Assignment_1_SVM

September 18, 2021

1 CPSC 393 Assignment 1 - SVM

2 Introduction

SVM (Support Vector Machine) is a supervised machine learning algorithm that can be used as a Regression or Classification algorithm, however the focus of this report is SVM being used as a classification algorithm. In classification, the SVM Model attempts to create decision boundaries that serve to separate the different classes of data. When working with two classes or features, the **decision boundary** is a line. When working with three or more classes/features, the decision boundary is a hyperplane. The decision boundary is created by finding the two nearest data points of different classes, which are called the **support vectors**. The goal of SVM is maximize the distance, margin, between the 2 support vectors so that the SVM Model will classify data points with the fewest possible errors. Often the case with data, different classes are not linearly separable and so the SVM applies a technique called the **Kernel Trick/Function**. The Kernel Trick is a transformation technique of transforming data points into a higher dimensional space than its original. The different kernel functions: polynomial, radial basis, sigmoid, and linear.

3 Background

A dataset of iris flowers is provided that consists of 2 classes: 'Iris-setosa' and 'Not-Iris-setosa' that can be described by 4 provided features: 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', and 'PetalWidthCm'. The goal of the SVM Models in this report is to correctly classify iris flowers with the fewest possible errors given their features without overfitting to training data.

4 Import packages

[1]:
```
# import packages
import warnings
warnings.filterwarnings('ignore')
import pandas as pd
import numpy as np
from plotnine import *
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split, KFold, GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, confusion_matrix, precision_score,␣
↪recall_score, classification_report
```

5 Import dataset

[2]:
```
# import data
data = pd.read_csv('iris-1.csv')
```

6 Visualizing Dataframe

[3]:
```
# view the data types of columns
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 6 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Id             150 non-null    int64
 1   SepalLengthCm  150 non-null    float64
 2   SepalWidthCm   150 non-null    float64
 3   PetalLengthCm  150 non-null    float64
 4   PetalWidthCm   150 non-null    float64
 5   Species        150 non-null    object
dtypes: float64(4), int64(1), object(1)
memory usage: 7.2+ KB
```

[4]:
```
# see what columns are in the dataframe; Species is the main class label
data.columns
```

[4]:
```
Index(['Id', 'SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm',
       'Species'],
      dtype='object')
```

7 Cleaning data

- Remove unnecessary column - Id
- Replace binary classes with binary numerical values
- Check for missing data
- Check for duplicate values and drop them if found

7.0.1 Remove unnecessary column - Id

```
[5]: # Do not need the Id column
     data.drop('Id', axis=1, inplace=True)
```

```
[6]: # Check if Id is removed
     data.columns
```

```
[6]: Index(['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm',
            'Species'],
           dtype='object')
```

7.0.2 Replace binary classes with binary numerical values

```
[7]: # Change Iris-setosa to 1 and Not-Iris-setosa to 0
     data = data.replace(to_replace = ['Iris-setosa', 'Not-Iris-setosa'],␣
      ↪value=[1,0])
```

```
[8]: # Check if the species was converted from obj to int
     data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   SepalLengthCm  150 non-null    float64
 1   SepalWidthCm   150 non-null    float64
 2   PetalLengthCm  150 non-null    float64
 3   PetalWidthCm   150 non-null    float64
 4   Species        150 non-null    int64
dtypes: float64(4), int64(1)
memory usage: 6.0 KB
```

```
[9]: # observe the different values used for species - 2 different values: 0, 1
     data['Species'].unique()
```

```
[9]: array([1, 0])
```

7.0.3 Check for missing data

```
[10]: # check if there are missing values in the dataframe - there are no missing␣
       ↪values
      data.isnull().sum(axis=0)
```

```
[10]: SepalLengthCm    0
      SepalWidthCm     0
```

```
PetalLengthCm    0
PetalWidthCm     0
Species          0
dtype: int64
```

7.0.4 Check for duplicate values and drop them if found

[11]:
```
# check if there is duplicated data - there is 3
data.duplicated().sum(axis=0)
```

[11]:
```
3
```

[12]:
```
# Remove duplicates
data = data.drop_duplicates()
```

[13]:
```
# check if there is duplicated data - there is 3
data.duplicated().sum(axis=0)
```

[13]:
```
0
```

7.0.5 Count the number of samples in each class

[14]:
```
num_of_species = {
    'Iris-setosa': 0,
    'Not-Iris-setosa': 0
}
for curr in data['Species']:
    if curr == 0:
        num_of_species['Not-Iris-setosa'] += 1
    if curr == 1:
        num_of_species['Iris-setosa'] += 1
print(num_of_species)
```

```
{'Iris-setosa': 48, 'Not-Iris-setosa': 99}
```

8 Setup model variables - features and output

[15]:
```
# do not include Id as a feature
features = ['SepalLengthCm', 'SepalWidthCm', 'PetalLengthCm', 'PetalWidthCm']
output = ['Species']
```

9 Create train and test variables for SVM Model

[16]:
```
# Split Data in test/train (80/20)
z = StandardScaler()
X_train, X_test, y_train, y_test = train_test_split(data[features],␣
 ↪data[output], test_size=0.3)
print(X_train)
```

```
     SepalLengthCm  SepalWidthCm  PetalLengthCm  PetalWidthCm
117            7.7           3.8            6.7           2.2
137            6.4           3.1            5.5           1.8
67             5.8           2.7            4.1           1.0
65             6.7           3.1            4.4           1.4
79             5.7           2.6            3.5           1.0
..             …             …              …             …
19             5.1           3.8            1.5           0.3
105            7.6           3.0            6.6           2.1
90             5.5           2.6            4.4           1.2
136            6.3           3.4            5.6           2.4
27             5.2           3.5            1.5           0.2

[102 rows x 4 columns]
```

10 z-score train and test data

[17]:
```
# Z-Score the train and test data sets
# for train, transform the data and then fit it to the model
X_train = z.fit_transform(X_train)
# for test, transform the data, but do not fit it to the model to prevent bias/
 ↪overfitting
X_test = z.transform(X_test)
```

11 Create the SVM Model

[18]:
```
# Init the SVM Model
SVM_Model = SVC()
```

12 Fit the train data to the SVM Model

[19]:
```
# Train the SVM Model
SVM_Model.fit(X_train, y_train)
```

[19]:
```
SVC()
```

13 Calculate predicted y values given test data

[20]:
```
# Testing the SVM Model
y_pred = SVM_Model.predict(X_test)
```

14 Evaluating Model's performance with:

14.0.1 Performance Metrics defined

- **Accuracy score** is the ratio of data that the model correctly classified.
- **Precision score** is a ratio of: TP / (TP + TN) where TP = number of True Positives and TN = number of True Negatives. Sklearn states in its documentation that the precision score is "is intuitively the ability of the classifier not to label as positive a sample that is negative".
- **Recall score** is the ratio of: TP / (TP + FN) where TP = number of True Positives and TN = number of True Negatives. Sklearn states in its documentation that the recall score is "is intuitively the ability of the classifier to find all the positive samples".
- **Confusion matrix** is a matrix that is used to understand and evaluate the accuracy of a classification. The top left of matrix represents the number of true negatives, top right is the number of false positives, bottom left is the number of false negatives, and the bottom right is the number of true positives.

[21]:
```
acc_score = accuracy_score(y_test, y_pred)
print(acc_score)
```

```
1.0
```

[22]:
```
print("Precision:", precision_score(y_test, y_pred))
```

```
Precision: 1.0
```

[23]:
```
print("Recall:", recall_score(y_test, y_pred))
```

```
Recall: 1.0
```

[24]:
```
cm = confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[34  0]
 [ 0 11]]
```

15 Discussion: Initial Results

An SVM Model was created above using the technique of train/test split to divide the dataset into data that trains the model and data that tests the model's effectiveness. Before fitting the data to the model, the data is z-scored to standardize the data. After fitting the z-scored data, the SVM Model's performance was evaluated with accuracy score, precision score, recall score, and a confusion matrix (all of which are defined a few cells above). The SVM Model got a perfect score for each metric that was used. Normally this is concerning because machine learning models can overfit to their training data. Overfitting means that the machine learning model learned the patterns of the training data too well from the model learning noise or irrelevant patterns. And so an overfit model typically performs poorly with outside data. It is possible that the data is overfitting because the data that was chosen as the training set was not optimal in terms of the model learning the patterns of the data. This is especially common in data sets with a small number of samples. The data set provided only contains 150 entries (147 after removing duplicates) which is very small for a machine learning algorithm. To ensure that an unoptimal training set is not chosen, the K-Folds Cross Validation technique is applied below with 10 K-Folds:

16 Employing K-Folds

[25]:
```
acc_scores = []
prec_scores = []
recall_scores = []
conf_matrices = []

X = data[features]
y = data['Species']
n_splits = 10
kf = KFold(n_splits=n_splits)
for train_index, test_index in kf.split(X):
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]

    SVM_Model = SVC()
    SVM_Model.fit(X_train, y_train)
    y_pred = SVM_Model.predict(X_test)

    acc_scores.append(accuracy_score(y_test, y_pred))
    prec_scores.append(precision_score(y_test, y_pred))
    recall_scores.append(recall_score(y_test, y_pred))
    conf_matrices.append(confusion_matrix(y_test, y_pred))

avg_acc_score = np.mean(acc_scores)
avg_prec_score = np.mean(prec_scores)
avg_recall_score = np.mean(recall_scores)
print(acc_scores)
print(prec_scores)
print(recall_scores)
print(f"The average accuracy score is: {avg_acc_score}")
print(f"The average precision score is: {avg_prec_score}")
print(f"The average recall score is: {avg_recall_score}")
for cm in conf_matrices:
    print(cm)
```

```
[1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
[1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

```
[1.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
The average accuracy score is: 1.0
The average precision score is: 0.4
The average recall score is: 0.4
[[15]]
[[15]]
[[15]]
[[12  0]
 [ 0  3]]
[[15]]
[[15]]
[[15]]
[[14]]
[[14]]
[[14]]
```

17 Discussion: KFolds Results

10 K-Folds cross validation was applied to the data set for the SVM Model as shown above. The accuracy scores for each K-Fold is 100% like observed before, but the precision and recall scores differ than before. The reason that the average precision and recall scores are 0.4 is because the precision and recall scores are 0 for some KFold iterations. A precision score of 0 indicates no positive predictions were true and a recall score of 0 indicates positive values were not predicted. These results are most likely indicative of the unbalanced data set. Earlier it was calculated that the data set consists of 48 'Iris-setosa' samples and 99 'Not-Iris-setosa' samples. The K iterations in which a 0 precision score and 0 recall score is calculated is most likely due to that specific K iteration training set not having enough samples of the 'Iris-setosa' class. Without having enough of those samples, the SVM Model is not able to learn the patterns of the 'Iris-setosa' class.

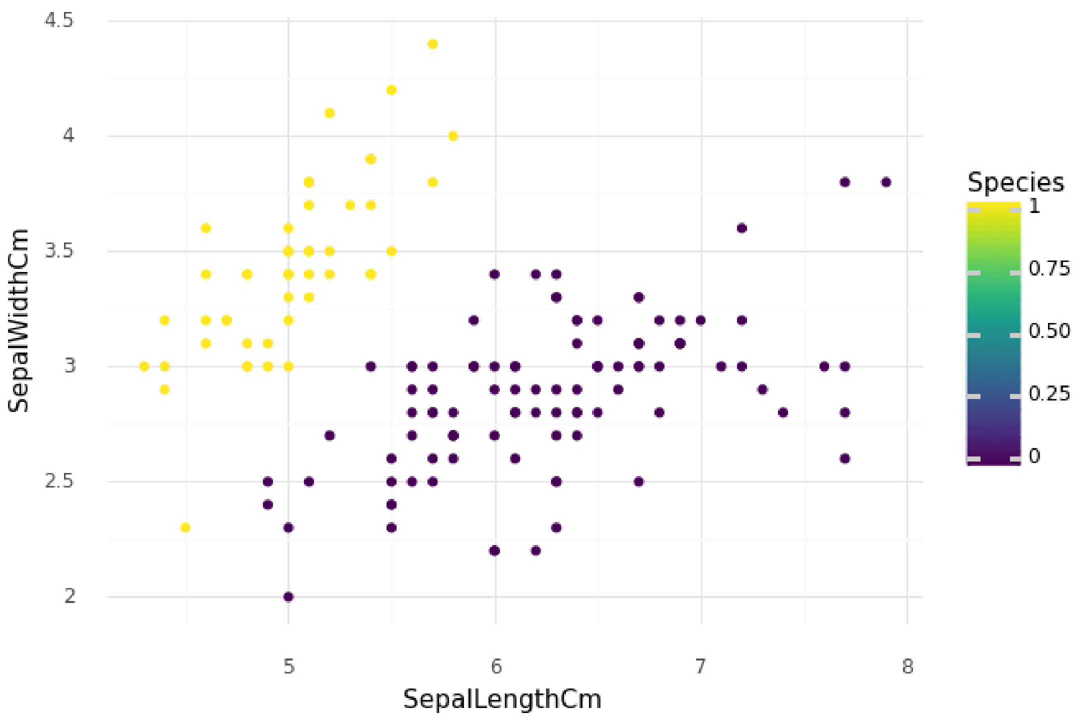
The unbalanced data set most likely explains the precison and recall scores, but the 100% accuracy scores still need to be investigated. To investigate what could be causing the perfect performance metric scores, graphs for each feature relation in a 2D vector space are created below:

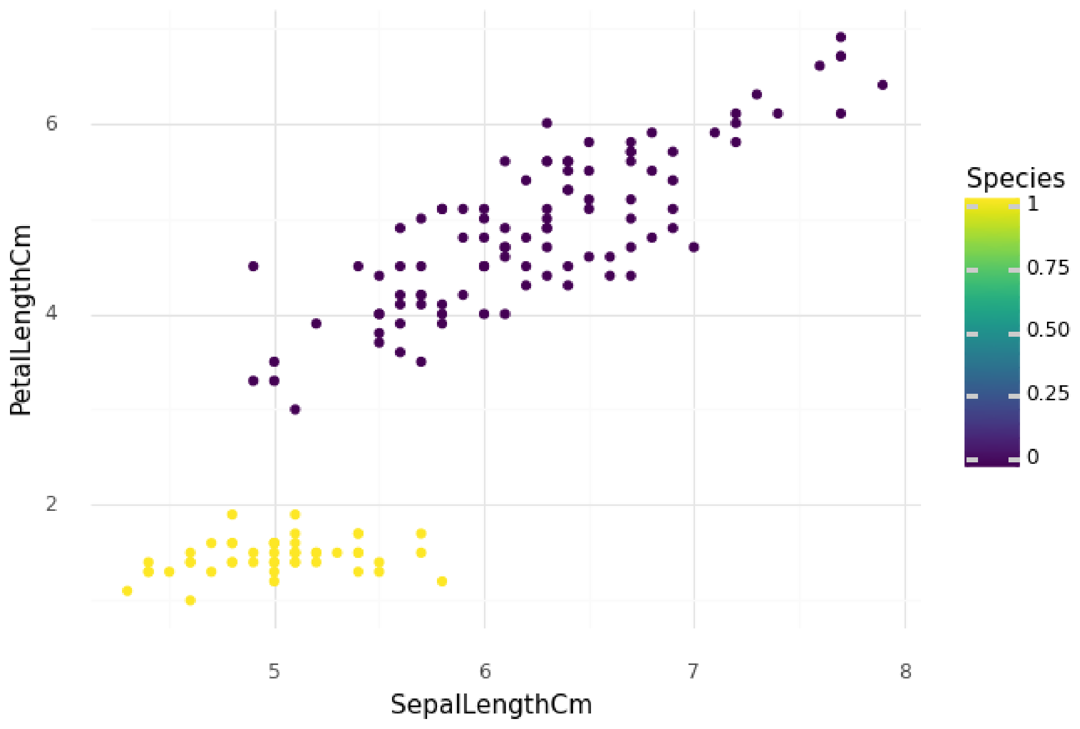
[26]:
```
def graph_features_by_species(feature1, feature2):
    return (ggplot(data, aes(x=feature1, y=feature2, color='Species'))
     + geom_point()
     + theme_minimal()
    )
```

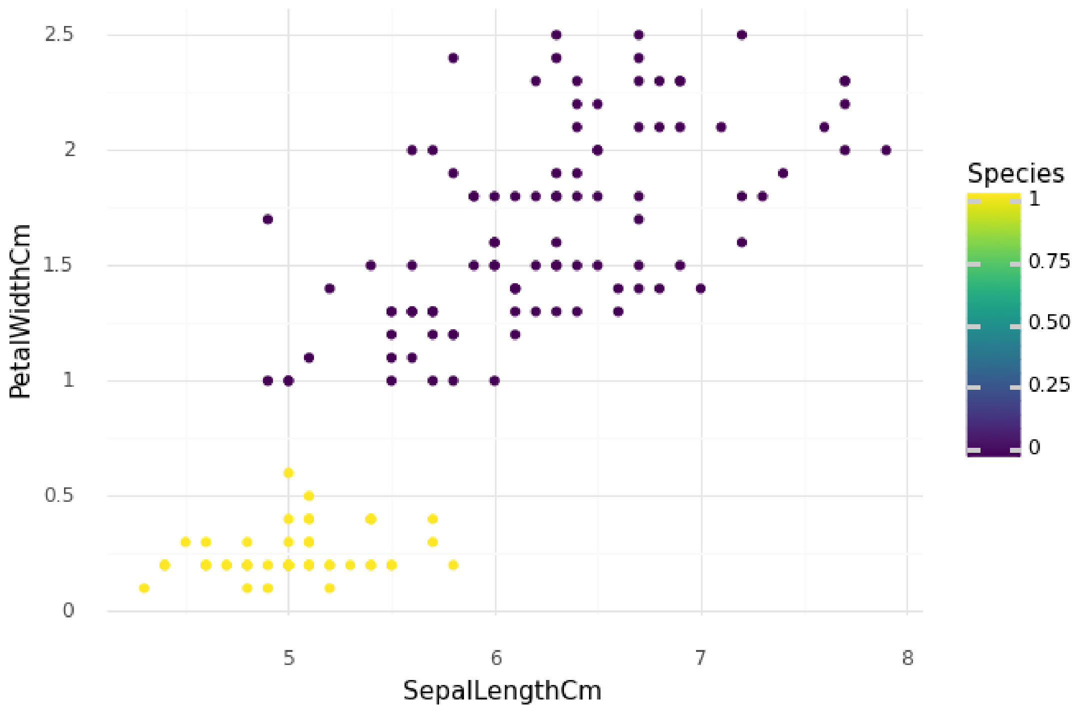
18 Graphing the features against one another in the R2 vector space

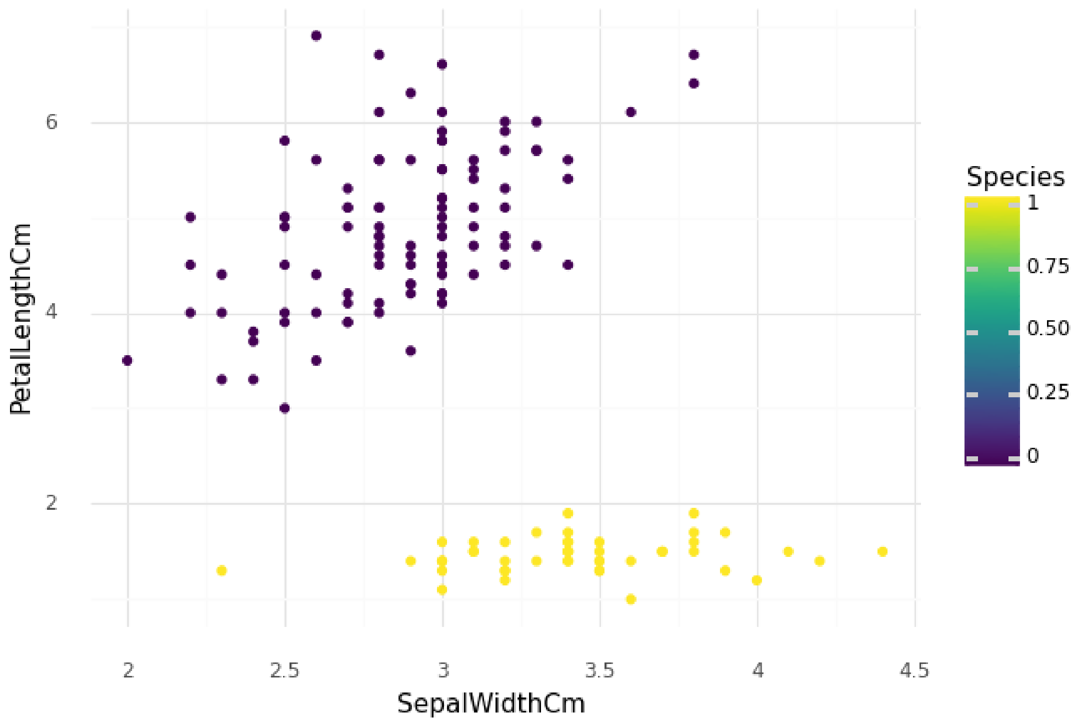
[27]:
```
# Create a graph for each feature relation - Note: every feature relation␣
 ↪appears to be linearly separable

# Graph SepalLengthCm against SepalWidthCm
```

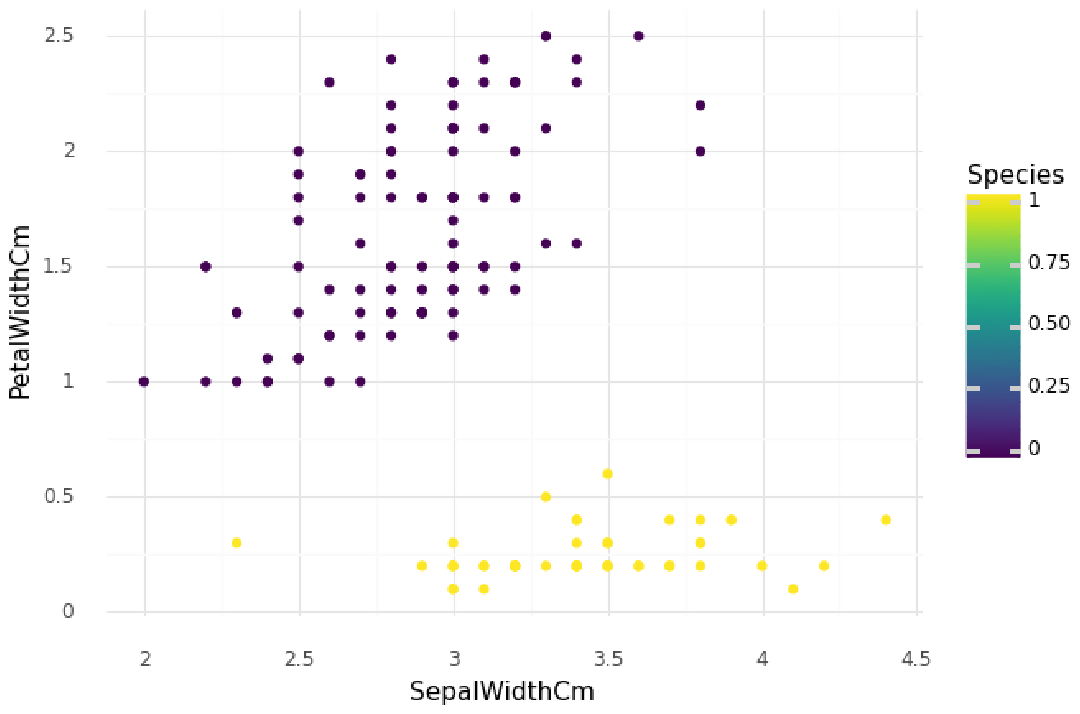
```
print(graph_features_by_species('SepalLengthCm', 'SepalWidthCm'))
# Graph SepalLengthCm against PetalLengthCm
print(graph_features_by_species('SepalLengthCm', 'PetalLengthCm'))
# Graph SepalLengthCm against PetalWidthCm
print(graph_features_by_species('SepalLengthCm', 'PetalWidthCm'))
# Graph SepalWidthCm against PetlWidthCm
print(graph_features_by_species('SepalWidthCm', 'PetalLengthCm'))
# Graph SepalWidthCm against PetalLengthCm
print(graph_features_by_species('SepalWidthCm', 'PetalWidthCm'))
# Graph PetalLengthCm against PetalWidthCm
print(graph_features_by_species('PetalLengthCm', 'PetalWidthCm'))
```

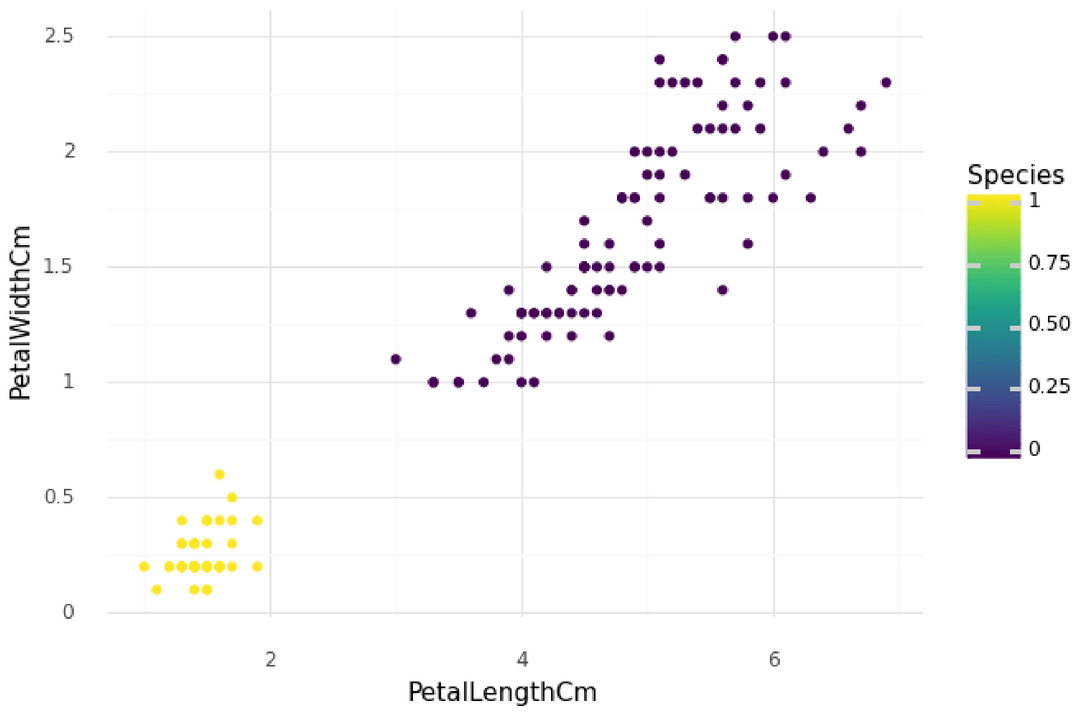



6
4
2
5
6
7
8
PetalLengthCm
SepalLengthCm
Species
1
0.75
0.50
0.25
0


2.5
2
1.5
1
0.5
0
5
6
7
8
PetalWidthCm
SepalLengthCm
Species
1
0.75
0.50
0.25
0


PetalLengthCm
SepalWidthCm
Species
1
0.75
0.50
0.25
0
2
4
6
2
2.5
3
3.5
4
4.5


PetalWidthCm
SepalWidthCm
Species
1
0.75
0.50
0.25
0
2.5
2
1.5
1
0.5
0
2
2.5
3
3.5
4
4.5


PetalWidthCm
PetalLengthCm
Species
1
0.75
0.50
0.25
0
2.5
2
1.5
1
0.5
0
2
4
6

19 Discussion: Interpreting 2D Feature Relations

A graph of each feature relation is created above to better understand why the SVM Model was producing a perfect score for each performance metric. The 'Iris-setosa' class is represented by the purple color points and the 'Not-Iris-setosa' class is represented by the yellow color points. In each graph, the two groups of classes are separable. And so a decision boundary line could be drawn in each of these graphs that separate the two classes: 'Iris-setosa' and 'Not-Iris-setosa'. Although these graphs are not representative of the model's vector space with all of the features together, it still provides insight on how the features differ across the 2 classes. **Given the clear linear separability observed in feature relations between the two classes, the SVM model is not overfitting. The SVM Model is performing perfectly because the data provided consists of 2 classes that are fully separable with the provided labels.**

Note - Since 4 features are used in total for the SVM Model, the actual decision boundary between the 2 classes would be a hyperplane, not a line.

To further ensure that overfitting was not occurring, parameter tuning was performed below:

19.1 Parameter Tuning: Kernel Methods

[28]:
```
# trying different kernel methods
kernel_methods = ['linear', 'poly', 'rbf', 'sigmoid']
kernel_methods_dict = {}

for km in kernel_methods:
    X_train, X_test, y_train, y_test = train_test_split(data[features],␣
 ↪data[output], test_size=0.3)
    # for train, transform the data and then fit it to the model
    X_train = z.fit_transform(X_train)
    # for test, transform the data, but do not fit it to the model to prevent␣
 ↪bias/overfitting
    X_test = z.transform(X_test)
    SVM_Model = SVC(kernel=km)
    SVM_Model.fit(X_train, y_train)
    # Testing the SVM Model
    y_pred = SVM_Model.predict(X_test)
    curr_acc_score = accuracy_score(y_test, y_pred)
    kernel_methods_dict[km] = curr_acc_score

for key, value in kernel_methods_dict.items():
    print(f'{key} : {value},')
```

```
linear : 1.0,
poly : 1.0,
rbf : 1.0,
sigmoid : 1.0,
```

20 Discussion: Kernel Method tuning

The Kernel Trick is a transformation technique of transforming data points into a higher dimensional space than its original. It is employed when data points of different classes are non separable. In the cell above, each kernel method resulted in a model with a 100% accuracy score. The results are not suprising because the kernel method choice should not have an impact on a model containing linearly separable classes. **These results support the notion that the SVM Model is not overfitting but rather reflective of the data having perfectly separable classes.**

Since the kernel method did not make a difference in the results, the default sklearn kernel method,'rbf', is used as the kernel method when modifying other parameters. The 'rbf' is the most commonly used kernel method.

21 Parameter Tuning: Regularization Value

[29]:
```
# tuning regularization parameter
regularization_params = [i for i in range(5,101,5)]
regularization_params.insert(0, 1.0)
regularization_params.insert(0, 0.1)
regularization_dict = {}
for c in regularization_params:
    X_train, X_test, y_train, y_test = train_test_split(data[features],␣
 ↪data[output], test_size=0.3)
    # for train, transform the data and then fit it to the model
    X_train = z.fit_transform(X_train)
    # for test, transform the data, but do not fit it to the model to prevent␣
 ↪bias/overfitting
    X_test = z.transform(X_test)
    SVM_Model = SVC(kernel='rbf', C=c)
    SVM_Model.fit(X_train, y_train)
    # Testing the SVM Model
    y_pred = SVM_Model.predict(X_test)
    curr_acc_score = accuracy_score(y_test, y_pred)
    regularization_dict[str(c)] = curr_acc_score

for key, value in regularization_dict.items():
    print(f'{key} : {value},')
```

```
0.1 : 1.0,
1.0 : 1.0,
5 : 1.0,
10 : 1.0,
15 : 1.0,
20 : 1.0,
25 : 0.9777777777777777,
30 : 1.0,
35 : 1.0,
40 : 1.0,
45 : 1.0,
50 : 1.0,
55 : 1.0,
60 : 1.0,
65 : 1.0,
70 : 1.0,
75 : 1.0,
80 : 1.0,
85 : 1.0,
90 : 1.0,
95 : 1.0,
100 : 1.0,
```

22 Discussion: Regularization tuning

The regularization parameter, lambda, affects how the SVM algorithm handles misclassifications. It is typically used in SVM when classes are non separable. Generally speaking, a higher lambda implies less misclassifications are allowed and a lower lambda implies that more misclassifications are allowed. Different regularization values were attempted above and it can be observed that tweaking the regularization values has little effect on the model. This is not suprising considering that the regularization parameter is intended to be used for non separable class data. **Once again, the results above support the notion that the SVM Model is not overfitting but rather reflective of the data having perfectly separable classes.**

The next parameter that is being tested is gamma, which is only for the 'rbf' kernel method.

23 Parameter Tuning: Gamma Value

[30]:
```python
# tuning gamma parameter
gammas = [i for i in range(1,11,1)]
gammas.insert(0, 1)
gammas.insert(0, 0.1)
gammas.insert(0, 0.01)
gammas.insert(0, 0.001)
gammas.insert(0, 0.0001)
gamma_dict = {}
for gamma in gammas:
    X_train, X_test, y_train, y_test = train_test_split(data[features],␣
 ↪data[output], test_size=0.3)
    # for train, transform the data and then fit it to the model
    X_train = z.fit_transform(X_train)
    # for test, transform the data, but do not fit it to the model to prevent␣
 ↪bias/overfitting
    X_test = z.transform(X_test)
    SVM_Model = SVC(kernel='rbf', gamma=gamma)
    SVM_Model.fit(X_train, y_train)
    # Testing the SVM Model
    y_pred = SVM_Model.predict(X_test)
    curr_acc_score = accuracy_score(y_test, y_pred)
    gamma_dict[str(gamma)] = curr_acc_score

for key, value in gamma_dict.items():
    print(f'{key} : {value},')
```

```
0.0001 : 0.6888888888888889,
0.001 : 0.6444444444444445,
0.01 : 1.0,
0.1 : 1.0,
1 : 1.0,
2 : 1.0,
3 : 0.9777777777777777,
4 : 1.0,
5 : 0.9111111111111111,
6 : 1.0,
7 : 0.9333333333333333,
8 : 1.0,
9 : 0.9555555555555556,
10 : 0.8888888888888888,
```

24 Discussion: Gamma tuning

Sklearn defines the gamma parameter as "how far the influence of a single training example reaches, with low values meaning 'far' and high values meaning 'close' ". It can be observed that the different gamma values did have affect accuracy scores especially when the gamma value is a low value (less than 0.1). The reason the accuracy scores decreased so much with low valued gammas is because low gamma imply that more points are grouped together. Having a low gamma actually significantly lowered the accuracy score because the SVM Model is broadened it's classification too much making it less effective than before. **These results further indicate that the data is separable and tuning parameters meant for non separable data actually worsens performance metric results or does not affect them altogether.**

25 Conclusion

SVM can be a very effect classification algorithm given the labled data and employment of proper techniques. This report demonstrated how the SVM Model was able to easily distinguish classes of iris flowers. The classes of the iris flower data set are separable, however SVM can be very effective with non separable data with the use of Kernel functions that increase the dimensionality of the vector space. SVM will continue to be used as an effective supervised machine learning algorithm.