Penguins Species-Prediction Machine Learning Group-Project

Instructor: Dr. Michael Murray

Course: PIC 16A

Names: Johnathan Harding and Emily Shi

Date: December 2nd, 2022

Group Contributions Statement

Both of us wrote the data acquisition and preparation. Emily started a couple of the figures, which Johnathan added axes to and polish. Johnathan also added his own figures and Johnathan completed the explanations for the exploratory analysis. Johnathan wrote the feature selection section and the cross-validation tests for both models, including the functions used. Emily led construction of the models and training (with complexity parameters provided by Johnathan). Emily produced the decision region plots and confusion matrices for all models, and Johnathan wrote both the discussion about the decision regions and the general discussion. Johnathan conducted the data writing and analysis of model performance, in both the model used in feature selection and the two actual prediction models. Johnathan wrote almost all of the text cells in this report, explaining both his own code and his partners. We both helped each other, and much of the project was collaborative, including text and actual code revisions. Johnathan also went to office hours multiple times to get information to assist our project.

Data Import, Initial Test-Train Split, and Selective-Cleaning

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split

url = 'https://philchodrow.github.io/PIC16A/datasets/palmer_penguins.csv'
penguins = pd.read_csv(url) #import our data frame

train, test = train_test_split(penguins, test_size = 0.2) # hold out 20% of data
```

Above we split out data directly after import into our training and testing sets. That way, we avoid polluting our testing set.

In [2]:
```
from sklearn import preprocessing
```

```
def clean_data(df): #Emily and Johnathan
  '''
  Clean data function to make dataframe usable. First, we drop NAN-values for
  select categories. Then we drop the comments column. Next, we drop cells
  where sex isn't encoded properly. Lastly, we encode qualitative data.
  param: df The dataframe to be cleaned.
  returns: the cleaned dataframe.
  '''

  # #drop empty NAN values
  df = df.dropna(subset=['Island', 'Culmen Length (mm)', 'Flipper Length (mm)',
                         'Body Mass (g)', "Delta 15 N (o/oo)"])

  df = df.drop(["Comments"],axis=1) #drop comments column

  df = df[df['Sex'] != '.'] #boolean indexing to remove empty entries for sex

  le = preprocessing.LabelEncoder() #shorthand

  df['Species'] = le.fit_transform(df['Species']) #encode Region
  df['Region'] = le.fit_transform(df['Region']) #encode Region
  df['Island'] = le.fit_transform(df['Island']) #encode Island

  return df
```

Our **clean_data** function makes our data **simpler and usable** for our purposes. We do not need the entire data set! Also, we have to **drop NAN-values** and **encode qualitative** so we can analyze the data and model in the first place.

In [3]:
```
cleaned_data_train = clean_data(train)
```

Now, we have **cleaned training data** to allow us to do all the analyses we need to do before we get to the actual modeling. Analyses will include automated feature selection and model-specific **cross-validation** so we can use optimized **complexity parameters** (more on that later...).

NOTE: The only data that has been cleaned thus far is our *training* data. We want to avoid polluting our test set.

EXPLORATORY ANALYSIS:

Displayed Tables (Total 2): (Johnathan)

In [4]:
```
pert_vals = ["Culmen Length (mm)", "Flipper Length (mm)",
             "Body Mass (g)", 'Delta 15 N (o/oo)' ] #the pertinent values
table_groups = ["Species", "Sex", "Island"] #the groupings we are interested in

#display average quatitatives, as by the above groups, with rounded values
train.groupby(table_groups)[pert_vals].mean().round(2)
```

Out[4]:

Species	Sex	Island	Culmen Length (mm)	Flipper Length (mm)	Body Mass (g)	Delta 15 N (o/oo)
Adelie Penguin (Pygoscelis adeliae)	FEMALE	Biscoe	37.79	187.25	3384.38	8.72
		Dream	36.94	188.00	3344.00	8.93
		Torgersen	37.65	187.95	3421.43	8.68
	MALE	Biscoe	40.06	190.71	4039.29	8.87
		Dream	40.24	192.75	4056.25	9.02
		Torgersen	40.98	195.50	4073.75	8.93
Chinstrap penguin (Pygoscelis antarctica)	FEMALE	Dream	46.66	191.72	3495.69	9.29
	MALE	Dream	51.18	200.28	3937.00	9.50
Gentoo penguin (Pygoscelis papua)	.	Biscoe	44.50	217.00	4875.00	8.04
	FEMALE	Biscoe	45.78	212.80	4719.38	8.18
	MALE	Biscoe	49.50	221.67	5468.75	8.30

Using **groupby()**, we can generate a nice simplified version of the data set showing a breakdown of the important qualitative information about the three penguin species. With this table we can get an idea of how the species might be easily grouped.

In [5]:
```
#display average quatitatives, as by the above groups, with rounded values
islands_species_GBobj = train.groupby(["Species", "Island"])
# Reset index of grouped data to make usable table
islands_species_df = islands_species_GBobj.size().reset_index()

islands_species_df
```

Out[5]:

	Species	Island	0
0	Adelie Penguin (Pygoscelis adeliae)	Biscoe	30
1	Adelie Penguin (Pygoscelis adeliae)	Dream	49
2	Adelie Penguin (Pygoscelis adeliae)	Torgersen	46
3	Chinstrap penguin (Pygoscelis antarctica)	Dream	54
4	Gentoo penguin (Pygoscelis papua)	Biscoe	96

Especially in this simplified table, one can see that **Chinstraps** and **Gentoos** are pretty much confined to **Biscoe** and **Dream** island, while the **Adelies** are not.

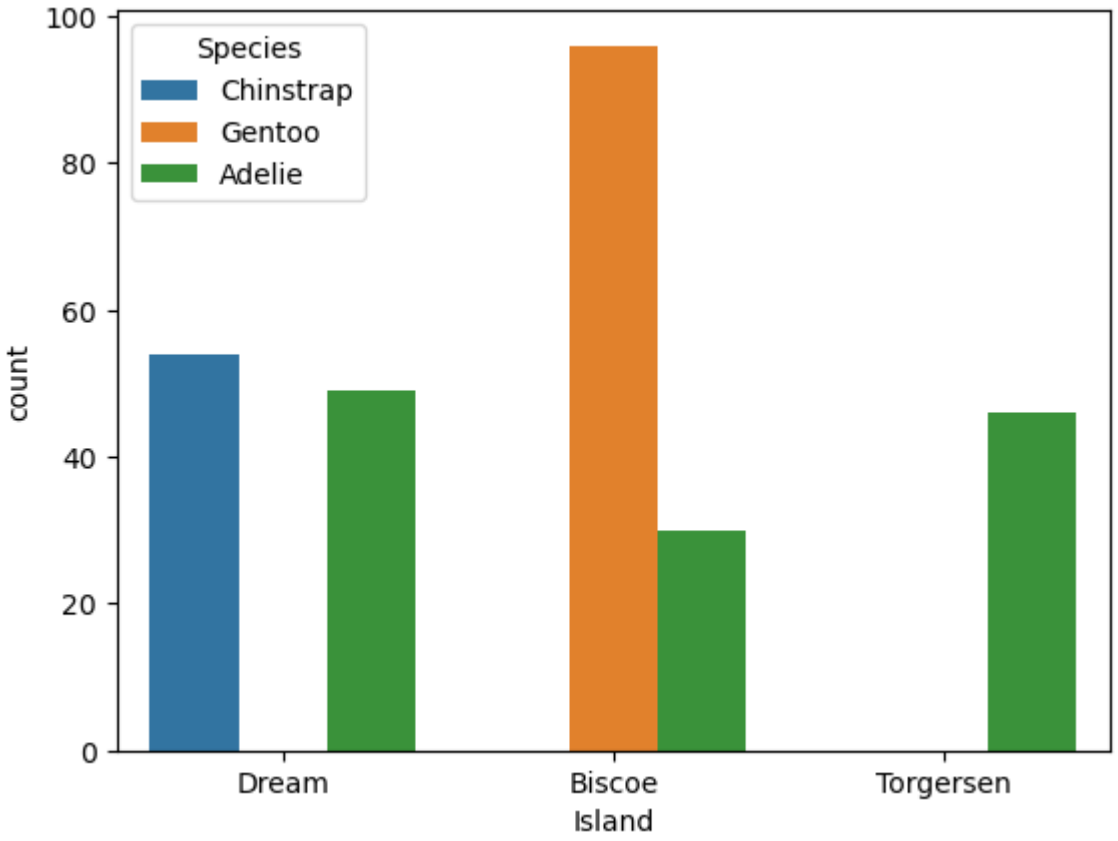
Our goal in this project is to predict **species**, so we will use **Island** as our qualitative predictor variable because the penguins are seemingly fairly organized by species-island anyway.

NOTE: BOTH tables employ the groupby function in order to to summarize data across multiple qualitative categories. \

Figures (Total 12):

FIGURE I: Species Count by Island (Emily and Johnathan)

In [6]:
```
# ## Qualitative: Island

#make a series from penguins::species column
speciesCount = penguins[['Species']].copy()

#make a copy so clipping species name not applied to actual dataframe
trainCp = train

#clip species name for brevity
trainCp["Species"] = trainCp["Species"].str.split().str.get(0)

#encode species
speciesCount.replace(['Adelie', 'Chinstrap','Gentoo'], [1, 2, 3])

#plot countplot of species by island
fgrid = sns.countplot(x = 'Island', #x axis is island
                      hue = "Species", #color based on species
                      data = trainCp) #pull from copy of full dataframe
```



Explanation (Johnathan): This plot is especially important because it highlights the value of the Island data. It is clear that there is a substantial correlation between island and the particular species that live on each island. As one can make out from the figure, all Gentoos live on Biscoe island, and all Chinstraps live on Dream island. Adelies are much more evenly distributed, but

Torgersen has only Adelies. Therefore, we can definitively predict if a penguin lives on Torgersen, it is an adelie.

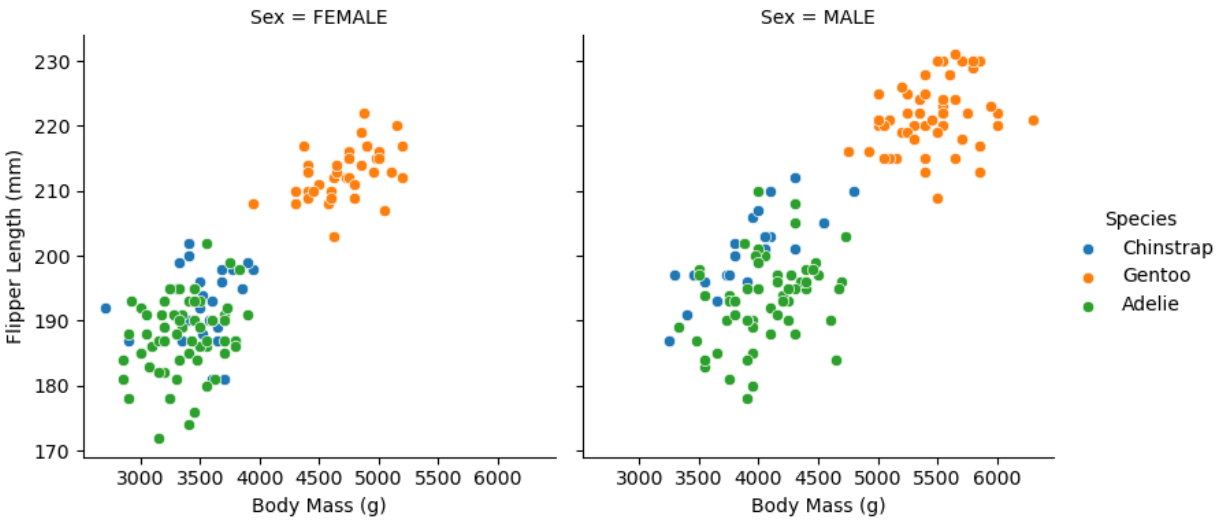
If a penguin is from another island, we still have valuable information, because we can rule out Gentoos if the island is Biscoe and likewise for Chinstraps if the penguin is from Dream island. Island is a great way to start the prediction process.

FIGURE II: Body Mass and Flipper Length Across Sexes (Johnathan)

In [7]:
```
#Johnathan, our figure with multiple axes

trainCp = trainCp[trainCp['Sex'] != '.'] #clip one unknown value for sex
#make facet grid object with columns
plot = sns.FacetGrid(trainCp, col= "Sex", hue="Species", size=4, aspect=1)
plot.map(sns.scatterplot, "Body Mass (g)", "Flipper Length (mm)") #map scatter
plot.add_legend() #add nice legend
```

```
C:\Users\emily\anaconda3\lib\site-packages\seaborn\axisgrid.py:337: UserWarning: The
`size` parameter has been renamed to `height`; please update your code.
  warnings.warn(msg, UserWarning)
```

Out[7]:
```
<seaborn.axisgrid.FacetGrid at 0x207321e9100>
```



Explanation (Johnathan): We include this plot because we needed to determine if important features of a penguin, here **body mass and flipper length**, have **a) notable differences between species, and b) these differences between species are consistent across sexes.**

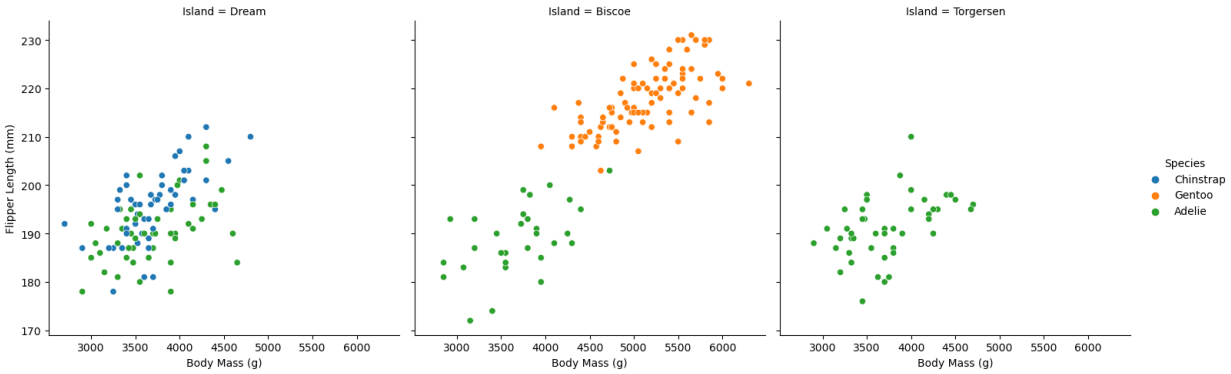
Point a's value is obvious; we can differentiate species by such features as body mass and flipper length.

Point b is important, too, however, because if the differences are not consistent, then we might confuse a male member of one species for a female of another because, while features might differ between species, the difference is less so depending on the sex of the individual. *Perhaps, for example, male Gentoos generally have small flippers than Chinstraps, but male Gentoos do not have smaller flippers than female Chinstraps, or something of that nature.* bold text Here we can see that, at least for Gentoos, **body mass** and **flipper length** are indeed quite different

across the sexes than Chinstraps and Adelies, and can perhaps be used as a pairing for prediction.

FIGURE III: Body Mass and Flipper Length, Across Islands (Emily)

In [8]:
```
fgrid = sns.relplot(x = 'Body Mass (g)', #set x and y axes
                    y = 'Flipper Length (mm)',
                    hue = 'Species', #set hue
                    kind = 'scatter', #set type
                    ci = None,
                    data = trainCp, #data
                    col = "Island") #make columns

axes = fgrid.axes.flatten() #collapse into 1 dimension
```



Explanation (Johnathan): Figure III is quite interesting because it demonstrates that differences in body mass and flipper length are even across the islands. Consistency across islands is important because it supports the idea that the differences between species are not simply because of localized trends, like the amount of food in an area, that could be the real cause of differences between groups.

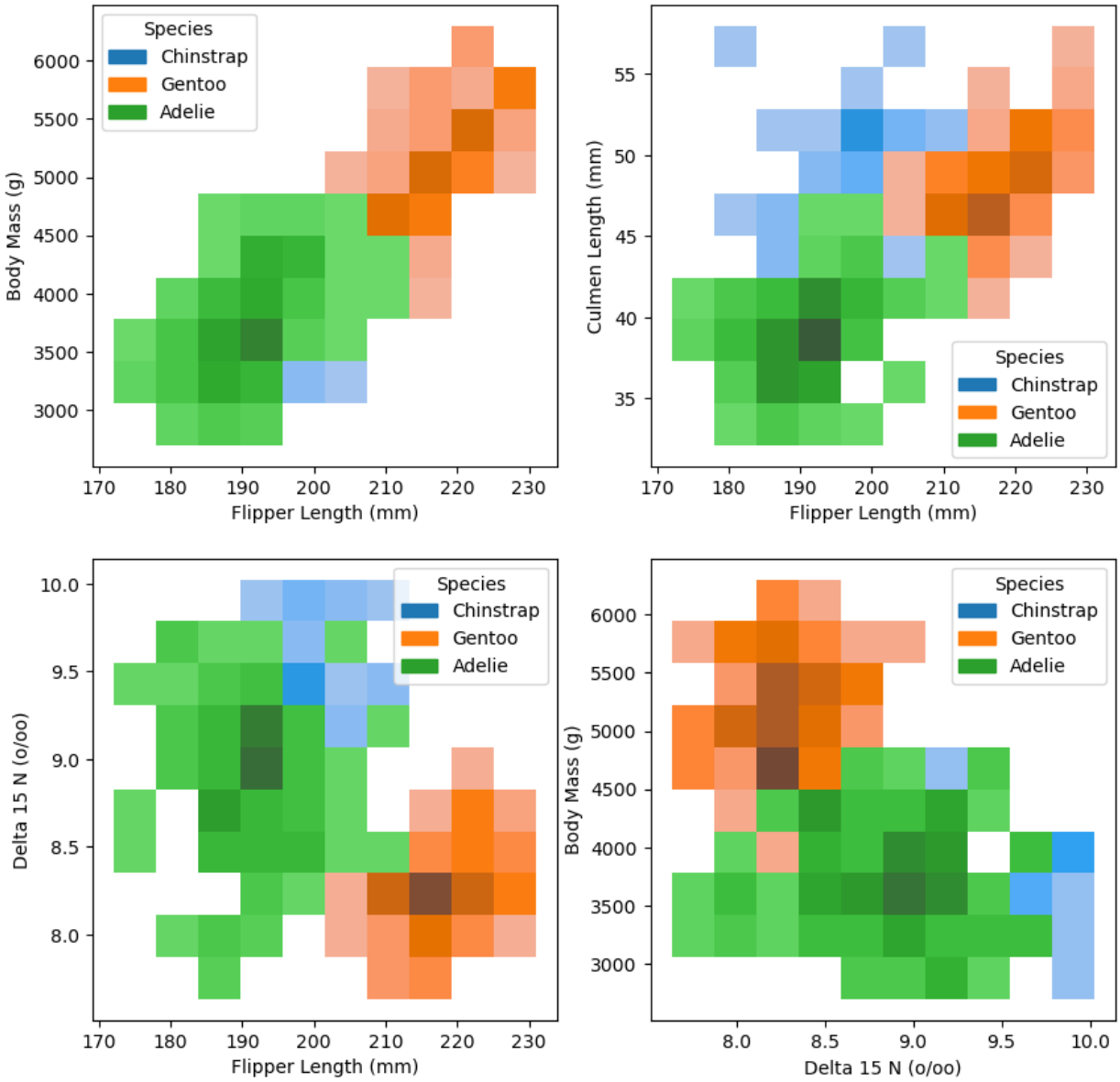
If measures for a species were inconsistent across islands for each species, it would harder to distinguish species because any differences might just be due to local effects. However, as the plot shows, Adelies, for example, have consistently lower body mass and flipper length across the islands. This usefully suggests these trends are not simply due to lack of food or another environmental effect but truly features of the species.

Therefore, it makes sense to consider body mass and flipper length for predicting purposes, though further analysis is certainly necessary.

FIGURE IV: HEATMAPS BY SPECIES, FOR MULTIPLE QUANTITATIVE PAIRINGS (Johnathan and Emily)

In [9]:
```
#Johnathan
fig, axs = plt.subplots(2, 2, figsize=(10, 10)) #create our figure

sns.histplot(x = 'Flipper Length (mm)', #setting axes for plot
             y = 'Body Mass (g)',
             hue = 'Species',
             data = trainCp, ax=axs[0, 0]) #first quadrant, histogram
```

```
sns.histplot(x = 'Flipper Length (mm)', #setting axes for plot
             y = 'Culmen Length (mm)',
             hue = 'Species',
             data = trainCp, ax=axs[0, 1]) #second quadrant, histogram

sns.histplot(x = 'Flipper Length (mm)', #setting axes for plot
             y = 'Delta 15 N (o/oo)',
             hue = 'Species',
             data = trainCp, ax=axs[1, 0]) #third quadrant, histogram

sns.histplot(x = 'Delta 15 N (o/oo)',#setting axes for plot
             y = 'Body Mass (g)',
             hue = 'Species',
             data = trainCp, ax=axs[1, 1]) #fourth quadrant, histogram

#setting legends in suitable places
sns.move_legend(axs[0, 0], "upper left")
sns.move_legend(axs[0, 1], "lower right")
```



Explanation (Johnathan): Figure V contains a lot of useful information that helped motivate our modeling choices. We sought to show side-by-side a lot of distributions of **quantitative**

data by **species** to see which pair might be a good start for our feature selection.

The **upper left plot** demonstrated that the pairing of flipper length and body mass would be a **poor pairing** because the data is fairly inter-mixed, especially in the range of body mass as flipper length increases (penguins with similar sized flippers have similar mass, regardless of species). Here, we can see body mass is really not a great predictor because the species are fairly similar in mass.

The **upper right sub-figure** yields a **much better pairing**, one we were excited to verify with our automated feature modeling: flipper length and culmen length. We have solid grouping by species along these two measures, in both the x and x axes.

The **lower left plot** demonstrates another **poor pairing:** delta 15 N and flipper length. It would seem that for Adelies and Chinstraps, both species had higher delta 15 N levels, regardless of flipper length, making these two species less distinguishable on these two measures, even if it is a fairly good pairing to separate them at least from Gentoos.

The **bottom right sub-figure** shows another **less-than-ideal pairing:** delta 15 N levels versus body mass. For all three species, these measure were fairly similar at mid-range body mass and delta 15 N levels, so it would simply be too confusing to make a good model, though we did test it in our automated feature selection part of our project.

NOTE: We have multiple figures with more than one axis.

FEATURE SELECTION (using a logistic-regression model and cross-validation): Johnathan

1 *Qualitative Feature:* ISLAND

We have settled on choosing "Island" as our qualitative feature for prediction because, as stated in the exploratory analysis, the penguins are fairly organized by island-species. This situation will be useful in modeling because we have a solid qualitative base to guide our predictions which we can then add nuance to via the right quantitatives.

2 *Quanitative* Features:

In order to pick our **quantitatives**, we will perform a series of **cross-validations** on a **logistic regression** model that uses all possible pairs from the set of four quantitative features used in the exploratory analysis.

Then, we compare the various **cross-validation scores** to determine which ought to be used to train our three models.

Before we move on, we need to import some useful modules...

In [10]:
```
from sklearn.linear_model import LogisticRegression #for our LR model
from sklearn.model_selection import cross_val_score #to cross validate
```

```
from sklearn.pipeline import make_pipeline #to help us scale our data

from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
```

Below is Johnathan's **exhaustive seach** for the **feature selection**, based on the one from lecture. We will use this algorithm to explore all possible explorations of the pairs of quantitatives with our selected-qualitative feature.

In [11]:
```
def exhaustive_search(model, X, y, quals, quants): #Johnathan
    '''
    Exhaustive search algorithm here:
    Iterates through all subsets of included quantitatives with added
    qualitative identify best subset of columns in terms of cross validation
    error. Prints best combination before returning both
    parameters:
      model: to use
      X: predictor data
      y: target data
      quals: the qualitative variable
      quants: list of quantitative variables
    returns:
      best_cv: calculated best CV-score
      best_feature_comb: calculated best feature combination

    '''
    best_cv = 0 #initialize our best CV score and feature
    best_feature_comb = None

    #for every given qualitative value
    for qual in quals:

        #for every pair-wise combination of given quantitative values
        for quant_comb in combinations(quants, 2):

            qual_quant_comb = list(quant_comb)+list(["Island"])

            #print which combo we're on...
            print("Current qual_quant_comb is: " + str(qual_quant_comb))

            model.fit(X[qual_quant_comb].values, y) #fit model

            #conduct cross-validation score
            cv = cross_val_score(model, X[qual_quant_comb], y, cv = 10).mean()

            #print the score for viewing
            print("\tCurrent qual_quant_comb CV Score is: " + str(cv))

            if cv > best_cv: #if we've a new best CV, replace!
                best_cv = cv
                best_feature_comb = qual_quant_comb
                #^^better CV score = better combo! Replace old-best combo!

    #nice line for clear print-out
    print("-----------------------------------------------------")
    #nice message about CV score
    print("Best CV Score combination is: " + str(best_cv))
```

```
    #nice message about feature score
    print("Best feature combination is: " + str(best_feature_comb))

    return best_cv, best_feature_comb
```

Below is Johnathan's **test_column** function for the feature selection, based on the one from lecture. We will use this algorithm to make sure our selected set of features really is as good as our **exhaustive_search** reports.

In [12]:
```
def test_column_score(model, test_cols, X, y ):
    """
    Trains and evaluates a model on the test set using the columns of the data
    with selected indices
    param: model the model to be used
    param: test_cols the columns to test with.
    param: X the predictor data
    param: y the target data
    returns: score of model with trained with provided columns.
    """
    print("TESTING WITH COLUMNS: " + str(test_cols))
    model.fit(X[test_cols].values, y)

    return model.score(X[test_cols].values, y)
```

The next cell sets up all our **data** that we will use in **feature selection**, completeing another split from the cleaned *training* data to funnel into our exhaustive search and test_columns functions.

See comments for details...

In [13]:
```
from itertools import combinations #for easy generation of combinations

quals = ["Island"] #our selected qualitative feature
#possible quantitative features
quants = ["Culmen Length (mm)", "Flipper Length (mm)",
          "Body Mass (g)",      "Delta 15 N (o/oo)"]

#making testing and training data for automated feature selection
X = cleaned_data_train[['Island', 'Culmen Length (mm)', 'Flipper Length (mm)',
                        'Body Mass (g)', 'Delta 15 N (o/oo)']]
y = cleaned_data_train['Species']

#SPLIT data for cross-validation stuff.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

#the model we will use for the exhaustive search
my_model = tree.DecisionTreeClassifier(max_depth=3, random_state = 42)

#exhaustive search returns two outputs, the best score AND best combination
best_CVS, best_cols = exhaustive_search(my_model, X_train, y_train,
                                        quals, quants)
```

```
Current qual_quant_comb is: ['Culmen Length (mm)', 'Flipper Length (mm)', 'Island']
        Current qual_quant_comb CV Score is: 0.971190476190476
Current qual_quant_comb is: ['Culmen Length (mm)', 'Body Mass (g)', 'Island']
        Current qual_quant_comb CV Score is: 0.9664285714285713
Current qual_quant_comb is: ['Culmen Length (mm)', 'Delta 15 N (o/oo)', 'Island']
        Current qual_quant_comb CV Score is: 0.9664285714285713
Current qual_quant_comb is: ['Flipper Length (mm)', 'Body Mass (g)', 'Island']
        Current qual_quant_comb CV Score is: 0.7988095238095239
Current qual_quant_comb is: ['Flipper Length (mm)', 'Delta 15 N (o/oo)', 'Island']
        Current qual_quant_comb CV Score is: 0.8707142857142858
Current qual_quant_comb is: ['Body Mass (g)', 'Delta 15 N (o/oo)', 'Island']
        Current qual_quant_comb CV Score is: 0.8078571428571427
----------------------------------------------------------
Best CV Score combination is: 0.971190476190476
Best feature combination is: ['Culmen Length (mm)', 'Flipper Length (mm)', 'Island']
```

As we can see from the print out above, our ***exhaustive search*** yield's the best combination of ***quantitative features*** with "Island" and accompanying ***CV scores***. The power of this method is we've tried *every* option for the four quanitatives and our qualitative selected above and found the best one...

The algorithm tells us the best combination to use (based on CV score) is...

['Culmen Length (mm)', 'Flipper Length (mm)', 'Island'] !

Great! We trained our ***decision tree*** model and assessed performance with a cross-validation. We still need to assess performance with non-training data. We will use our ***test_column_score*** function (defined above).

Alright! As we can tell from the above print-out, scores with ***training*** and ***testing*** data are both high and within 5% of one another. We have found a good set of predictor variables to use for our two models.

Our selected features for our ***multinomial regression*** and ***random forest*** models will thus be... *drum-roll! ...*

Culmen Length, Flipper Length, and Island!

In [14]:
```
#test the "best" columns
best_cols_test_score = test_column_score(my_model, best_cols, X_test, y_test)
print("\nTRAINING score with " + str(best_cols) + ": " + str(best_cols_test_score) + '
      str(best_cols) + ": " +str(best_CVS)) #snaggle-tooth, but then we can print in c
```

```
TESTING WITH COLUMNS: ['Culmen Length (mm)', 'Flipper Length (mm)', 'Island']

TRAINING score with ['Culmen Length (mm)', 'Flipper Length (mm)', 'Island']: 0.981132
0754716981
vs.
TESTING score with ['Culmen Length (mm)', 'Flipper Length (mm)', 'Island']: 0.9711904
76190476
```

Machine-Learning Models:

Importing necessary modules...

In [15]:
```
import sklearn
from sklearn import metrics
from sklearn.linear_model import LogisticRegression
```

Here, we at last clean our testing data and conduct the final test-train split.

In [16]:
```
cleaned_data_test = clean_data(test)
```

In [17]:
```
### Data for models (final cleaning)

X_train = cleaned_data_train[['Culmen Length (mm)', 'Flipper Length (mm)',
                              'Island']]
y_train = cleaned_data_train['Species']

X_test = cleaned_data_test[['Culmen Length (mm)', 'Flipper Length (mm)',
                            'Island']]
y_test = cleaned_data_test['Species']
```

NOTE: Before creating each model object, we need to define a function that performs the fitting of train data and generating of an accuracy score. Additionally, evaluation and visualization of the models are performed through functions generating confusion matrices and decision region plots. All three of these functions are defined prior to the instantiation of a new model object.

FOR BOTH MODELS: Functions for Fitting/Scoring, Confusion Matrices, & Decision Regions (Emily)

Fitting and Scoring

In [18]:
```
# EMILY
def modelFitScore(ML):
  '''
  Fits the model to the predictor and target variables. Computes the accuracy
  score of the model.
  param: Scaled ML model
  param: pred Predictions of values based on test set
  returns: nothing.
  '''
  # fit model
  ML.fit(X_train.values, y_train)
  # test the model using X_test
  y_test_pred = ML.predict(X_test)
  # score model
  test_ML_score = ML.score(X_test.values, y_test)
  print('test_ML_score: ' + str(test_ML_score))
```

Confusion Matrix Generator

In [19]:
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
from sklearn.metrics import precision_score

def confusionMatrix(ML):
  '''
  Creates a confusion matrix of size N x N, where N is the number of outputs.
  The confusion matrix allows for comparison between actual and predicted values.
  param: ML model
  returns: nothing.
  '''
  # test the model using X_test
  y_test_pred = ML.predict(X_test.values)
  # generating confusion matrix comparing the actual and predicted values
  cm = confusion_matrix(y_test, y_test_pred, labels=ML.classes_)
  # displaying the confusion matrix
  disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=ML.classes_)
  disp.plot()
  plt.show()
```

For each model, the confusion matrix is a 3 x 3 matrix because there are 3 classes of species that are outputted.

Grid of cells:

1 2 3

4 5 6

7 8 9

Cell 1 = True Positive: actual value and predicted values are the same

Cell 2 + Cell 3 = False Negative: actual value is positive but the model predicted it as negative (model has given the wrong prediction)

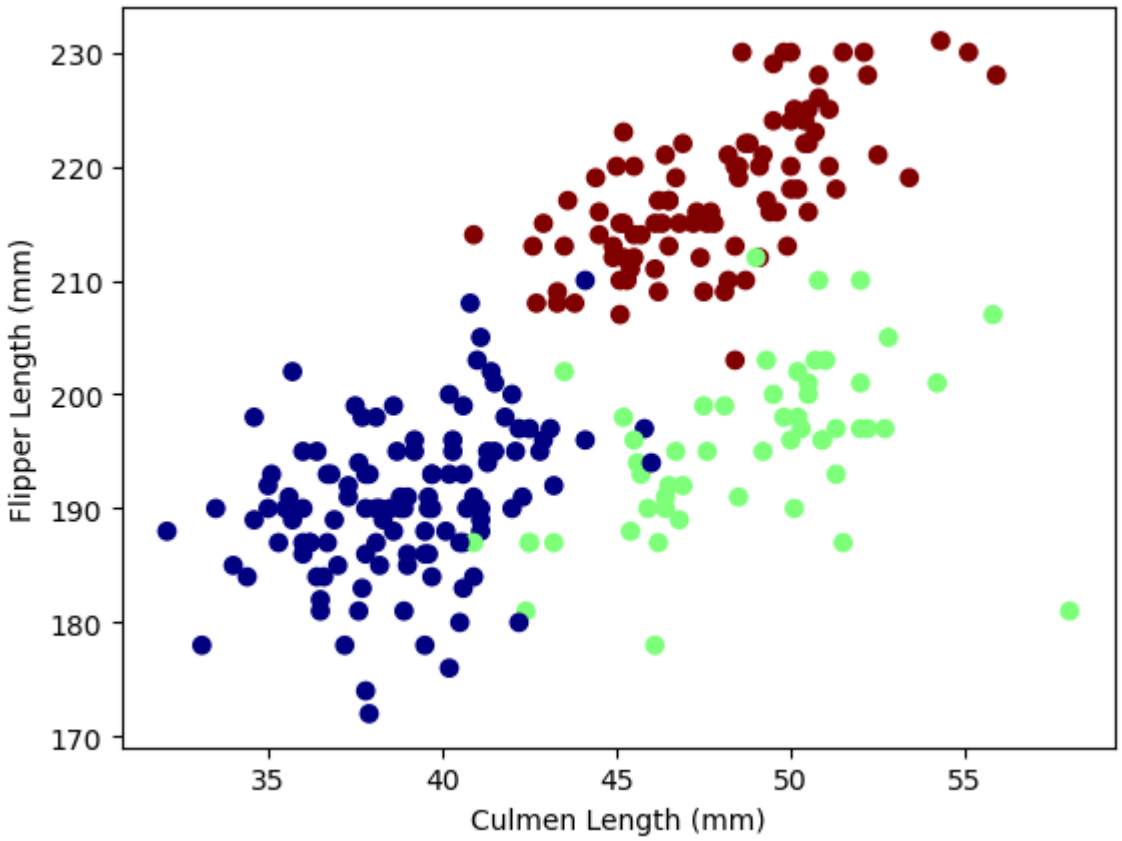
Cell 4 + Cell 7 = False Positive: actual value is negative but the model predicted it as postive (model has given the wrong prediction)

Cell 5 + Cell 6 + Cell 8 + Cell 9 = True Negative: actual value and predicted values are the same

Scatter Plot of Selected Quantitatives

In [20]:
```
fig, ax = plt.subplots(1)

ax.scatter(X['Culmen Length (mm)'],
           X['Flipper Length (mm)'],
           c = y,
           cmap = "jet")

ax.set(xlabel = "Culmen Length (mm)",
       ylabel = "Flipper Length (mm)")
```

Out[20]:
```
[Text(0.5, 0, 'Culmen Length (mm)'), Text(0, 0.5, 'Flipper Length (mm)')]
```



Decision Regions Generator

In [21]:
```
from matplotlib import patches as mpatches

def plot_regions(m, X, y, t):
  '''
  Fits the model to the predictor and target variables. Computes the accuracy
  score of the model.
  param:
  returns: nothing.
  '''
  islands = [i for i in X['Island'].unique()]
  fig, ax = plt.subplots(1, len(islands), figsize = (12, 7))


  # ive names to the two columns of the data
  x0 = X['Culmen Length (mm)'].values
  x1 = X['Flipper Length (mm)'].values

  grid_x = np.linspace(x0.min() - 1, x0.max() + 1)
  grid_y = np.linspace(x1.min() - 1, x1.max() + 1)
  f1, f2 = np.meshgrid(grid_x, grid_y)


  for i in islands:

    Z = m.predict(np.c_[f1.ravel(), f2.ravel(),
                        np.ones(f1.ravel().shape) * 1.0 * i])
    Z = Z.reshape(f1.shape)

    ax[i].set_xlabel('Culmen Length (mm)')
    ax[i].set_title('Island = ' + str(i))
```

```
    recode={0:'blue',1:'green',2:'red'}
    color_map=y.map(recode)

    # 0 for train set (scatter plot)
    if t == 0:
      ax[i].scatter(X[X['Island']==i]['Culmen Length (mm)'],
                    X[X['Island']==i]['Flipper Length (mm)'],
                    c=color_map[X['Island']==i], cmap='jet')
      fig.suptitle('Decision Regions: Train Set')
    # 1 for test set (scatter plot)
    else:
      ax[i].scatter(X[X['Island']==i]['Culmen Length (mm)'],
                    X[X['Island']==i]['Flipper Length (mm)'],
                    c=color_map[X['Island']==i].values, cmap='jet')
      fig.suptitle('Decision Regions: Test Set')
    # plot decision regions
    ax[i].contourf(f1, f2, Z, alpha=0.2, cmap='jet')


  # sets only one y-axis label for all plots
  ax[0].set_ylabel('Flipper Length (mm)')

  # creating the legend
  legend0 = mpatches.Patch(color = 'red', label = 'Adelie', alpha = 0.2)
  legend1 = mpatches.Patch(color = 'green', label = 'Chinstrap', alpha = 0.2)
  legend2 = mpatches.Patch(color = 'blue', label = 'Gentoo', alpha = 0.2)
  fig.legend(handles = [legend0, legend1, legend2],loc = (0.9,0.92),
             fontsize = 'medium',framealpha = 1)

  fig.tight_layout()
  plt.show()
```

MODEL #1: Complexity

Cross Validation for L2 Regularization Constant Complexity Hyperparameter (Johnathan)

Here, we will **cross-validate** against different levels of regularization to find the best amount for our model. Then, we will know how to set our complexity parameters for regularization for our **multinomial logistic regression** model.

In [22]:

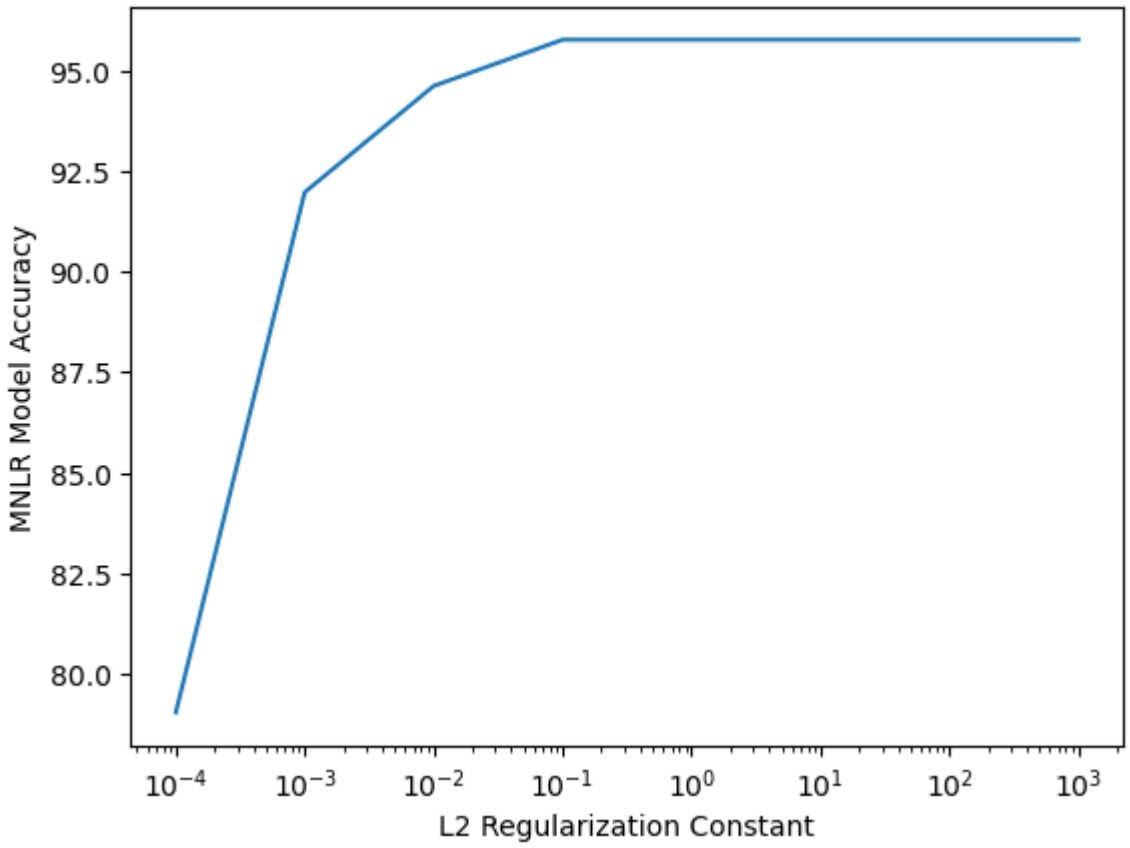
```
def M1cross_val(X, y, vary): #Johnathan
  '''
  This function will run cross_validation for our first model to determine the best
  level for complexity parameter C. (specialized for multinomial Log. regression)
  param: X Training data.
  param: y Target data.
  param: vary List containing values for complexity parameter to test on.
  returns: nothing.
  '''

  best_cv = 0 #initialize values for cross-validation
  top_C = None
  scores = []
```

```
  for c in vary:
    #instantiate model with proper regularization penalty and C value
    CV_score_me = LogisticRegression(penalty = "l2",
                                     multi_class = 'multinomial',
                                     solver = 'newton-cg',
                                     max_iter = 1000,
                                     C=c) #
    CV_score_me.fit(X, y) #fit the model
    cv = cross_val_score(CV_score_me, X, y, cv = 10).mean()
    scores.append(cv*100)

    #print nice message per cv score for easy reading
    print( "CV score for complexity parameter of C=" + str(c) + ": " + str(cv))

    if cv > best_cv: #if we found a better C value
      best_cv = cv #new cv
      top_C = c #new C value

  #line so it doens't hurt as much to read
  print( "-----------------------------------------")

  #print outs for the best CV score and C. needed for human-comparison
  print( "Top CV score for complexity parameter of L2 Regularization: " +
        str(best_cv) + ", constant C of: " + str(top_C))

  fig = plt.figure() #initialize plot
  ax1 = fig.add_subplot() #add subplot
  ax1.plot(vary, scores)
  plt.xscale("log") #make x-axis on logarithmic scale

  #set axis labels
  ax1.set_xlabel('L2 Regularization Constant')
  ax1.set_ylabel('MNLR Model Accuracy')


  return
```

Johnathan's function above compartmentalizes all the necessary steps, so now we just have to pass it the values we want to try. Johnathan got most of these C values to try from the professor and from reading some documentation.

In [23]:
```
try_vals = [ 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000]
M1cross_val(X_train, y_train, try_vals)
```

```
CV score for complexity parameter of C=0.0001: 0.7901709401709403
CV score for complexity parameter of C=0.001: 0.91994301994302
CV score for complexity parameter of C=0.01: 0.9464387464387464
CV score for complexity parameter of C=0.1: 0.9579772079772081
CV score for complexity parameter of C=1: 0.9579772079772081
CV score for complexity parameter of C=10: 0.9579772079772081
CV score for complexity parameter of C=100: 0.9579772079772081
CV score for complexity parameter of C=1000: 0.9579772079772081
-----------------------------------------
Top CV score for complexity parameter of L2 Regularization: 0.9579772079772081, const
ant C of: 0.1
```



Explanation/Justification for Chosen L2 Regularization Constant (Johnathan):

We will choose our our level of **L2 regularization** according to the **maximum accuracy** for **minimum complexity** paradigm, meaning we do *not* want to compromise on **model accuracy**, but we do not want an overly complex model that will suffer from overfitting.

As shown in the plot above, performance with the **training data** peaks with a constant of **1.0**, but this constant might give our model too much leeway to fit different curves to the training data and thus result overfitting.

We will use **0.01** as our constant because it gives the model enough flexibility to be accurate with the training data, but will lower the risk of overfitting because we are still regularizing our model significantly. Johnathan settled on 0.01 essentially through trial-and-error on a logarithmic scale for C. As we can see this constant still has about **92.5% accuracy**, which is promising for our model.

Keeping the paradigm in mind, we will choose the **lowest number** of layers that still has **high accuracy**. The lowest number of layers is, of course, one, but the model simply does not score as well for that level of **max_depth**, as shown in the plot above.

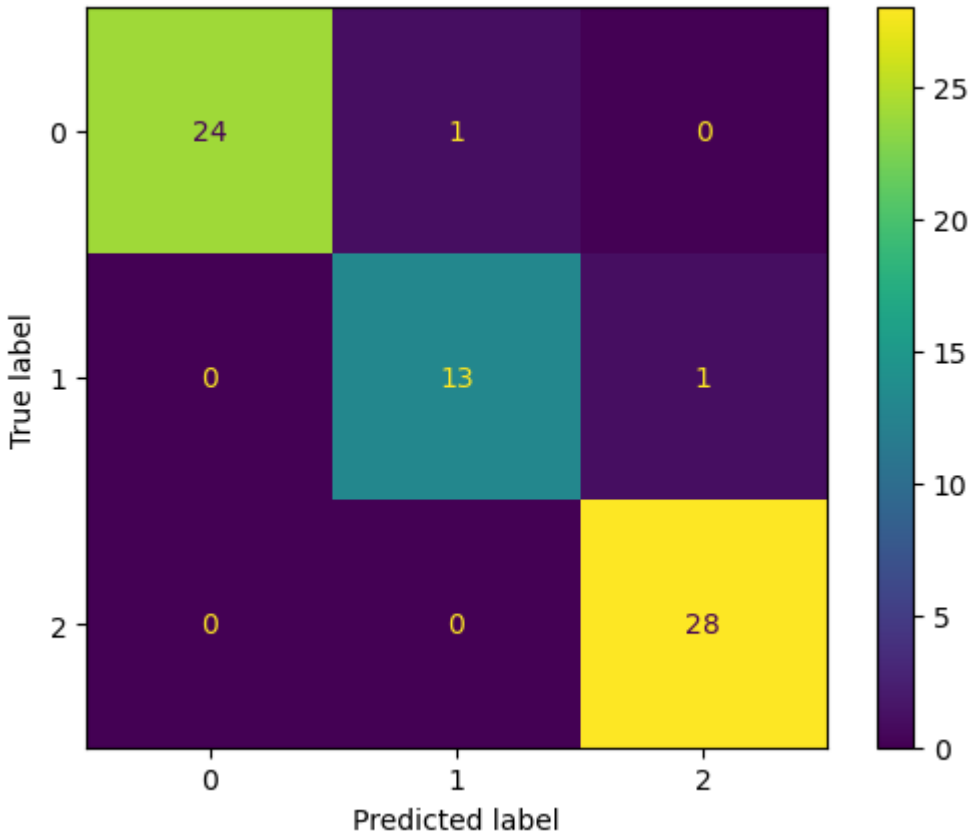
Now we have a best complexity parameter for the L2 regularization constant for our **multinomoial logistic regression model:** 0.01! On-to modeling.

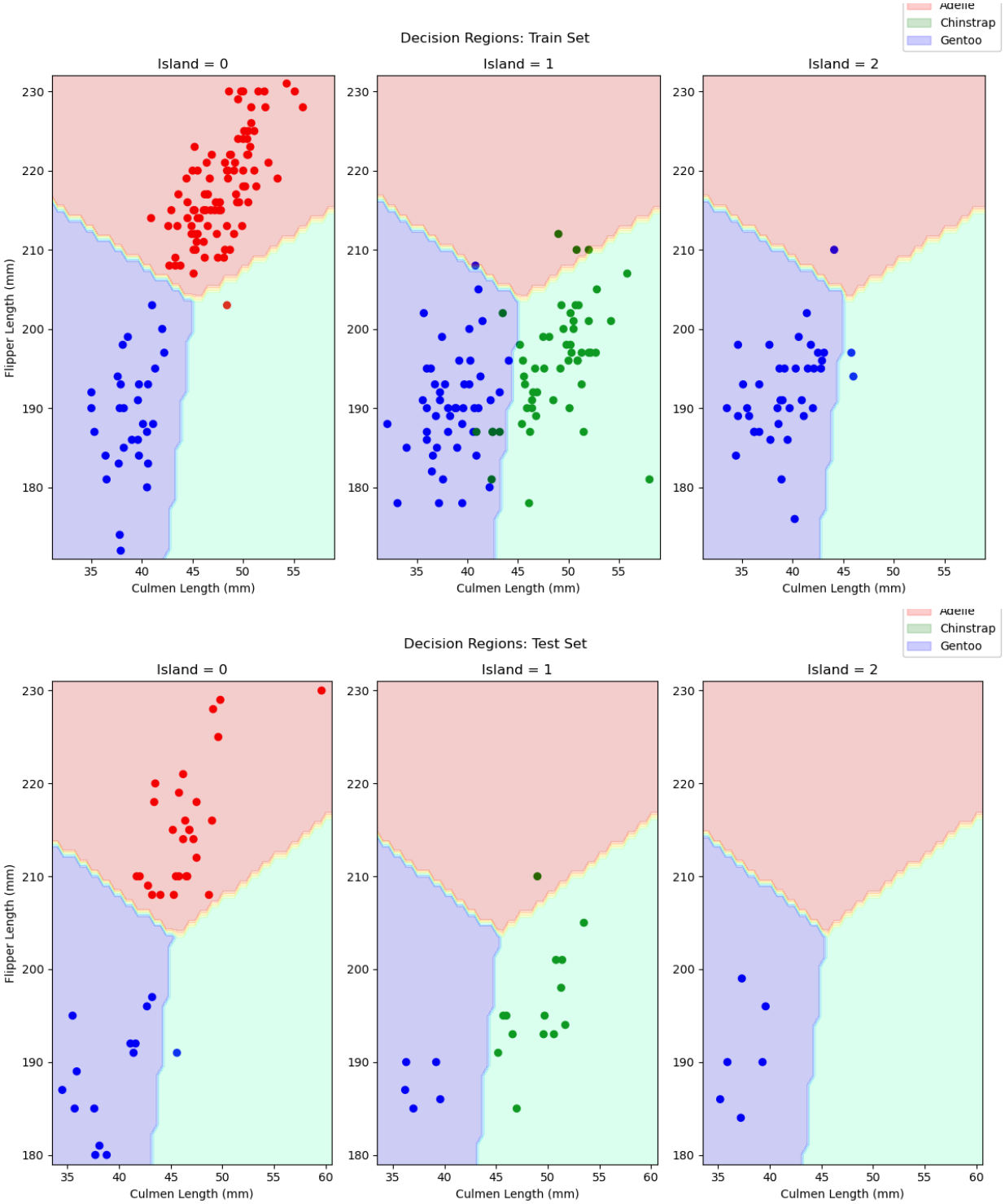
Okay, we have our best complexity parameter for regularizing our **multinomoial logistic regression** model: 1! On-to modeling.

MODEL #1: Multinomial Logistic Regression

In [24]:
```
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

#justified above in model cross-validation section
chosen_L2_RegConstant = 0.01

MNLR_model = LogisticRegression(multi_class = 'multinomial',
                                solver = 'newton-cg',
                                penalty = "l2", C=chosen_L2_RegConstant)

modelFitScore(MNLR_model)
confusionMatrix(MNLR_model)
plot_regions(MNLR_model, X_train, y_train, 0)
plot_regions(MNLR_model, X_test, y_test, 1)
```

```
test_ML_score: 0.9701492537313433
C:\Users\emily\anaconda3\lib\site-packages\sklearn\base.py:443: UserWarning: X has feature names, but LogisticRegression was fitted without feature names
  warnings.warn(
```





Multinomial Regression Model Performance Discussion: (Johnathan and Emily)

The score of the scaled multinomial logistic regression model is about 97% in terms of accuracy.

Despite our best efforts to regularize our multinomial regression model, it does suffer from a bit of overfitting. This overfitting limits the value of our model at measurements especially close to the decision boundaries, as penguins with features close to these boundaries might be mistaken for the wrong species.

The **confusion matrix** of the scaled multinomial logistic regression model indicates that the number of true positive values is 24, meaning the model accurately predicted 24 values. There is

1 false negative value and 0 false positive values. However, there are 42 true negative values, meaning the model accurately predicted these values as well.

While this shortcoming is non-ideal, it interestingly only matters a great deal depending on the island from which a penguin comes. We can notice this trend when we look at the proximity of species near the central **decision boundaries** on the island of Torgensen. Without more data from this island, the model was not able to learn the subtleties of the penguin population there well enough for accurate prediction with the test set.

While it is not totally clear why this learning gap exists, it may very well be that in the training data there was mostly one species found on the island of Torgensen (see exploratory analysis above). Still, our model is quite successful at distinguishing penguins on different islands, proving the usefulness of multinomial regression for this problem.

MODEL #2: Complexity

Cross Validation for max_depth Complexity Hyperparameter (Johnathan)

Here, we will cross-validate against different levels of **max_depth** to find the best level for our **random forest model**.

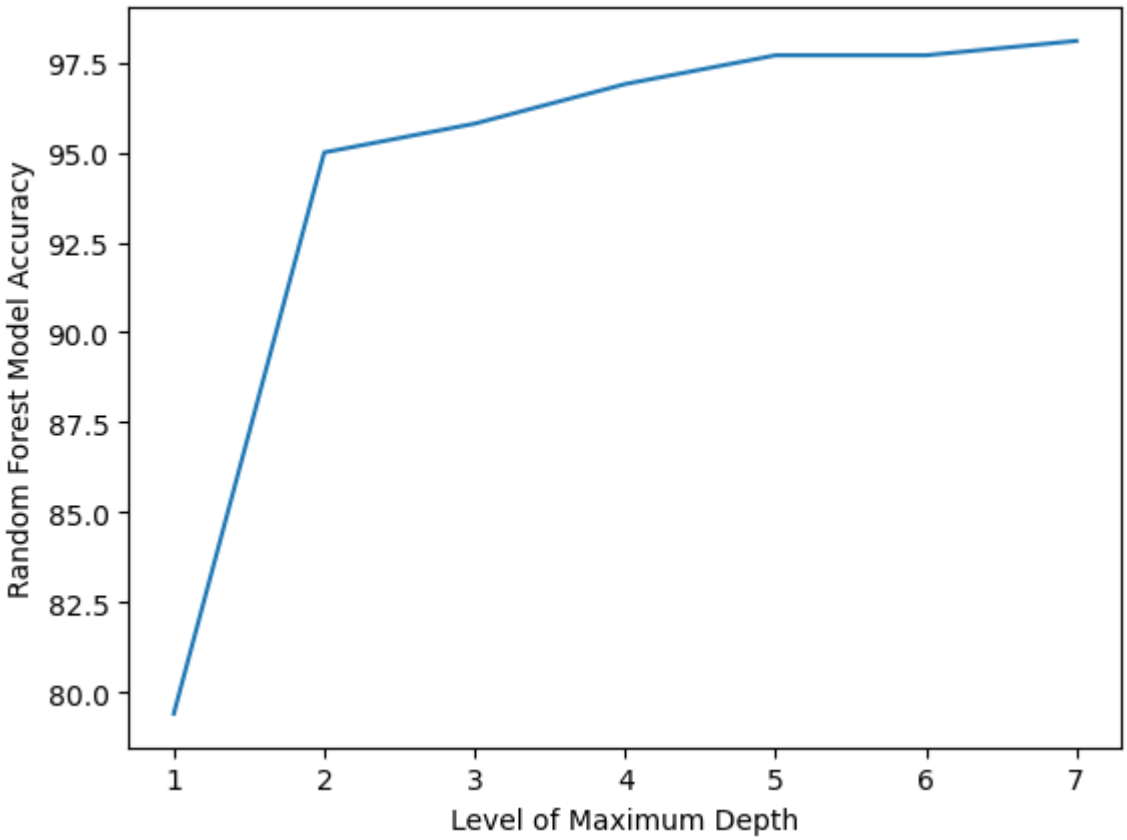
In [25]:
```
def M2cross_val(X, y, vary): #Johnathan
  '''
  This function will run cross_validation for our second model to determine the
  best level for complexity parameter C. (specialized for random forest model)
  param: X Training data.
  param: y Target data.
  param: vary List containing values for complexity parameter to test on.
  returns: The best value for the complexity parameter.
  '''

  best_cv = 0 #initialize values for cross-validation
  top_D = None
  scores = []

  for d in vary:

    #instantiate model with our a possible max_depth
    CV_score_me = RandomForestClassifier(max_depth=d, random_state=0)
    CV_score_me.fit(X, y) #fit the model

    #run our cross validation score.
    cv = cross_val_score(CV_score_me, X, y, cv = 10).mean().round(3)
    scores.append(cv*100)

    #print nice message per cv score for easy reading
    print( "CV score for complexity parameter of C=" + str(d) + ": " + str(cv))

    if cv > best_cv: #if we found a better C value
      best_cv = cv #new cross-validation score
      top_D = d #new depth value
```

```
  #line so it doens't hurt as much to read
  print( "-----------------------------------------")

  #print outs for the best CV score and C. needed for human-comparison
  print( "Top CV score for complexity parameter of max_depth: " +
        str(best_cv) + ", with max_depth of: " + str(top_D))

  #plot for hyperparameter-score relationship
  fig = plt.figure() #initialize plot
  ax1 = fig.add_subplot() #add subplot
  ax1.plot(vary, scores)

  #set axis labels
  ax1.set_xlabel('Level of Maximum Depth')
  ax1.set_ylabel('Random Forest Model Accuracy')

  return
```

Johnathan's function above compartmentalizes all the necessary steps, so now we just have to pass it the values we want to try. Johnathan got these **max_depth** values from lecture where we discussed that more than 3 levels in a decision tree/random forest model is not necessarily a good idea...

In [26]:
```
try_vals = [ 1, 2, 3, 4, 5, 6, 7]
M2cross_val(X_train, y_train, try_vals)
```

```
CV score for complexity parameter of C=1: 0.794
CV score for complexity parameter of C=2: 0.95
CV score for complexity parameter of C=3: 0.958
CV score for complexity parameter of C=4: 0.969
CV score for complexity parameter of C=5: 0.977
CV score for complexity parameter of C=6: 0.977
CV score for complexity parameter of C=7: 0.981
-----------------------------------------
Top CV score for complexity parameter of max_depth: 0.981, with max_depth of: 7
```



Explanation/Justification for Chosen max_depth: (Johnathan)

We will choose our our level of **max_depth** according to the **maximum accuracy** for **minimum complexity** paradigm, meaning we do *not* want to compromise on **model accuracy**, but we do not want an overly complex model that will suffer from overfitting.

As shown in the plot above, performance with the **training data** tops out at around **four levels**. However, such a choice for **max_depth** would like result in an over-complicated model that will not generalize to new data well.

Keeping the paradigm in mind, we will choose the **lowest number** of layers that still has **high accuracy**. The lowest number of layers is, of course, one, but the model simply does not score as well for that level of **max_depth**, as shown in the plot above.

What about **two** layers? At about 2 layers, accuracy is still high (within **2.5%** of overall highest) *and* two layers is still simple enough to avoid overfitting as best we can. Thus, use a model with a max_depth of **two**, balancing simplicity and generalization to testing.

Now we have a best complexity parameter for the maximum depth of our **random forest model:** TWO layers! On-to modeling.

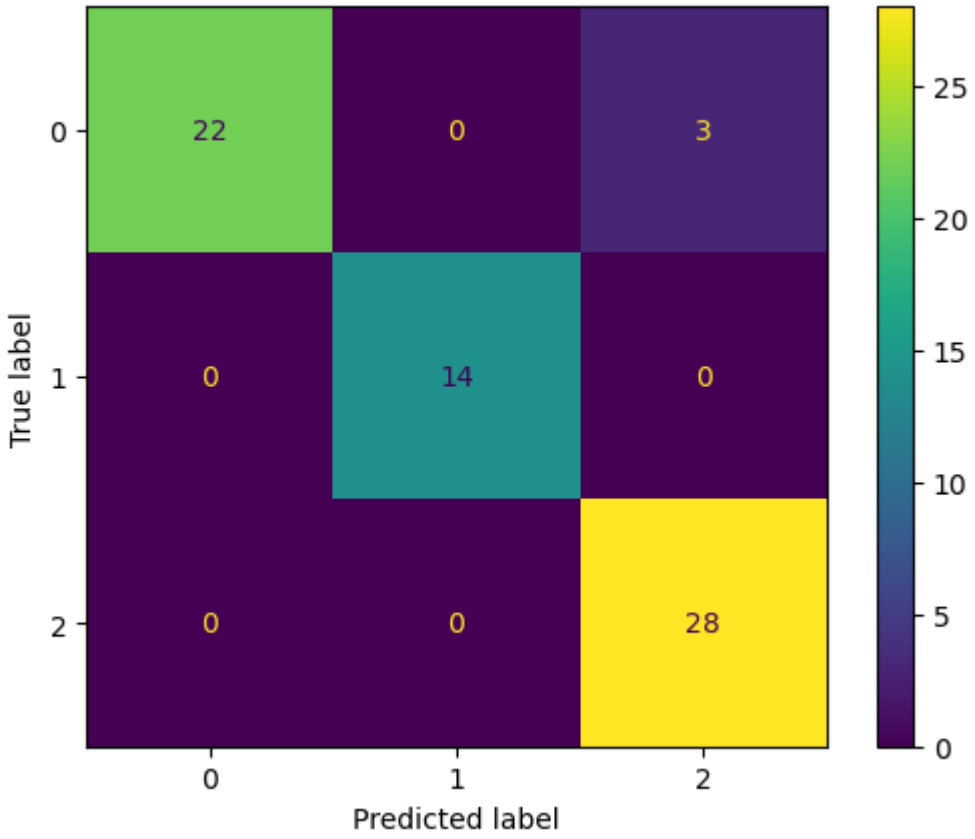
MODEL #2: Random Forest Classifier

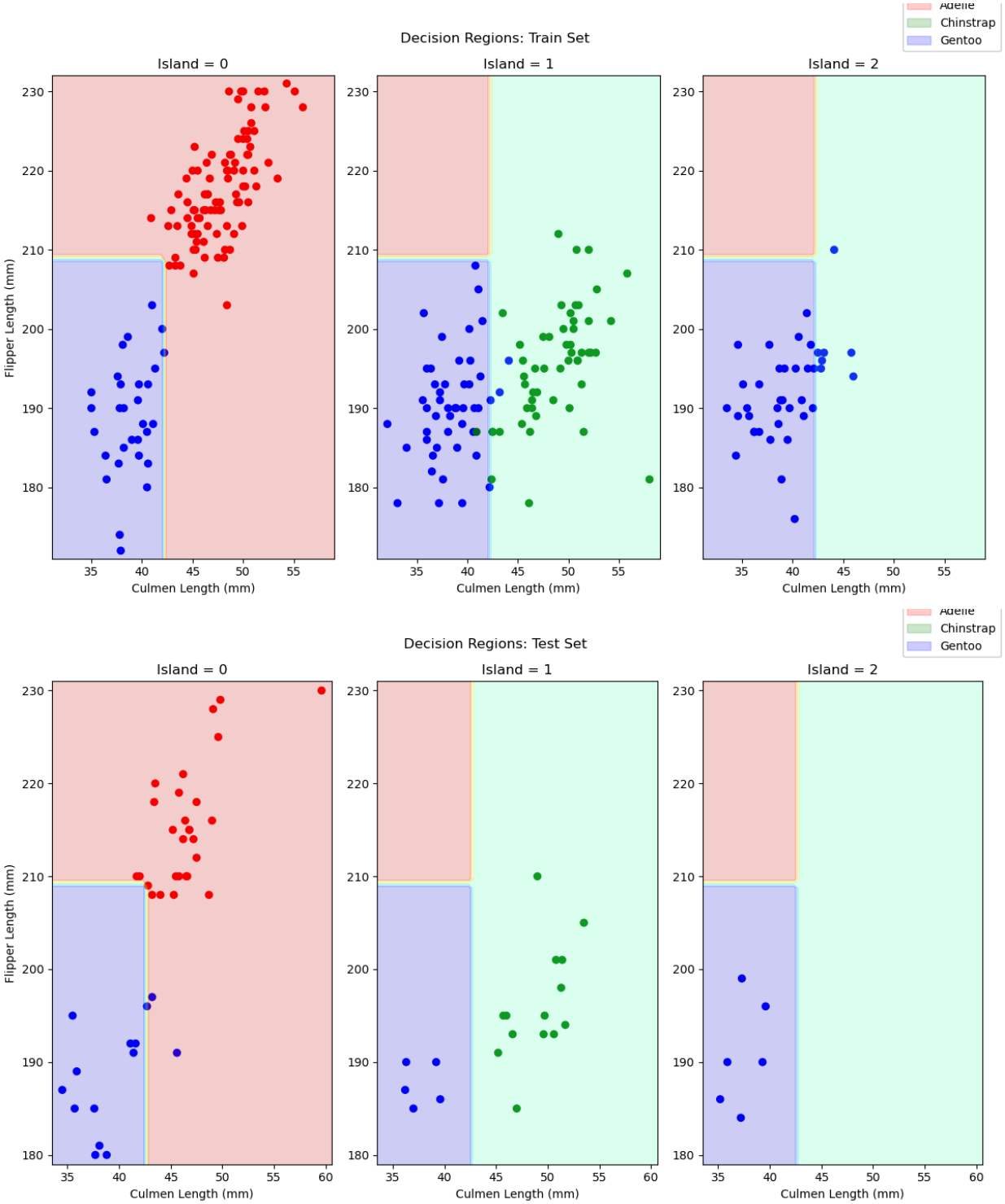
In [27]:
```
from sklearn.ensemble import RandomForestClassifier
```

```
chosen_max_depth = 2 #justified above in model cross-validation section

#initialize our model
RandForest_model = RandomForestClassifier(n_estimators = 3,
                                          max_depth = chosen_max_depth,
                                          random_state = 42)

print('SCALED: ')
modelFitScore(RandForest_model)
confusionMatrix(RandForest_model)
plot_regions(RandForest_model, X_train, y_train, 0)
plot_regions(RandForest_model, X_test, y_test, 1)
```

```
SCALED:
test_ML_score: 0.9552238805970149
```

```
C:\Users\emily\anaconda3\lib\site-packages\sklearn\base.py:443: UserWarning: X has fe
ature names, but RandomForestClassifier was fitted without feature names
  warnings.warn(
```





Random Forest Model Performance Discussion: (Johnathan and Emily)

The score of the scaled multinomial logistic regression model is about **96%** in terms of accuracy.

All things considered, our **Random Forest** model performed quite well, with only a few misfires. These misfires accompanied close measurement in **culmen length**. This shortcoming makes sense, as culmen length varies a great deal species to species, but there is a lot of overlap in the midranges of this measure.

The **confusion matrix** of the scaled multinomial logistic regression model indicates that the number of true positive values is 22, meaning the model accurately predicted 22 values. There are 3 false negative values and 0 false positive values. However, there are 42 true negative values, meaning the model accurately predicted these values as well.

Our **random forest** uses an amalgamation of many smaller models (each its own randomly generated decision tree model). These many random trees come together to make an "average" decision about all the data.

This feature allows the model to benefit from the idea that the **average of many estimates** being better than **most single estimates** (as the number of estimates increases, so does the likelihood that the *average* estimate is close to the actual **target value**).

However, the many different models, the overall **decision region border** hold more subtleties (jaggedness) than other models might. That being said, the model is still useful due to its general accuracy.

General Discussion: (Johnathan)

Sadly, the multinomial regression model has some overfitting, which limits the usefulness of our model. If we included data with more variance we might be able to reduce this, or if we simply had a better means of regularizing our model (better than L2 regularization perhaps).

As stated above, it seems prudent to include more varied data from the island of Torgensen, specifically, because the penguins there appear to have a lot of similar measurements for the included features of culmen length and flipper length. It would also make sense to simply examine other features for the penguins that live on the island of Torgensen.

As the random forest predictions derive from the collective learning of myriad smaller decision trees, this model could learn the nuances of the many subsets of the data to make the best model possible. Though, as previously stated, this can result in too much subtlety, and cause some overfitting. With fewer features, it might be possible to reduce this problem.

The random forest model has fewer misfires, it seems like it is the better model for the three selected features, though perhaps these three are not the ideal set of features to select in the first place (using a logistic regression model is not without its faults, such as difficulty balancing overfitting).

As far as different features to include, it would be prudent to include different features that have less clustering around mid-range values for each species, as culmen length did unfortunately. Such useful features might include body mass, which for this dataset, appears to have more variance than the feature used for prediction here.

In [ ]: