Penguin Classification Group Mini-Project

- By Isha Bola Kamath, Asmita Majumder, Rachel Yu
- Due Friday, August 13, 2021

We affirm that we personally wrote the text, code, and comments in this assignment.
We received help from Professor Perlmutter (whose lecture notes and code we referenced) and Wonjun.

- Isha Bola Kamath, Asmita Majumder, Rachel Yu; 2021/08/13

Group Contributions Statement

Rachel led the Exploratory Analysis; Isha led the Modeling; Asmita led the Writing. However, there was a lot of overlap in the contributions of the group members.

In more detail: Rachel did the majority of the Exploratory Analysis section, but Isha created the summary table. In the Modeling section, Rachel created the first two models, and Isha created the third (random forest classifier) model. Isha created all of the confusion matrices. Rachel created the decision region plots. Asmita did the Writing, both the expository and the discussion, but Isha helped with the content for the discussion section. All three helped plan, compile, and edit the project.

Data Acquisition

We first import all the modules we anticipate needing in this project. We also acquire our penguin data and store it as a data frame.

In [1]:
```
import pandas as pd
import numpy as np
import urllib
from matplotlib import pyplot as plt
from sklearn import preprocessing

#import data from given URL
url = 'https://philchodrow.github.io/PIC16A/datasets/palmer_penguins.csv'
penguins = pd.read_csv(url)
```

Exploratory Analysis

Data Preparation

Before we begin the main part of our exploratory analysis, we must clean up the data so that we can more easily deal with it.

There are a number of columns that will likely not be useful in our attempts to classify penguins by species. For instance, region and stage are the same for every penguin; the study name, sample numbers, and individual IDs are markers produced by the scientists gathering the penguin data, rather than traits of the penguins themselves; clutch completion and date egg seem to be too specific to individual penguins, and do not seem to tell us anything about their species. We get rid of these aforementioned columns, leaving in only the numerical physical measurements, as well as the penguins' species, sex, and island. We want to remove any NaN values to prevent errors and NaN propagation, so we call `dropna()` on our data frame. We shorten the species names, and change the strings representing the different sexes to numerical values for easier analysis.

In [2]:
```
#We will only use these specific columns here:
cols =  ["Species","Body Mass (g)", "Culmen Length (mm)", "Culmen Depth (mm)",
         "Flipper Length (mm)","Delta 15 N (o/oo)", "Delta 13 C (o/oo)",
         "Sex", "Island"]
penguins = penguins[cols]

#Remove NaN values:
penguins = penguins.dropna()

#Shorten 'Species' name:
penguins["Species"] = penguins["Species"].str.split().str.get(0)

#Change 'Sex' to numerical values:
le=preprocessing.LabelEncoder()
penguins["Sex"]=le.fit_transform(penguins["Sex"])
```

Summary Table

We are now ready to move on to the bulk of the exploratory analysis. We begin by creating a function, called `penguin_summary_table`, that produces a table that summarizes the average attributes of different groups. This table allows us to observe at a glance whether or not there are, on average, any drastic differences between the features of different groups.

We call this function on our penguin data set, which we choose to group by species.

In [3]:
```
#Summary table function:
def penguin_summary_table(group_cols, value_cols):
    """
    Creates a Summary Table
    ---
    Parameters

    group_cols, is a list of strings
    value_cols, is a list of strings
    ---
    The function groups a set of data on the basis of group_cols.
    Then, it calculates the mean of each column in value_cols for
    each group created.
    ---
```

```
    Returns a Summary Table
    """
    return penguins.groupby(group_cols)[value_cols].mean().round(2)

#Create a summary table in which the penguin are grouped by species.
penguin_summary_table(["Species"], ["Body Mass (g)", "Culmen Length (mm)",
                                    "Culmen Depth (mm)","Flipper Length (mm)",
                                    "Delta 15 N (o/oo)", "Delta 13 C (o/oo)"])
```

Out[3]:

Species	Body Mass (g)	Culmen Length (mm)	Culmen Depth (mm)	Flipper Length (mm)	Delta 15 N (o/oo)	Delta 13 C (o/oo)
Adelie	3702.70	38.79	18.32	190.32	8.86	-25.81
Chinstrap	3729.85	48.79	18.40	195.67	9.36	-24.56
Gentoo	5089.29	47.54	15.00	217.19	8.25	-26.18

In our summary table, we notice the following:

- The average body mass of the Gentoo penguins is over 1000g greater than the other penguin species; in other words, the average body mass of the Gentoo penguins is about 136% to 137% of the average body mass of the Adelie and Chinstrap penguins.
- The average culmen length of the Adelie penguins is nearly 10mm shorter than the other two species; this means that the Adelie penguin's average culmen length is about 80% of the Chinstrap's average culmen length, and about 82% of the Gentoo's average culmen length. The average culmen lengths of the Chinstrap and Gentoo penguins appear quite close, however.
- The average culmen depth of the Gentoo penguins is smaller than the Adelie and Chinstrap penguins, by about 3mm (a decrease of about 18%). The Adelie and Chinstrap penguins, however, have very similar average culmen depths.
- There are discrepancies between the flipper lengths of each of the species. The Adelie penguins have the shortest average flipper length, followed by the Chinstrap penguins, which have an average flipper length that is about 5mm greater (a roughly 3% increase), and then the Gentoo penguins, which have an average flipper length of 217.19 (an increase of roughly 14% compared to the Adelie penguins and 11% compared to the Chinstrap penguins).
- There are slight discrepancies between the average Delta N 15 levels of the different penguin species: the Gentoo penguins have the lowest average; the Adelie penguins have an average that is 0.61 o/oo (or about 7%) higher; and the Chinstrap penguins have the highest average, which is 1.11 o/oo (or about 13%) higher than the Gentoo penguins and 0.5 o/oo (or about 6%) higher than the Adelie penguins.
- There are also slight discrepancies between the average Delta 13 C levels across penguin species. We notice that the order is the same, with Gentoo penguins having the lowest (most negative) average, followed by the Adelie penguins in the middle, and Chinstrap penguins with the highest average.

Although this table gives us only a cursory glance at how the physical features of the various penguin species differ, it allows us to develop a set of hypotheses. By looking at the summary table, we can look to body mass and culmen depth as features that may allow us to distinguish Gentoo

penguins, or to culmen length as a feature that may allow us to distinguish Adelie penguins. We also note that we may be able to use flipper lengths and Delta 15 N/Delta 13 C levels as part of our penguin classifiers, as these traits appear to differ between species, although not drastically.

These hypotheses are a good starting point, but we also need to check whether or not they apply to each of the species as a whole, rather than just on average.

Data Visualization

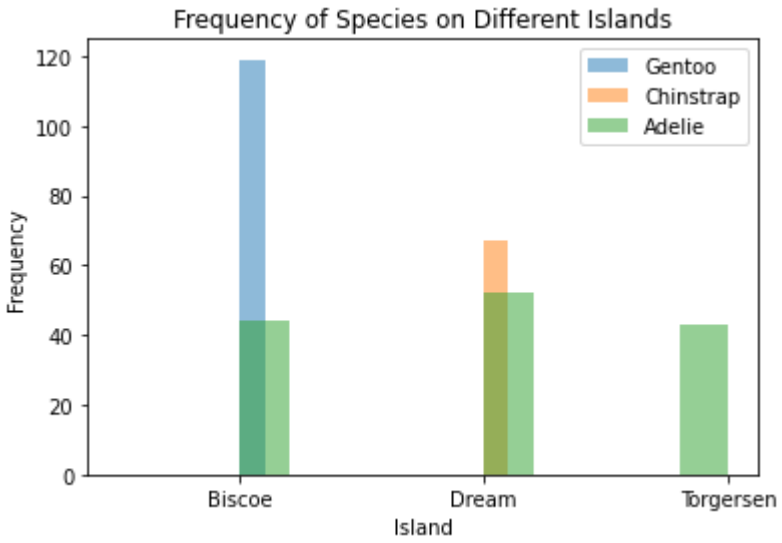
Using the information we gleaned from the penguin summary table to guide us, we want to further explore the relationships that exist between penguin species and various physical features, and see if we can pinpoint any features that are specific to each species. We will use scatterplots and histograms to help us visualize these relationships.

1. Islands

Although our penguin summary table did not include any information about the islands on which the penguins reside, a quick look at our data set shows us that not all of the species reside on all of the islands. Thus, we create a histogram that displays the frequencies of penguin species on each island.

In [4]:
```
fig, ax = plt.subplots(1)

#Loops over every species
for species in set(penguins["Species"]):
    v = penguins[penguins["Species"] == species]
    #plots histogram of 'Island'' for each species
    ax.hist(v["Island"], label = str(species.split()[0]), alpha = 0.5)

#labels
ax.set(xlabel = "Island", ylabel = "Frequency",
       title = "Frequency of Species on Different Islands")
ax.legend()
```

Out[4]: <matplotlib.legend.Legend at 0x20918d9dd30>



We see that Gentoo penguins only reside on Biscoe, Chinstrap penguins only reside on Dream, and

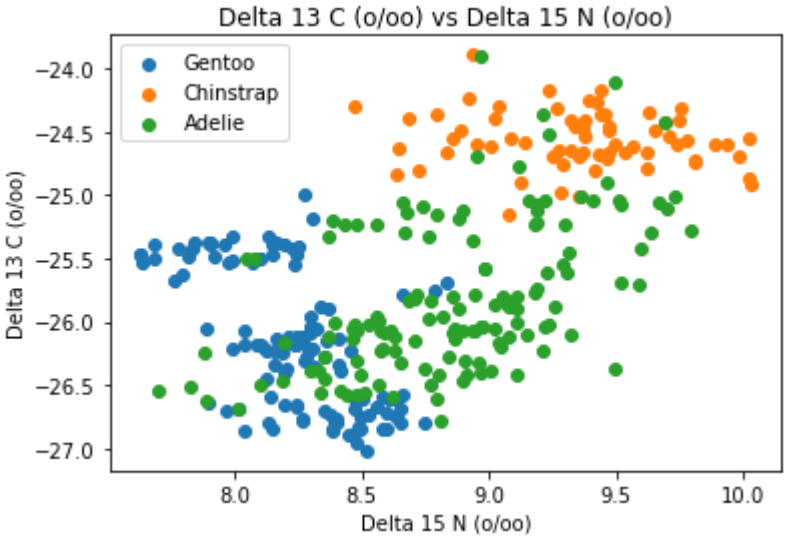
Adelie penguins reside on all three islands. Thus, if we know which island a penguin was found on, we can quickly narrow its species down to at most two options: if found on Biscoe, it must be Adelie or Gentoo; if found on Dream, it must be Adelie or Chinstrap; and if found on Torgersen, it must be Adelie.

2. Delta 13 C (o/oo) vs Delta 15 N (o/oo)

Recall from our summary table that there was a slight difference between the Delta 13 C and Delta 15 N levels across species, although it was not drastic enough for us to surmise whether this was the result of the different species of penguins. A particular point of interest was that the ordering was the same for both the Delta 13 C and Delta 15 N levels--in both cases, the Gentoo penguin had the lowest average, followed by the Adelie penguins with a moderate average and then by the Chinstrap penguin with the highest average. This indicates that looking at Delta 15 N and Delta 13 C in conjunction with each other may tell us something about the penguin species that we could not find when looking at each of the columns in isolation; thus, we plot the two columns against each other.

In [5]:
```
#scatter plot of Delta 13C and Delta 15N
fig,ax = plt.subplots(1)

#loops over every species
for species in set(penguins["Species"]):
    v = penguins[penguins["Species"] == species]

    #plots data point for each species
    ax.scatter(v["Delta 15 N (o/oo)"], v["Delta 13 C (o/oo)"],
               label = str(species))

#label
ax.set (xlabel = "Delta 15 N (o/oo)", ylabel = "Delta 13 C (o/oo)",
        title = "Delta 13 C (o/oo) vs Delta 15 N (o/oo)")
ax.legend()
```

Out[5]:
```
<matplotlib.legend.Legend at 0x20918689370>
```



In the plot, we observe that there is a notable distinction between the region occupied by the Gentoo penguins and the region occupied by the Chinstrap penguins; the Gentoo penguins tend to

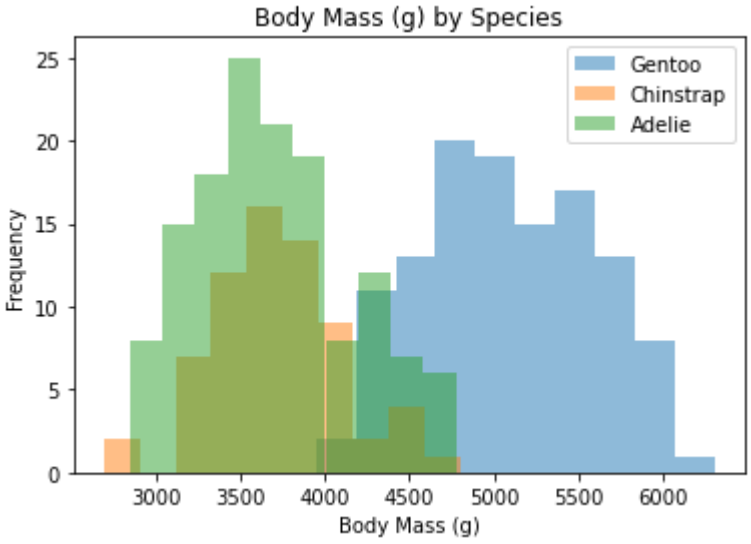
have lower Delta 15 N (o/oo) and lower Delta 13 C (o/oo) values, while the Chinstrap penguins have higher values for both. The Adelie penguins, however, are fairly spread out throughout the plot. Thus, Delta 13 C and Delta 15 N measurements may be useful for distinguishing between Gentoo and Chinstrap penguins, but are unlikely to help us separate out the Adelie species.

3. Body Mass (g) by Species

Recall that our summary table found that Gentoo penguins were, on average, significantly heavier than penguins of the other species. We wish to confirm that this high body mass is something that applies to the Gentoo species as a whole, rather than being the byproduct of a high variance. Thus, we create a histogram in which we plot the body masses of each of the penguins in our dataset.

In [6]:
```
#histogram of Body Mass by species
fig, ax = plt.subplots(1)

#loops over every species
for species in set(penguins["Species"]):
    v = penguins[penguins["Species"] == species]
    #plots histogram of 'Body Mass (g)' for each species
    ax.hist(v["Body Mass (g)"].dropna(),
            label = str(species.split()[0]), alpha = 0.5)

#label
ax.set(xlabel = "Body Mass (g)", ylabel = "Frequency",
       title = "Body Mass (g) by Species")
ax.legend()
```

Out[6]: <matplotlib.legend.Legend at 0x20918ec1220>



We find a lot of overlap in the body masses of the Adelie and Chinstrap penguins, but little overlap between the body masses of the Gentoo penguins and penguins of the other two species. This demonstrates that, indeed, the Gentoo penguin species as a whole tends to have higher body masses when compared to the Adelie and Chinstrap penguins.

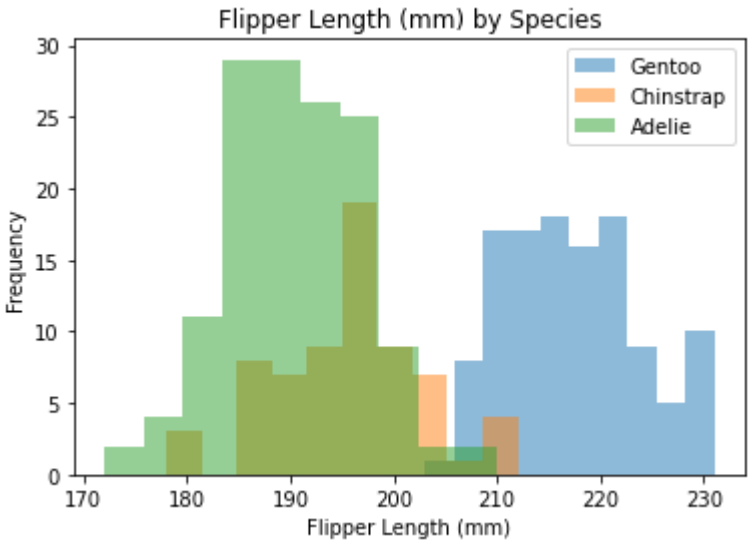
If we know the body mass of a penguin, then, we can most likely determine whether or not it is a Gentoo penguin; however, we will have a difficult time distinguishing between Adelie and Chinstrap penguins, due to the large overlapping regions.

4. Flipper Length (mm) by Species

Recall from our summary table that there was a slight difference between the average flipper lengths of the penguin species, particularly between the Adelie/Chinstrap penguins and the Gentoo penguins. Again, we wish to see if this is something that holds across the Gentoo species as a whole, so we create another histogram that plots the flipper lengths of each of the penguins in our dataset.

In [7]:
```
#histogram of flipper lengths by species
fig, ax = plt.subplots(1)

#loops over every species
for species in set(penguins["Species"]):
    v = penguins[penguins["Species"] == species]
    #plots histogram of 'Flipper Length (mm)' for each species
    ax.hist(v["Flipper Length (mm)"].dropna(),
            label = str(species.split()[0]), alpha = 0.5)

#labels
ax.set(xlabel = "Flipper Length (mm)", ylabel = "Frequency",
       title = "Flipper Length (mm) by Species")
ax.legend()
```

Out[7]: <matplotlib.legend.Legend at 0x20918efec70>



Similar to body mass, we find that there is quite a bit of overlap between the Adelie and Chinstrap penguins, but very little overlap between the latter two and the Gentoo penguins. We confirm that the Gentoo penguins do tend to have longer flippers than penguins of the other two species.

Just like with body mass, knowing a penguin's flipper length will help us determine whether or not the penguin is a Gentoo penguin, but if it is not, the body mass will not tell us much regarding whether it is an Adelie or Chinstrap penguin.

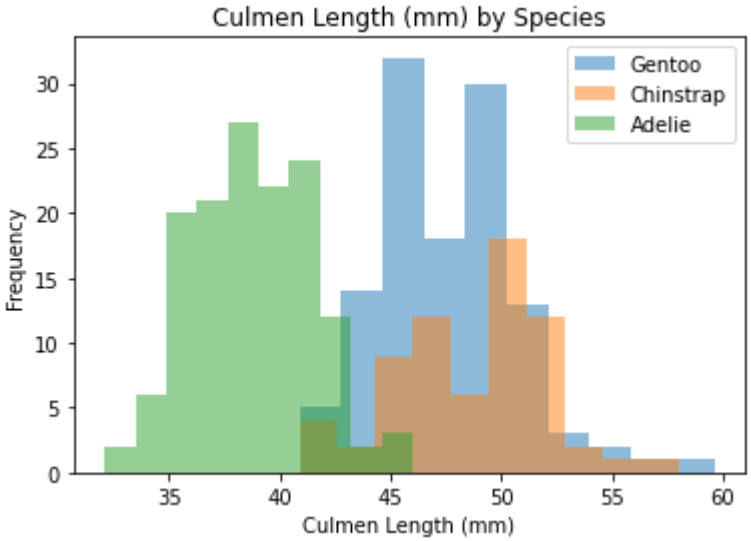
5. Culmen Length (mm) by Species

Recall from our penguin summary table that Adelie penguins had a lower average culmen length than the other two species. We again want to confirm that this is something that tends to hold for

all Adelie penguins, rather than just on average. Let us create a histogram in which we plot the culmen length (in mm) of each of the penguins in our data set.

In [8]:
```
fig, ax = plt.subplots(1)

#loops over every species
for species in set(penguins["Species"]):
    v = penguins[penguins["Species"] == species]
    #plots histogram of 'Culmen Length (mm)' for each species
    ax.hist(v["Culmen Length (mm)"].dropna(),
            label = str(species.split()[0]), alpha = 0.5)

#labels
ax.set(xlabel = "Culmen Length (mm)", ylabel = "Frequency",
       title = "Culmen Length (mm) by Species")
ax.legend()
```

Out[8]: <matplotlib.legend.Legend at 0x20918fbf790>



We see that the majority of Adelie penguins in our data set do indeed have shorter culmens than the majority of Gentoo and Chinstrap penguins. There is a lot of overlap in the culmen lengths of Gentoo and Chinstrap penguins, but little overlap between penguins of the latter two species and the Adelie penguins.

This tells us that if we know the culmen length of a given penguin, then we can likely determine whether it is an Adelie penguin. However, we will have a much harder time distinguishing between Gentoo and Chinstrap penguins given just culmen length, due to the large amount of overlap.

Final Decisions

From our above figures, we can make some decisions regarding which columns we want to use in creating our models. Our island histogram is the most decisive; knowing a penguin's island will necessarily narrow its possible species down to at most two options, so we can deduce that the island column will be useful. Once we know a penguin's island, we only need two more sets of information--we need some way to distinguish between Adelie and Gentoo penguins, and between Adelie and Chinstrap penguins. Recall that both body mass and flipper lengths were features that

we found to be useful for distinguishing between Adelie and Gentoo penguins; the flipper length histogram appears to contain less overlap between the two species, so we choose the latter column as our second measurement. To distinguish between the Adelie and Chinstrap penguins, meanwhile, we can use culmen length. Culmen length is doubly useful, as it also allows us to distinguish between Adelie and Gentoo penguins.

Thus, we decide to use a penguin's island, flipper length (mm), and culmen length (mm) to produce the machine learning models that will predict its species.

Modeling

Processing Data and Preparation for Modeling

Before we begin the modelling, we must prepare our data. Since we have decided to use a penguin's island, flipper length, and culmen length to guess its species, we wish to isolate only these relevant columns and drop all the other columns. We also drop any NaN values that may exist in our columns to avoid possible errors and NaN propagation. Since machine learning algorithms have trouble dealing with text, we also encode the string values in the "Island" and "Species" columns to numerical values. Now, a value of 0 corresponds to Adelie penguins, a value of 1 to Chinstrap penguins, and a value of 2 to Gentoo penguins in the "Species" column. In the "Island" column, meanwhile, a value of 0 corresponds to Biscoe, a value of 1 to Dream, and a value of 2 to Torgersen. Finally, we split our data set into the predictor variables (X), which consists of all the information we will use to try to predict our target, and the target variable (y), which is the feature that we will try to predict using X.

In [9]:
```
#utilize three variables for models to use
cols = ["Species", "Island", "Flipper Length (mm)", "Culmen Length (mm)"]
penguins = penguins[cols]
penguins.dropna()

#process 'Island' to numeric values for models to use
le=preprocessing.LabelEncoder()
penguins["Island"]=le.fit_transform(penguins["Island"])

#process 'Species' to numeric values for models to use
le=preprocessing.LabelEncoder()
penguins["Species"]=le.fit_transform(penguins["Species"])

#separate into predictor and target variables
X=penguins.drop(["Species"], axis=1)
y=penguins["Species"]
```

Recall that using the same set of data to both train and test our models leaves us susceptible to overfitting; thus, we want to split our data up into a training set (which we will fit our model on) and a testing set (which we will use to test the accuracy of our model at the very end). This will allow us to ensure that the model follows the signal of the data, rather than just fitting to its noise, and that it is capable of predicting classifications even for unseen data.

We thus split our penguins data set into training and testing sets using the `train_test_split()` function from the `sklearn` module. This function takes in our entire penguin data set and splits it randomly into two parts--a training set and a testing set--according to a user-provided test size (in our case, 0.2). We end up with two data frames: `train`, which consists of 80% of the rows of the original penguins data set and which we will use to train our models, and `test`, which consists of 20% of the rows of the original penguins data set and which we will use to test our models. We will not touch the `test` set until the very end, after we have decided on a final model and its parameters, to ensure that we accurately assess our model's performance on entirely unseen data.

Just as we did previously, we also split each of these two sets into predictor and target variables. Our predictor variables are `x_train` and `x_test`, and they contain all the columns except for the `"Species"` column, which is what we're trying to predict. `y_train` and `y_test`, meanwhile, make up our target variables, and consist only of the `"Species"` column.

In [10]:
```
#split train and test data
from sklearn.model_selection import train_test_split

train, test = train_test_split(penguins,test_size=0.2)

#split into predictor and target variables
x_train = train.drop(["Species"], axis=1)
x_test = test.drop(["Species"], axis=1)

y_train=train["Species"]
y_test=test["Species"]
```

We also want to consider some ways we can analyze our models beyond simply scoring them. Some options are confusion matrices and decision region plots, which we can write functions for. Confusion matrices and decision region plots help provide a more complete picture of our model and give us some insight into why the models performed the way they did.

We first define a function that will allow us to produce confusion matrices, which provide a visualization of how well a model performs. In the confusion matrix that we create for the penguins, each row of the matrix indicates the true species of a penguin, and each column indicates its model-predicted species. Thus, the [i,j] element of the confusion matrix tells us the number of penguins in the test set that are actually of species i and are predicted to be of species j. When i = j (i.e. for the elements that lie along the diagonal of the matrix), the predictions match the actual classification.

In [11]:
```
#confusion matrix function
from sklearn.metrics import plot_confusion_matrix
from sklearn import metrics
import seaborn as sns

def Confusion_Matrix(MLmodel, name):
        """
         Plots a Confusion Matrix
         ---
         Parameters

         MLmodel, a model from sklearn Library
         name, a string representing the name of the model
```

```
    ---
    The function creates a confusion matrix
    and then plots it.
    ---

    Returns a graph of the confusion matrix along with a heatmap.
    """

    #predictions are based on x_test data
    y_pred=MLmodel.predict(x_test)

    #create confusion matrix
    matrix = metrics.confusion_matrix(y_test, y_pred)
    %matplotlib inline

    # name of classes we predict
    class_names= penguins["Species"]

    fig, ax = plt.subplots()

    #Labelling tick marks
    tick_marks = np.arange(len(class_names))
    plt.xticks(tick_marks, class_names)
    plt.yticks(tick_marks, class_names)

    #create heatmap
    sns.heatmap(pd.DataFrame(matrix), annot=True, cmap="Blues" ,
                fmt='g', linewidth = 0.2, )
    ax.xaxis.set_label_position("top")
    plt.tight_layout()

    #more labels
    plt.title('Confusion matrix for ' + name + ' model', y=1.1)
    plt.ylabel('Actual Species')
    plt.xlabel('Predicted Species')
```

We also define a function that will allow us to produce decision region plots. A decision region plot essentially provides a visualization for how a model assigns different parts of the data space to different classifications.

In our specific case, we will be plotting the decision regions of the flipper length versus the culmen length. Each plot will contain regions of three different colors, each representing the three species; the plot thus tells us which species a penguin with a given flipper length and culmen length would be classified as by the model.

In [68]:

```
def plot_regions(c,X,y):
    """
    Plots the decision regions of a classifier.
    ---
    Parameters

    c, a model from sklearn Library
    X, a data frame comtaining the predictor variables
    y, a data series containing the target variable
    ---
    The function identifies the decision regions of
    a model and then plots it.
```

```
    ---

    Returns a graph of the decision regions of a classifier.
    """

    #fit model
    c.fit(X,y)

    #need to make a 2d grid of all the points and then
    #attempt to predict at all these points
    grid_x=np.linspace(x0.min(),x0.max(),501)
    grid_y=np.linspace(x1.min(),x1.max(),501)
    xx,yy=np.meshgrid(grid_x,grid_y)

    #unfortunately machine learning models like decision trees like 1-d data
    XX=xx.ravel()
    YY=yy.ravel()

    #make predictions
    p=c.predict(np.c_[XX,YY])

    #reshape p into a 2d array
    p=p.reshape(xx.shape)

    fig,ax = plt.subplots(1) #creates the fig and the ax
    ax.contourf(xx,yy,p,cmap="jet",alpha=.2) #plot the decision boundaries
    ax.scatter(x0,x1,c=y,cmap="jet") #plot the points

    #extract string from column
    col_name = list(X.columns.values)

    #label our axis
    ax.set(xlabel=col_name[0],ylabel=col_name[1])
```

We have finished preprocessing our data and writing any necessary functions we may need for analysis; we are now ready to move on to the modeling.

Modeling

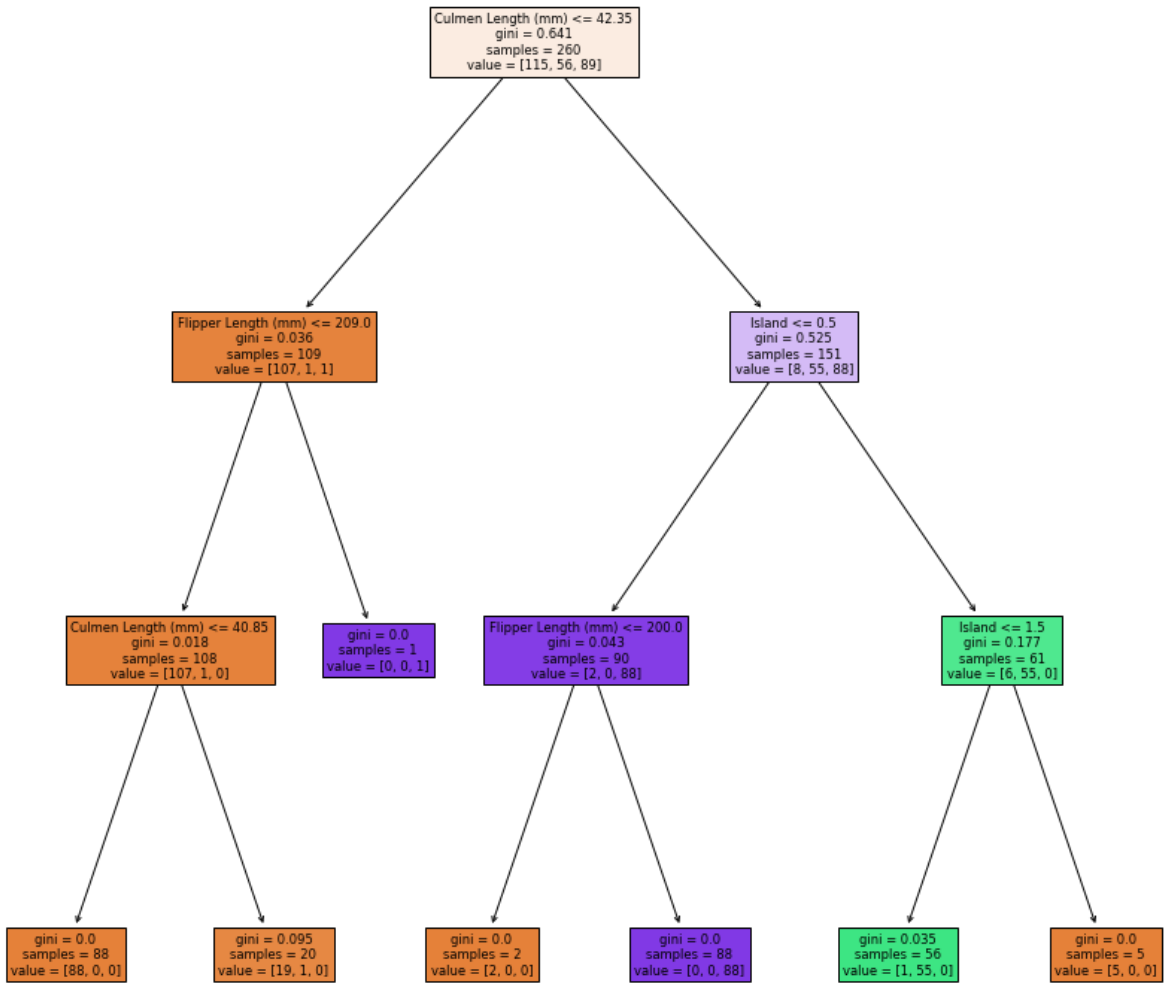
Model 1: Decision Tree

The first machine learning model we will use is the Decision Tree Classifier (DTC). The DTC works like a flowchart; at each step, it asks a question about a feature of our data set, and depending on whether the answer is yes or no, it narrows down the classification options and moves onto another question. The `DecisionTreeClassifier` model takes in the user-provided `max_depth`, which is a hyperparameter that controls how many layers the model's underlying decision tree can have. Just to illustrate this, we create a DTC on our penguin data set with `max_depth` arbitrarily set equal to 3 and plot the tree:

In [13]:

```
#Decision Tree model cross validation
from sklearn import tree
T = tree.DecisionTreeClassifier(max_depth=3)
T.fit(x_train, y_train)

#create Decision Tree plot visualization
```

```
fig, ax = plt.subplots(1, figsize = (15, 15))
p = tree.plot_tree(T, filled = True, feature_names = X.columns)
```



Above we can see a visualization of the DTC. At each step, a yes-or-no question is asked; depending on the answer, we progress along different branches of the tree. Regardless of which path we take down the flowchart, we arrive at some final rectangle, which informs the final classification decision by providing information regarding which species has the highest probability of having that given combination of physical features. For example, if we follow the tree and find that a penguin has a culmen length of <= 42.35 mm, a flipper length of <= 209 mm, and again a culmen length <= 40.85 mm, then our tree tells us that this penguin is most likely an Adelie penguin, because all 88 penguins in our training set which have that specific combination of physical features are Adelie penguins.

Different values of `max_depth` will create models with different levels of complexity, and so we can refine our results and score better accuracy if we can find the optimal value for `max_depth`.

Finding the best value for `max_depth` is a bit tedious; we have to test out a variety of values and score them, and see which value gives us the best score.

Recall that we don't want to touch our testing data until the very end. To test each `max_depth` using our set-aside testing data would be bad practice, since we want to ensure that our final score using the testing data is an entirely honest assessment of how well the model performs on unseen data. Thus, we must use cross-validation.

The idea behind k-fold cross validation (CV) is that, rather than using our testing set to score the various values of `max_depth`, we instead put aside a subset of our *training* set, fit our model to the remainder of the training set, and then score the model using the subset of the training set that we previously set aside. This allows us to see how the model will perform on data that it was not trained with without us having to directly involve our testing data. This process gets repeated *k* times, and averaging these *k* scores gives us an estimate of how well the model will perform on our real test set. We can also use these CV scores to pick an optimal `max_depth`; since the average CV score is a good predictor of what the actual score will be, we can also guess that the `max_depth` value that scores the highest CV score will also be the `max_depth` value that will maximize our real score.

In [15]:
```
#Decision Tree model cross validation
from sklearn.model_selection import cross_val_score

#cross validation to determine best max depth
best_score = -np.inf
best_depth = 0
for d in range (1, 21):
    T = tree.DecisionTreeClassifier(max_depth = d)
    scores = cross_val_score(T, x_train, y_train, cv = 5).mean()

    #save best score, best depth
    if scores > best_score:
        best_score = scores
        best_depth = d

best_depth, best_score
```

Out[15]:
```
(4, 0.9576923076923076)
```

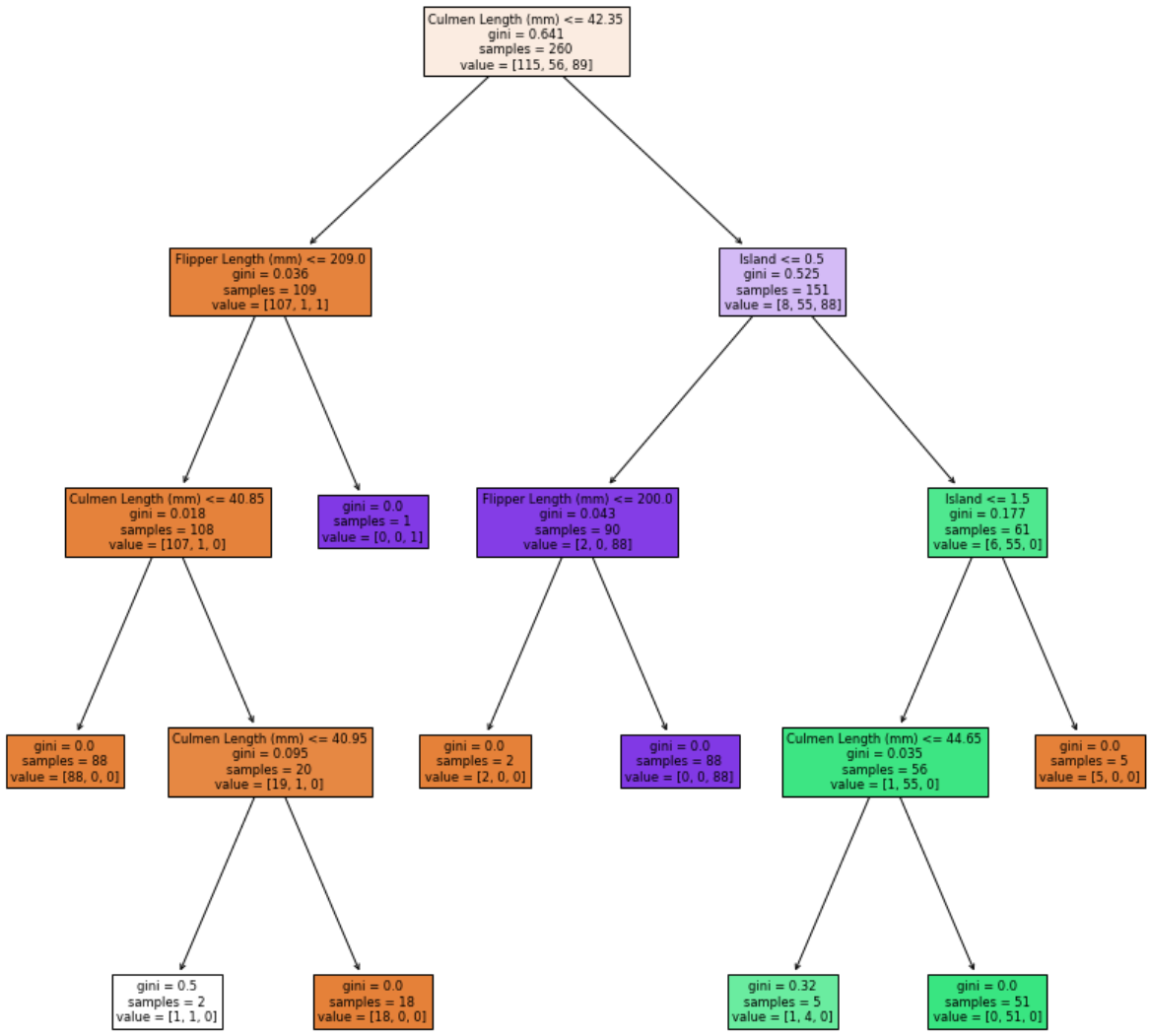
We find that the optimal `max_depth` is 4, with an average cross-validation score of 95.8%. We can use this optimal `max_depth` to create an improved DTC.

In [16]:
```
#we create Decision Tree model using best_depth from cross validation
T=tree.DecisionTreeClassifier(max_depth=best_depth)
T.fit(x_train, y_train)
```

Out[16]:
```
DecisionTreeClassifier(max_depth=4)
```

We can visualize our DTC model as follows:

In [18]:
```
#Our new tree
fig, ax = plt.subplots(1, figsize = (15, 15))
p = tree.plot_tree(T, filled = True, feature_names = X.columns)
```



Notice how this tree has 4 layers this time, which is more than the tree we used for illustrative purposes. In terms of how the model works, this means that more steps are taken (and more questions are asked) before a given penguin is classified. This allows a slightly higher level of specificity (although not too much specificity, which can lead to overfitting), which in turn allows for more accurate classification.

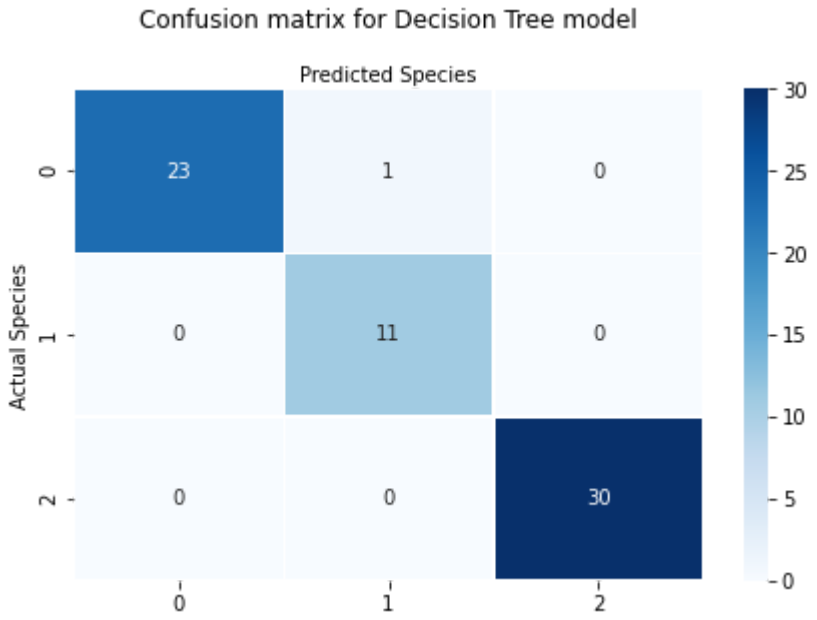
In [19]:
```
#testing Decision Tree model against unseen data
T.score(x_train, y_train), T.score(x_test,y_test)
```

Out[19]:
```
(0.9923076923076923, 0.9846153846153847)
```

We find that our DTC scores about 98.5% on our testing set, which means that our DTC performs extremely well on unseen data--even better than expected from our CV score!

We can further analyze our model results using confusion matrices and decision regions. We first create a confusion matrix for the test set using the `Confusion_Matrix` function that we defined earlier in the section.

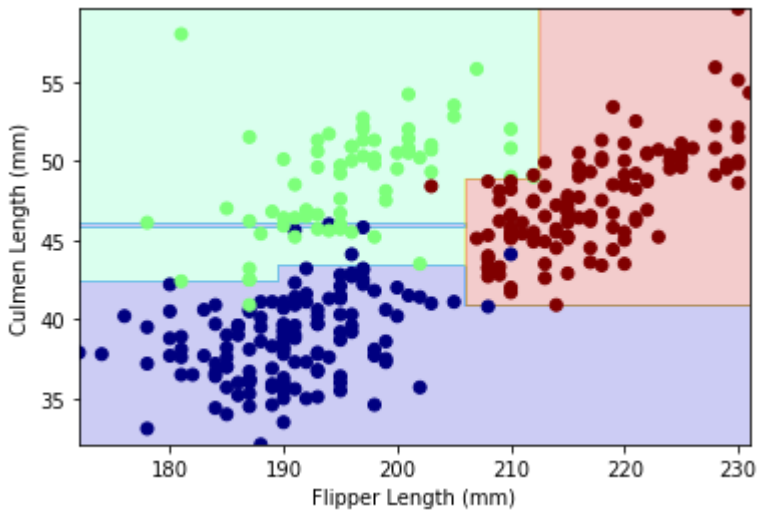
In [20]:
```
#create confusion matrix for Decision Tree model
Confusion_Matrix(T, "Decision Tree")
```



In the above confusion matrix, we see that 23 + 11 + 30 = 64 penguins out of 65 total were correctly classified using our DTC. This matches the accuracy rate of about 98.5% that we found using the `score()` function. The model incorrectly classified only one penguin: an Adelie penguin which the model predicted to be a Chinstrap.

We now create a decision region plot, which illustrates how different regions in the data space correspond to different species classifications.

In [22]:
```
#Decision Regions
penguins=penguins.dropna(subset=["Flipper Length (mm)", "Culmen Length (mm)"])
X=penguins[["Flipper Length (mm)", "Culmen Length (mm)"]]
y=penguins['Species']
x0=X["Flipper Length (mm)"]
x1=X["Culmen Length (mm)"]

plot_regions(T,X,y)
```



We see that these decision regions are mostly accurate. Out of all the penguins in our data set, there appear to be two or three penguins that are in the wrong region, but the vast majority of the penguins get classified correctly based on their culmen length and flipper length.

Model 2: Logistic Regression

Let us now try using another machine learning model: multinomial logistic regression (MLR). Logistic regression uses the logistic function as the basis of its model, and is used to predict probabilities and classifications. Multinomial logistic regression is a version of the latter that has been generalized to problems that have more than two possible outcomes.

To create a MLR classifier, we can use the `LogisticRegression` object from `sklearn` with the argument `multi_class='multinomial'`. To prevent hitting any iteration limits, we can also set `max_iter` to a very large number, like 1000. The `LogisticRegression` model additionally takes in a hyperparameter `C`, which is the inverse of regularization strength. In simpler terms, this hyperparameter controls how much of a penalty is applied for higher complexity models. Recall that increasing the complexity of a model can help make it more accurate, but can also lead to overfitting; thus, this argument `C` is a tool to help prevent overfitting, and the strength with which it does that depends on the value the user gives it.

We will use CV to test several different values for `C` and determine the optimal value, as well as to predict how the MLR model will perform on unseen data.

In [25]:
```
#Logistic Regression model, cross validation
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import cross_val_score

#cross validation to determine best C value
best_score = -np.inf
best_c = 0

for c in [0.1, 0.2, 0.5, 1, 2, 5, 10, 100]:
    LR = LogisticRegression(multi_class='multinomial', C=c, max_iter=1000)
    scores = cross_val_score(LR, x_train, y_train, cv = 5).mean()

    #save best score, best C value
    if scores > best_score:
        best_score = scores
        best_c = c

best_c, best_score
```

Out[25]: (100, 0.9653846153846153)

We find that a `C` value of 100 maximizes the MLR model's CV score (at about 96.5%). We thus choose `C` to be 100 to optimize our MLR model, and then fit it to our training data.

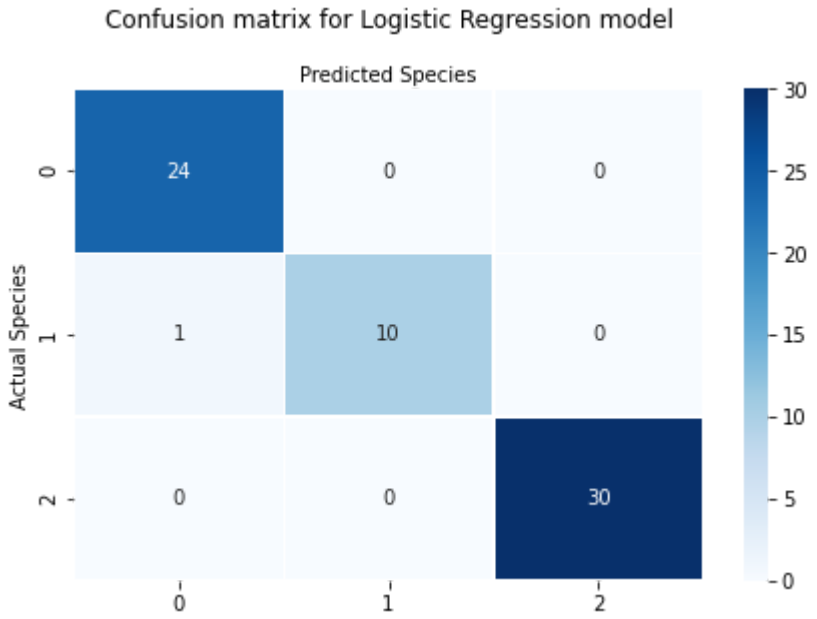
In [33]:
```
LR = LogisticRegression(multi_class='multinomial', C=best_c, max_iter=1000)
LR.fit(x_train, y_train)
LR.score(x_train, y_train), LR.score(x_test,y_test)
```

Out[33]: (0.9692307692307692, 0.9846153846153847)

Our optimized MLR model scores very highly on the testing set, with 98.5% accuracy. It again exceeds the expectations we had based on the CV score.

We move on to analyzing the model, and again create a confusion matrix for the testing set using the `Confusion_Matrix` function.

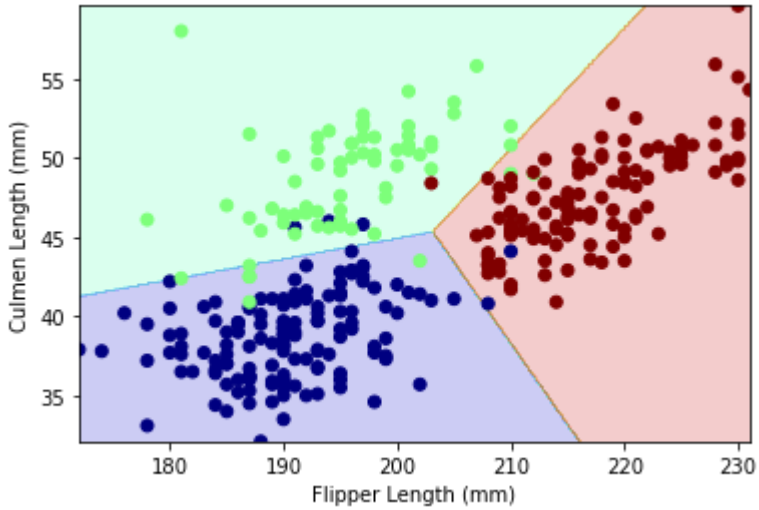
In [34]:
```
#create confusion matrix for Logistic Regression model
Confusion_Matrix(LR, "Logistic Regression")
```



The majority of our non-diagonal values are zero, which supports the high testing score we found. Only one of the testing set penguins were misclassified: a Chinstrap penguin that the model mistook for an Adelie penguin.

To further analyze our model, we plot the decision regions of the model as well, using all of the data from our penguins data set.

In [35]:
```
#Decision Regions
penguins=penguins.dropna(subset=["Flipper Length (mm)", "Culmen Length (mm)"])
X=penguins[["Flipper Length (mm)", "Culmen Length (mm)"]]
y=penguins['Species']
x0=X["Flipper Length (mm)"]
x1=X["Culmen Length (mm)"]

plot_regions(LR,X,y)
```



This decision region plot appears to be much simpler than that of our DTC; in this plot, the regions are separated only by three straight lines, whereas the previous plot displayed many more divisions, with some regions appearing within others. About 10 penguins are noticeably misplaced, which is far greater than the 2 or 3 misplaced penguins in the DTC decision region plot.

This initially seems a bit counterintuitive, as our DTC and MLR models got the same score on the testing set. However, we must keep two things in mind: first, the DTC scored higher on the training set, and the decision region plots take *all* penguins into account, not just the testing set. Second, these decision region plots only show culmen length and flipper length, and do not include island, which is another feature that our models use in their classifications. It is possible that the MLR model takes a penguin's island into higher regard than the DTC when making its classification decisions, and that this in turn causes its decision region plot, which does not include island, to appear less accurate than that of the DTC.

Model 3: Random Forest Classifier

For our third model, we choose to use a Random Forest Classifier (RFC). An RFC model works by using several of the decision tree classifiers that we used in our first model, and fitting them on many different subsets of the data set we are using to train the model. It then takes the average over all of these trees to determine the overall accuracy score. In our model, we provide two arguments: `max_depth`, as in the DTC, controls the complexity of each of the decision trees in the forest, and `random_state` controls the randomness of the data subsets that are used to build each of the trees in the forest.

Just as we did for our decision tree model, we again wish to optimize the `max_depth` hyperparameter by looping over different possible depths and finding which one gives us the best score on unseen data. Since we want to leave our testing set alone until the very end of the modeling process, we must again utilize cross-validation.

In [61]:
```
#RandomForestClassifier model
from sklearn.ensemble import RandomForestClassifier

#cross validation to determine best max depth
best_score = -np.inf
```

```
for d in range(1, 20):
    rfclf = RandomForestClassifier(max_depth=d, random_state=0)
    scores =  cross_val_score(rfclf, x_train, y_train, cv = 5).mean()

    #save best scores, best depth
    if scores > best_score:
        best_score = scores
        best_depth = d

best_depth, best_score
```

Out[61]: (6, 0.9807692307692308)

We find the best `max_depth` to be 6, with a CV score of about 98%. Let us apply this value for `max_depth` to our RFC model:

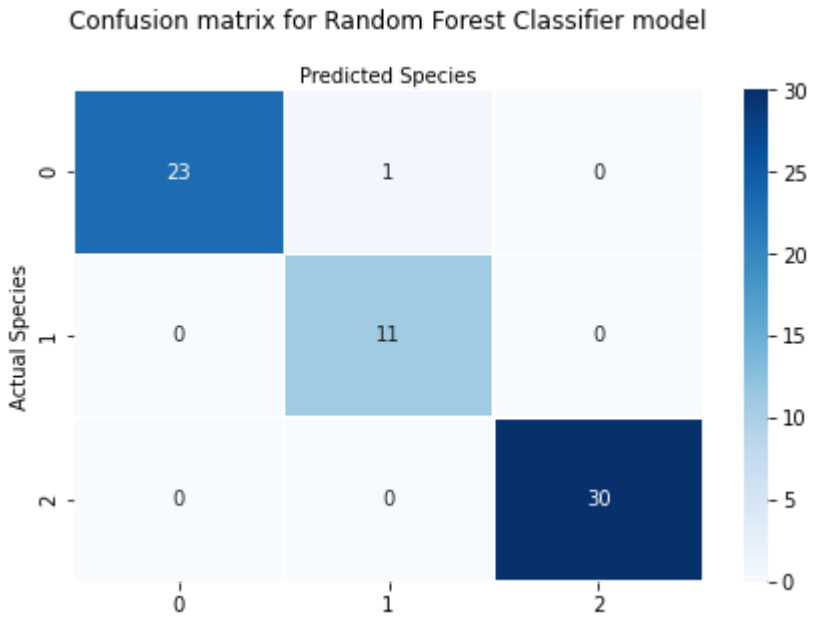
In [64]:
```
#we create Random Forest Classifier model using best_depth from cross validation
rfclf = RandomForestClassifier(max_depth=best_depth, random_state=0)

#testing Random Forest Classifier model against unseen data
rfclf.fit(x_train,y_train),
rfclf.score(x_train, y_train), rfclf.score(x_test,y_test)
```

Out[64]: (0.9961538461538462, 0.9846153846153847)

With the optimized `max_depth` value, we see that the RFC model scores approximately 98.5% accuracy on our testing set. Yet again, our actual score slightly exceeds the CV score.

We again use our previously-defined `Confusion_Matrix` function to create a confusion matrix for the RFC in order to further analyze our model.

In [65]:
```
#create confusion matrix for Random Forest Classifier model
Confusion_Matrix(rfclf, "Random Forest Classifier")
```

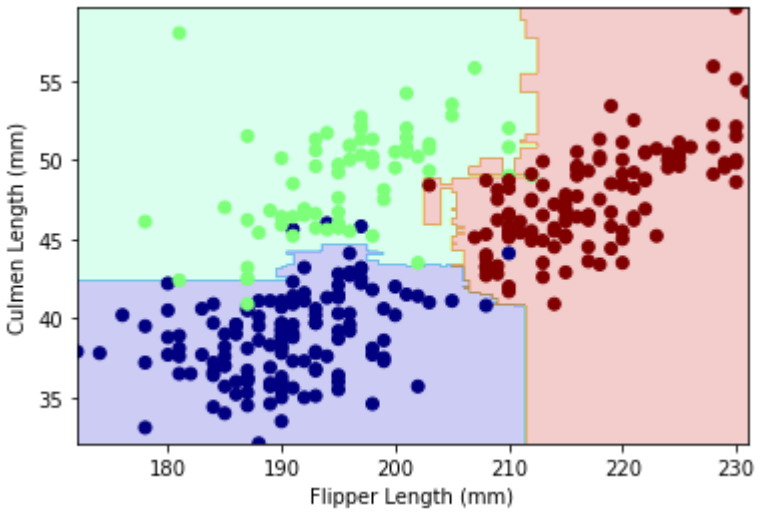


We see that in the above confusion matrix, the majority of the non-diagonal matrix elements are 0, which shows that most of the penguins in the testing set were correctly classified by the RFC model.

The only exception is an Adelie penguin that the classifier mistakenly predicted to be a Chinstrap penguin.

We also plot our decision regions using our `plot_regions()` function to visualize the way the model views the data space.

In [67]:
```
#Decision Regions
penguins=penguins.dropna(subset=["Flipper Length (mm)", "Culmen Length (mm)"])
X=penguins[["Flipper Length (mm)", "Culmen Length (mm)"]]
y=penguins['Species']
x0=X["Flipper Length (mm)"]
x1=X["Culmen Length (mm)"]

plot_regions(rfclf,X,y)
```



This decision region plot appears to be the most complex out of all the plots we have seen thus far; the borders of the regions shown here are the most irregular out of all three decision region plots. About 5 of the points are noticeably misplaced, despite the RFC having better accuracy for both the training and testing sets than the other two models. As we discussed previously, this may be because the decision region plots, unlike the classifiers, fail to take a penguin's island into account.

Discussion

All of the models performed fairly well. The RFC had the highest CV score, with approximately 98.08% accuracy (compared to a CV score of 95.77% for the DTC and 96.54% for the MLR). The RFC also fit the testing set most closely, with a score of 99.62% on the training set, followed closely by the DTC, which scored 99.23% on the training set, and finally the MLR model, which scored 96.92%. Strangely enough, despite these discrepancies, all three models achieved the exact same score on the data set that mattered the most: on the unseen testing set, they all scored 98.46% accuracy.

This leads us to conclude that all three of these models, when paired with the measurements we found (i.e., a `max_depth` of 4 for the DTC, a `C` value of 100 for the MLR, and a `max_depth` of 6 for the RFC), will give very good estimations when faced with unseen data. If we had to pick one, however, we would recommend the RFC--with a `max_depth` of 6 and with the columns

`"Island"`, `"Culmen Length (mm)"`, and `"Flipper Length (mm)"` --as this did have the highest CV score.

Of course, the best model to use in any given scenario depends largely on the type of data being processed, and each classifier has its own strengths and weaknesses. DTCs, for instance, are simple and easy to implement, but they are prone to overfitting and are overly sensitive; a small change in the data being fitted can easily change a tree's entire structure. MLR models, meanwhile, work well with data that is linearly independent and involves fewer dimensions, but has a tendency of overfitting when it comes to high-dimensional data sets. RFCs are good at dealing with high-dimensional data and provide much flexibility, but require a lot of time and resources due to the multiple decision trees that it consists of.

In our case, we had a fairly simple data set that involved only numbers (or strings that could be encoded to numbers) and few dimensions (since we chose only 3 features to use as predictors). Our sample size, especially for the testing set, was also quite small. Additionally, our penguin predictor variables--the length of their culmen, the length of their flipper, and their location--were quite distinct from each other, and this linear independence between our chosen columns further helped our models. The nature of our data set likely contributed a great deal to the accuracy of our models, and allowed them to score so highly.

Despite this, there is always room for improvement. One of the things we could do in the future to refine our models is to use CV to tune other hyperparameters. For instance, the RFC has a hyperparameter called `n_estimators`, which controls the number of trees in the forest and has a default value of 100. Playing around with different values for this argument could change the complexity of the RFC, which in turn could affect our model performance.

We could also try using different measurements as our predictor variables. For all of the models in this project, we used the same three columns: island, culmen length, and flipper length. It is possible that, if we try different sets of columns for different models, we may find that the same columns do not optimize all three of the models, since each of the models takes a different approach to the classification problem. Recall from our penguin summary table in the Exploratory Analysis section that body mass and culmen depth also distinguished certain penguin species from the others; perhaps utilizing one of these could have led to better results for certain models.