

# Seeds Classification – SVM and Perceptron

## Group 3, Assignment 1, Multi-Class classification

Ahmed Shehata Mahmoud AboMoustafa

Sarah Hossam Abdel-Hameed Elmowafy

### Introduction:

The goal of Multi-Class Classification algorithms is to assign a class label for each input, some algorithms are basically designed for Binary classification. However, they can be indirectly extended to the Multi-Class case and this what will be applied here using SVM and Perceptron

### Data Set:

Data is the basic pillar here so we didn't save any effort to understand its nature and collect any piece of information that might help us building the model.

Seeds data set, numerical data with 169 instances (0 - 168) with two features and one target.

	0	1	2
0	14.84	2.221	1
1	14.09	2.699	1
2	13.94	2.259	1
3	14.99	1.355	1
4	14.49	3.586	1

The target has three values (1 “*Kama*”, 2 “*Rosa*”, 3 “*Canadian*”), for 1 and 3 there're 58 values, and 53 for 2, All of this for Training data. for Testing data we have 42 values with different targets.

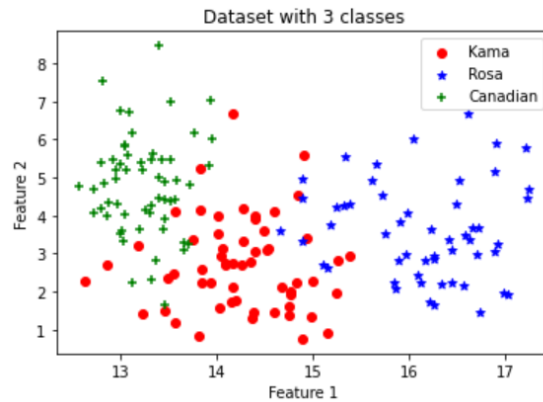
```
1    58
3    58
2    53
Name: 2, dtype: int64
```

And this is the discription of the data just to figure out how does it look like.

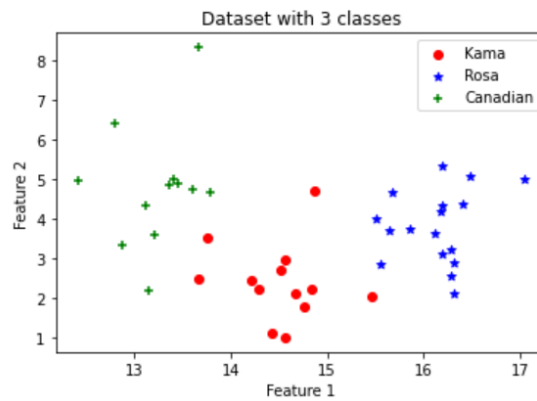
	0	1	2
count	169.000000	169.000000	169.000000
mean	14.500592	3.702427	2.000000
std	1.304375	1.520625	0.830949
min	12.570000	0.765100	1.000000
25%	13.410000	2.587000	1.000000
50%	14.180000	3.597000	2.000000
75%	15.380000	4.825000	3.000000
max	17.250000	8.456000	3.000000

## Data Plotting

In classification problems like SVM or Perceptron, the data need to be plotted to see if it's linearly separable or not, and in this case it seems to be linearly separable with acceptable accuracy.

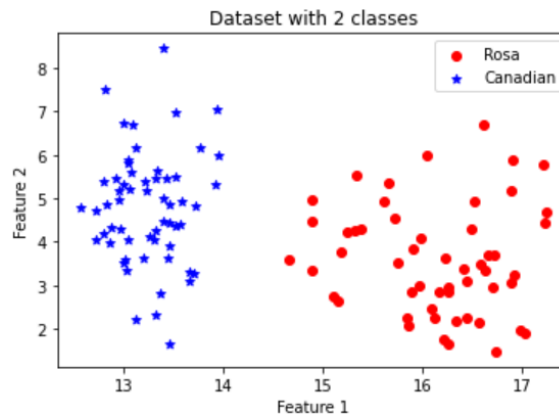


This is the test set that will be used to check the accuracy of the model that's going to be used later.



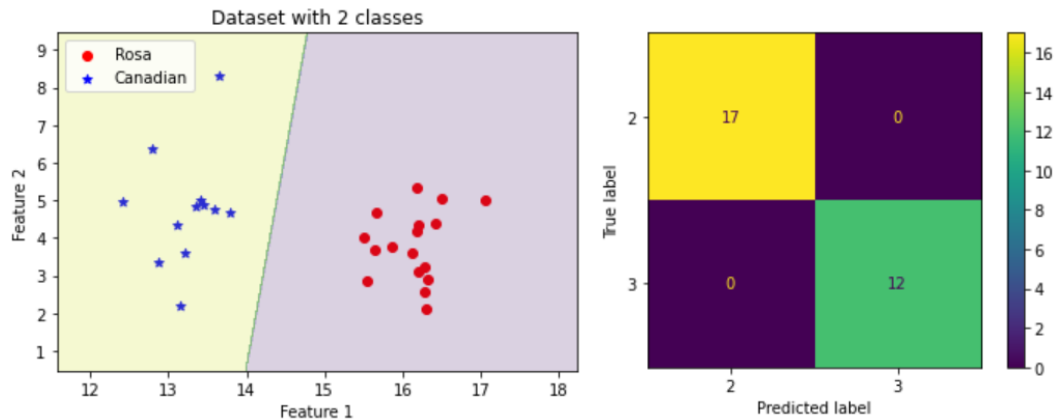
## Binary Classification

After dropping the Kama data, we will have two classes to be classified, the plot shows that the data is linearly separable, so let's see what our models will act like.



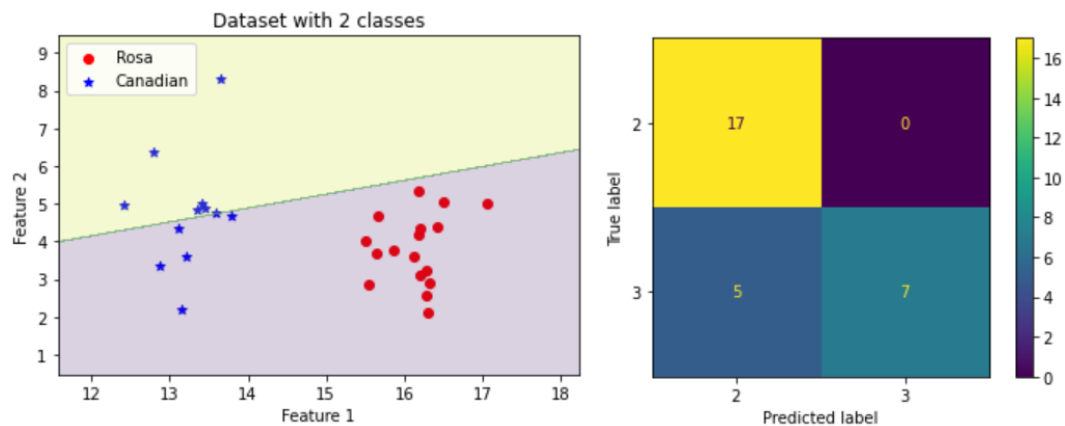
## SVM One vs One

Support Vector Machines separates the data perfectly with 100% accuracy with test data.



## Perceptron One vs One

Perceptron classifier didn't manage to draw the correct line between the two classes in the training data, that's why it shows worse accuracy with the testing points.



## Multi-class Classification

Multi-class classification is those tasks where examples are assigned exactly one of more than two classes, some algorithms are designed for this, but some are designed for binary classification like SVM and Perceptron. as such they cannot be used for multi-class classification directly.

Instead, heuristic methods can be used to split them into multiple binary classification datasets and train a binary classification model for each one. We used one of them is **One-vs-One (OvO)**, that we used before, after dropping one class "Kama" and the other is **One-vs-Rest (OvR)** that we are going to use with our models SVM and Perceptron after turning data into multiple binary classes.

## Data Binarization

As mentioned before, data need to be split into multiple binary classes first.

```
# method to make data binary
def binary_y(df_train, df_test):
    for i in df_train[2].unique():
        df_train['y{}'.format(i)] = df_train[2].apply(lambda x: 1 if x == i else 0)
        df_test['y{}'.format(i)] = df_test[2].apply(lambda x: 1 if x == i else 0)

    return df_train, df_test

df_train_ovr, df_test_ovr = binary_y(seeds_train_ovr, seeds_test_ovr)
```

This code turns the data from (1, 2, 3) labels as whole into 2 labels for each one. for example, data belongs to 1 class will be labeled as 1, otherwise will be 0, and the same with the two other labels so are going to have three more columns for each label as shown below.

	0	1	2	y1	y2	y3
0	14.84	2.221	1	1	0	0
1	14.09	2.699	1	1	0	0
...	...	...	...	...	...	...
164	13.41	8.456	3	0	0	1
165	13.47	3.919	3	0	0	1

Now, after we have the data binarized, it is ready to be classified using SVM or Perceptron OvR.


## SVM One vs Rest

One vs Rest will combine One vs One for all the three classes. let's talk about it in details.

### First step:

the model is trained on the training data separately using OvO methodology, first it trains the two features with y1, then it does the same with the two other classes y2 and y3, In the testing phase the outcome will be the probability by which the point is classified as class 1 or 0.

For example, the first line shows that the model predicts the point to be in class 1 with more than 0.97 probability and belongs otherwise with about 0.024 for y1, so we take the probability of being in class 1.

[0.97603808 0.02396192]		0.97603808
[0.60256673 0.39743327]		0.60256673
[0.31049698 0.68950302]		0.31049698

## Second step:

now we have the results of the three probability lists for each class y1, y2 and y3 respectively, so we will compare between them and choose which model has the higher probability.

for the first point y1 predicts it to be in “1” class with more than 0.97 and y3 with .13 but y2 with more than .99 prob. so the model will choose the highest probability, and this means that it will predict it belongs to “2” class.

y1 (1)	y2 (2)	y3 (3)
0.97603808	9.99872005e-01	1.35179849e-07
0.60256673	9.99946308e-01	3.19789557e-02
0.31049698	9.95674778e-01	9.38321539e-01

The output for these three points will be like [2, 2, 2]

But sometimes the output is inaccurate because the probabilities of two or more classes may be all high or low and this makes it hard to choose the best. in this case, classes one and two are both higher than 0.9 for the first point and here misclassification comes in.

The maximum probability here can be calculated using argmax or any code that does the same.

```
def Model_SVM(df_train_ovr, df_test_ovr):
    tmp = []
    svm_models = []
    for i in range(3):
        x_train = df_train_ovr.iloc[:, 0:2].to_numpy()
        y_train = df_train_ovr.iloc[:, (i+3)].to_numpy()

        x_test = df_test_ovr.iloc[:, 0:2].to_numpy()
        y_test = df_test_ovr.iloc[:, (i+3)].to_numpy()

        svm_model = svm.SVC(kernel='linear', decision_function_shape='ovo', probability=True)
        svm_model = svm_model.fit(x_train, y_train)

        svm_models.append(svm_model)

    svm_pred = svm_model.predict(x_test)

    print('Accuracy of SVM model: {:.2f}'.format(getAccuracy(svm_model, x_test, y_test)))
    print(" ")
    print('\nConfusion Matrix:\n')
    print(confusion_matrix(y_test, svm_pred))

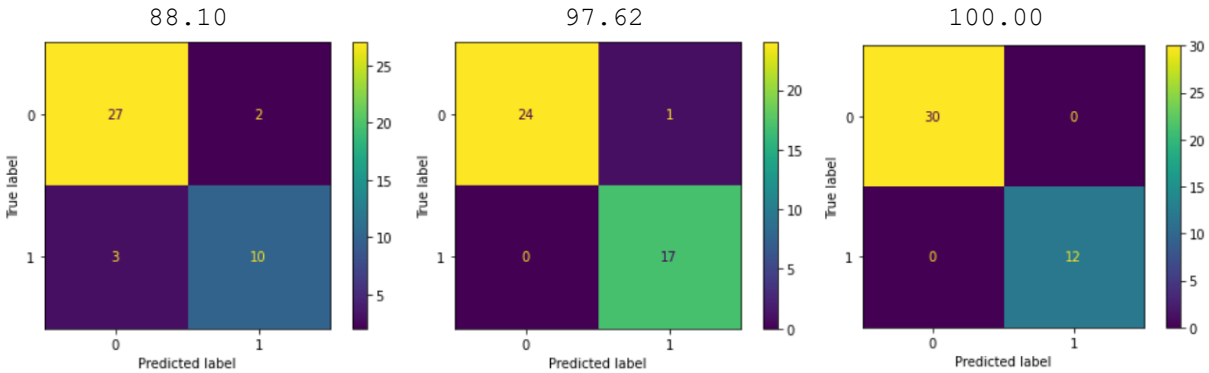
    print('\nClassification Report:\n')
    print(classification_report(y_test, svm_pred))

    plot_confusion_matrix(svm_model, x_test, y_test, xticks_rotation='horizontal')

    tmp.append(svm_model.predict_proba(x_test))
    t = svm_model.predict_proba(x_test)
    return tmp, svm_models
```

This function for SVM model takes the training and testing data as inputs and in range of 3 for the three labels y1, y2 and y3 it trains and fits the model then it predicts on the testing data and print the confusion matrix and the classification report.

The accuracy and confusion matrices for y1, y2 and y3 respectively:



Classification report for y1:

Classification Report:

	precision	recall	f1-score	support
0	0.90	0.93	0.92	29
1	0.83	0.77	0.80	13
accuracy			0.88	42
macro avg	0.87	0.85	0.86	42
weighted avg	0.88	0.88	0.88	42

This function will be used to calculate the final accuracy of the three models aggregated comparing the predictions with the real targets.

```
def get_final_acc(act, pred):
    correct = []
    errors = []
    for i in zip(act, pred):
        if i[1] == i[0]:
            correct.append(i)
        else:
            errors.append(i)
    acc = len(correct)/len(act)
    return correct, errors, acc
```

This is the final step aggregating predictions using argmax and getting final accuracy using the last function and printing the confusion matrix and accuracy of the final model.

```
predictions_svm=[]
for i in zip(ls_svm_1, ls_svm_2, ls_svm_3):
    x = np.argmax(i)
    predictions_svm.append(x+1)
print(predictions_svm)

actuals = seeds_test.iloc[:, 2]

print('\nConfusion Matrix:\n')
print(confusion_matrix(actuals, predictions_svm))

correct, errors, accuracy = get_final_acc(actuals, predictions_svm)
print(" ")
print("Final Accuracy: ", accuracy)
```

The final output:

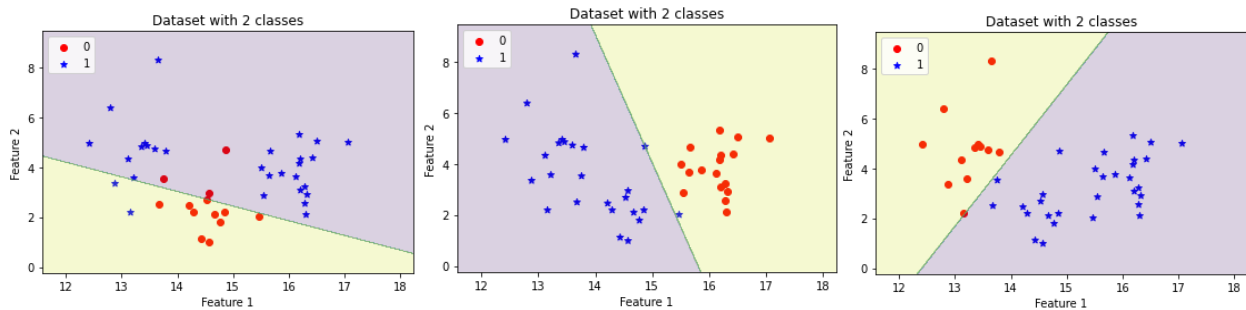
```
[1, 2, 2, 1, 1, 2, 1, 2, 3, 3, 2, 3, 1, 3, 2, 2, 2, 1, 3, 1, 3, 1, 2, 3,
2, 1, 1, 3, 2, 2, 2, 3, 3, 1, 2, 2, 2, 3, 2, 1, 2, 2]
```

Confusion Matrix:

```
[[11  2  0]
 [ 0 17  0]
 [ 1  0 11]]
```

Final Accuracy: 0.9285714285714286

The plots for the three binarized models with y1, y2 and y3:

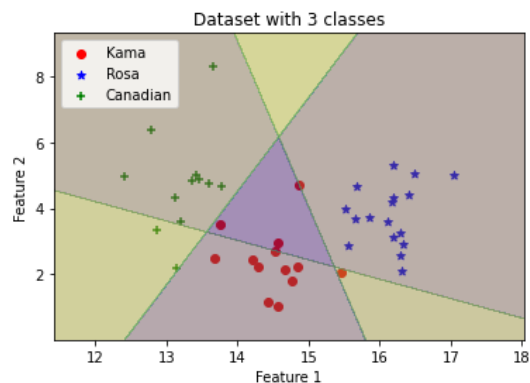


This function shows the three models combined as follows:

```
cls_final = [1, 2, 3]
class_names_final = getClassNames(c_names)

plotData(X_test_all, y_test_all, cls_final, class_names_final)
plotRegions(svm_models[0], X_test_all)
plotRegions(svm_models[1], X_test_all)
plotRegions(svm_models[2], X_test_all)
plt.legend(loc= "upper left")
plt.show()
```

Every line of these three separates one class from the rest as the plot depicts below and this illustrate the one vs rest methodology on the test data set.



## Perceptron One vs Rest

Perceptron has almost all the steps that have been applied in SVM, the below function for Perceptron model takes the training and testing data as inputs and in range of 3 for the three labels y1, y2 and y3 it trains and fits the model then it predicts on the testing data and print the confusion matrix and the classification report.

```
def Model_Perceptron(df_train_ovr, df_test_ovr):
    tmp = []
    prec_models = []

    for i in range(3):
        x_train = df_train_ovr.iloc[:, 0:2].to_numpy()
        y_train = df_train_ovr.iloc[:, (i+3)].to_numpy()

        x_test = df_test_ovr.iloc[:, 0:2].to_numpy()
        y_test = df_test_ovr.iloc[:, (i+3)].to_numpy()

        perceptron_model = Perceptron(alpha = 0.01, tol=1e-2, penalty= 'l2', max_iter = 4000
        ,n_iter_no_change = 8, validation_fraction = 0.3, random_state=0)
        prec_model = CalibratedClassifierCV(perceptron_model, method='sigmoid')
        prec_model_fitted = prec_model.fit(x_train, y_train)

        prec_models.append(prec_model_fitted)

        prec_pred = prec_model_fitted.predict(x_test)

        print('Accuracy of Perceptron model {}: {:.2f}'.format((i+1),getAccuracy(prec_model_fitted, x_test, y_test)))
        print(" ")
        print('\nConfusion Matrix:\n')
        print(confusion_matrix(y_test, prec_pred))

        print('\nClassification Report:\n')
        print(classification_report(y_test, prec_pred))

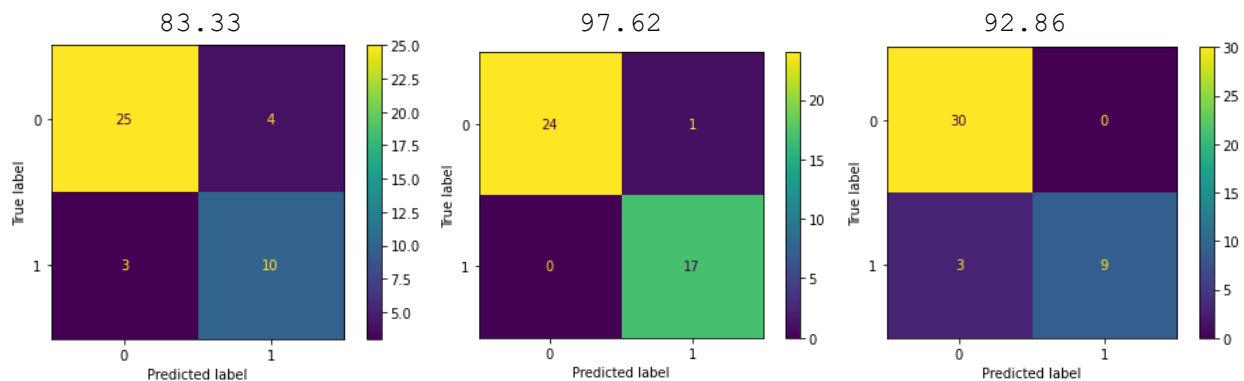
        plot_confusion_matrix(prec_model_fitted, x_test, y_test, xticks_rotation='horizontal')

    tmp.append(prec_model_fitted.predict_proba(x_test))

    return tmp, prec_models
```

We tuned the parameters of the perceptron model to get the best accuracy we could like setting the max\_iter to 4000 and Validation fraction to 0.3 and so on, but the Perceptron model accuracy overall is not good enough, the output is as follows:

The accuracy and confusion matrices for y1, y2 and y3 respectively:



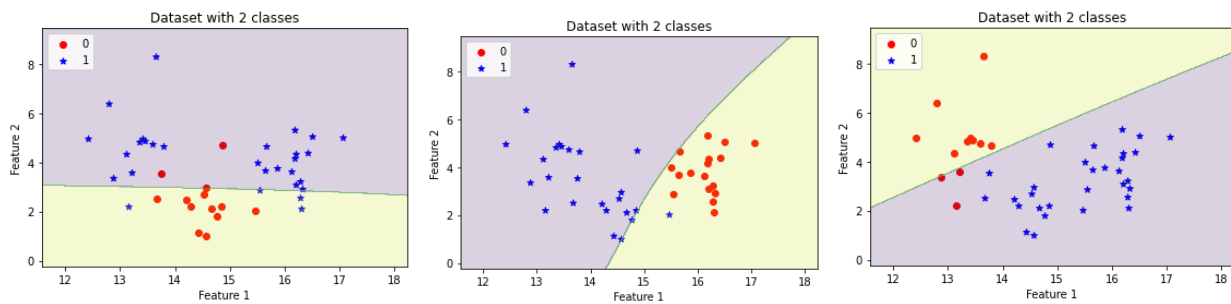


Classification report for y2:

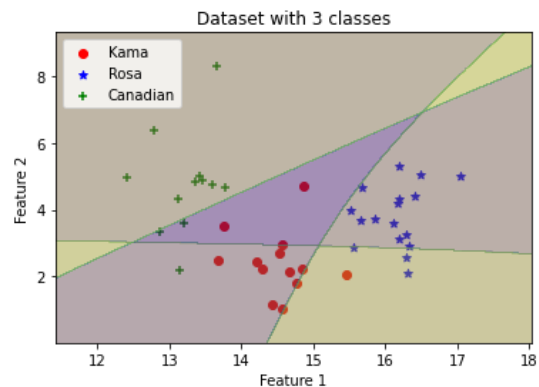
Classification Report:

	precision	recall	f1-score	support
0	1.00	0.96	0.98	25
1	0.94	1.00	0.97	17
accuracy			0.98	42
macro avg	0.97	0.98	0.98	42
weighted avg	0.98	0.98	0.98	42

The plots for the three binarized models with y1, y2 and y3:



Here is the model predicting on the test data sets making a lot of misclassifications out of the worst training phase.



The final output and Acc

[1, 2, 2, 1, 1, 2, 1, 2, 3, 3, 2, 3, 1, 3, 2, 2, 2, 1, 3, 1, 3, 1, 3, 3, 2, 1, 1, 3, 2, 2, 2, 3, 3, 1, 2, 2, 2, 1, 2, 1, 2, 2]

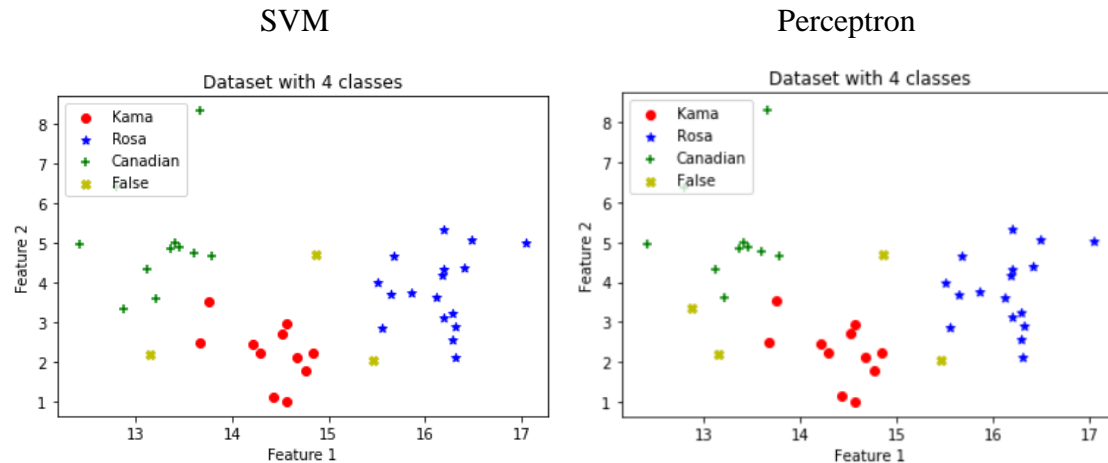
Confusion Matrix:

```
[[11  1  1]
 [ 0 17  0]
 [ 2  0 10]]
```

Final Accuracy: 0.9047619047619048

## True Label Wrong Label Predictions for SVM and Perceptron

The yellow crosses are the false predictions and it's just three in SVM, while in Perceptron they are 4 and this illustrates that SVM gives the higher accuracy and would be the perfect model to use here.



## Argmax function replacement

Instead of using the argmax function we may use this function that will do the same job which is returning the maximum value out of the three probabilities.

```
def max_value(ls1, ls2, ls3):  
    ls_predictions = []  
    for x, y, z in zip(ls1, ls2, ls3):  
        temp, num = x, 1  
        if (y > x):  
            temp = y  
            num = 2  
        if (z > temp):  
            temp = z  
            num = 3  
        ls_predictions.append(num)  
    return ls_predictions
```

## Our Strategy

Having this problem, we thought about building a strategy and it's as follows:

We are going to build on the SVM model as it shows higher probability but of course it's applicable on perceptron too. But let's check on SVM for now,

After having the values out of each model (y1 here) is like that:

```
[0.97603808 0.02396192]  
[0.60256673 0.39743327]  
[0.31049698 0.68950302]
```

Now we know that the first column is the probability that it belongs to class "1" and the second column is otherwise, so we will return the maximum value with threshold (0.6), If it is in the first column then it will be "1", if not, it will be -1. After that we will check if the probability is higher without threshold, and this will be used in the second case as we will illustrate in the example below

### Step 1:

These are the three lists of SVM binary classification, each of which has two columns for 1 (1, 2, 3) and 0 (otherwise).

<pre>ls_svm_1 = f1[0][:] ls_svm_2 = f1[1][:] ls_svm_3 = f1[2][:]</pre>		<pre>[0.97603808 0.02396192] [0.60256673 0.39743327] [0.31049698 0.68950302]</pre>
--	---	--

### Step 2:

```
def ls_back(ls):
    l = []
    temp = 0
    for y, j in ls:
        if (j > y and j > .6):
            temp = 1
        elif (y > j and y > .6):
            temp = -1
        elif (j > .5):
            temp = .5
        elif (y > .5):
            temp = -.5
        else:
            temp = 0
        l.append(temp)
    return l
```

This function takes each list as a parameter and check "y" which is the first column 1 and "j" the second column 0.

For the first list:

no	input	output
1	[0.41657677, 0.58342323],	.5
2	[0.97541643, 0.02458357],	-1
3	[0.59959479, 0.40040521],	-1
4	[0.3088149 , 0.6911851 ],	1
5	[0.5808871 , 0.4191129 ],	-.5
6	[0.85961112, 0.14038888],	-1
7	[0.20844619, 0.79155381],	1
8	[0.92868899, 0.07131101],	-1
9	[0.80955309, 0.19044691],	-1
10	[0.81253638, 0.18746362],	-1

Now let's see how it works

- Point 1 returns .5, this means it belongs to class "1" with prob less than the threshold (0.6)
- Point 2 returns -1, this means it belongs otherwise ("2" or "3") with prob higher than the threshold.
- Point 4 returns 1, this means it belongs to class "1" with prob higher than the threshold
- Point 5 returns -.5, this means it belongs otherwise with prob less than the threshold.


The actual output for test set is like that for each model.

```
[0.5, -1, -1, 1, -0.5, -1, 1, -1, -1, -1, -1, -1, 1, -1, -1, -1, -1, 1, -0.5, -0.5, -1, 1, -1, -1, -1, 1, 0.5, -1, -1, -1, 0.5, -1, -1, 1, -1, -1, -1, 0.5, -1, 1, -1, -1]
```

### Step 3:

Now after we have three lists like the above for each model, this function takes them and compare the values.

```
def Final(l1, l2, l3):
    final = []
    for x, y, z in zip(l1, l2, l3):
        if (y == 1):
            temp = 2
        elif (x == 1):
            temp = 1
        elif (z == 1):
            temp = 3
        elif (x == .5):
            temp = 1
        elif (y == .5):
            temp = 2
        elif (z == .5):
            temp = 3
        else:
            temp = 3
        final.append(temp)
    return final
```



no	l1	l2	l3	output
1	0.5	-1	-1	--> 1
2	-1	1	-1	--> 2
3	-1	1	-1	--> 2
4	1	-1	-1	--> 1
5	-0.5	-1	-1	--> 3
6	-1	1	-1	--> 2
7	1	-1	-0.5	--> 1
8	-1	1	-1	--> 2
9	-1	-1	1	--> 3
10	-1	-1	1	--> 3

the function compares the 3 values and give the output as follows:

- **Point number 1:** l1 indicates that the value belongs to class one with prob higher than .5 and less than the threshold (.6) but the other two classes saying that the point doesn't belong to them, so the output is "1".
- **Point number 2:** l1 and l3 saying that the value doesn't belong to them and point two says it belongs to it, so the output is "2"

- **Point number 4:** l1 says it belongs otherwise with prob less than the threshold and the other 2 classes saying it doesn't belong to them, in this case we use the default value as it's quite vague to the system to decide.
- **Point number 7:** l1 indicates that the value belongs to class one with prob higher than the threshold (.6), l2 says it belongs otherwise and l3 says it belongs otherwise with prob less than the threshold, the output is "1".

And so on, the final output is the predictions:

[1, 2, 2, 1, 3, 2, 1, 2, 3, 3, 2, 3, 1, 3, 2, 2, 2, 1, 3, 3, 3,  
1, 2, 3, 2, 1, 1, 3, 2, 2, 1, 3, 3, 1, 2, 2, 2, 3, 2, 1, 2, 2]

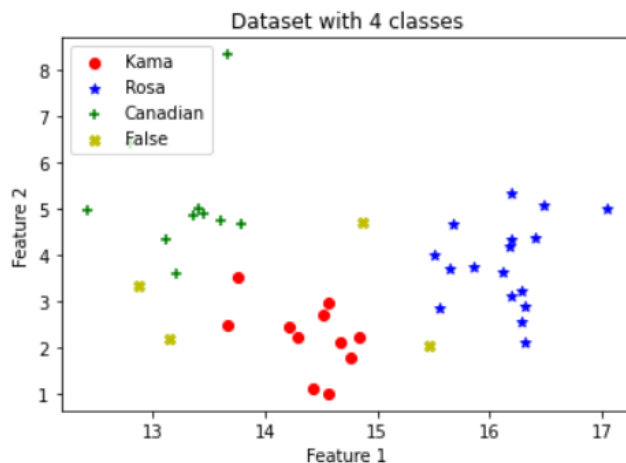
The accuracy and confusion matrix of our strategy:

Confusion Matrix:

```
[[10  1  2]
 [ 0 17  0]
 [ 1  0 11]]
```

Final Accuracy: 0.9047619047619048

The accuracy is more than 90%, which is lower than SVM but the same as Perceptron, Our strategy also misclassifies the same points that the perceptron misclassify.



These points reside in the critical areas between the classes that's why it's not easy for them to be correctly classified, and although we are satisfied with our strategy result but we will consider refine it in the future work.