Assignment 2

By Marc Mailloux

Initial Code

Importing the necessary libraries:

```
from __future__ import division
import time
import pandas as p
from sklearn import svm
from sklearn.model_selection import train_test_split, GridSearchCV, RepeatedKFold
from sklearn.metrics import confusion_matrix
```

Pulling the data set next, and establishing glass. Additionally using the drop function allows the Id number and the classifier columns to be dropped for the observations

```
data = 'https://archive.ics.uci.edu/ml/machine-learning-databases/glass/glass.data'

glass = p.read_csv(data, sep=',', names=['Id number', 'RI', 'Na ', 'Mg', 'Al', 'Si', 'K','Ca', 'Ba','Fe', 'Classifier'])
#splitting predictors and observations
predictors= glass.Classifier
observation = glass.drop(['Classifier','Id number'], axis=1)
```

Using train test split to split the data sets into a training and validation sets for both X and Y variables. Additionally using the repeated Kfold function allows for the proper evaluation of the models using cross-validation where the test set will be iterated on through the entire data set.

```
train_obs, test_obs, train_predictors, test_predictors = train_test_split(observation, predictors,test_size=0.2,random_state=333, stratify=predictors)
cross_val = RepeatedKFold(n_splits=5, n_repeats=5, random_state=333)
```

Below are the hyperparameters for parts 1 to 3

```
hyperparameters_part1 = [{'kernel': ['rbf'], 'gamma': [1, .1, .01, 0.001, 0.0001],  'C': [1, 10, 100, 1000]},
                 {'kernel': ['linear'], 'gamma': [1, .1, .01, 0.001, 0.0001],  'C': [1, 10, 100, 1000]},
                 {'kernel': ['poly'], 'gamma': [1, .1, .01, 0.001, 0.0001],  'C': [1, 10, 100, 1000]},
                 {'kernel': ['sigmoid'], 'gamma': [1, .1, .01, 0.001, 0.0001],  'C': [1, 10, 100, 1000]}]


hyperparameters_part2 = [{'kernel': ['rbf'], 'gamma': [1, .1, .01, 0.001, 0.0001],   'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']},#,'class_weight' :['balanced' , None]},
                 {'kernel': ['linear'], 'gamma': [1, .1, .01,0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']},#,'class_weight' :['balanced' , None]},
                 {'kernel': ['poly'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']},#,'class_weight' :['balanced' , None]},
                 {'kernel': ['sigmoid'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']}]#,'class_weight' :['balanced' , None]}]

hyperparameters_part3 = [{'kernel': ['rbf'], 'gamma': [1, .1, .01, 0.001, 0.0001],   'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']},
                 {'kernel': ['linear'], 'gamma': [1, .1, .01,0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']},
                 {'kernel': ['poly'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']},
                 {'kernel': ['sigmoid'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']}]
```

The function function was made so that when m = 1 it is refering to part 1, m=2 refers to part 2 and so on. The only difference is in the outputs and the param_grid function.

```
def function(m):
    start = time.time()
    if m == 1:
```

calling the gridsearchCV function to perform Crossvalidation using the repeartedKfold function. After we train the classifier on the training sets. Also calling the specific hyperparameters for part 1 2 and 3.

```
# starting the gridsearch with the crossvalidation
classifier = GridSearchCV(estimator=svm.SVC(),
                          param_grid=hyperparameters_part1, cv=cross_val)

classifier.fit(train_obs, train_predictors)
```

The code below displays the accuracies for the dictionary given by the hyperparameters of 5 Kfold CV

```
print "generated results for the data"
means = classifier.cv_results_['mean_test_score']
stds = classifier.cv_results_['std_test_score']
for mean, std, params in zip(means, stds, classifier.cv_results_['params']):
    print("%0.4f (+/-%0.03f) for %r" % (mean, std * 2, params))
print ''
print "The Best Training Score was:", classifier.best_score_
print ''
print "The Best Parameters were: ", classifier.best_params_
print ' '
```

Setting the variables for the best parameters for the test set

```
# setting the optimal values to variables
kern = classifier.best_estimator_.kernel
c = classifier.best_estimator_.C
g = classifier.best_estimator_.gamma
d = classifier.best_estimator_.decision_function_shape
```

Setting the optimal parameters to determine the test accuracy for the optimal settings. Additionally calculating the the time it takes from beginning to end of the code.

```
optimalrun = svm.SVC(kernel=kern, C=c, gamma=g, decision_function_shape= d , random_state= 333).fit(train_obs, train_predictors)
pred = optimalrun.predict(test_obs)

# model accuracy creating confusion matrix
accuracy = optimalrun.score(test_obs, test_predictors)
cm = confusion_matrix(test_predictors, pred)

print accuracy, 'is the models test accuracy part 1'

end = time.time()
print 'Completion Time:' (end - start)
```

The beginning of the code as a whole:

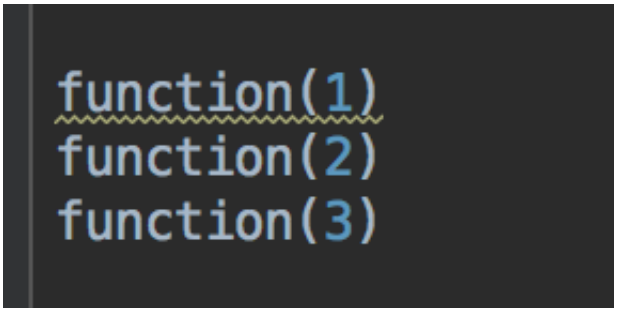
```
def function(m):
    start = time.time()
    if m == 1:

        # starting the gridsearch with the crossvalidation
        classifier = GridSearchCV(estimator=svm.SVC(),
                                  param_grid=hyperparameters_part1, cv=cross_val)

        classifier.fit(train_obs, train_predictors)

        print "generated results for the data"
        means = classifier.cv_results_['mean_test_score']
        stds = classifier.cv_results_['std_test_score']
        for mean, std, params in zip(means, stds, classifier.cv_results_['params']):
            print("%0.4f (+/-%0.03f) for %r" % (mean, std * 2, params))
        print ''
        print "The Best Training Score was:", classifier.best_score_
        print ''
        print "The Best Parameters were: ", classifier.best_params_
        print ' '

        # setting the optimal values to variables
        kern = classifier.best_estimator_.kernel
        c = classifier.best_estimator_.C
        g = classifier.best_estimator_.gamma
        d = classifier.best_estimator_.decision_function_shape

        optimalrun = svm.SVC(kernel=kern, C=c, gamma=g, decision_function_shape= d , random_state= 333).fit(train_obs, train_predictors)
        pred = optimalrun.predict(test_obs)

        # model accuracy creating confusion matrix
        accuracy = optimalrun.score(test_obs, test_predictors)
        cm = confusion_matrix(test_predictors, pred)

        print accuracy, 'is the models test accuracy part 1'

        end = time.time()
        print 'Completion Time:', (end - start)
```

The rest of the code for parts 2 & 3:

```
    elif m==2:
        # starting the gridsearch with the crossvalidation
        classifier = GridSearchCV(estimator=svm.SVC(),
                                  param_grid=hyperparameters_part2, cv=cross_val)
        # only need balanced for the last questions
        # not sure why the accuacryies arent changing in the part classifier
        classifier.fit(train_obs, train_predictors)

        print "generated results for the data"
        means = classifier.cv_results_['mean_test_score']
        stds = classifier.cv_results_['std_test_score']
        for mean, std, params in zip(means, stds, classifier.cv_results_['params']):
            print("%0.4f (+/-%0.03f) for %r" % (mean, std * 2, params))
        print ''
        print "The Best Training Score was:", classifier.best_score_
        print ''
        print "The Best Parameters were: ", classifier.best_params_
        print ' '

        # setting the optimal values to variables
        kern = classifier.best_estimator_.kernel
        c = classifier.best_estimator_.C
        g = classifier.best_estimator_.gamma
        d = classifier.best_estimator_.decision_function_shape

        optimalrun = svm.SVC(kernel=kern, C=c, gamma=g, decision_function_shape=d, random_state=333).fit(train_obs,
                                                                                                          train_predictors)
        pred = optimalrun.predict(test_obs)

        # model accuracy creating confusion matrix
        accuracy = optimalrun.score(test_obs, test_predictors)
        cm = confusion_matrix(test_predictors, pred)

        print accuracy, 'is the models test accuracy part 2'

        print 'One Vs rest'
        end = time.time()
        print 'completion time:', (end - start)
```

```
elif m == 3:
    # starting the gridsearch with the crossvalidation
    classifier = GridSearchCV(estimator=svm.SVC(),
                              param_grid=hyperparameters_part3, cv=cross_val)

    classifier.fit(train_obs, train_predictors)

    print "generated results for the data"
    means = classifier.cv_results_['mean_test_score']
    stds = classifier.cv_results_['std_test_score']
    for mean, std, params in zip(means, stds, classifier.cv_results_['params']):
        print("%0.4f (+/-%0.03f) for %r" % (mean, std * 2, params))
    print ''
    print "The Best Training Score was:", classifier.best_score_
    print ''
    print "The Best Parameters were: ", classifier.best_params_
    print ' '

    # setting the optimal values to variables
    kern = classifier.best_estimator_.kernel
    c = classifier.best_estimator_.C
    g = classifier.best_estimator_.gamma
    d = classifier.best_estimator_.decision_function_shape
    cw = classifier.best_estimator_.class_weight

    optimalrun = svm.SVC(kernel=kern, C=c, gamma=g, decision_function_shape=d, random_state=333, class_weight=cw).fit(train_obs,
                                                                                                                        train_predictors)
    pred = optimalrun.predict(test_obs)

    # model accuracy creating confusion matrix
    accuracy = optimalrun.score(test_obs, test_predictors)
    cm = confusion_matrix(test_predictors, pred)

    print accuracy, 'is the models test accuracy part 3'

    end = time.time()
    print 'Completion Time:', (end - start)
```

Below is the code that runs the entire code for all three parts



So in summary the code takes the glass data set and creates the observation and the predicting variables. From there it splits the data into a validation and training sets. From there a 5 fold cv takes places where it is 5 times repeated so that validation set is iterated through the entire dataset accordingly to find the optimal hyperparameters. The best hyper parameters are then run against the validation set for each part for the “test” accuracies

Part 1 Results

```
generated results for the data
0.7006 (+/-0.177) for {'kernel': 'rbf', 'C': 1, 'gamma': 1}
0.5860 (+/-0.194) for {'kernel': 'rbf', 'C': 1, 'gamma': 0.1}
0.4386 (+/-0.225) for {'kernel': 'rbf', 'C': 1, 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'rbf', 'C': 1, 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'rbf', 'C': 1, 'gamma': 0.0001}
0.7146 (+/-0.153) for {'kernel': 'rbf', 'C': 10, 'gamma': 1}
0.7158 (+/-0.166) for {'kernel': 'rbf', 'C': 10, 'gamma': 0.1}
0.5637 (+/-0.168) for {'kernel': 'rbf', 'C': 10, 'gamma': 0.01}
0.4351 (+/-0.219) for {'kernel': 'rbf', 'C': 10, 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'rbf', 'C': 10, 'gamma': 0.0001}
0.6807 (+/-0.175) for {'kernel': 'rbf', 'C': 100, 'gamma': 1}
0.7228 (+/-0.143) for {'kernel': 'rbf', 'C': 100, 'gamma': 0.1}
0.6959 (+/-0.160) for {'kernel': 'rbf', 'C': 100, 'gamma': 0.01}
0.5637 (+/-0.192) for {'kernel': 'rbf', 'C': 100, 'gamma': 0.001}
0.4316 (+/-0.204) for {'kernel': 'rbf', 'C': 100, 'gamma': 0.0001}
0.6596 (+/-0.178) for {'kernel': 'rbf', 'C': 1000, 'gamma': 1}
0.6819 (+/-0.177) for {'kernel': 'rbf', 'C': 1000, 'gamma': 0.1}
0.7064 (+/-0.151) for {'kernel': 'rbf', 'C': 1000, 'gamma': 0.01}
0.6690 (+/-0.141) for {'kernel': 'rbf', 'C': 1000, 'gamma': 0.001}
0.5579 (+/-0.154) for {'kernel': 'rbf', 'C': 1000, 'gamma': 0.0001}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'gamma': 1}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'gamma': 0.1}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'gamma': 0.01}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'gamma': 0.001}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'gamma': 0.0001}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'gamma': 1}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'gamma': 0.1}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'gamma': 0.01}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'gamma': 0.001}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'gamma': 0.0001}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'gamma': 1}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'gamma': 0.1}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'gamma': 0.01}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'gamma': 0.001}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'gamma': 0.0001}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'gamma': 1}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'gamma': 0.1}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'gamma': 0.01}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'gamma': 0.001}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'gamma': 0.0001}
```

```
0.6667 (+/-0.180) for {'kernel': 'poly', 'C': 1, 'gamma': 1}
0.6725 (+/-0.164) for {'kernel': 'poly', 'C': 1, 'gamma': 0.1}
0.6737 (+/-0.149) for {'kernel': 'poly', 'C': 1, 'gamma': 0.01}
0.5485 (+/-0.201) for {'kernel': 'poly', 'C': 1, 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'poly', 'C': 1, 'gamma': 0.0001}
0.6690 (+/-0.172) for {'kernel': 'poly', 'C': 10, 'gamma': 1}
0.6702 (+/-0.157) for {'kernel': 'poly', 'C': 10, 'gamma': 0.1}
0.6784 (+/-0.156) for {'kernel': 'poly', 'C': 10, 'gamma': 0.01}
0.6409 (+/-0.159) for {'kernel': 'poly', 'C': 10, 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'poly', 'C': 10, 'gamma': 0.0001}
0.6643 (+/-0.170) for {'kernel': 'poly', 'C': 100, 'gamma': 1}
0.6737 (+/-0.161) for {'kernel': 'poly', 'C': 100, 'gamma': 0.1}
0.6737 (+/-0.145) for {'kernel': 'poly', 'C': 100, 'gamma': 0.01}
0.6655 (+/-0.148) for {'kernel': 'poly', 'C': 100, 'gamma': 0.001}
0.4269 (+/-0.212) for {'kernel': 'poly', 'C': 100, 'gamma': 0.0001}
0.6702 (+/-0.175) for {'kernel': 'poly', 'C': 1000, 'gamma': 1}
0.6713 (+/-0.168) for {'kernel': 'poly', 'C': 1000, 'gamma': 0.1}
0.6643 (+/-0.146) for {'kernel': 'poly', 'C': 1000, 'gamma': 0.01}
0.6713 (+/-0.154) for {'kernel': 'poly', 'C': 1000, 'gamma': 0.001}
0.5485 (+/-0.201) for {'kernel': 'poly', 'C': 1000, 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1, 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1, 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1, 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 1, 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 1, 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 10, 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 10, 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 10, 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 10, 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 10, 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 100, 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 100, 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 100, 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 100, 'gamma': 0.001}
0.4058 (+/-0.164) for {'kernel': 'sigmoid', 'C': 100, 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1000, 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1000, 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1000, 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 1000, 'gamma': 0.001}
0.5181 (+/-0.191) for {'kernel': 'sigmoid', 'C': 1000, 'gamma': 0.0001}

The Best Training Score was: 0.7228070175438597

The Best Parameters were:  {'kernel': 'rbf', 'C': 100, 'gamma': 0.1}

0.6511627906976745 is the models test accuracy part 1
Completion Time: 78.4111089706
```

The above results are the combinations tried for the hyperparameters and the best accuracy calculated in the cross validation process for each combination.

The best parameters were RBF, c=100 , gamma = .1

With training accuracy of 72.28% and a test Accuracy is 65.12%

With a total calculating time of 78.411 seconds

Part 2 Results

For this section of the assignment i changed the hyperparameters to fit the conditions tasked with:

```
hyperparameters_part2 = [{'kernel': ['rbf'], 'gamma': [1, .1, .01, 0.001, 0.0001],   'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']},#,'class_weight' :['balanced' , None]},
                {'kernel': ['linear'], 'gamma': [1, .1, .01,0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']},#,'class_weight' :['balanced' , None]},
                {'kernel': ['poly'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']},#,'class_weight' :['balanced' , None]},
                {'kernel': ['sigmoid'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovr']}]#,'class_weight' :['balanced' , None]}]
```

```
generated results for the data
0.7006 (+/-0.177) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 1}
0.5860 (+/-0.194) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.4386 (+/-0.225) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.7146 (+/-0.153) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 1}
0.7158 (+/-0.166) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.5637 (+/-0.168) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.4351 (+/-0.219) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6807 (+/-0.175) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 1}
0.7228 (+/-0.143) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6959 (+/-0.160) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.5637 (+/-0.192) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.4316 (+/-0.204) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6596 (+/-0.178) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6819 (+/-0.177) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.7064 (+/-0.151) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6690 (+/-0.141) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.5579 (+/-0.154) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.6339 (+/-0.177) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.6538 (+/-0.157) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.6491 (+/-0.143) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
```

```
0.6526 (+/-0.151) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6667 (+/-0.180) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6725 (+/-0.164) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6737 (+/-0.149) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.5485 (+/-0.201) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6690 (+/-0.172) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6702 (+/-0.157) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6784 (+/-0.156) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6409 (+/-0.159) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6643 (+/-0.170) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6737 (+/-0.161) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6737 (+/-0.145) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6655 (+/-0.148) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.4269 (+/-0.212) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.6702 (+/-0.175) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 1}
0.6713 (+/-0.168) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.6643 (+/-0.146) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.6713 (+/-0.154) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.5485 (+/-0.201) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.4058 (+/-0.164) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.0001}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.1}
0.3287 (+/-0.154) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.01}
0.3216 (+/-0.149) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.001}
0.5181 (+/-0.191) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovr', 'gamma': 0.0001}

The Best Training Score was: 0.7228070175438597

The Best Parameters were:  {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovr', 'gamma': 0.1}

0.6511627906976745 is the models test accuracy part 2
One Vs rest
77.6086928844
```

The above results are the combinations tried for the hyperparameters and the best accuracy calculated in the cross validation process.

The best parameters for One vs Rest/All were RBF, c=100 , gamma = .1

With training accuracy of 72.28% and a test Accuracy is 65.12%
With a total calculating time of 77.61 seconds

Part 3 Results

The only difference in the results were in the time it took to complete the calculations in where one vs rest calculated quicker than one vs one but only by under a second. This makes sense since One vs rest is is more effcient.

The results below were running an additional time. Note that the accuracies are the same for the test but the one vs all procedure again finished quicker than the one vs one

```
0.6511627906976745 is the models test accuracy part 1
Completion Time: 82.8205120564
```

```
0.6511627906976745 is the models test accuracy part 2
One Vs rest
completion time: 79.920758009
```

Part 4 Results

For this section of the assignment i changed the hyperparameters to fit the conditions tasked with:

```
hyperparameters_part3 = [{'kernel': ['rbf'], 'gamma': [1, .1, .01, 0.001, 0.0001],   'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']},
                         {'kernel': ['linear'], 'gamma': [1, .1, .01,0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']},
                         {'kernel': ['poly'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']},
                         {'kernel': ['sigmoid'], 'gamma': [1, .1, .01, 0.001, 0.0001], 'C': [1, 10, 100, 1000], 'decision_function_shape': ['ovo'],'class_weight':['balanced']}]
```

```
generated results for the data
0.6643 (+/-0.164) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.5404 (+/-0.127) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.3860 (+/-0.274) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.2865 (+/-0.284) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.2830 (+/-0.261) for {'kernel': 'rbf', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.7006 (+/-0.150) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6304 (+/-0.176) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.5099 (+/-0.147) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.4140 (+/-0.264) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.2515 (+/-0.256) for {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.6924 (+/-0.181) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6480 (+/-0.130) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6164 (+/-0.131) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.4889 (+/-0.189) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.4363 (+/-0.272) for {'kernel': 'rbf', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.6596 (+/-0.178) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6795 (+/-0.134) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6468 (+/-0.155) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.6140 (+/-0.115) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.4936 (+/-0.224) for {'kernel': 'rbf', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.5661 (+/-0.125) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.5661 (+/-0.125) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.5661 (+/-0.125) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.5661 (+/-0.125) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.5661 (+/-0.125) for {'kernel': 'linear', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.6070 (+/-0.130) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6070 (+/-0.130) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6070 (+/-0.130) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.6070 (+/-0.130) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.6070 (+/-0.130) for {'kernel': 'linear', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.5895 (+/-0.147) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.5895 (+/-0.147) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.5895 (+/-0.147) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.5895 (+/-0.147) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.5895 (+/-0.147) for {'kernel': 'linear', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.5836 (+/-0.138) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.5836 (+/-0.138) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.5836 (+/-0.138) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.5836 (+/-0.138) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.5836 (+/-0.138) for {'kernel': 'linear', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
```

```
0.6433 (+/-0.174) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6222 (+/-0.173) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6281 (+/-0.159) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.4480 (+/-0.212) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.2725 (+/-0.265) for {'kernel': 'poly', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.6398 (+/-0.185) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6129 (+/-0.169) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6515 (+/-0.153) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.5801 (+/-0.126) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.2444 (+/-0.247) for {'kernel': 'poly', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.6433 (+/-0.183) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6164 (+/-0.146) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6281 (+/-0.170) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.6058 (+/-0.113) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.4292 (+/-0.274) for {'kernel': 'poly', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.6398 (+/-0.190) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.6211 (+/-0.161) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.6023 (+/-0.167) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.6269 (+/-0.157) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.4444 (+/-0.219) for {'kernel': 'poly', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.2409 (+/-0.280) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.2409 (+/-0.280) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.2409 (+/-0.280) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.2468 (+/-0.294) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.2795 (+/-0.278) for {'kernel': 'sigmoid', 'C': 1, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.2409 (+/-0.280) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.2409 (+/-0.280) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.2409 (+/-0.280) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.2269 (+/-0.264) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.2503 (+/-0.250) for {'kernel': 'sigmoid', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.2047 (+/-0.291) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.2047 (+/-0.291) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.2047 (+/-0.291) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.2117 (+/-0.284) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.3860 (+/-0.276) for {'kernel': 'sigmoid', 'C': 100, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}
0.1380 (+/-0.255) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}
0.1380 (+/-0.255) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.1, 'class_weight': 'balanced'}
0.1380 (+/-0.255) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.01, 'class_weight': 'balanced'}
0.1602 (+/-0.260) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.001, 'class_weight': 'balanced'}
0.4526 (+/-0.205) for {'kernel': 'sigmoid', 'C': 1000, 'decision_function_shape': 'ovo', 'gamma': 0.0001, 'class_weight': 'balanced'}

The Best Training Score was: 0.7005847953216374

The Best Parameters were:  {'kernel': 'rbf', 'C': 10, 'decision_function_shape': 'ovo', 'gamma': 1, 'class_weight': 'balanced'}

0.5813953488372093 is the models test accuracy part 3
Completion Time: 84.0543859005

Process finished with exit code 0
```

The results for the Balanced one vs one were:

Hyperparameters RBF , C=10 , gamma = 1

The best training score resulted to be 70.05% and the testing accuracy to be 58.14% which is the lowest out of all of the experiments. This makes sense as there was some bias to the weights, yet the results of this bias of course show more in the test accuracies than in the training accuracies. This again helps to show that classifiers should be properly balanced in order to predict the proper population in the given situation. Making sure that that we have the proper amount of bias and variance is important. To talk about variance it can be noted that indeed the answers vary alot yet the optimal hyperparameters does not vary too much. For example in parts 1 & 2 the hyperparameters were the same for the optimal setting. It was not until the last part where the hyperparameters changed from C = 100 to C =10, and gamma = .1 to gamma = 1, while the kernal remains the same. The also helps to show the low variance in the models and more of a higher bias.

Laslty it can be noted that the time for this process to complete took a lot longer than the other two processes. This may because of the function svm.SVC() is changing the weights of the class to be balanced. So in doing so this takes more processing time.

```
0.5813953488372093 is the models test accuracy part 3
Completion Time: 84.0543859005
```