Group Contributions Statement

Group 14: Dara Tan and Duc Hoang

We decided on the strategy for data cleaning and preparation, exploratory analysis and modeling together. Dara led the exploratory analysis and the code for the models. Duc led the write-up for the models and the discussion section. We checked and edited each other's work.

Introduction

Cataloging the various species of penguins is an important task in Antarctic ecology. In this project, we use the Palmer Penguins dataset, collected by Dr. Kristen Gorman and the Palmer Station, Antarctica LTER, a member of the Long Term Ecological Research Network, to make predictions about the species of a penguin using only two quantitative and one qualitative measurement.

Import and Cleaning

In [1]:
```
import itertools
from matplotlib.colors import ListedColormap
from matplotlib import lines as mlines
from matplotlib import patches as mpatches
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
from sklearn import preprocessing
from sklearn.ensemble import RandomForestClassifier
from sklearn.feature_selection import mutual_info_classif
from sklearn.metrics import confusion_matrix
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
```

Before conducting our analysis, we import the Palmer Penguins dataset and split it to obtain a training set and a test set, reserving 30% of observations as test data and using a `random_state` of 0 to ensure that the results of this project are reproducible. By performing our exploratory analysis and feature selection using only information from our training data, we ensure that our test data remains 'unseen' and prevent information from our cleaning process from unintentionally 'polluting' our test data. In splitting our data, we also make sure to stratify on the label variable, Species. This ensures that our training set has a sufficient number of observations of each species for our models to learn from. After obtaining our training set, we perform some preliminary data cleaning to (1) remove columns which have the same value for every observation and (2) reformat columns. These steps are implemented in the cells below.

In [2]:

```
penguins = pd.read_csv('palmer_penguins.csv')
train, test = train_test_split(penguins, test_size = 0.3, random_state = 0,
                               stratify = penguins['Species'])
```

In [3]:

```
def clean_data(raw_df):
    '''
    Cleans a subset of the penguins data

    Input ----
    raw_df: DataFrame, the data to be cleaned

    Output ----
    clean_df: DataFrame, the cleaned data
    '''
    clean_df = raw_df.copy()

    # species: keep only the first word
    clean_df['Species'] = clean_df['Species'].str.split().str.get(0)

    # region and stage: drop columns since every value is the same
    clean_df = clean_df.drop(columns = ['Region', 'Stage'])

    # clutch completion: recode as logical
    clean_df['Clutch Completion'] = clean_df['Clutch Completion'] == 'Yes'

    # sex: '.' is equivalent to NA and title format
    clean_df = clean_df.replace({'Sex': '.'}, np.nan)
    clean_df['Sex'] = clean_df['Sex'].str.title()

    return clean_df
```

In [4]:

```
train = clean_data(train)
```

Exploratory Analysis

In [5]:

```
species = ['Adelie', 'Chinstrap', 'Gentoo']
colors = dict(zip(species, ['r', 'g', 'b']))
quantitative = ['Culmen Length (mm)', 'Culmen Depth (mm)',
                'Flipper Length (mm)', 'Body Mass (g)',
                'Delta 15 N (o/oo)', 'Delta 13 C (o/oo)']
```

After splitting our data and performing preliminary cleaning of our training set, we proceed with our exploratory analysis. In this section, we create tables and figures to visualize our training data. In particular, we use tables to get a better understanding of two qualitative variables, Sex and Island. We use figures to better understand the six quantitative variables in our data, namely Culmen Length, Culmen Depth, Flipper Length, Body Mass, Delta 15N and Delta 13C. In doing so, we look out for one qualitative feature and two quantitative features that are strong predictors of the species of a penguin, as these are features that we may want to use in our models later on.

Table 1: Percentage of Species by Sex

In [6]:
```
species_sex = train.groupby(['Species', 'Sex']).size()
species_sex = species_sex.unstack('Sex')
species_sex = species_sex.apply(lambda r: (r / r.sum()) * 100, axis = 1)
display(species_sex.round(1))
```

Sex	Female	Male
Species		
Adelie	56.9	43.1
Chinstrap	57.4	42.6
Gentoo	48.2	51.8

In Table 1, we compare the percentage of female and male penguins by species. We observe that Adelie and Chinstrap penguins are more likely to be female than male, with about 56.9% of Adelie penguins and 57.4% of Chinstrap penguins in our training set being female. In contrast, Gentoo penguins are slightly more likely to be male than female, with about 51.8% of Gentoo penguins in our training set being male.

Table 2: Count of Species by Island

In [7]:
```
species_island = train.groupby(['Species', 'Island']).size()
species_island = species_island.unstack('Island', fill_value = 0)
display(species_island)
```

Island	Biscoe	Dream	Torgersen
Species			
Adelie	32	38	36
Chinstrap	0	47	0
Gentoo	87	0	0

In Table 2, we look at how the penguins in our training set are distributed over the three islands — Biscoe, Dream and Torgersen. We see that only Adelie and Gentoo penguins are found on Biscoe island, in a ratio of about 1 to 3. On Dream island, we find only Adelie and Chinstrap penguins, in a relatively balanced ratio. Finally, we observe that only Adelie penguins are found on Torgersen island.

Comparing Tables 1 and 2, we find that Table 2 will likely be more helpful in determining the species of a given penguin. As we observed, once we know the island that a penguin was found on, we can narrow the species of that penguin down to two of three options in the case of Biscoe and Dream, or even definitively say that the penguin is an Adelie penguin in the case of Torgersen. This hints to us that Island is likely a good choice for the qualitative variable that we will include in our model later on.

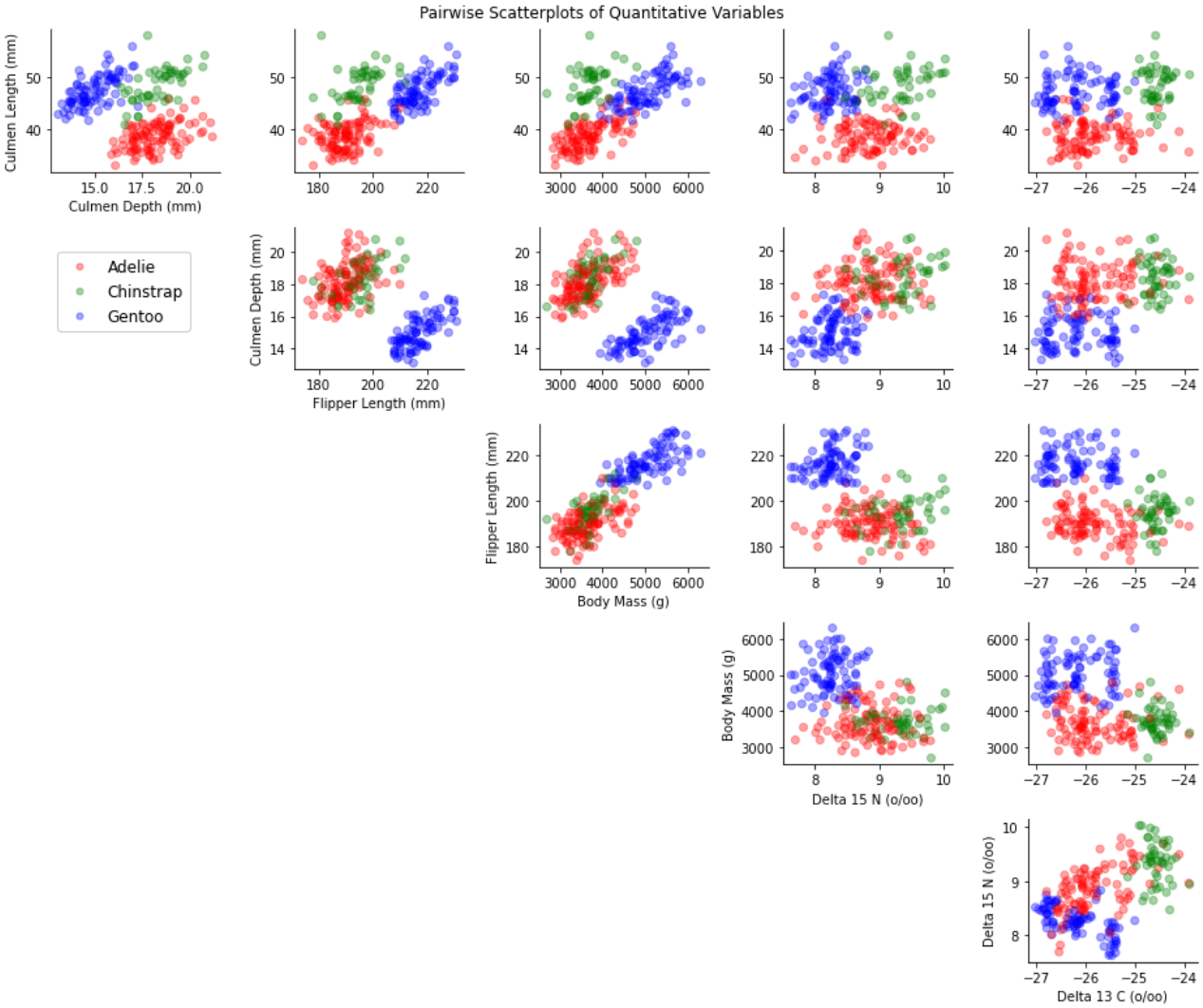
Figure 1: Pairwise Scatterplots

In [8]:

```
def pairwise_scatterplots(variables, ax):
    '''
    Plots a pairwise scatterplot of two quantitative variable

    Inputs ----
    variables: tuple, formatted as (y-axis variable name,
               x-axis variable name)
    ax: tuple, row and column where the plot is to be located
    '''
    color_vector = train['Species'].apply(lambda x: colors[x])

    axes[ax].scatter(train[variables[1]], train[variables[0]],
                     c = color_vector, alpha = 0.35)
    axes[ax].spines['top'].set_visible(False)
    axes[ax].spines['right'].set_visible(False)

    # axis labels only for plots on the diagonals
    if ax[0] == ax[1]:
        axes[ax].set(xlabel = variables[1], ylabel = variables[0])
```

In [9]:

```
variables = list(itertools.combinations(quantitative, 2))
axs = [(row, column) for row in range(5) for column in range(row, 5)]

# plot pairwise scatterplots
fig, axes = plt.subplots(5, 5, figsize = (13, 11))
for i in range(15):
    pairwise_scatterplots(variables[i], axs[i])

# hide unused subplots
empty = list(itertools.product(list(range(5)), list(range(5))))
empty = [e for e in empty if e not in axs]
for j in empty:
    axes[j].spines['top'].set_visible(False)
    axes[j].spines['bottom'].set_visible(False)
    axes[j].spines['left'].set_visible(False)
    axes[j].spines['right'].set_visible(False)
    axes[j].tick_params(axis = 'both', bottom = False, left = False,
                        labelbottom = False, labelleft = False)

# add title, legend and other formatting
lkwargs = dict(marker = '.', linestyle = 'None', markersize = 10, alpha = 0.35)
legend_adelie = mlines.Line2D([], [], color = colors['Adelie'],
                              label = 'Adelie', **lkwargs)
legend_chinstrap = mlines.Line2D([], [], color = colors['Chinstrap'],
                                 label = 'Chinstrap', **lkwargs)
legend_gentoo = mlines.Line2D([], [], color = colors['Gentoo'],
                              label = 'Gentoo', **lkwargs)
fig.legend(handles = [legend_adelie, legend_chinstrap, legend_gentoo],
           loc = (0.05, 0.67), fontsize = 'large', framealpha = 1)
plt.suptitle('Pairwise Scatterplots of Quantitative Variables')
plt.tight_layout()
```



In Figure 1, we have a pairwise scatterplot, where each of the six quantitative variables in our dataset is plotted against the other five quantitative variables. x-axis labels are indicated on the bottom-most plot in each column, while y-axis labels are indicated on the left-most plot in each row. Each point corresponds to an observation in our dataset and is colored according to the species of the penguin observed.

Since we will be using two quantitative variables to predict the species of a penguin in the modeling section of this project, when reading these pairwise scatterplots we pay special attention to the pairs of variables that will best allow us to distinguish penguins of different species from one another. Visually, this corresponds to scatterplots in which points of one color have minimal overlap with points of the other two colors.

Looking at the plot for Culmen Depth against Body Mass, for instance, we see that there is a distinct cluster of blue points that are clearly separated from the rest of the points. This suggests that this pair of variables will enable us to clearly distinguish Gentoo penguins from Adelie and Chinstrap penguins. However, the red and green points in this plot appear to have significant overlap, suggesting that this pair of variables is not effective when it comes to distinguishing between Adelie and Chinstrap penguins. This overlap is what we wish to avoid when selecting the quantitative variables for our model later on.

Overall, we observe three pairs of quantitative variables that offer minimal overlap across all three colors of points, namely:

1. Culmen Depth and Delta 13C
2. Flipper Length and Delta 13C
3. Body Mass and Delta 13C

We will thus focus on these pairs of variables when selecting features.

Figure 2: Grouped Boxplots

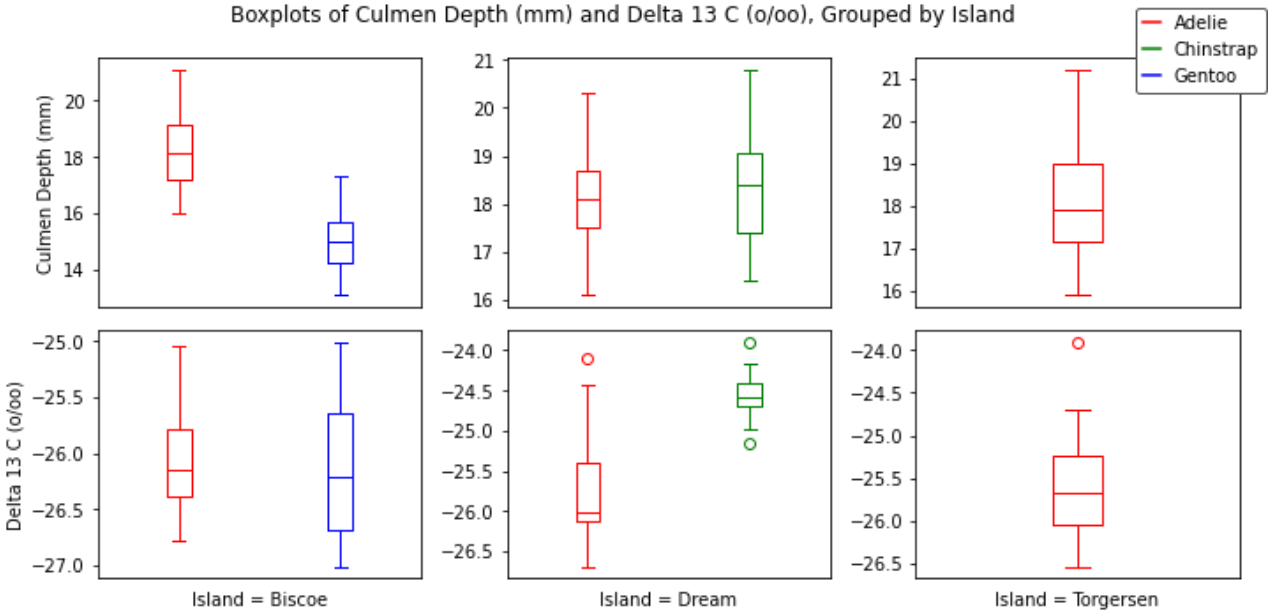
In [10]:

```
def grouped_boxplots(cat, nums):
    '''
    Plots side-by-side boxplots on rows of subplots; one numerical variable
    is used on each row of subplots and each column of subplots
    corresponds to a unique value of the categorical variable supplied

    Inputs ----
    cat: string, name of the categorical variable
    nums: tuple of strings of at least length 2,
          names of the numerical variables
    '''
    fig, ax = plt.subplots(len(nums), len(set(train[cat])), figsize = (10, 5))
    title = 'Boxplots of '
    for row in range(len(nums)):
        column = 0
        for name, group in train.groupby(cat):
            bp = group.boxplot(nums[row], by = 'Species', ax = ax[row, column],
                               grid = False, return_type = 'both')
            group_species = pd.unique(group['Species'])

            # boxplot aesthetics: whiskers, caps, boxes and medians
            aesthetics = {'whiskers': 2, 'caps': 2, 'boxes': 1, 'medians': 1}
            for key in aesthetics:
                A = bp[nums[row]][1][key]
                for a in range(len(A)):
                    A[a].set_color(colors[np.repeat(group_species,
                                                    aesthetics[key])[a]])

            # boxplot aesthetics: fliers
            F = bp[nums[row]][1]['fliers']
            for f in range(len(F)):
                F[f].set_markeredgecolor(colors[group_species[f]])

            # remove titles; keep x-axis labels only for the bottom row
            ax[row, column].set(title = '', xlabel = '')
            ax[row, column].tick_params(axis = 'both', bottom = False,
                                        labelbottom = False)
            if row == len(nums) - 1:
                ax[row, column].set_xlabel(cat + ' = ' + name, labelpad = 7.5)

            column += 1

        # add y-axis labels only to the left
        ax[row, 0].set(ylabel = nums[row])
```

```
        # add numerical variable name to the figure title
        title += nums[row]
        # 'and' between the second last and last name
        if row == len(nums) - 2:
            title += ' and '
        # ',' between all other names
        else:
            title += ', '
    title += 'Grouped by ' + cat
    fig.suptitle(title)

    # add legend to explain the species colors
    legend_adelie = mlines.Line2D([], [], color = colors['Adelie'],
                                  label = 'Adelie', linestyle = '-')
    legend_chinstrap = mlines.Line2D([], [], color = colors['Chinstrap'],
                                     label = 'Chinstrap', linestyle = '-')
    legend_gentoo = mlines.Line2D([], [], color = colors['Gentoo'],
                                  label = 'Gentoo', linestyle = '-')
    fig.legend(handles = [legend_adelie, legend_chinstrap, legend_gentoo],
               loc = (0.8953, 0.8395), fontsize = 'medium',
               framealpha = 1, edgecolor = '#555555', handlelength = 1)

    plt.tight_layout()
```

In [11]:

```
grouped_boxplots('Island', ('Culmen Depth (mm)', 'Delta 13 C (o/oo)'))
```



To create Figure 2, we wrote a `grouped_boxplots` function, which takes the name of a qualitative variable and the names of at least two quantitative variables as inputs, then creates an array of plots, in which each column corresponds to a category of the qualitative variable and each row corresponds to a quantitative variable of interest. In each individual plot of the array, one boxplot is drawn per species and colored accordingly.

Based on our observations from Tables 1 and 2, we decided to use Island as the qualitative variable for our grouped boxplots. Additionally, we chose one of the three pairs of variables that we had shortlisted using our findings from Figure 1, namely Culmen Depth and Delta 13C, as our

quantitative variables for this plot. Through Figure 2, we hoped to confirm our earlier suspicions that this combination of three variables will enable us to distinguish between the three species of penguins with relatively high accuracy.

As observed previously from Table 2, we know that penguins found on Biscoe must be Adelies or Gentoos, penguins found on Dream must be Adelies or Chinstraps and penguins found on Torgersen are definitely Adelies. Then, looking at Figure 2, we see that:

- Our choice of Culmen Depth as one of the quantitative variables offers us a way to distinguish between Adelies and Gentoos on Biscoe: penguins with deeper culmens, especially those with Culmen Depth values of 17 mm or more, are likely to be Adelies, while penguins with shallower culmens, especially those with Culmen Depth values of less than 16 mm, are likely to be Gentoos.
- By choosing Delta 13C, we can distinguish between Adelies and Chinstraps on Dream: penguins with lower Delta 13C values, particularly of -25.5 o/oo or lower, are likely to be Adelies, while penguins with higher Delta 13C values, particularly those higher than -25 o/oo, are likely to be Chinstraps.

In effect, this combination of three variables — Island, Culmen Depth and Delta 13C, enables us to (1) narrow the species of a new penguin down to one or two options, depending on where it was found and (2) make an educated guess between the two options if the penguin was found on Biscoe or Dream, based on the depth of its culmen or the amount of carbon isotopes in its blood respectively. This serves to 'confirm' our earlier suspicions that using Island as our qualitative variable and one of the three pairs of quantitative variables selected using Figure 1 will allow us to build a fairly accurate classification model.

Feature Selection

Building on the insights gleaned through our exploratory analysis, we select Island as our qualitative variable and Delta 13C as one of our quantitative variables. However, as described earlier, based on Figure 1, there seem to be three good options for the second quantitative variable: Culmen Depth, Flipper Length and Body Mass. In this section, we choose one of these three variables to include in our models.

We do so by leveraging the `sklearn` implementation of mutual information as a feature selector. This is a model-neutral method which, as described in the documentation, allows us to estimate the mutual information between each of our three shortlisted features and our discrete target variable, Species. In other words, this feature selection method helps us identify which of our three shortlisted features the species of a penguin is most dependent on and thus, which feature we should include as our second quantitative variable.

In [12]:
```
# training set, focused on the shortlisted features and the labels
# NAs must be removed before evaluating the features
# data must be split into x and y
shortlist = ['Culmen Depth (mm)', 'Flipper Length (mm)', 'Body Mass (g)']
fs = train[shortlist + ['Species']].dropna()
```

```
fs_x = fs[shortlist]; fs_y = fs['Species']

# calculate mutual information for each feature and display results
mi = mutual_info_classif(fs_x, fs_y, random_state = 0)
mi_results = pd.DataFrame({'Feature': shortlist, 'Mutual Information': mi})
mi_results = mi_results.sort_values('Mutual Information', ascending = False)
display(mi_results.style.hide_index())
```

Feature	Mutual Information
Flipper Length (mm)	0.635310
Culmen Depth (mm)	0.554440
Body Mass (g)	0.492610

As shown in the results above, the Flipper Length variable produced the highest mutual information value of about 0.635. This indicates that of the shortlisted features, Species is most dependent on Flipper Length, so we choose it as our second quantitative variable.

Modeling

Now that we have selected our three features — Island, Delta 13C and Flipper Length, we can move on to the modeling section of this project. To begin, we will prepare our data, using the following steps:

1. For simplicity, we drop the features that we are not using in our models.
2. We remove any row in which there is a missing value. By removing missing values at this stage when unused features have already been dropped, we avoid losing observations that have missing values only in features we are not interested in, thereby minimizing the loss of useful data.
3. We standardize both quantitative variables using the `StandardScaler` from `sklearn`. This step is particularly important for models such as k-nearest neighbors, which make decisions based on distances.
4. We encode our qualitative variable using the `LabelEncoder` from `sklearn`.
5. We split our data into features and labels.

In [13]:

```
def prepare_data(raw_df, quantitative, qualitative):
    '''
    Prepares a subset of the penguins data for use in modeling

    Inputs ----
    raw_df: DataFrame, the data to be prepared
    quantitative: list of length 2, the selected quantitative features
    qualitative: string, the selected qualitative feature

    Outputs ----
    clean_x: DataFrame, the features of the prepared data
    clean_y: Series, the labels of the prepared data
    scaler: StandardScaler object, the scaler used to scale the
            quantitative features
    '''
```

```
    clean_df = raw_df.copy()

    # keep only selected features
    variables = quantitative + [qualitative]
    variables.append('Species')
    clean_df = clean_df[variables]

    # missing: drop NA rows
    clean_df = clean_df.dropna()

    # quantitative: standardize
    scaler = preprocessing.StandardScaler()
    clean_df[quantitative] = scaler.fit_transform(clean_df[quantitative])

    # qualitative: label encoding
    le = preprocessing.LabelEncoder()
    clean_df[qualitative] = le.fit_transform(clean_df[qualitative])

    # split into x and y
    clean_x = clean_df[quantitative + [qualitative]]
    clean_y = clean_df['Species']

    return clean_x, clean_y, scaler
```

In [14]:

```
quant = ['Delta 13 C (o/oo)', 'Flipper Length (mm)']; qual = 'Island'
train_x, train_y, train_s = prepare_data(train, quant, qual)
test_x, test_y, test_s = prepare_data(clean_data(test), quant, qual)
```

Next, we define two key functions that we will use in this section. First, we have `decision_regions`, which we will use to visualize how each model that we fit 'sees' the data, i.e. how each model divides the sample space up and makes predictions. Since we use one qualitative and two quantitative variables in each model, `decision_regions` creates multiple subplots, with one subplot created for each unique value of the qualitative variable and the quantitative variables plotted on the horizontal and vertical axes of each subplot. Additionally, both the points in our data and the decision regions of each model are color-coded by species, using the same color scheme as in our exploratory analysis: red for Adelie, green for Chinstrap and blue for Gentoo.

In [15]:

```
def decision_regions(model, features, labels, scaler, limits = (),
                     description = '', categories = {}, legend = False):
    '''
    Plots the decision regions for a given model with one subplot for each
    category of the qualitative variable used in the model and the
    quantitative variables plotted on the horizontal and vertical axes

    Inputs ----
    model: model object that has been fitted with training data
    features: DataFrame, features; with the quantitative variables as columns
              0 and 1 and the qualitative variable as column 2
    labels: Series, corresponding labels
    scaler: StandardScaler object, the scaler to be used to undo the scaling
            of the quantitative variables
    limits: tuple of length 4, horizontal and vertical limits of each subplot,
            formatted as (horizontal minimum, horizontal maximum, vertical
```

```
        minimum, vertical maximum)
description: string, description of the model to be used as suptitle
categories: dictionary, keys are integer-encoded categories of the
            qualitative variable and values are strings indicating the
            pre-encoding category names
legend: logical, whether to add a legend to the plot
'''
# create a grid for the quantitative variables
# use feature minimums and maximums if no limits are supplied
if len(limits) == 0:
    horizontal = features.iloc[:, 0]; vertical = features.iloc[:, 1]
    h = np.linspace(horizontal.min(), horizontal.max(), 151)
    v = np.linspace(vertical.min(), vertical.max(), 151)
else:
    h = np.linspace(limits[0], limits[1], 151)
    v = np.linspace(limits[2], limits[3], 151)
H, V = np.meshgrid(h, v)

# iterate through the categories of the qualitative variable
C = pd.unique(features.iloc[:, 2])
fig, ax = plt.subplots(1, len(C), figsize = (8.5, 3), sharey = True)
for c in C:

    # obtain model predictions
    X = np.c_[H.ravel(), V.ravel(), np.repeat(c, H.ravel().shape)]
    predictions = model.predict(X)

    # map predictions from strings to integers
    SPECIES = {'Adelie': 0, 'Chinstrap': 1, 'Gentoo': 2}
    P = np.array([SPECIES[p] for p in predictions]).reshape(H.shape)

    # undo the scaling done in the data preparation stage and
    # plot the data, showing only within the category
    within = features.iloc[:, 2] == c
    f_unscaled = scaler.inverse_transform(features[within].iloc[:, 0:2])
    f_unscaled = pd.DataFrame(f_unscaled)
    ax[c].scatter(f_unscaled.iloc[:, 0], f_unscaled.iloc[:, 1],
                  color = [colors[l] for l in labels[within]])

    # define colormap based on predictions and plot the predictions,
    # taking care to undo the scaling before plotting
    palette = [colors[s] for s in species if s in predictions]
    colormap = ListedColormap(palette)
    X_unscaled = scaler.inverse_transform(pd.DataFrame(X).iloc[:, 0:2])
    H_unscaled = np.array([r[0] for r in X_unscaled]).reshape(H.shape)
    V_unscaled = np.array([r[1] for r in X_unscaled]).reshape(V.shape)
    ax[c].contourf(H_unscaled, V_unscaled, P,
                   cmap = colormap, alpha = 0.15)

    # subplot labels: use pre-encoding category names in title if provided
    # via categories, otherwise use integer-encoded categories
    if len(categories) == len(C):
        ax[c].set(title = features.columns[2] + ' = ' + categories[c],
                  xlabel = features.columns[0])
    else:
        ax[c].set(title = features.columns[2] + ' = ' + str(c),
                  xlabel = features.columns[0])

# ADDITIONAL FORMATTING ----
# add suptitle only if description is supplied
```

```
    if len(description) > 0:
        plt.suptitle('Decision Regions of ' + description)

    # y-axis labels for the left-most plot only
    ax[0].set(ylabel = features.columns[1])

    # add legend if requested
    if legend:
        legend_adelie = mpatches.Patch(color = colors['Adelie'],
                                       label = 'Adelie', alpha = 0.35)
        legend_chinstrap = mpatches.Patch(color = colors['Chinstrap'],
                                          label = 'Chinstrap', alpha = 0.35)
        legend_gentoo = mpatches.Patch(color = colors['Gentoo'],
                                       label = 'Gentoo', alpha = 0.35)
        fig.legend(handles = [legend_adelie, legend_chinstrap, legend_gentoo],
                   loc = (0.855, 0.55), fontsize = 'medium',
                   framealpha = 1, edgecolor = '#555555')

    plt.tight_layout(pad = 0.35, rect = (0, 0, 0.85, 0.95))
```

The other function that we define is `run_model`, which does the following for each type of model that we run:

1. Given a hyperparameter grid, perform 10-fold cross-validation to select the model hyperparameters. This prevents the model from overfitting, a phenomenon in which the model picks up on the idiosyncrasies of the training set, as opposed to learning the actual trends from the data. 10-fold cross-validation, as applied to the selection of model hyperparameters, involves:
    - Splitting our training set into 10 parts.
    - For each set of hyperparameters, as given by permutations of the hyperparameter grid:
        - For each of the 10 parts that the training set was split into:
            - Setting that part of the training set aside, for use as a validation set.
            - Fitting a model with that set of hyperparameters, using the remaining 9 parts of the training set.
            - Scoring the fitted model using the part of the training set that was set aside initially.
        - Averaging the scores from the 10 parts to get the cross-validation score for the set of hyperparameters.
    - The best set of hyperparameters is the set with the highest cross-validation score.
2. Fit a model with the hyperparameters identified in #1, then use that model to predict labels for the test set.
3. Evaluate the performance of the model from #2 on the test set.
4. Visualize the decision regions of the model using `decision_regions` as defined above. For completeness, we have included two sets of decision region plots, in which the points represent actual observations in the training set and test set respectively. However, in our discussion of model performance, we focus on the plots using observations from the test set, as they allow us to better explain the insights from our confusion matrix.

By defining this function, we can implement several important steps of the Machine Learning pipeline — identifying a good model class, fitting the model, evaluating the model and visualizing the results — easily, with just a few lines of code.

In [16]:

```
def run_model(e, hp, names, description):
    '''
    Implements these steps for a given estimator type and hyperparameter grid:
    (1) Performs 10-fold cross-validation to find the best hyperparameters
        for the estimator and prints the results
    (2) Fits a model with the best hyperparameters, then uses the model to
        generate predictions using the test data, test_x
    (3) Evaluates the performance of the fitted model by printing (a) the score
        of the model on test_x and (b) the confusion matrix
    (4) Visualizes the decision regions of the model using decision_regions

    Inputs ----
    e: estimator object, type of estimator to be used
    hp: dictionary, keys are names of parameters used by the estimator and
        values are values to try for the corresponding model parameter
    names: dictionary, keys are names of parameters as in hp and values are
           names of parameters as they should be printed in step (1)
    description: string, model description to be printed at the top of the
                 model results, as well as passed to decision_regions
    '''
    # find the best hyperparameters and generate predictions
    gscv = GridSearchCV(estimator = e, param_grid = hp, cv = 10)
    gscv.fit(train_x, train_y)
    p = gscv.predict(test_x)

    # print the best hyperparameters found
    dashes = '\n' + ('-' * 75) + '\n'
    best = gscv.best_params_
    print('\n' + description.upper().center(75, ' ') + dashes +
          'Best Hyperparameters'.center(75, ' ') + dashes)
    for parameter in best:
        if best[parameter] == 'rbf':
            # for rbf, print in all capital letters
            print(names[parameter] + ': RBF')
        else:
            # print other string hyperparameter values with title formatting
            print(names[parameter] + ': ' + str(best[parameter]).title())

    # prints cv results: best hyperparamteres
    print(dashes + 'Cross-validation Scores'.center(75, ' ') + dashes)
    results = pd.DataFrame(gscv.cv_results_)
    print('Best Hyperparameters: ' +
          str(round(results['mean_test_score'].max(), 5)) + '\n')

    # prints cv results: overall mean and sd
    print('Overall Statistics' +
          '\n\tMean = ' + str(round(results['mean_test_score'].mean(), 5)) +
          '\n\tSD = ' + str(round(results['mean_test_score'].std(), 5)) + '\n')

    # prints cv results: mean and sd when varying one hyperparameter
    print('Varying One Hyperparameter with the Others Fixed')
    for parameter in best:
        filtered = results.copy()
        # fixes other parameters to their best values
```

```
        for key in best:
            if key != parameter:
                filtered = filtered[filtered['param_' + key] == best[key]]
        # gets statistics with other hyperparameters fixed
        scores = pd.DataFrame(filtered)['mean_test_score']
        print('\t' + names[parameter] +
              '\n\t\tMean = ' + str(round(scores.mean(), 5)),
              '\n\t\tSD = ' + str(round(scores.std(), 5)))

    # evaluate performance on test data: score
    print(dashes + 'Performance on Test Data'.center(75, ' ') +
          dashes + '\nScore: ' + str(gscv.score(test_x, test_y)))

    # evaluate performance on test data: confusion matrix
    cm = pd.DataFrame(confusion_matrix(y_true = test_y, y_pred = p),
                      index = ['Actual ' + s for s in species],
                      columns = ['Predicted ' + s for s in species])
    print('Confusion Matrix:', end = '')
    display(cm)

    # visualize decision regions using decision_regions
    print(dashes + 'Decision Regions'.center(75, ' ') + dashes, end = '')
    islands = {0: 'Biscoe', 1: 'Dream', 2: 'Torgersen'}
    limits = (-1.75, 2.45, -2.15, 2.25)
    decision_regions(gscv, train_x, train_y, train_s, limits,
                     description + ': Training Set', islands, True)
    decision_regions(gscv, test_x, test_y, test_s, limits,
                     description + ': Test Set', islands)
```

Finally, we select a different algorithm to use for each of our three models. For this project, we have chosen the following three algorithms:

1. k-nearest Neighbors: As we saw in our exploratory analysis, points representing penguins of the same species tend to be clustered together. Hence, this algorithm, which classifies a new point based on the points that it is surrounded by, will likely be helpful.
2. Random Forest: As we saw in our exploratory analysis, there is a logical flow of steps by which we can divide penguins up into different species. Specifically, we can first consider where the penguin was found, then for islands with two species, we can use either the depth of the culmen or the amount of carbon isotopes in the blood to guess which species a penguin belongs to. This flow of reasoning reminded us of the Decision Tree algorithm, so we chose the Random Forest algorithm, which makes use of multiple decision trees.
3. Support Vector Machines: We chose this model because it would allow us to experiment with both linear and non-linear decision boundaries, simply by using a different kernel.

In the cells below, we use `run_model` and functions from `sklearn` to implement these algorithms, as well as to evaluate, visualize and discuss their performance. Note that we use a `random_state` of 0 when implementing the Random Forest and Support Vector Machines algorithms to ensure that the results of this project are reproducible, as well as an `n_jobs` of -1 when implementing the Random Forest algorithm to allow all routines to be run in parallel.

Model 1: k-nearest Neighbors

For our first model, we use the k-nearest Neighbors algorithm. For each new penguin whose species is unknown, this algorithm:

1. Calculates the distance between this new penguin and every penguin whose species is known. This quantifies the similarity between the new penguin's set of measurements and every other penguin's set of measurements.
2. Identifies the k penguins whose species are known that are closest to the new penguin, based on the distances calculated in #1.
3. Assigns the new penguin to the species that most of the k penguins in #2 belong to.

For this algorithm, we tune the following hyperparameters:

- Number of Neighbors (k): the number of neighboring points upon which each prediction is based. As k increases, the bias of the model increases, but the variance decreases.
- Type of Distance: this determines how distances between points are calculated. 1 indicates that the l_1 distance is used, while 2 indicates that the l_2 distance is used.
- Type of Weights: this controls how neighboring points are weighted. When uniform weights are used, all neighboring points are weighted equally, as opposed to being weighted by their distance from the point of interest.

In [17]:

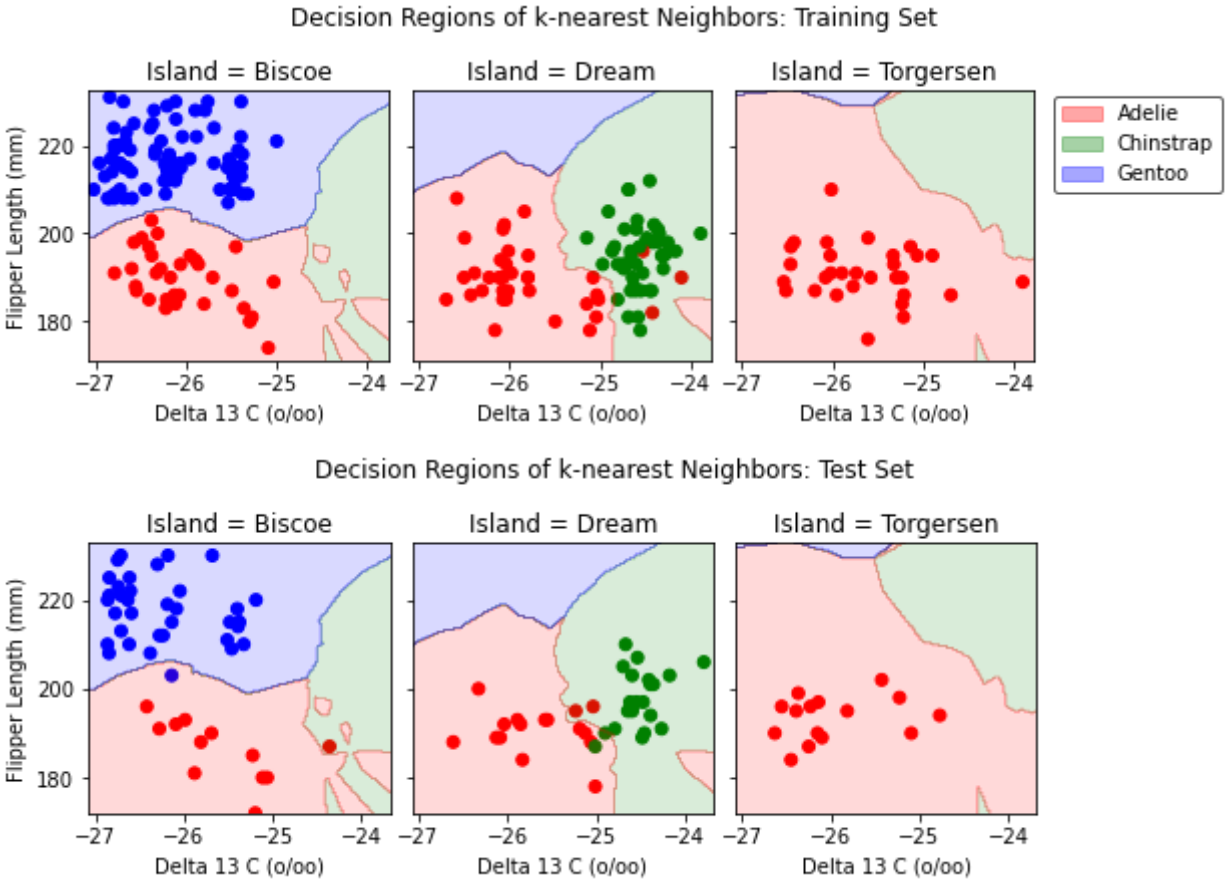
```
e1 = KNeighborsClassifier()
hp1 = {'n_neighbors': [1, 3, 5, 7, 10, 15, 25],
       'weights': ['uniform', 'distance'],
       'p': [1, 2]}
names1 = {'n_neighbors': 'Number of Neighbors (k)',
          'weights': 'Type of Weights',
          'p': 'Type of Distance'}
run_model(e1, hp1, names1, 'k-nearest Neighbors')
```

```
                            K-NEAREST NEIGHBORS
--------------------------------------------------------------------------------
                            Best Hyperparameters
--------------------------------------------------------------------------------

Number of Neighbors (k): 3
Type of Distance: 2
Type of Weights: Uniform

--------------------------------------------------------------------------------
                          Cross-validation Scores
--------------------------------------------------------------------------------

Best Hyperparameters: 0.98696

Overall Statistics
        Mean = 0.97032
        SD = 0.01447

Varying One Hyperparameter with the Others Fixed
        Number of Neighbors (k)
                Mean = 0.96726
                SD = 0.0162
        Type of Distance
```

```
                Mean = 0.98478
                SD = 0.00307
        Type of Weights
                Mean = 0.98696
                SD = 0.0

--------------------------------------------------------------------------------
                        Performance on Test Data
--------------------------------------------------------------------------------

Score: 0.94
Confusion Matrix:
```

	Predicted Adelie	Predicted Chinstrap	Predicted Gentoo
Actual Adelie	40	3	0
Actual Chinstrap	2	19	0
Actual Gentoo	1	0	35

```
--------------------------------------------------------------------------------
                        Decision Regions
--------------------------------------------------------------------------------
```



Hyperparameters

As shown above, the k-nearest Neighbors algorithm works best with $k = 3$, uniform weights and the l_2 distance, achieving a cross-validation score of about 0.987 on the training set and a score of 0.94 on the test set. Additionally, of the three hyperparameters that were tuned, k appears to have the greatest impact on the performance of the model, since the standard deviation of the

cross-validation scores obtained when it was allowed to vary while the other hyperparameters were held at their best values is the largest, at 0.0162.

Error Analysis

From the confusion matrix, we observe that the model misclassified 3 Adelie penguins as Chinstrap penguins, 2 Chinstrap penguins as Adelie penguins and 1 Gentoo penguin as an Adelie penguin. To get an idea of why Model 1 failed in these cases, we turn to the test set row of our decision region plots.

Looking at the Biscoe and Dream island plots, we see that a small number of Adelie penguins have higher than normal Delta 13C levels, which are more commonly observed in Chinstrap penguins. This explains why 3 Adelie penguins were misclassified as Chinstrap penguins. Similarly, a small number of Chinstrap penguins have lower than normal Delta 13C levels, which are more commonly observed in Adelie penguins. This explains why 2 Chinstrap penguins were misclassified as Adelie penguins. In other words, 5 of our 6 misclassified points under Model 1 can be explained by penguins having unusual Delta 13C values, relative to the other penguins of their species.

Other Observations

- **Unexpected Observations from the Decision Region Plots:** Aside from the errors discussed above, we note that all three decision region plots contain predictions for more species than we had expected based on the insights from our exploratory analysis. In the case of Biscoe and Torgersen islands, Chinstrap penguins are mistakenly predicted when high Delta 13C values are observed and in the case of Dream and Torgersen islands, Gentoo penguins are mistakenly predicted when higher Flipper Length values and relatively lower Delta 13C values are observed. This could be because the k-nearest Neighbors algorithm works by finding the nearest training observations to a new observation of interest; for the specific regions described, it may be that points in the training set with similar Delta 13C or Flipper Length values are closer in distance than points with similar Island categories to the new observation (especially since the Delta 13C and Flipper Length features were scaled using `StandardScaler`, whereas it does not make sense to scale a categorical variable such as Island, even after it has been encoded using `LabelEncoder`).
- **Run Time:** Since the k-nearest Neighbors algorithm has to calculate the distances of a new observation of interest to every training observation before making a prediction, the time taken to run the algorithm may become prohibitive as the number of training observations increases.

Model 2: Random Forest

For our second model, we use the Random Forest algorithm, which:

- Fits multiple decision trees, each of which:

    - Segments the sample space by asking multiple Yes/No questions regarding the values of the features used in the model. One such question that might be asked is, 'Was the penguin found on Biscoe island?' These questions are layered, i.e. after an initial question is asked to make the first partition of the sample space, a second question is posed to the observations that fall into each part of the segmented space. This second question can be the same, or different, for each part of the segmented space. Once all Yes/No questions have been asked, each part of the segmented space is associated with the species of penguins that is most abundant in that segment.
    - Predicts the species of a new penguin whose species is unknown by determining which part of the segmented space the new penguin falls in.
- For each new penguin whose species is unknown, the algorithm obtains predictions from each of the fitted trees and aggregates them to produce a single species prediction.

For this algorithm, we tune the following hyperparameters:

- Maximum Depth: the maximum depth of a tree. As the maximum depth increases, the complexity of the model increases.
- Minimum Samples per Leaf Node: the minimum number of samples required to be at a leaf node. As the value of this hyperparameter decreases, the model complexity increases.
- Number of Estimators/Trees: the number of trees in the forest. As the number of trees increases, the bias in each tree increases, but the variance of the model decreases.

In [18]:
```
e2 = RandomForestClassifier(random_state = 0, n_jobs = -1)
hp2 = {'n_estimators': [50, 100, 200],
       'max_depth': [2, 3, 5, 10, 25],
       'min_samples_leaf': [1, 2, 3, 5, 7]}
names2 = {'n_estimators': 'Number of Estimators/Trees',
          'max_depth': 'Maximum Depth',
          'min_samples_leaf': 'Minimum Samples per Leaf Node'}
run_model(e2, hp2, names2, 'Random Forest')
```

```
                                RANDOM FOREST
--------------------------------------------------------------------------------
                             Best Hyperparameters
--------------------------------------------------------------------------------

Maximum Depth: 3
Minimum Samples per Leaf Node: 1
Number of Estimators/Trees: 100

--------------------------------------------------------------------------------
                           Cross-validation Scores
--------------------------------------------------------------------------------

Best Hyperparameters: 0.97826

Overall Statistics
        Mean = 0.96278
        SD = 0.00774

Varying One Hyperparameter with the Others Fixed
        Maximum Depth
```

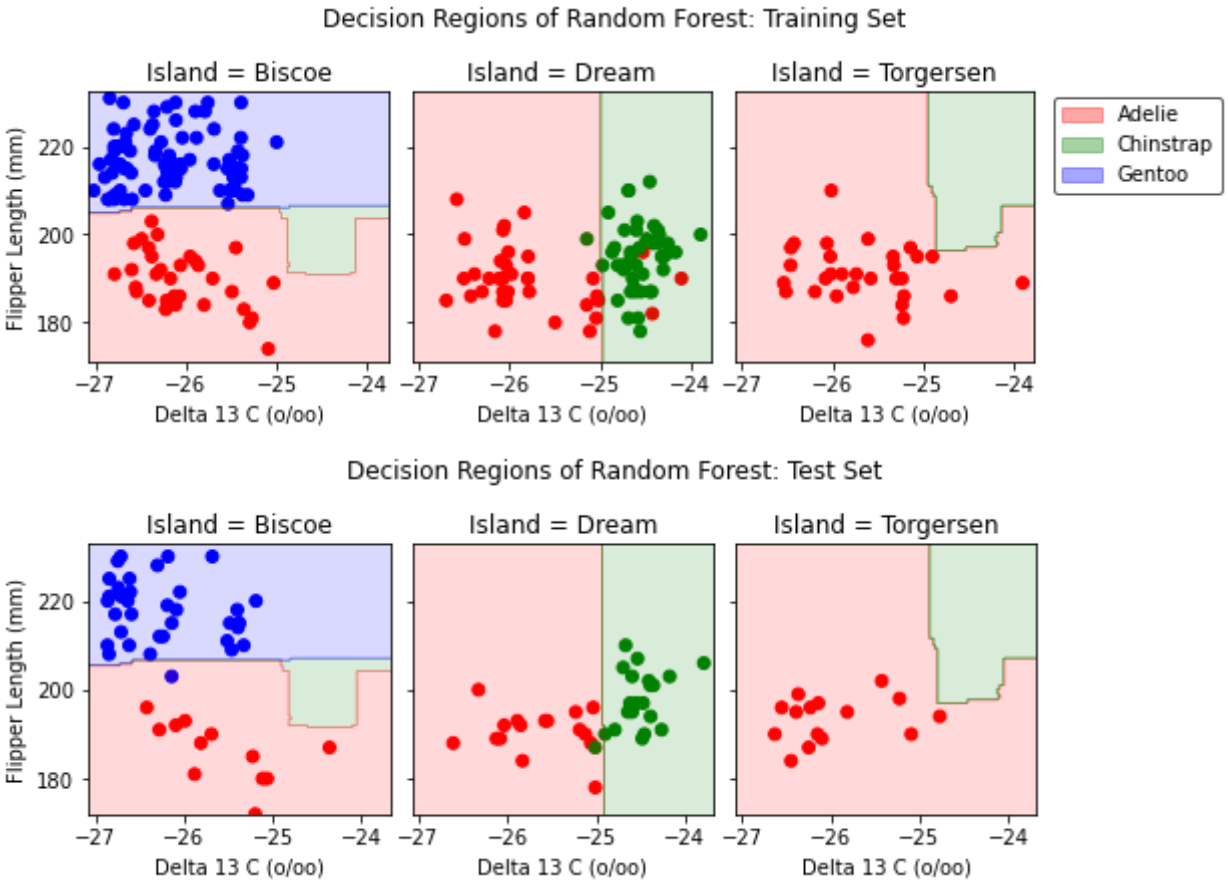
```
                Mean = 0.97391
                SD = 0.00532
        Minimum Samples per Leaf Node
                Mean = 0.96348
                SD = 0.01001
        Number of Estimators/Trees
                Mean = 0.97681
                SD = 0.00251
```

Performance on Test Data

Score: 0.98
Confusion Matrix:

	Predicted Adelie	Predicted Chinstrap	Predicted Gentoo
Actual Adelie	43	0	0
Actual Chinstrap	1	20	0
Actual Gentoo	1	0	35

Decision Regions



Hyperparameters

As shown above, the Random Forest algorithm works best with 100 trees, a maximum depth of 3 and a minimum of 1 sample per leaf node, achieving a cross-validation score of about 0.978 on

the training set and a score of 0.98 on the test set. Additionally, of the three hyperparameters that were tuned, the minimum samples per leaf node appears to have the greatest impact on the performance of the model, since the standard deviation of the cross-validation scores obtained when it was allowed to vary while the other hyperparameters were held at their best values is the largest, at about 0.01.

Error Analysis

From the confusion matrix, we observe that the model did not misclassify any Adelie penguins, but did misclassify 1 Chinstrap penguin and 1 Gentoo penguin as Adelie penguins. Looking at test set row of our decision region plots, we observe that, much like with Model 1, the misclassification of the Chinstrap and Gentoo penguins can be attributed to slightly lower than normal Delta 13C and Flipper Length values respectively.

Other Observations

- **Shapes of the Decision Regions:** Compared to the decision regions plotted for Model 1, we observe that the decision regions for Model 2 tend to be more rectangular, with straighter edges. This can be explained by the fact that the Random Forest algorithm makes use of decision trees, which individually tend to produce more rectangular decision regions.
- **Unexpected Observations from the Decision Region Plots:** Aside from the errors discussed above, we note that, similar to Model 1, the decision region plots for Biscoe and Torgersen island contain unexpected predictions of Chinstrap penguins. Similar to Model 1, these unexpected predictions occurred at higher Flipper Length values. However, Model 2 performed better than Model 1 in that (1) the size of the region allocated to Chinstrap penguins seems smaller on Biscoe island and (2) no Gentoo penguins were unexpectedly predicted on Dream or Torgersen island.
- **Run Time:** Similar to the k-nearest Neighbors algorithm, the Random Forest algorithm can be computationally-intensive for large datasets. In fact, we observed that implementing Model 2 took longer than implementing either of Model 1 or Model 3, even on our relatively small dataset.

Model 3: Support Vector Machines

For our third model, we use the Support Vector Machines algorithm. This algorithm functions differently based on the type of kernel used, but in essence:

- It first tries to find a plane that perfectly separates the classes in the feature space.
- If it fails to do so, it seeks the plane in the feature space that separates the classes with minimal mistakes.
- If the problem is too complex such that a plane is unsatisfactory, it enlarges the feature space so that separation is possible.

For this algorithm, we tune the following hyperparameters:

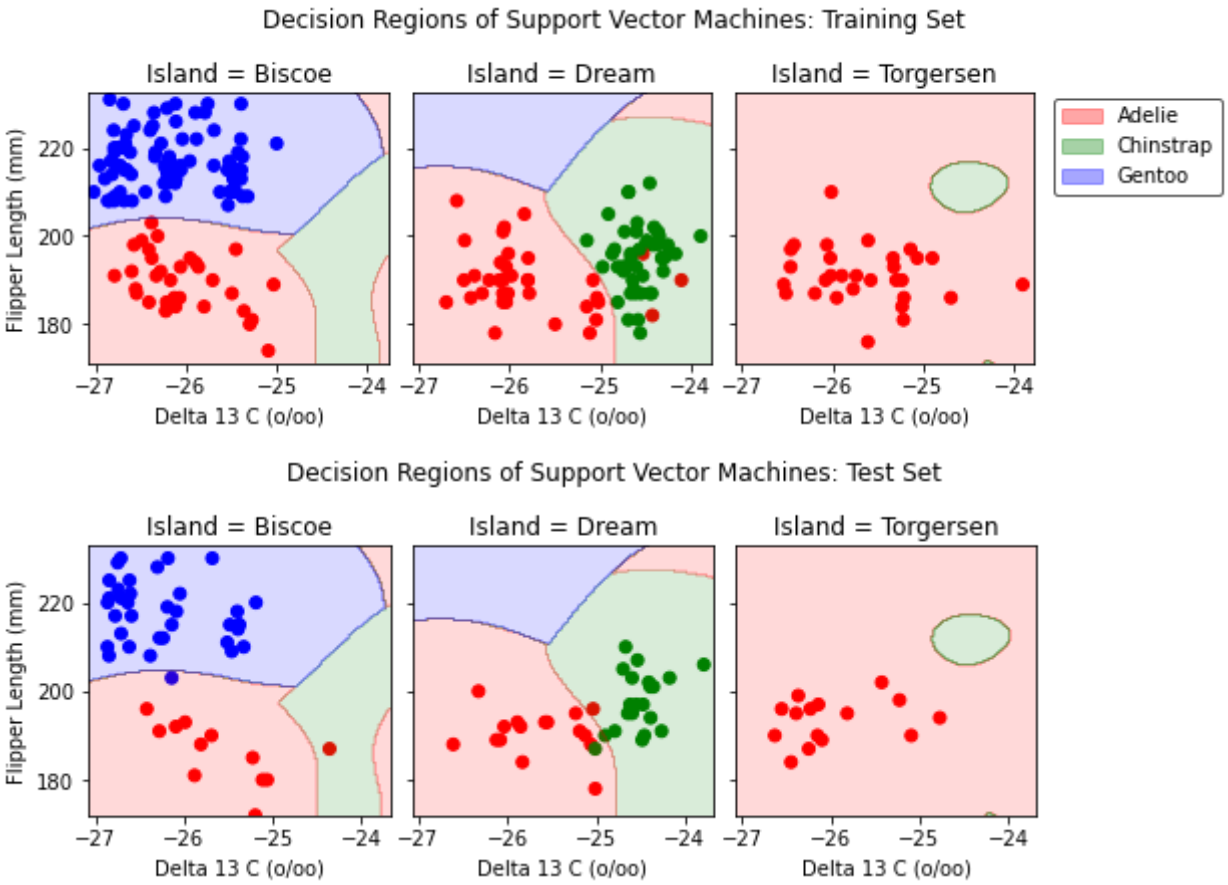
- Regularization Parameter (C): this allows us to strike a balance between bias and variance. According to `sklearn` documentation, the strength of the regularization decreases as the value of C increases.
- Kernel Type: the type of kernel to be used. This influences the general shape of the decision regions.

In [19]:
```
e3 = SVC(random_state = 0)
hp3 = {'C': [10**i for i in range(-2, 3)],
       'kernel': ['linear', 'poly', 'rbf']}
names3 = {'C': 'Regularization Parameter (C)',
          'kernel': 'Kernel Type'}
run_model(e3, hp3, names3, 'Support Vector Machines')
```

```
                          SUPPORT VECTOR MACHINES
--------------------------------------------------------------------------------
                            Best Hyperparameters
--------------------------------------------------------------------------------

Regularization Parameter (C): 10
Kernel Type: RBF

--------------------------------------------------------------------------------
                          Cross-validation Scores
--------------------------------------------------------------------------------

Best Hyperparameters: 0.98261

Overall Statistics
        Mean = 0.90129
        SD = 0.14841

Varying One Hyperparameter with the Others Fixed
        Regularization Parameter (C)
                Mean = 0.86322
                SD = 0.24546
        Kernel Type
                Mean = 0.97397
                SD = 0.00861

--------------------------------------------------------------------------------
                          Performance on Test Data
--------------------------------------------------------------------------------

Score: 0.96
Confusion Matrix:
```

	Predicted Adelie	Predicted Chinstrap	Predicted Gentoo
Actual Adelie	41	2	0
Actual Chinstrap	1	20	0
Actual Gentoo	1	0	35

```
--------------------------------------------------------------------------------
                             Decision Regions
--------------------------------------------------------------------------------
```



Hyperparameters

As shown above, the Support Vector Machines algorithm works best with $C = 10$ and when using the radial basis function (RBF) kernel, achieving a cross-validation score of about 0.983 on the training set and a score of 0.96 on the test set. Additionally, of the two hyperparameters that were tuned, C appears to have the greatest impact on the performance of the model, since the standard deviation of the cross-validation scores obtained when it was allowed to vary while the other hyperparameters were held at their best values is the largest, at about 0.245.

Error Analysis

From the confusion matrix, we observe that the model misclassified 2 Adelie penguins as Chinstrap penguins, 1 Chinstrap penguin as an Adelie penguin and 1 Gentoo penguin as an Adelie penguin. Looking at test set row of our decision region plots, we observe that, much like with Models 1 and 2, the misclassification of the Adelie and Chinstrap penguins can be attributed to slightly higher and lower than normal Delta 13C values respectively. Additionally, we see that the Gentoo penguin was misclassified due to having flippers that are slightly shorter than normal.

Other Observations

- **Unexpected Observations from the Decision Region Plots:** Aside from the errors discussed above, we note that, similar to Models 1 and 2, all three decision region plots

contain predictions for more species than we had expected based on the insights from our exploratory analysis. These unexpected species predictions occurred at roughly the same Delta 13C and Flipper Length values for Model 3 as in the previous two models.

Discussion

Overall Model Performance

Overall, the features that we chose proved useful and all three models performed relatively well, scoring above 0.9 across the board when applied to the test set. Model 2, implemented using the Random Forest algorithm with 100 trees, a maximum depth of 3 and a minimum of 1 sample per leaf node performed the best, scoring 0.98. Model 3, which used the Support Vector Machines algorithm came in second, scoring 0.96 and Model 1, which used the k-nearest Neighbors algorithm was not far behind, scoring 0.94. Considering our results above, we recommend using **Island, Delta 13C and Flipper Length** as features, along with the **Random Forest** algorithm, to predict the Species of a penguin using a limited number of measurements.

Suggestions

Looking at the mistakes made across our three models, we note that differentiating between Adelie and Chinstrap penguins, particularly for Delta 13C values close to -25 o/oo, was a challenge. As such, if we were to improve our models further, we would seek additional data that could help us better classify penguins with these specifications. In particular, as exemplified by the picture below, the Chinstrap penguin has a visually distinctive line running across its face, near its chin. Hence, we might suggest that researchers make a note of whether a penguin has such a line running across its face during the data collection process. This measurement, while relatively easy to collect, will likely improve the accuracy of our models further.

Chinstrap Penguin: https://www.penguins-world.com/chinstrap-penguin/