LAB04 - NAIVE BAYES CLASSIFICATION IN DATA

Naive Bayes Models

In this lab you will work with **naive Bayes models**. Naive Bayes models are a surprisingly useful and effective simplification of the general Bayesian models. Naive Bayes models make the naive assumption of statistical independence of the features. In many cases, naive Bayes module are surprisingly effective despite violating the assumption of independence.

In simple terms, naive Bayes models use empirical distributions of the features to compute probabilities of the labels. The naive Bayes models can use most any family of distributions for the features. It is important to select the correct distribution family for the data you are working with. Common cases are:

- **Gaussian;** for continuous or numerical features.
- **Multinomial;** for features with more than two categories.

These is one pit fall, the model fails if a zero probability is encountered. This situation occurs when there is a 'hole' in the sample space where there are no samples. A simple smoothing procedure can deal with this problem. The smoothing hyperparameter, usually called alpha, is one of the few required for naive Bayes models.

Some properties of naive Bayes models are:

- Computational complexity is linear in number of parameter/features, making naive Bayes models highly scalable. There are out or core approaches suitable for massive datasets.
- Requires minimal data to produce models that generalizes well. If there are only a few cases per category to train a model a naive Bayes model can be a good choice.
- Have a simple and inherent regularization.

Naive Bayes models are used in many situations including:

- Document classification
- SPAM detection
- Image classification

Example: Iris dataset

As a first example you will use a naive Bayes model to classify the species of iris flowers.

As a first step, execute the code in the cell below to load the required packages to run the rest of this notebook.

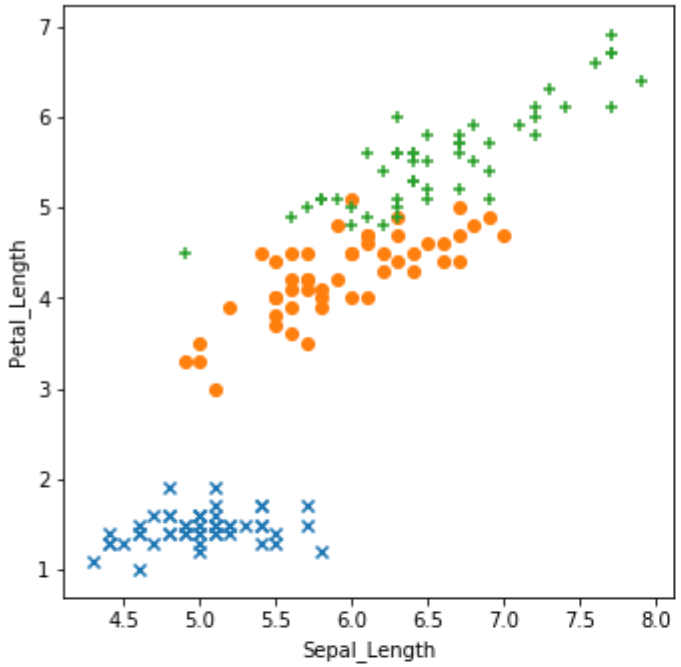
In [1]:
```
from sklearn import preprocessing
from sklearn.naive_bayes import GaussianNB, BernoulliNB
#from statsmodels.api import datasets
from sklearn import datasets ## Get dataset from sklearn
import sklearn.model_selection as ms
import sklearn.metrics as sklm
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import numpy.random as nr

%matplotlib inline
```

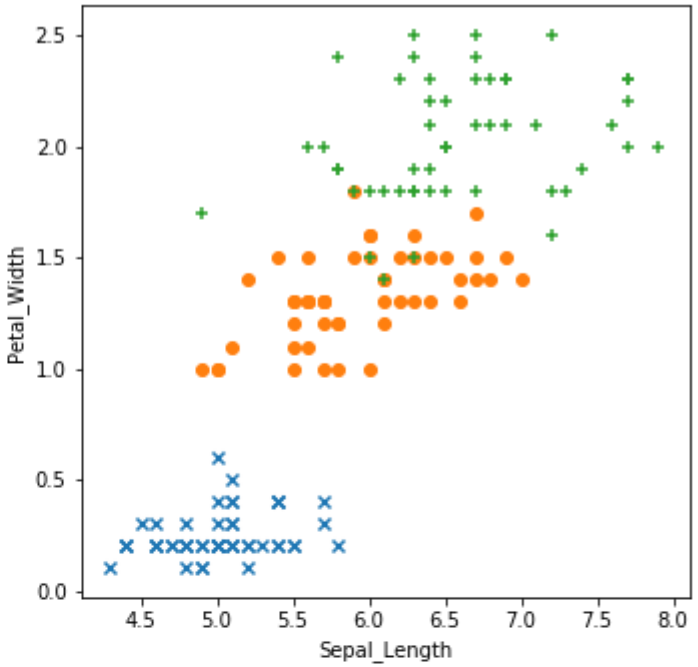
To get a feel for these data, you will now load and plot them. The code in the cell below does the following:

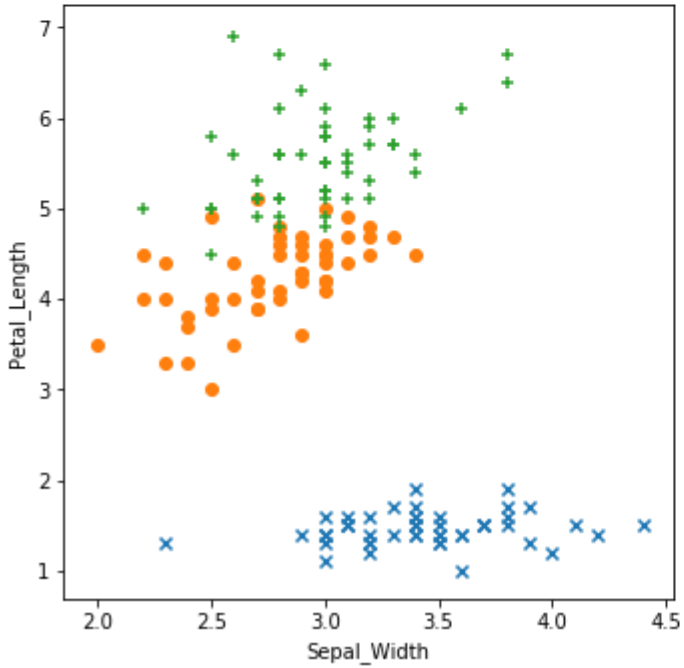
1. Loads the iris data as a Pandas data frame.
2. Adds column names to the data frame.
3. Displays all 4 possible scatter plot views of the data.

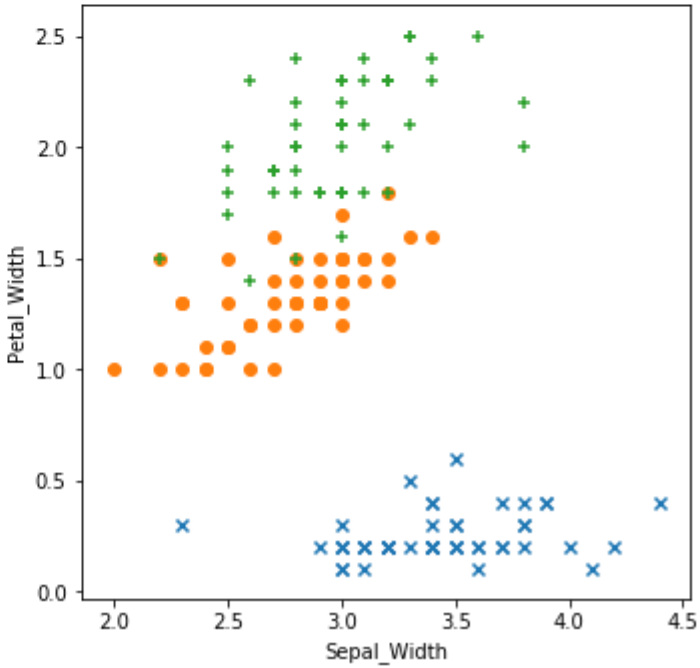
Execute this code and examine the results.

In [2]:
```
def plot_iris(iris):
    '''Function to plot iris data by type'''
    setosa = iris[iris['Species'] == 'setosa']
    versicolor = iris[iris['Species'] == 'versicolor']
    virginica = iris[iris['Species'] == 'virginica']
    fig, ax = plt.subplots(2, 2, figsize=(12,12))
    x_ax = ['Sepal_Length', 'Sepal_Width']
    y_ax = ['Petal_Length', 'Petal_Width']
    for i in range(2):
        for j in range(2):
            ax[i,j].scatter(setosa[x_ax[i]], setosa[y_ax[j]], marker = 'x')
            ax[i,j].scatter(versicolor[x_ax[i]], versicolor[y_ax[j]], marker = 'o')
            ax[i,j].scatter(virginica[x_ax[i]], virginica[y_ax[j]], marker = '+')
            ax[i,j].set_xlabel(x_ax[i])
            ax[i,j].set_ylabel(y_ax[j])

## Import the dataset from sklearn.datasets
iris = datasets.load_iris()

## Create a data frame from the dictionary
species = [iris.target_names[x] for x in iris.target]
iris = pd.DataFrame(iris['data'], columns = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width'])
iris['Species'] = species

## Plot views of the iris data
plot_iris(iris)
```


Petal_Length
Sepal_Length
1
2
3
4
5
6
7
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Petal_Width
Sepal_Length
0.0
0.5
1.0
1.5
2.0
2.5
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


Petal_Length
Sepal_Width
1
2
3
4
5
6
7
2.0
2.5
3.0
3.5
4.0
4.5


Petal_Width
Sepal_Width
0.0
0.5
1.0
1.5
2.0
2.5
2.0
2.5
3.0
3.5
4.0
4.5

You can see that Setosa (blue) is well separated from the other two categories. The Versicolor (orange) and the Virginica (green) show considerable overlap. The question is how well our classifier will separate these categories.

Scikit Learn classifiers require numerically coded numpy arrays for the features and as a label. The code in the cell below does the following processing:

1. Creates a numpy array of the features.
2. Numerically codes the label using a dictionary lookup, and converts it to a numpy array.

Execute this code.

In [3]:
```
Features = np.array(iris[['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width']])

levels = {'setosa':0, 'versicolor':1, 'virginica':2}
Labels =  np.array([levels[x] for x in iris['Species']])
```

Next, execute the code in the cell below to split the dataset into test and training set. Notice that unusually, 100 of the 150 cases are being used as the test dataset.

In [4]:
```
## Randomly sample cases to create independent training and test data
nr.seed(1115)
indx = range(Features.shape[0])
indx = ms.train_test_split(indx, test_size = 100)
X_train = Features[indx[0],:]
y_train = np.ravel(Labels[indx[0]])
X_test = Features[indx[1],:]
y_test = np.ravel(Labels[indx[1]])
```

In [6]:
```
display(X_train)
```

```
array([[6.1, 3. , 4.9, 1.8],
       [5. , 3.2, 1.2, 0.2],
       [6.8, 2.8, 4.8, 1.4],
       [4.9, 3.1, 1.5, 0.2],
       [6. , 2.7, 5.1, 1.6],
       [4.9, 3. , 1.4, 0.2],
       [5.4, 3.4, 1.7, 0.2],
       [6. , 2.2, 5. , 1.5],
       [6.2, 3.4, 5.4, 2.3],
       [5.6, 2.5, 3.9, 1.1],
       [7.6, 3. , 6.6, 2.1],
       [5.7, 3. , 4.2, 1.2],
       [6.3, 2.3, 4.4, 1.3],
       [6.7, 3.3, 5.7, 2.1],
       [5.2, 4.1, 1.5, 0.1],
       [6. , 3. , 4.8, 1.8],
       [6.4, 2.9, 4.3, 1.3],
       [5.8, 2.6, 4. , 1.2],
       [5.7, 2.8, 4.5, 1.3],
       [5.2, 3.4, 1.4, 0.2],
       [5.5, 3.5, 1.3, 0.2],
       [5.8, 2.7, 5.1, 1.9],
       [5.7, 2.6, 3.5, 1. ],
       [6. , 2.9, 4.5, 1.5],
       [5. , 3. , 1.6, 0.2],
       [4.8, 3.1, 1.6, 0.2],
       [5.1, 3.8, 1.5, 0.3],
       [5.1, 3.4, 1.5, 0.2],
       [6.3, 3.3, 6. , 2.5],
       [5.9, 3. , 5.1, 1.8],
       [4.8, 3. , 1.4, 0.1],
       [7.7, 3.8, 6.7, 2.2],
       [5.4, 3.9, 1.7, 0.4],
       [4.9, 2.5, 4.5, 1.7],
       [7.7, 3. , 6.1, 2.3],
       [6.9, 3.1, 5.1, 2.3],
       [5.5, 2.5, 4. , 1.3],
       [6.2, 2.8, 4.8, 1.8],
       [4.6, 3.4, 1.4, 0.3],
       [5. , 2. , 3.5, 1. ],
       [4.4, 3. , 1.3, 0.2],
       [7.1, 3. , 5.9, 2.1],
       [6.5, 3. , 5.8, 2.2],
```

```
[6.5, 2.8, 4.6, 1.5],
[7.7, 2.8, 6.7, 2. ],
[5.5, 2.6, 4.4, 1.2],
[5. , 2.3, 3.3, 1. ],
[5.8, 2.8, 5.1, 2.4],
[7.9, 3.8, 6.4, 2. ],
[5.5, 2.4, 3.8, 1.1]])
```

As is always the case with machine learning, numeric features must be scaled. The code in the cell below performs the following processing:

1. A Zscore scale object is defined using the `StandarScaler` function from the scikit-learn preprocessing package.
2. The scaler is fit to the training features. Subsequently, this scaler is used to apply the same scaling to the test data and in production.
3. The training features are scaled using the `transform` method.

Execute this code.

In [7]:
```
scale = preprocessing.StandardScaler()
scale.fit(X_train)
X_train = scale.transform(X_train)
```

Now you will define and fit a Gaussian naive Bayes model. A Gaussian model is appropriate here since all of the features are numeric.

The code in the cell below defines a Gaussian naive Bayes model object using the `GaussianNB` function from the scikit-learn naive_bayes package, and then fits the model. Execute this code.

In [8]:
```
NB_mod = GaussianNB()
NB_mod.fit(X_train, y_train)
```

Out[8]:
```
GaussianNB(priors=None, var_smoothing=1e-09)
```

Notice that the Gaussian naive Bayes model object has only one hyperparameter.

Next, the code in the cell below performs the following processing to score the test data subset:

1. The test features are scaled using the scaler computed for the training features.
2. The `predict` method is used to compute the scores from the scaled features.

Execute this code.

In [9]:
```
X_test = scale.transform(X_test)
scores = NB_mod.predict(X_test)
```

It is time to evaluate the model results. Keep in mind that the problem has been made deliberately difficult, by having more test cases than training cases.

The iris data has three species categories. Therefore it is necessary to use evaluation code for a three category problem. The function in the cell below extends code from previous labs to deal with a three category problem.

Execute this code, examine the results, and answer **Question 1** on the course page.

In [10]:
```
def print_metrics_3(labels, scores):

    conf = sklm.confusion_matrix(labels, scores)
    print('                 Confusion matrix')
    print('                 Score Setosa   Score Versicolor    Score Virginica')
    print('Actual Setosa      %6d' % conf[0,0] + '            %5d' % conf[0,1] + '             %5d' % conf[0,
2])
    print('Actual Versicolor  %6d' % conf[1,0] + '            %5d' % conf[1,1] + '             %5d' % conf[1,
2])
    print('Actual Vriginica   %6d' % conf[2,0] + '            %5d' % conf[2,1] + '             %5d' % conf[2,
2])
    ## Now compute and display the accuracy and metrics
    print('')
    print('Accuracy        %0.2f' % sklm.accuracy_score(labels, scores))
    metrics = sklm.precision_recall_fscore_support(labels, scores)
    print(' ')
    print('           Setosa  Versicolor  Virginica')
    print('Num case   %0.2f' % metrics[3][0] + '     %0.2f' % metrics[3][1] + '      %0.2f' % metrics[3][2])
    print('Precision   %0.2f' % metrics[0][0] + '      %0.2f' % metrics[0][1] + '       %0.2f' % metrics[0][2
])
    print('Recall      %0.2f' % metrics[1][0] + '      %0.2f' % metrics[1][1] + '       %0.2f' % metrics[1][2
])
    print('F1          %0.2f' % metrics[2][0] + '      %0.2f' % metrics[2][1] + '       %0.2f' % metrics[2][2
])

print_metrics_3(y_test, scores)
```

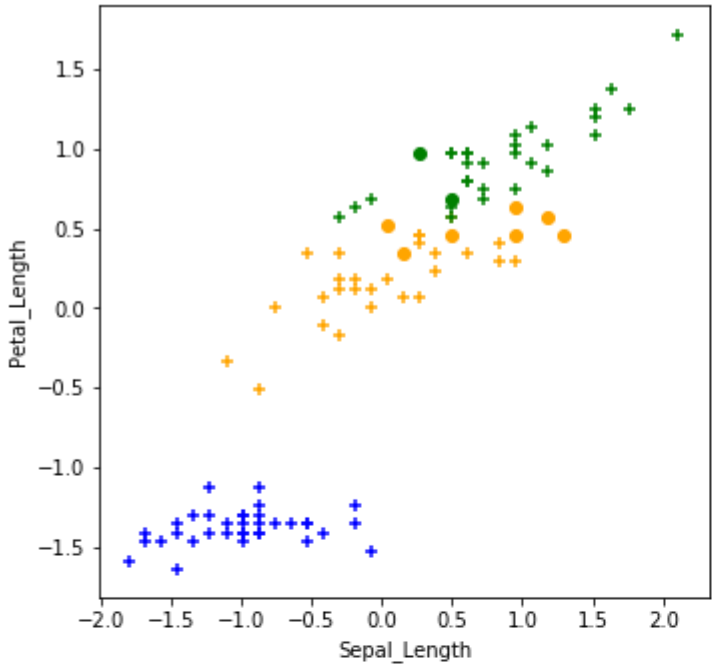
```
                 Confusion matrix
                 Score Setosa   Score Versicolor    Score Virginica
Actual Setosa          35                0                0
Actual Versicolor       0               27                7
Actual Vriginica        0                2               29

Accuracy        0.91

           Setosa  Versicolor  Virginica
Num case   35.00     34.00      31.00
Precision   1.00      0.93       0.81
Recall      1.00      0.79       0.94
F1          1.00      0.86       0.87
```

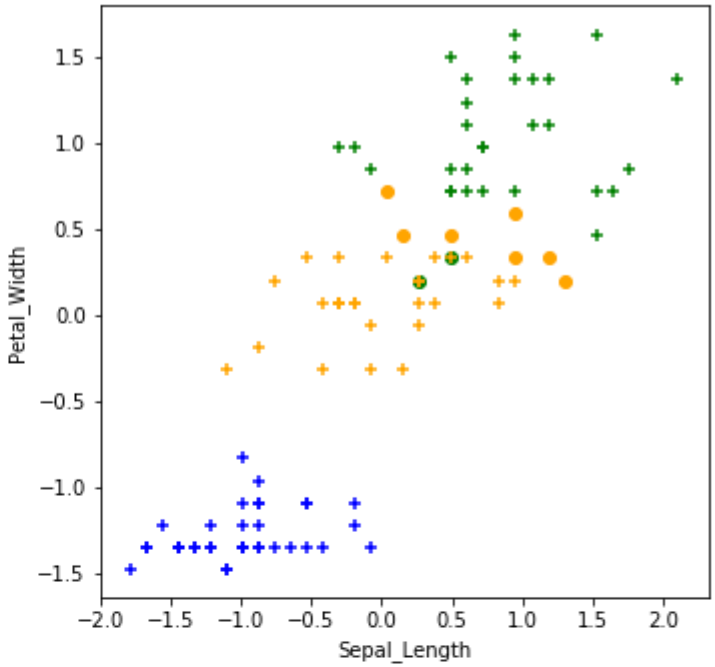
Examine these results. Notice the following:

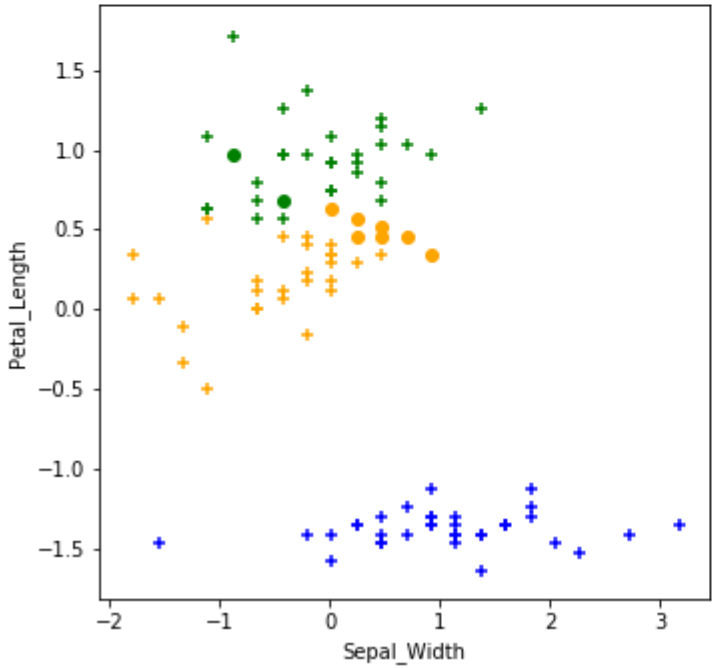
1. The confusion matrix has dimension 3X3. You can see that most cases are correctly classified.
2. The overall accuracy is 0.91. Since the classes are roughly balanced, this metric indicates relatively good performance of the classifier, particularly since it was only trained on 50 cases. As was mentioned previously, naive Bayes models require only small amounts of training data.
3. The precision, recall and F1 for each of the classes is relatively good. Versicolor has the worst metrics since it has the largest number of misclassified cases.

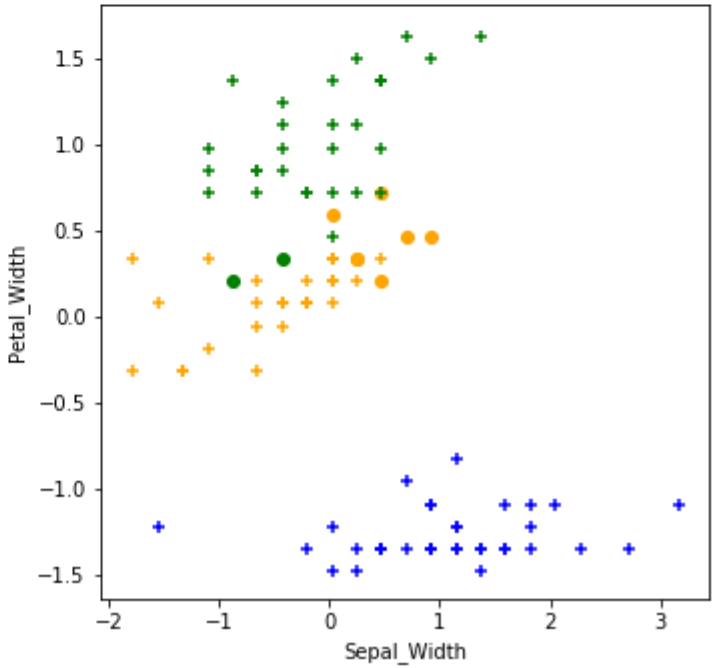
To get a better feel for what the classifier is doing, the code in the cell below displays a set of plots showing correctly (as '+') and incorrectly (as 'o') cases, with the species color-coded. Execute this code and examine the results.

In [11]:
```
def plot_iris_score(iris, y_test, scores):
    '''Function to plot iris data by type'''
    ## Find correctly and incorrectly classified cases
    true = np.equal(scores, y_test).astype(int)

    ## Create data frame from the test data
    iris = pd.DataFrame(iris)
    levels = {0:'setosa', 1:'versicolor', 2:'virginica'}
    iris['Species'] = [levels[x] for x in y_test]
    iris.columns = ['Sepal_Length', 'Sepal_Width', 'Petal_Length', 'Petal_Width', 'Species']

    ## Set up for the plot
    fig, ax = plt.subplots(2, 2, figsize=(12,12))
    markers = ['o', '+']
    x_ax = ['Sepal_Length', 'Sepal_Width']
    y_ax = ['Petal_Length', 'Petal_Width']

    for t in range(2): # loop over correct and incorect classifications
        setosa = iris[(iris['Species'] == 'setosa') & (true == t)]
        versicolor = iris[(iris['Species'] == 'versicolor') & (true == t)]
        virginica = iris[(iris['Species'] == 'virginica') & (true == t)]
        # loop over all the dimensions
        for i in range(2):
            for j in range(2):
                ax[i,j].scatter(setosa[x_ax[i]], setosa[y_ax[j]], marker = markers[t], color = 'blue')
                ax[i,j].scatter(versicolor[x_ax[i]], versicolor[y_ax[j]], marker = markers[t], color = 'orang
e')
                ax[i,j].scatter(virginica[x_ax[i]], virginica[y_ax[j]], marker = markers[t], color = 'green')
                ax[i,j].set_xlabel(x_ax[i])
                ax[i,j].set_ylabel(y_ax[j])

plot_iris_score(X_test, y_test, scores)
```


Petal_Length
Sepal_Length


Petal_Width
Sepal_Length


Petal_Length
Sepal_Width


Petal_Width
Sepal_Width

Examine these plots. You can see how the classifier has divided the feature space between the classes. Notice that most of the errors occur in the overlap region between Virginica and Versicolor. This behavior is to be expected.

Another example

Now, you will try a more complex example using the credit scoring data. You will use the prepared data which had the the following preprocessing:

1. Cleaning missing values.
2. Aggregating categories of certain categorical variables.
3. Encoding categorical variables as binary dummy variables.
4. Standardizing numeric variables.

Execute the code in the cell below to load the features and labels as numpy arrays for the example.

In [41]:
```
Data = pd.read_csv('data/Credit_Dataset.csv')
Data = Data.dropna()
Data = Data.reset_index(drop = True)
Data.head()
```

Out[41]:

	0	1	2	3	4	5	6	7	8	9	...	26	27	28	29	30	31	32	33	34	Class
0	-1.864869	-0.933901	0.918477	2.271006	0	1	0	0	0	1	...	1	0	0	0	1	0	0	1	0	0
1	1.708369	1.163046	-0.870183	-1.446152	1	0	0	0	0	0	...	0	0	1	0	0	0	0	1	0	1
2	-0.673790	-0.181559	-0.870183	1.226696	0	0	0	1	0	1	...	1	0	0	1	0	0	0	1	0	0
3	1.478913	1.525148	-0.870183	0.942455	0	1	0	0	0	0	...	1	0	0	0	1	1	0	0	0	0
4	0.517289	0.904743	0.024147	1.488620	0	1	0	0	0	0	...	1	0	0	0	1	0	0	0	1	1

5 rows × 36 columns

In [42]:
```
FeatureList = Data.columns[:-1]
Features = Data[FeatureList]
display(FeatureList)
display(Features.head())

Labels = Data["Class"]
display(Labels.head())
```

```
Index(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12',
       '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24',
       '25', '26', '27', '28', '29', '30', '31', '32', '33', '34'],
      dtype='object')
```

	0	1	2	3	4	5	6	7	8	9	...	25	26	27	28	29	30	31	32	33	34
0	-1.864869	-0.933901	0.918477	2.271006	0	1	0	0	0	1	...	0	1	0	0	0	1	0	0	1	0
1	1.708369	1.163046	-0.870183	-1.446152	1	0	0	0	0	0	...	0	0	0	1	0	0	0	0	1	0
2	-0.673790	-0.181559	-0.870183	1.226696	0	0	0	1	0	1	...	0	1	0	0	1	0	0	0	1	0
3	1.478913	1.525148	-0.870183	0.942455	0	1	0	0	0	0	...	0	1	0	0	0	1	1	0	0	0
4	0.517289	0.904743	0.024147	1.488620	0	1	0	0	0	0	...	0	1	0	0	0	1	0	0	0	1

5 rows × 35 columns

```
0    0
1    1
2    0
3    0
4    1
Name: Class, dtype: int64
```

In [43]:
```
print(Features.shape)
print(Labels.shape)
```

```
(1000, 35)
(1000,)
```

In [44]:
```
from sklearn.model_selection import train_test_split
X = Features
y = Labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=42)
```

In [45]:
```
from sklearn.metrics import accuracy_score
NB_credit = GaussianNB()
NB_credit.fit(X_train,y_train)
y_pred = NB_credit.predict(X_test)
print(accuracy_score(y_test, y_pred))
```

```
0.6727272727272727
```

In [46]:
```
nr.seed(321)
cv_folds = ms.KFold(n_splits=10, shuffle = True)

nr.seed(498)
NB_credit = GaussianNB()
cv_estimate = ms.cross_val_score(NB_credit, Features, Labels, cv = cv_folds) # Use the outside folds

print('Mean performance metric = %4.3f' % np.mean(cv_estimate))
print('SDT of the metric       = %4.3f' % np.std(cv_estimate))
print('Outcomes by cv fold')
for i, x in enumerate(cv_estimate):
    print('Fold %2d    %4.3f' % (i+1, x))
```

```
Mean performance metric = 0.691
SDT of the metric       = 0.093
Outcomes by cv fold
Fold  1    0.720
Fold  2    0.660
Fold  3    0.770
Fold  4    0.690
Fold  5    0.750
Fold  6    0.430
Fold  7    0.690
Fold  8    0.760
Fold  9    0.710
Fold 10    0.730
```

Examine these results. Notice that the standard deviation of the mean of the AUC is more than an order of magnitude smaller than the mean. This indicates that this model is likely to generalize well.

Now, you will build and test a model using a single split of the dataset. As a first step, execute the code in the cell below to create training and testing dataset.

The code in the cell below defines a naive Bayes model object and then fits the model to the training data. Execute this code:

In [51]:
```
def score_model(probs, threshold):
    return np.array([1 if x > threshold else 0 for x in probs[:,1]])

def print_metrics(labels, probs, threshold):
    scores = score_model(probs, threshold)
    metrics = sklm.precision_recall_fscore_support(labels, scores)
    conf = sklm.confusion_matrix(labels, scores)
    print('                 Confusion matrix')
    print('                 Score positive    Score negative')
    print('Actual positive    %6d' % conf[0,0] + '             %5d' % conf[0,1])
    print('Actual negative    %6d' % conf[1,0] + '             %5d' % conf[1,1])
    print('')
    print('Accuracy        %0.2f' % sklm.accuracy_score(labels, scores))
    print('AUC             %0.2f' % sklm.roc_auc_score(labels, probs[:,1]))
    print('Macro precision %0.2f' % float((float(metrics[0][0]) + float(metrics[0][1]))/2.0))
    print('Macro recall    %0.2f' % float((float(metrics[1][0]) + float(metrics[1][1]))/2.0))
    print(' ')
    print('           Positive      Negative')
    print('Num case   %6d' % metrics[3][0] + '        %6d' % metrics[3][1])
    print('Precision  %6.2f' % metrics[0][0] + '        %6.2f' % metrics[0][1])
    print('Recall     %6.2f' % metrics[1][0] + '        %6.2f' % metrics[1][1])
    print('F1         %6.2f' % metrics[2][0] + '        %6.2f' % metrics[2][1])

NB_credit = GaussianNB()
NB_credit.fit(X_train,y_train)
probabilities = NB_credit.predict_proba(X_test)
print_metrics(y_test, probabilities, 0.5)
```

Confusion matrix		
	Score positive	Score negative
Actual positive	180	50
Actual negative	58	42

Accuracy	0.67
AUC	0.69
Macro precision	0.61
Macro recall	0.60

	Positive	Negative
Num case	230	100
Precision	0.76	0.46
Recall	0.78	0.42
F1	0.77	0.44