Softmax Regression, One-vs-All and One-vs-One for Multi-class Classification

Estimated time needed: **1** hour

In this lab, we will study how to convert a linear classifier into a multi-class classifier, including multinomial logistic regression or softmax regression, One vs. All (One-vs-Rest) and One vs. One.

Objectives

After completing this lab you will be able to:

- Understand and apply some theory behind:
  - Softmax regression
  - One vs. All (One-vs-Rest)
  - One vs. One

Introduction

In Multi-class classification, we classify data into multiple class labels. Unlike classification trees and k-nearest neighbor, the concept of multi-class classification for linear classifiers is not as straightforward. We can convert logistic regression to multi-class classification using multinomial logistic regression or softmax regression; this is a generalization of logistic regression, this will not work for support vector machines. One vs. All (One-vs-Rest) and One vs. One are two other multi-class classification techniques can convert any two-class classifier into a multi-class classifier.

Install and Import the required libraries

For this lab, we are going to be using several Python libraries such as scit-learn, numpy, and matplotlib for visualizations. Some of these libraries might be installed in your lab environment, and others may need to be installed by you by removing the hash signs. The cells below will install these libraries when executed.

In [1]:
```
!pip install scikit-learn==1.0.2
```

```
Collecting scikit-learn==1.0.2
  Downloading scikit_learn-1.0.2-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.w
hl (24.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 24.8/24.8 MB 63.1 MB/s eta 0:00:0000:0100:
01
Requirement already satisfied: numpy>=1.14.6 in /home/jupyterlab/conda/envs/python/lib/p
ython3.7/site-packages (from scikit-learn==1.0.2) (1.21.6)
```

```
Requirement already satisfied: scipy>=1.1.0 in /home/jupyterlab/conda/envs/python/lib/py
thon3.7/site-packages (from scikit-learn==1.0.2) (1.7.3)
Requirement already satisfied: joblib>=0.11 in /home/jupyterlab/conda/envs/python/lib/py
thon3.7/site-packages (from scikit-learn==1.0.2) (1.3.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /home/jupyterlab/conda/envs/pytho
n/lib/python3.7/site-packages (from scikit-learn==1.0.2) (3.1.0)
Installing collected packages: scikit-learn
  Attempting uninstall: scikit-learn
    Found existing installation: scikit-learn 0.23.1
    Uninstalling scikit-learn-0.23.1:
      Successfully uninstalled scikit-learn-0.23.1
Successfully installed scikit-learn-1.0.2
```

In [2]:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import pandas as pd
```

Utility Function

This function plots a different decision boundary.

In [3]:

```
plot_colors = "ryb"
plot_step = 0.02

def decision_boundary (X,y,model,iris, two=None):
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, plot_step),
                         np.arange(y_min, y_max, plot_step))
    plt.tight_layout(h_pad=0.5, w_pad=0.5, pad=2.5)

    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    cs = plt.contourf(xx, yy, Z,cmap=plt.cm.RdYlBu)

    if two:
        cs = plt.contourf(xx, yy, Z,cmap=plt.cm.RdYlBu)
        for i, color in zip(np.unique(y), plot_colors):

            idx = np.where( y== i)
            plt.scatter(X[idx, 0], X[idx, 1], label=y,cmap=plt.cm.RdYlBu, s=15)
        plt.show()

    else:
        set_={0,1,2}
        print(set_)
        for i, color in zip(range(3), plot_colors):
            idx = np.where( y== i)
            if np.any(idx):

                set_.remove(i)

                plt.scatter(X[idx, 0], X[idx, 1], label=y,cmap=plt.cm.RdYlBu, edgecolor
```

```
        for  i in set_:
            idx = np.where( iris.target== i)
            plt.scatter(X[idx, 0], X[idx, 1], marker='x',color='black')

        plt.show()
```

This function will plot the probability of belonging to each class; each column is the probability of belonging to a class and the row number is the sample number.

In [4]:
```
def plot_probability_array(X,probability_array):

    plot_array=np.zeros((X.shape[0],30))
    col_start=0
    ones=np.ones((X.shape[0],30))
    for class_,col_end in enumerate([10,20,30]):
        plot_array[:,col_start:col_end]= np.repeat(probability_array[:,class_].reshape(
        col_start=col_end
    plt.imshow(plot_array)
    plt.xticks([])
    plt.ylabel("samples")
    plt.xlabel("probability of 3 classes")
    plt.colorbar()
    plt.show()
```

In ths lab we will use the iris dataset, it consists of three different types of irises' (Setosa y=0, Versicolour y=1, and Virginica y=2), petal and sepal length, stored in a 150x4 numpy.ndarray.

The rows being the samples and the columns: Sepal Length, Sepal Width, Petal Length and Petal Width.

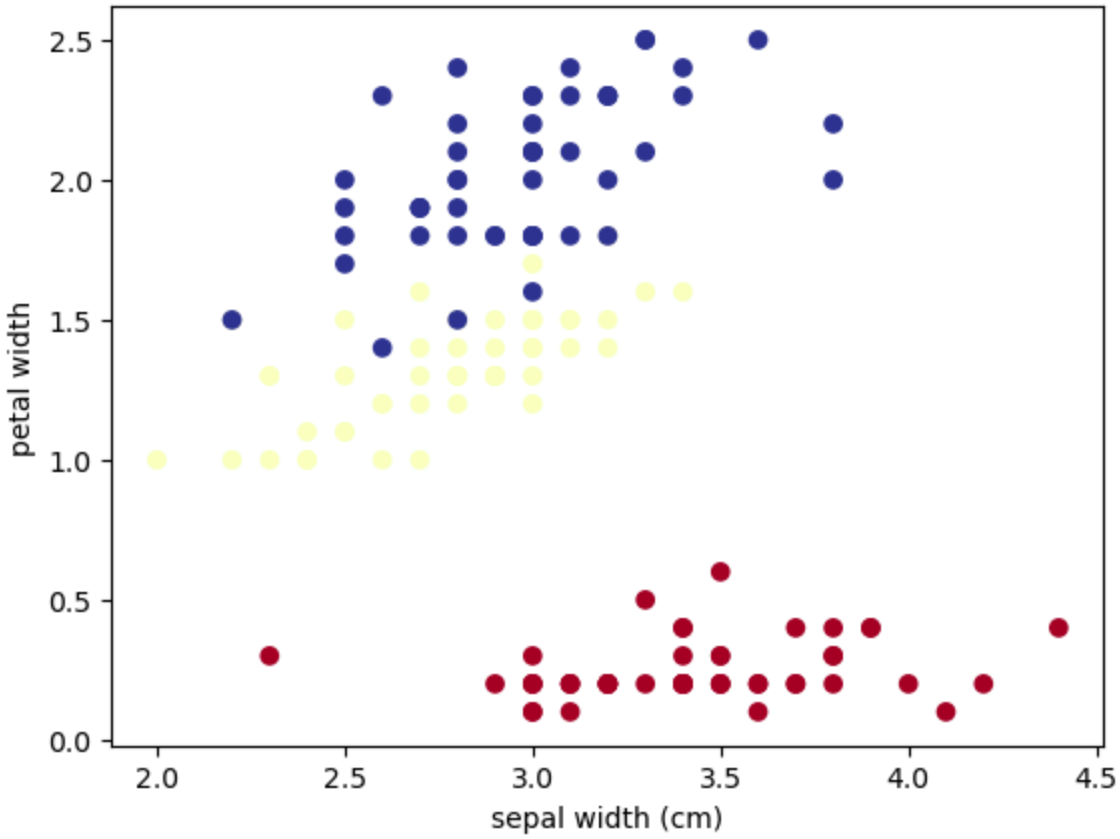
The following plot uses the second two features:

In [5]:
```
pair=[1, 3]
iris = datasets.load_iris()
X = iris.data[:, pair]
y = iris.target
np.unique(y)
```

Out[5]:
```
array([0, 1, 2])
```

In [6]:
```
plt.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.RdYlBu)
plt.xlabel("sepal width (cm)")
plt.ylabel("petal width")
```

Out[6]:
```
Text(0, 0.5, 'petal width')
```



Softmax Regression

SoftMax regression is similar to logistic regression, and the softmax function converts the actual distances, that is, dot products of x with each of the parameters θ_i for the K classes. This is converted to probabilities using the following:

$$softmax(x, i) = \frac{e^{\theta_i^T \mathbf{x}}}{\sum_{j=1}^{K} e^{\theta_j^T x}}$$

The training procedure is almost identical to logistic regression. Consider the three-class example where $y \in \{0, 1, 2\}$ we would like to classify x_1. We can use the softmax function to generate a probability of how likely the sample belongs to each class:

$$[softmax(x_1, 0), softmax(x_1, 1), softmax(x_1, 2)] = [0.97, 0.2, 0.1]$$

The index of each probability is the same as the class. We can make a prediction using the argmax function:

$$\hat{y} = argmax_i\{softmax(x, i)\}$$

For the previous example, we can make a prediction as follows:

$$\hat{y} = argmax_i\{[0.97, 0.2, 0.1]\} = 0$$

The `sklearn` does this automatically, but we can verify the prediction step, as we fit the model:

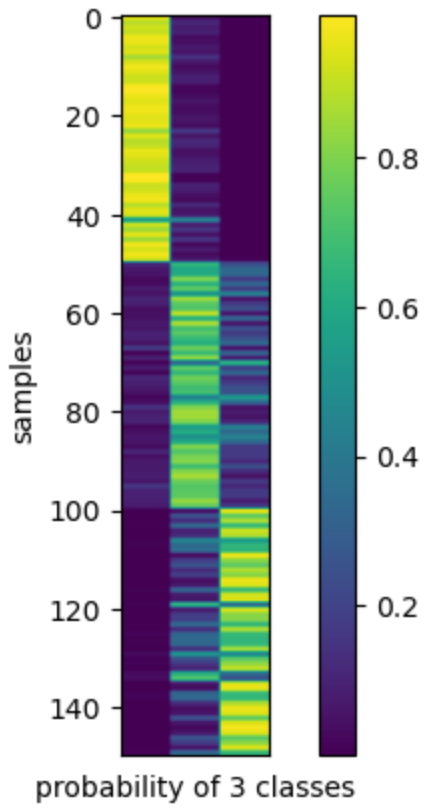
In [7]:
```
lr = LogisticRegression(random_state=0).fit(X, y)
```

We generate the probability using the method predict_proba</code>:

In [8]:
```
probability=lr.predict_proba(X)
```

We can plot the probability of belonging to each class; each column is the probability of belonging to a class and the row number is the sample number.

In [9]:
```
plot_probability_array(X,probability)
```



Here, is the output for the first sample:

In [10]:
```
probability[0,:]
```

Out[10]:
```
array([9.57671606e-01, 4.22321095e-02, 9.62845517e-05])
```

We see it sums to one.

In [11]:
```
probability[0,:].sum()
```

Out[11]:
```
1.0
```

We can apply the $argmax$ function.

In [12]:
```
np.argmax(probability[0,:])
```

Out[12]:
```
0
```

We can apply the $argmax$ function to each sample.

In [13]:
```
softmax_prediction=np.argmax(probability,axis=1)
softmax_prediction
```

Out[13]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2,
       2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

We can verify that sklearn does this under the hood by comparing it to the output of the method `predict` .

In [14]:
```
yhat =lr.predict(X)
accuracy_score(yhat,softmax_prediction)
```

Out[14]:
```
1.0
```

We can't use Softmax regression for SVMs, Let's explore two methods of Multi-class Classification that we can apply to SVM.

SVM

Sklean performs Multi-class Classification automatically, we can apply the method and calculate the accuracy. Train a SVM classifier with the `kernel` set to `linear` , `gamma` set to `0.5` , and the `probability` paramter set to `True` , then train the model using the `X` and `y` data.

In [15]:
```
model = SVC(kernel='linear', gamma=.5, probability=True)

model.fit(X,y)
```

Out[15]:
```
SVC(gamma=0.5, kernel='linear', probability=True)
```

▶ Click here for the solution

Find the `accuracy_score` on the training data.

In [16]:
```
yhat = model.predict(X)

accuracy_score(y,yhat)
```

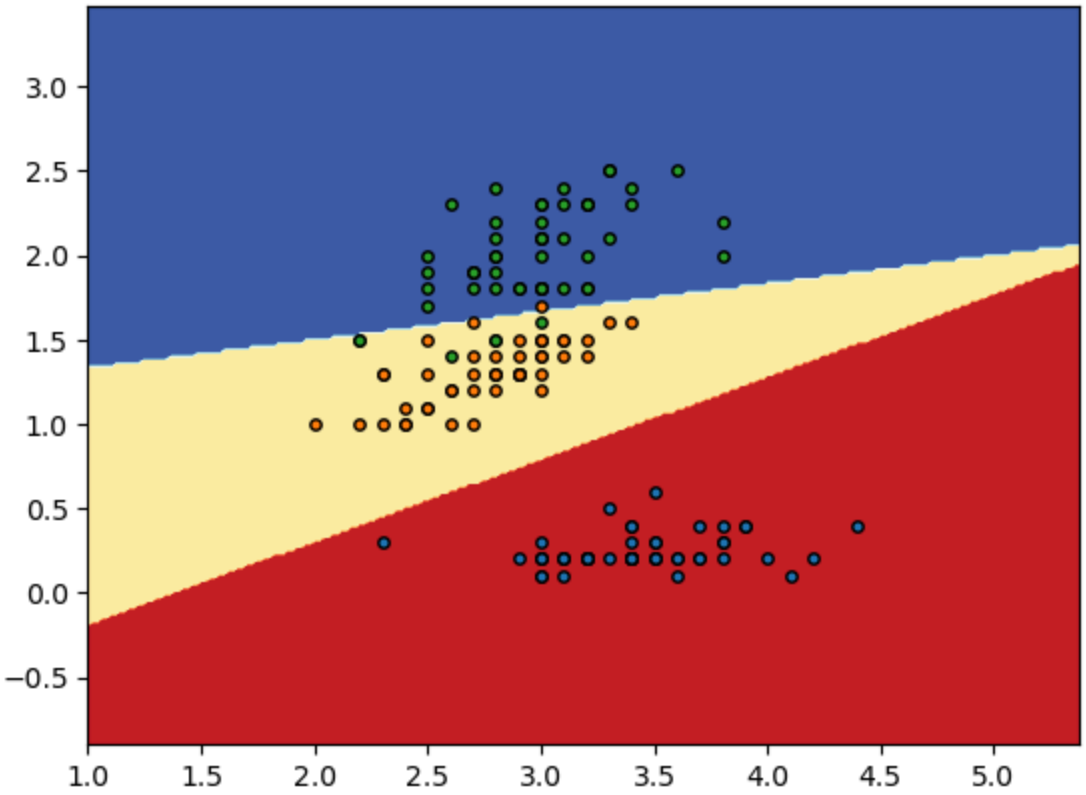
Out[16]:
```
0.96
```

▶ Click here for the solution

We can plot the decision_boundary.

In [17]:
```
decision_boundary (X,y,model,iris)
```

```
{0, 1, 2}
```



Let's implement One vs. All and One vs One:

One vs. All (One-vs-Rest)

For one-vs-all classification, if we have K classes, we use K two-class classifier models. The number of class labels present in the dataset is equal to the number of generated classifiers. First, we create an artificial class we will call this "dummy" class. For each classifier, we split the data into two classes. We take the class samples we would like to classify, the rest of the samples will be labelled as a dummy class. We repeat the process for each class. To make a classification, we use the classifier with the highest probability, disregarding the dummy class.

Train Each Classifier

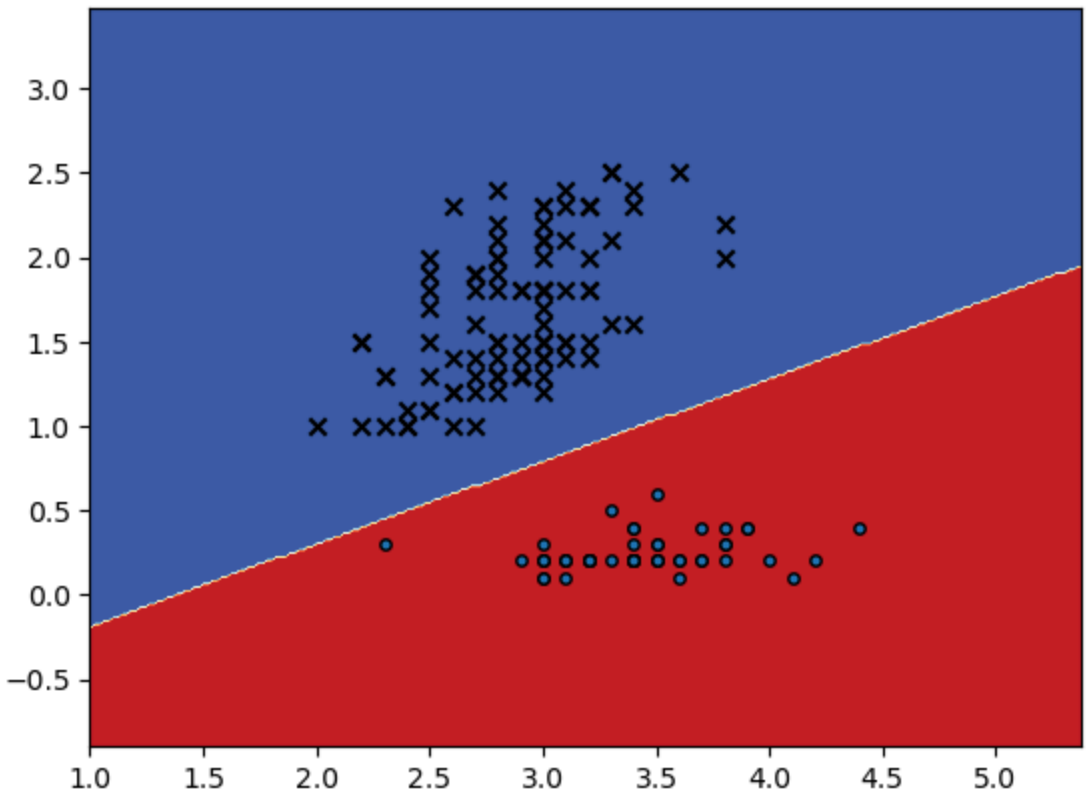
Here, we train three classifiers and place them in the list `my_models` . For each class we take the class samples we would like to classify, and the rest will be labelled as a dummy class. We repeat the process for each class. For each classifier, we plot the decision regions. The class we are interested in is in red, and the dummy class is in blue. Similarly, the class samples are marked in blue, and the dummy samples are marked with a black x.

In [18]:

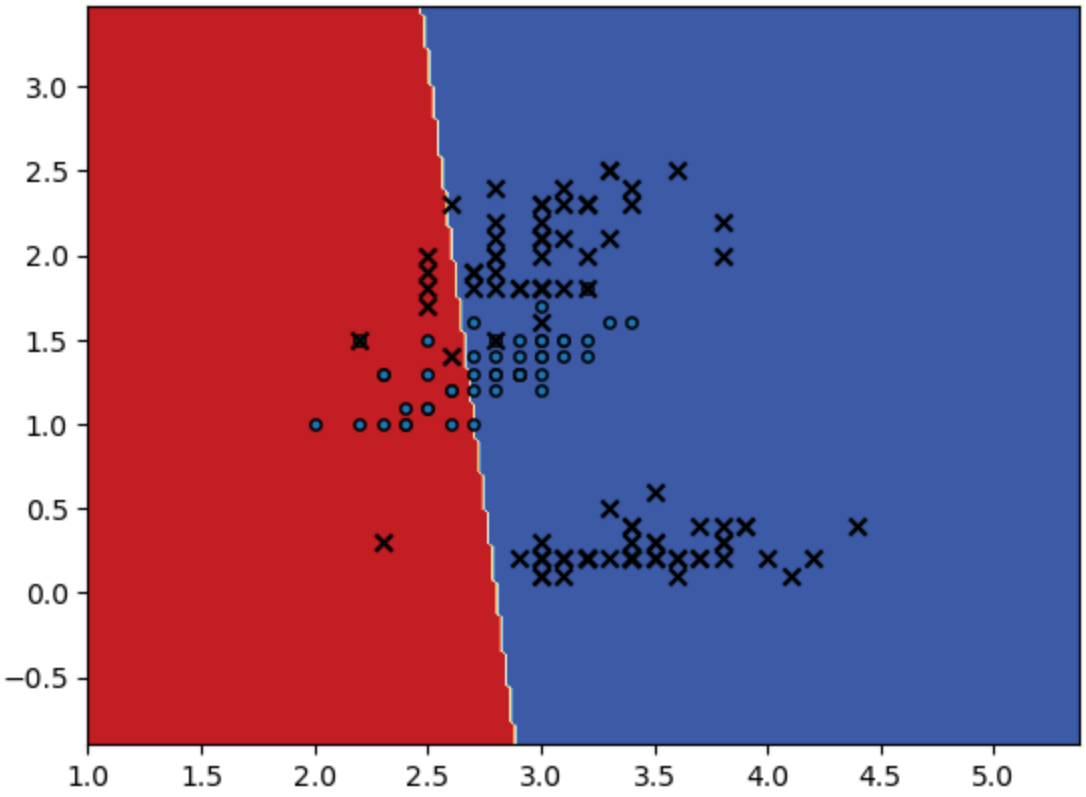
```
#dummy class
dummy_class=y.max()+1
#list used for classifiers
my_models=[]
#iterate through each class
for class_ in np.unique(y):
    #select the index of our  class
    select=(y==class_)
    temp_y=np.zeros(y.shape)
```

```
    #class, we are trying to classify
    temp_y[y==class_]=class_
    #set other samples  to a dummy class
    temp_y[y!=class_]=dummy_class
    #Train model and add to list
    model=SVC(kernel='linear', gamma=.5, probability=True)
    my_models.append(model.fit(X,temp_y))
    #plot decision boundary
    decision_boundary (X,temp_y,model,iris)
```

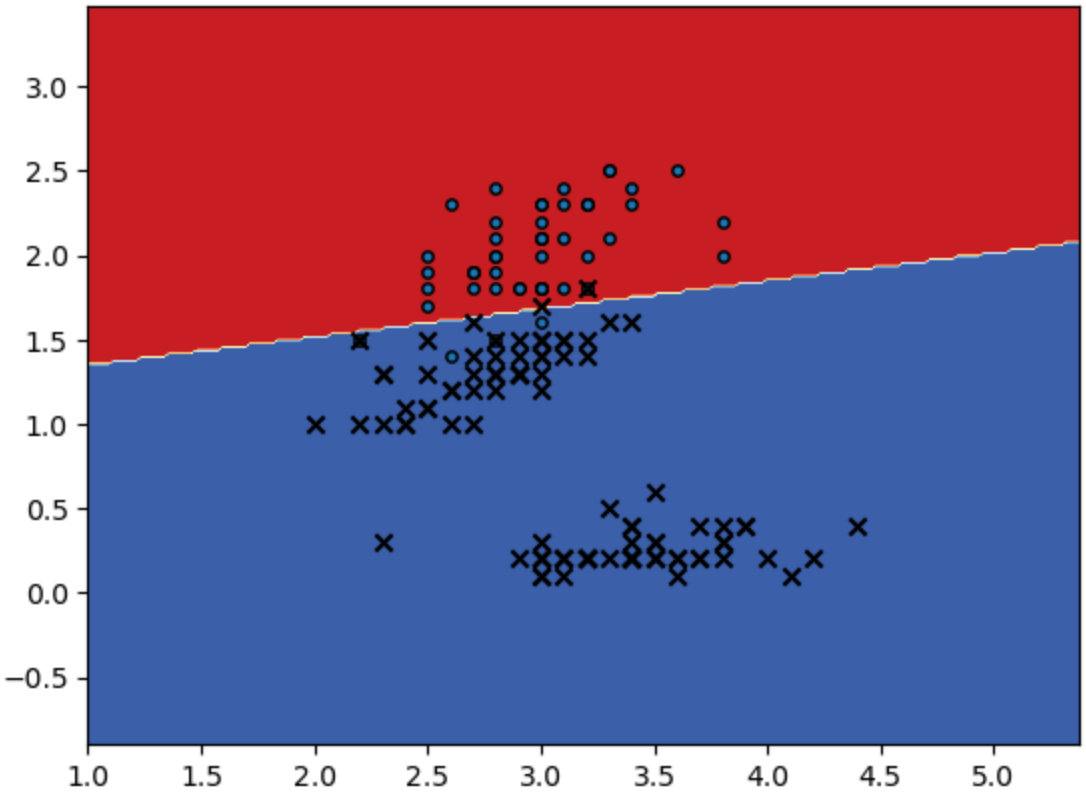
```
{0, 1, 2}
```




```
{0, 1, 2}
```




```
{0, 1, 2}
```



For each sample we calculate the probability of belonging to each class, not including the dummy class.

In [19]:

```
probability_array=np.zeros((X.shape[0],3))
for j,model in enumerate(my_models):
```

```
    real_class=np.where(np.array(model.classes_)!=3)[0]

    probability_array[:,j]=model.predict_proba(X)[:,real_class][:,0]
```

Here, is the probability of belonging to each class for the first sample.

In [20]:
```
probability_array[0,:]
```

Out[20]:
```
array([9.91054606e-01, 1.36406631e-01, 7.46328061e-12])
```

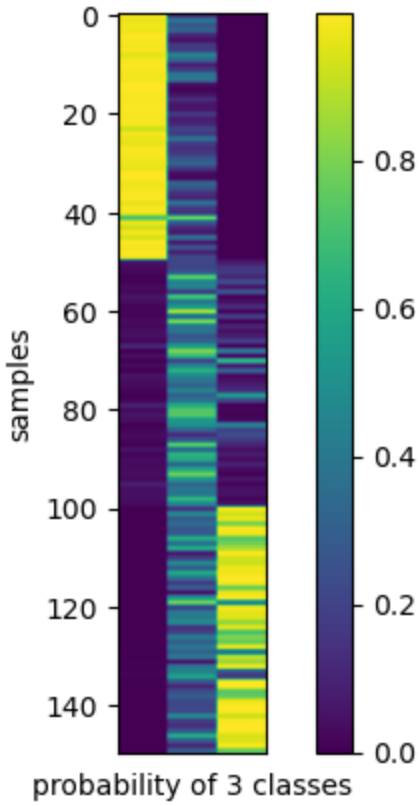
As each is the probability of belonging to the actual class and not the dummy class, it does not sum to one.

In [21]:
```
probability_array[0,:].sum()
```

Out[21]:
```
1.127461237070478
```

We can plot the probability of belonging to the class. The row number is the sample number.

In [22]:
```
plot_probability_array(X,probability_array)
```



We can apply the $argmax$ function to each sample to find the class.

In [23]:
```
one_vs_all=np.argmax(probability_array,axis=1)
one_vs_all
```

Out[23]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

We can calculate the accuracy.

In [24]:
```
accuracy_score(y,one_vs_all)
```

Out[24]: 0.9466666666666667

We see the accuracy is less than the one obtained by sklearn, and this is because for SVM, sklearn uses one vs one; let's verify it by comparing the outputs.

In [25]:
```
accuracy_score(one_vs_all,yhat)
```

Out[25]: 0.9733333333333334

We see that the outputs are different, now lets implement one vs one.

One vs One

In One-vs-One classification, we split up the data into each class, and then train a two-class classifier on each pair of classes. For example, if we have class 0,1,2, we would train one classifier on the samples that are class 0 and class 1, a second classifier on samples that are of class 0 and class 2, and a final classifier on samples of class 1 and class 2.

For K classes, we have to train $K(K-1)/2$ classifiers. So, if $K = 3$, we have $(3x2)/2 = 3$classes.

To perform classification on a sample, we perform a majority vote and select the class with the most predictions.

Here, we list each class.

In [26]:
```
classes_=set(np.unique(y))
classes_
```

Out[26]: {0, 1, 2}

Determine the number of classifiers:

In [27]:
```
K=len(classes_)
K*(K-1)/2
```

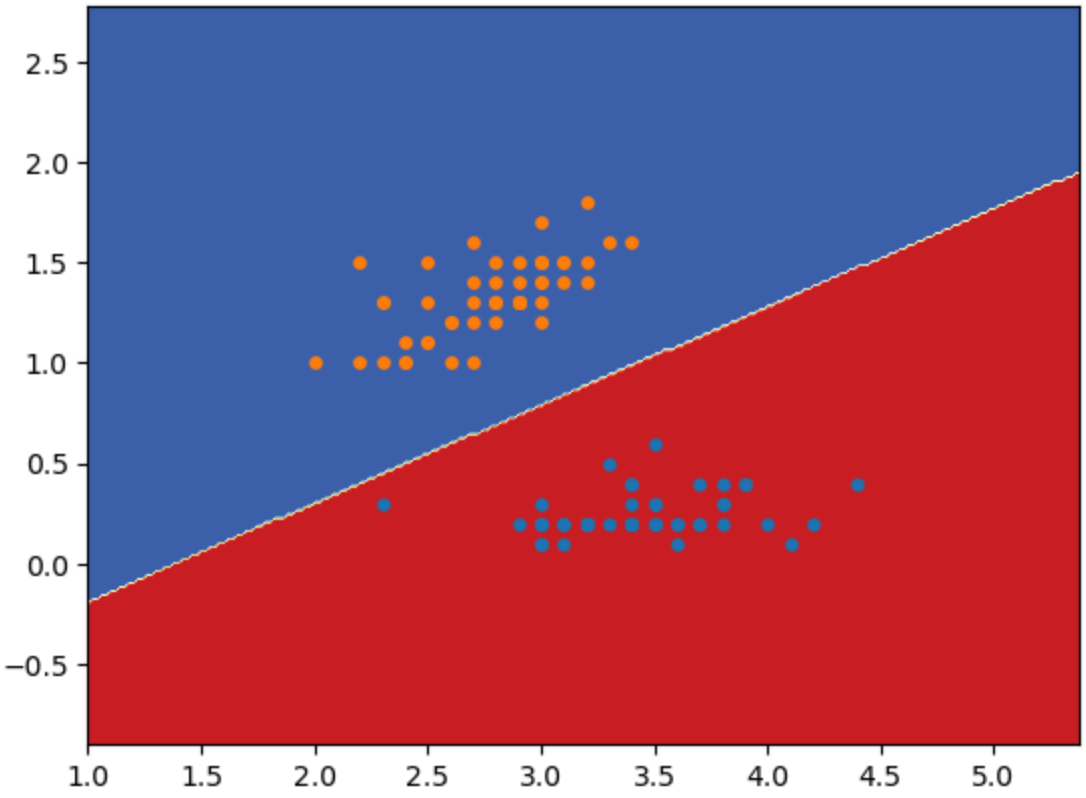
Out[27]: 3.0

We then train a two-class classifier on each pair of classes. We plot the different training points for each of the two classes.

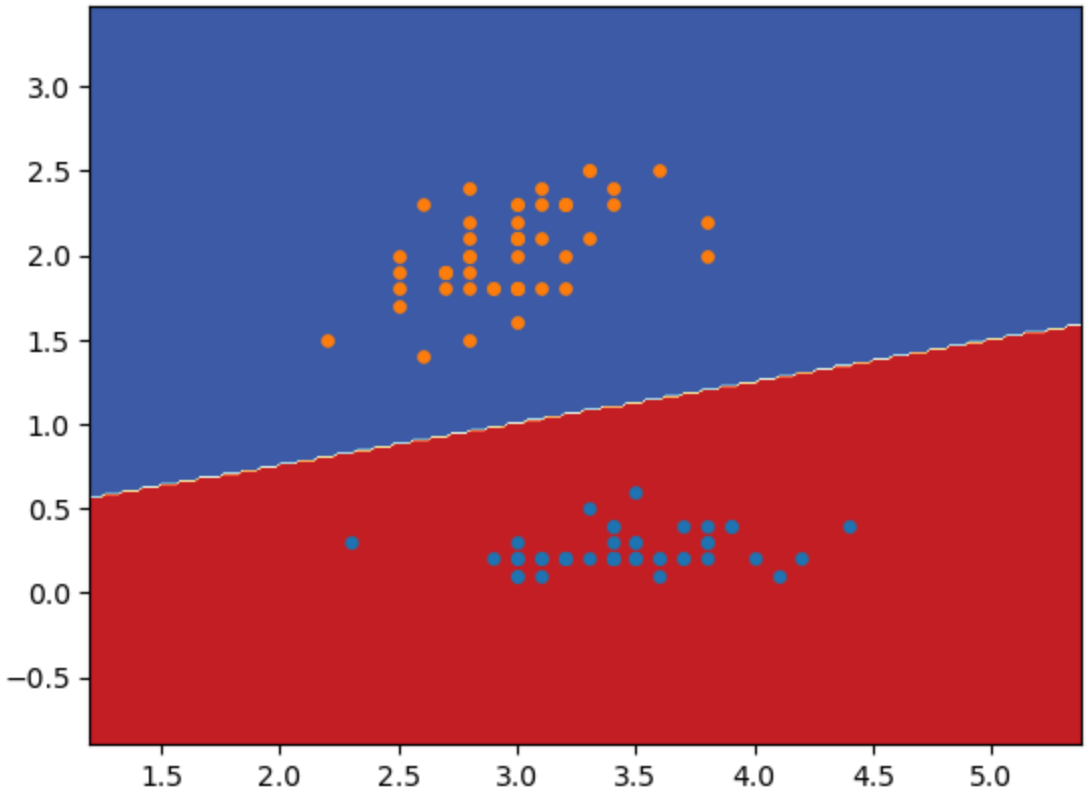
In [28]:
```
pairs=[]
left_overs=classes_.copy()
```

```
#list used for classifiers
my_models=[]
#iterate through each class
for class_ in classes_:
    #remove class we have seen before
    left_overs.remove(class_)
    #the second class in the pair
    for second_class in left_overs:
        pairs.append(str(class_)+' and '+str(second_class))
        print("class {} vs class {} ".format(class_,second_class) )
        temp_y=np.zeros(y.shape)
        #find classes in pair
        select=np.logical_or(y==class_ , y==second_class)
        #train model
        model=SVC(kernel='linear', gamma=.5, probability=True)
        model.fit(X[select,:],y[select])
        my_models.append(model)
        #Plot decision boundary for each pair and corresponding Training samples.
        decision_boundary (X[select,:],y[select],model,iris,two=True)
```

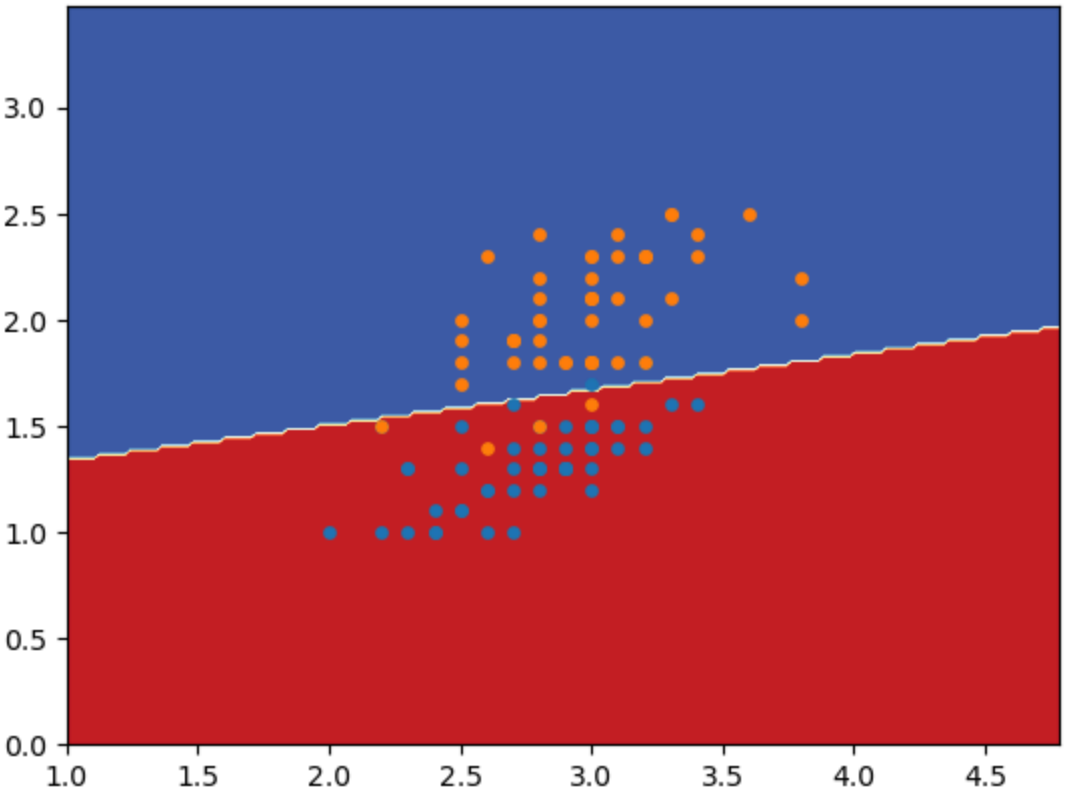
```
class 0 vs class 1
```




```
class 0 vs class 2
```



class 1 vs class 2



In [29]:
```
pairs
```

Out[29]:
```
['0 and 1', '0 and 2', '1 and 2']
```

As we can see, our data is left-skewed, containing more "5" star reviews.

Here, we are plotting the distribution of text length.

In [30]:
```
pairs
majority_vote_array=np.zeros((X.shape[0],3))
majority_vote_dict={}
for j,(model,pair) in enumerate(zip(my_models,pairs)):

    majority_vote_dict[pair]=model.predict(X)
    majority_vote_array[:,j]=model.predict(X)
```

In the following table, each column is the output of a classifier for each pair of classes and the output is the prediction:

In [31]:
```
pd.DataFrame(majority_vote_dict).head(10)
```

Out[31]:

	0 and 1	0 and 2	1 and 2
0	0	0	1
1	0	0	1
2	0	0	1
3	0	0	1
4	0	0	1
5	0	0	1
6	0	0	1
7	0	0	1
8	0	0	1
9	0	0	1

To perform classification on a sample, we perform a majority vote, that is, select the class with the most predictions. We repeat the process for each sample.

In [32]:
```
one_vs_one=np.array([np.bincount(sample.astype(int)).argmax() for sample  in majority_v
one_vs_one
```

Out[32]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 2, 2, 1, 2, 2,
       2, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2])
```

We calculate the accuracy:

In [33]:
```
accuracy_score(y,one_vs_one)
```

Out[33]: 0.96

If we compare it to `sklearn` , it's the same!

In [34]:
```
accuracy_score(yhat,one_vs_one)
```

Out[34]: 1.0

Author

Joseph Santarcangelo

Other Contributors

Azim Hirjani

<!--

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-07-20	0.2	Azim	Modified Multiple Areas
2020-07-17	0.1	Azim	Created Lab Template
2022-08-31	0.3	Steve Hord	QA pass edits

--!>