excessive)

    Js = Js_good_cold + Js_good_excessive

    sorted_indices = np.argsort(Js)

    best_features = feature_names[sorted_indices][::-1][:3]
    _ = Js[sorted_indices][::-1][:3]

    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    ax.set_title("Separability of Best 3 Features")

    ax.set_xlabel("Rise Peak")
    ax.set_ylabel("Rise Slope")
    ax.set_zlabel("Welding Pressure")

    ax.grid(True)
```

```
    ax.scatter(
        df_cold[best_features[0]], df_cold[best_features[1]], df_cold[best_f
        color='blue', alpha=0.6, s=20, label="Cold Weld"
    )

    ax.scatter(
        df_excessive[best_features[0]], df_excessive[best_features[1]], df_e
        color='red', alpha=0.6, s=20, label="Excessive Weld"
    )

    ax.scatter(
        df_good[best_features[0]], df_good[best_features[1]], df_good[best_f
        color='green', alpha=0.6, s=20, label="Good Weld"
    )

    ax.legend()
    plt.show()

    return best_features
```

Part 4a

In [32]:

```
import numpy as np
import matplotlib.pyplot as plt

def fisher_ratio(feature_names, df_1, df_2):

    Js = []

    for col_name in feature_names:

        mu_c1 = df_1[col_name].mean()
        std_c1 = df_1[col_name].std()

        mu_c2 = df_2[col_name].mean()
        std_c2 = df_2[col_name].std()

        J = (mu_c1 - mu_c2) ** 2 / (std_c1 ** 2 + std_c2 ** 2)

        Js.append(J)

    return np.array(Js)

def get_best_features(df, plot=False, threshold=0.8):

    mask_cold = df["Class"] == 1
    mask_excessive = df["Class"] == 2
    mask_good = df["Class"] == 3

    features = df.iloc[:, 2:]
    feature_names = features.columns

    df_cold = features[mask_cold]
    df_excessive = features[mask_excessive]
    df_good = features[mask_good]
```

```
Js_good_cold = fisher_ratio(feature_names, df_good, df_cold)
Js_good_excessive = fisher_ratio(feature_names, df_good, df_excessive)

Js = Js_good_cold + Js_good_excessive

sorted_indices = np.argsort(Js)

sorted_features = feature_names[sorted_indices][::-1]
sorted_Js = Js[sorted_indices][::-1]

# Choose cumulative sum of ratios to be less than .80

ws = []

Js_total = np.sum(sorted_Js)
cum_sum = 0

for J in sorted_Js:
    w = J / Js_total
    cum_sum += w

    if cum_sum > threshold:
        break

    ws.append(cum_sum)

best_features = sorted_features[:len(ws)]

print("Cumulative sum of ratios:", ws)
print(f"Best {len(ws)} features are:", best_features)

if plot and len(ws) >= 3:

    fig = plt.figure(figsize=(10, 8))
    ax = fig.add_subplot(111, projection='3d')

    ax.set_title(f"Separability of Best 3 Features")

    ax.set_xlabel(best_features[0])
    ax.set_ylabel(best_features[1])
    ax.set_zlabel(best_features[2])

    ax.grid(True)

    ax.scatter(
        df_cold[best_features[0]], df_cold[best_features[1]], df_cold[be
        color='blue', alpha=0.6, s=20, label="Cold Weld"
    )

    ax.scatter(
        df_excessive[best_features[0]], df_excessive[best_features[1]],
        color='red', alpha=0.6, s=20, label="Excessive Weld"
    )

    ax.scatter(
```

```
            df_good[best_features[0]], df_good[best_features[1]], df_good[be
            color='green', alpha=0.6, s=20, label="Good Weld"
        )

        ax.legend()
        plt.show()

    return best_features
```

Part 4b

In [33]:

```
import numpy as np
import warnings

from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold, LeaveOneOut

warnings.filterwarnings("ignore")

# We do not need to scale X as the data is already normalized

def classify(X, y, classifier, classifier_name, splits=5):

    # Train error
    classifier.fit(X, y)
    y_pred = classifier.predict(X)

    train_accuracy = accuracy_score(y, y_pred)
    print(f"{classifier_name} Train Accuracy", train_accuracy)

    kf = KFold(n_splits=splits, random_state=42, shuffle=True)
    # loo = LeaveOneOut()

    accuracies = []
    confusion_matrices = []

    for i, (train_index, test_index) in enumerate(kf.split(X)):

        X_train = X.iloc[train_index, :]
        X_test = X.iloc[test_index, :]
        y_train = y.iloc[train_index]
        y_test = y.iloc[test_index]

        classifier.fit(X_train, y_train)
        y_pred = classifier.predict(X_test)

        accuracy = accuracy_score(y_test, y_pred)
        ConfMatrix = confusion_matrix(y_test, y_pred)

        #print(f"Fold {i + 1}")
        #print(f"{classifier_name} Accuracy: {accuracy}")
        #print("#" * 70)

        accuracies.append(accuracy)
```

```
        confusion_matrices.append(ConfMatrix)

    avg_accuracy = np.mean(accuracies)
    print(f"Average {classifier_name} Validation Accuracy: {avg_accuracy}")

    return accuracies, confusion_matrices
```

Part 4c

In [34]:

```
import numpy as np
import matplotlib.pyplot as plt

def error_rate(cm):
    return 1 - np.trace(cm) / np.sum(cm)

def class_errors(cm):
    return 1 - np.diag(cm) / cm.sum(axis=1)

def error_plots(cms, classifier_names, wspace=None):

    folds_ = np.arange(1,6,1)
    classes = ["Class 1", "Class 2", "Class 3"]

    _, axs = plt.subplots(2,len(cms), sharey=True)

    for j, cm in enumerate(cms):
        error_rates = []
        print(classifier_names[j])

        for i in folds_:
            print(f"Fold {i}")
            print(cm[i-1])
            print("#" * 20)

            error_rates.append(error_rate(cm[i-1]))
            ce = class_errors(cm[i-1])
            axs[0,j].plot(classes, ce, marker='o', label=f"Fold {i}")

        axs[0,j].set_title(f"Per-Class Error for {classifier_names[j]}")
        axs[0,j].set_ylabel("Error Rate")
        axs[0,j].set_ylim((0, 1))

        axs[0,j].legend()
        axs[0,j].grid(True)

        axs[1,j].bar(folds_, error_rates)
        axs[1,j].set_title(f"Overall Error Rate for {classifier_names[j]}")
        axs[1,j].set_ylabel("Error Rate")
        axs[1,j].set_xlabel("Fold number")
        axs[1,j].set_ylim((0, 1))
        axs[1,j].grid(True)

    plt.tight_layout()
```

```
    if wspace is not None:
        plt.subplots_adjust(wspace=wspace)
```