CS109A Introduction to Data Science

Homework 7: Classification with Logistic Regression, LDA/QDA, and Trees

Harvard University
Fall 2018
Instructors: Pavlos Protopapas, Kevin Rader

In [1]:
```
#RUN THIS CELL
import requests
from IPython.core.display import HTML
styles = requests.get("https://raw.githubusercontent.com/Harvard-IACS/2018-CS109A
HTML(styles)
```

Out[1]:

INSTRUCTIONS

- To submit your assignment follow the instructions given in Canvas (https://canvas.harvard.edu/courses/42693/pages/homework-policies-and-submission-instructions).
- If needed, clarifications will be posted on Piazza.
- This homework can be submitted in pairs.
- If you submit individually but you have worked with someone, please include the name of your **one** partner below.

Name of the person you have worked with goes here:

In [2]:
```
%matplotlib inline
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()

from sklearn.decomposition import PCA
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import LogisticRegressionCV
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis
from sklearn.preprocessing import PolynomialFeatures
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_graphviz
from sklearn.pipeline import make_pipeline
from sklearn.datasets import make_blobs
from sklearn.metrics import confusion_matrix
import itertools
```

Question 1 [20 pts]: Overview of Multiclass Thyroid Classification

In this problem set you will build a model for diagnosing disorders in a patient's thyroid gland. Given the results of medical tests on a patient, the task is to classify the patient either as:

- *normal* (class 1)
- having *hyperthyroidism* (class 2)
- or having *hypothyroidism* (class 3).

The data set is provided in the file `dataset_hw7.csv` . Columns 1-2 contain biomarkers for a patient (predictors):

- Biomarker 1: (Logarithm of) level of basal thyroid-stimulating hormone (TSH) as measured by radioimmuno assay
- Biomarker 2: (Logarithm of) maximal absolute difference of TSH value after injection of 200 micro grams of thyrotropin-releasing hormone as compared to the basal value.

The last column contains the diagnosis for the patient from a medical expert. This data set was obtained from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/datasets/Thyroid+Disease); for this assignment we chose two predictors so we can visualize the decision boundaries.

Notice that unlike previous exercises, the task at hand is a 3-class classification problem. We will explore different methods for multiclass classification.

For most of this problem set, we'll measure overall classification accuracy as the fraction of observations classified correctly.

1.1 Load the data and examine its structure. How many instances of each class are there in our dataset? In particular, what is the ratio of the number of observations in class 2 (hyperthyroidism) to the number of observations in class 3 (hypothyroidism)? We'll refer to this as the *hyper-to-hypo ratio*.

1.2: We're going to split this data into a 50% training set and a 50% test set. But since our dataset is small, we need to make sure we do it correctly. Let's see what happens when we *don't* split correctly: for each of 100 different random splits of the data into 50% train and 50% test, compute the hyper-to-hypo for the observations end up in the training set. Plot the distribution of the hyper-to-hypo ratio; on your plot, also mark the hyper-to-hypo ratio that you found in the full dataset. Discuss how representative the training and test sets are likely to be if we were to have selected one of these random splits.

1.3 Now, we'll use the `stratify` option to split the data in such a way that the relative class frequencies are preserved (the code is provided). Make a table showing how many observations of each class ended up in your training and test sets. Verify that the hyper-hypo ratio is roughly the same in both sets.

1.4 Provide the scatterplot of the predictors in the (training) data in a way that clearly indicates which class each observation belongs to.

1.5: When we first start working with a dataset or algorithm, it's typically a good idea to figure out what *baselines* we might compare our results to. For regression, we always compared against a baseline of predicting the mean (in computing R^2). For classification, a simple baseline is always predicting the *most common class*. What "baseline" accuracy can we achieve on the thyroid classification problem by always predicting the most common class? Assign the result to `baseline_accuracy` so we can use it later. (**note: don't look at the test set until instructed**)

1.6 Make a decision function to separate these samples using no library functions; just write out your logic by hand. Your manual classifier doesn't need to be well-tuned (we'll be exploring algorithms to do that!); it only needs to (1) predict each class at least once, and (2) achieve an accuracy at least 10% greater accurate than predicting the most likely class. Use the `overlay_decision_boundaries` function provided above to overlay the decision boundaries of your function on the training set. (Note that the function modifies an existing plot, so call it after plotting your points.)

Based on your exploration, do you think a linear classifier (i.e., a classifier where all decision boundaries are line segments) could achieve above 85% accuracy on this dataset? Could a non-linear classifier do better? What characteristics of the data lead you to these conclusions?

1.1

In [3]:
```
# your code here
df = pd.read_csv('dataset_hw7.csv')
display(df.describe())
```

	Biomarker 1	Biomarker 2	Diagnosis
count	215.000000	215.000000	215.000000
mean	0.414441	0.303155	1.441860
std	0.888106	2.174369	0.726737
min	-2.302485	-11.512925	1.000000
25%	0.000010	-0.510809	1.000000
50%	0.262372	0.693152	1.000000
75%	0.530634	1.410989	2.000000
max	4.032469	4.030695	3.000000

In [4]:
```
# your code here
print('Instances for each class:')
display(df['Diagnosis'].value_counts())

hyper = df.loc[df['Diagnosis']==2, 'Diagnosis'].count()
hypo = df.loc[df['Diagnosis']==3, 'Diagnosis'].count()
hyper_hypo_full = hyper/hypo
print('Hyper-to-hypo Ratio: %s'%hyper_hypo_full)
```

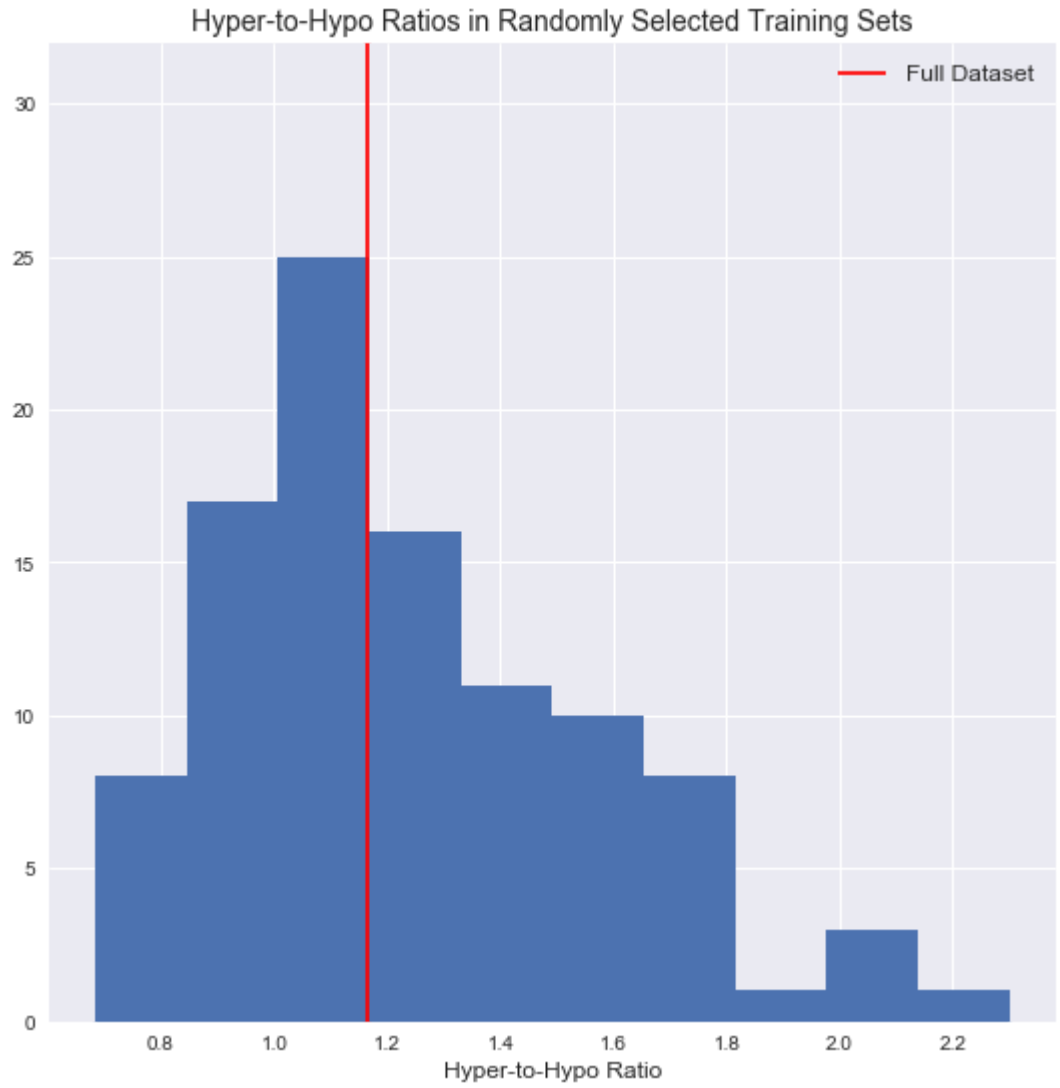
```
Instances for each class:

1    150
2     35
3     30
Name: Diagnosis, dtype: int64

Hyper-to-hypo Ratio: 1.1666666666666667
```

1.2

In [5]:
```
# your code here
hyper_hypo_rand = []
for i in range(100):
    # create random train/test
    msk = np.random.rand(len(df)) < 0.5
    df_train = df[msk]
    df_test = df[~msk]

    # ratio
    hyper = df_train.loc[df_train['Diagnosis']==2, 'Diagnosis'].count()
    hypo = df_train.loc[df_train['Diagnosis']==3, 'Diagnosis'].count()
    hyper_hypo_rand.append(hyper/hypo)

print('25th percentile: %s'%np.percentile(hyper_hypo_rand,25))
print('75th percentile: %s'%np.percentile(hyper_hypo_rand,75))

# plot ratios from randomly created train/test
fig, ax = plt.subplots(1,1, figsize=(9,9))
ax.hist(hyper_hypo_rand)
ax.vlines(x=hyper_hypo_full, ymin=0, ymax=40, color='red', label='Full Dataset')
ax.set_xlabel('Hyper-to-Hypo Ratio', fontsize=12)
ax.set_ylim(top=32)
ax.legend(loc='best', fontsize=12)
ax.set_title('Hyper-to-Hypo Ratios in Randomly Selected Training Sets', fontsize
```

```
25th percentile: 1.0394736842105263
75th percentile: 1.4615384615384615
```

Out[5]:
```
Text(0.5,1,'Hyper-to-Hypo Ratios in Randomly Selected Training Sets')
```



Answer: As we can see above, if we are to select the train and the test sets randomly, there is 50% chance data will have a hyper-to-hypo outside of the range (1,1.3). Hence, selecting the train and test sets randomly might misrepresent our priors while doing the classification and thus skew our results.

1.3

In [6]:
```
data_train, data_test = train_test_split(df, test_size=.5, stratify=df.Diagnosis
```

In [7]:
```
# your code here
print('Training Set Counts:')
display(pd.DataFrame(data_train['Diagnosis'].value_counts()))
print('\nTest Set Counts:')
display(pd.DataFrame(data_test['Diagnosis'].value_counts()))
```

```
Training Set Counts:
```

	Diagnosis
1	75
2	17
3	15

```
Test Set Counts:
```

	Diagnosis
1	75
2	18
3	15

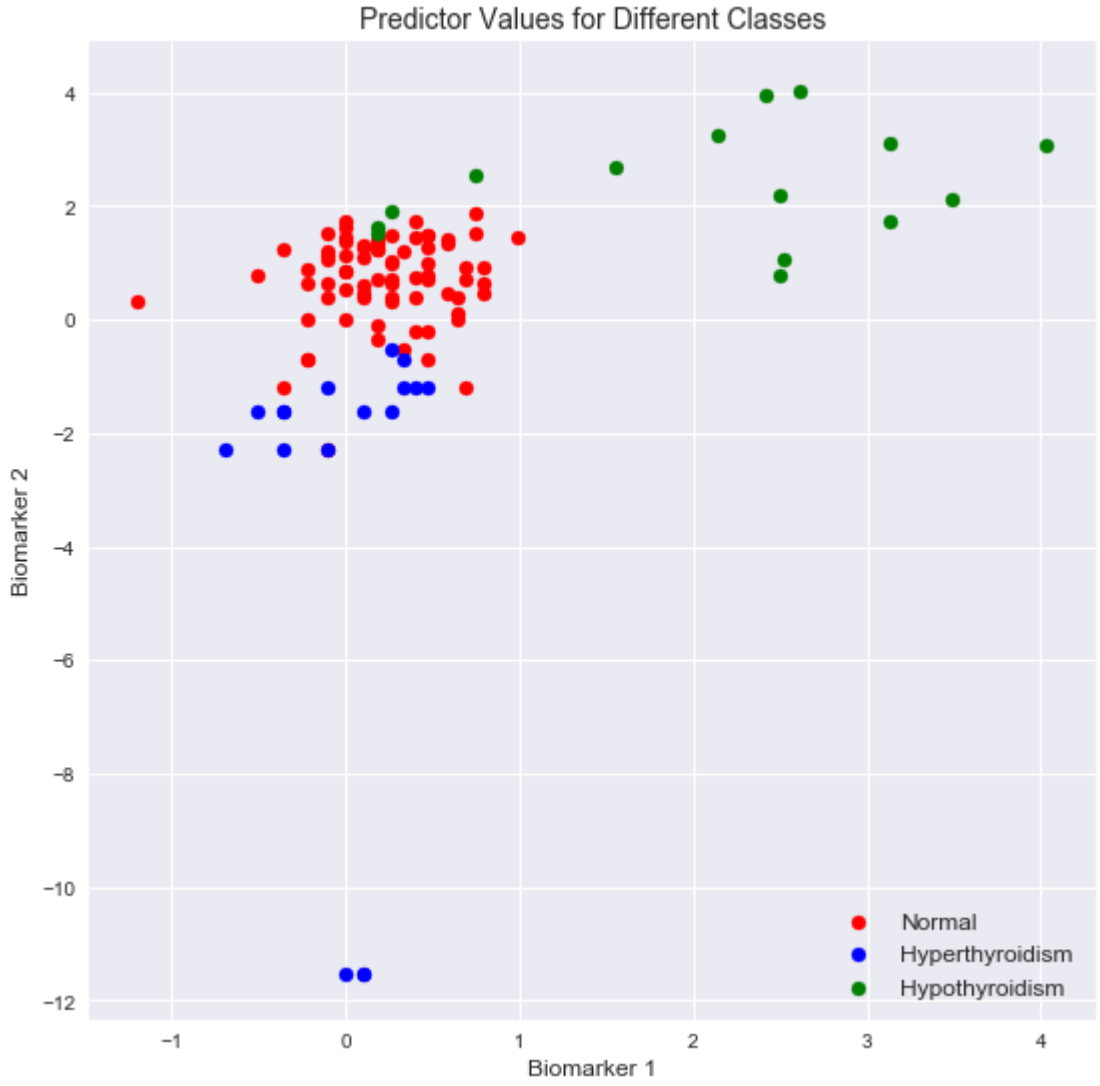
In [8]:
```
# your code here
hyper_train = data_train.loc[data_train['Diagnosis']==2, 'Diagnosis'].count()
hypo_train = data_train.loc[data_train['Diagnosis']==3, 'Diagnosis'].count()
hyper_hypo_train = hyper_train/hypo_train
print('Hyper-to-hypo Ratio (Train): %s'%hyper_hypo_train)

hyper_test = data_test.loc[data_test['Diagnosis']==2, 'Diagnosis'].count()
hypo_test = data_test.loc[data_test['Diagnosis']==3, 'Diagnosis'].count()
hyper_hypo_test = hyper_test/hypo_test
print('Hyper-to-hypo Ratio (Test): %s'%hyper_hypo_test)
```

```
Hyper-to-hypo Ratio (Train): 1.1333333333333333
Hyper-to-hypo Ratio (Test): 1.2
```

1.4

In [9]:
```
# your code here
color = ['red', 'blue', 'green']
classes = ['Normal', 'Hyperthyroidism','Hypothyroidism']
fig, ax = plt.subplots(1,1, figsize=(9,9))
for i in range(1, len(color)+1):
    ax.scatter(data_train.loc[data_train['Diagnosis']==i,'Biomarker 1'],
               data_train.loc[data_train['Diagnosis']==i,'Biomarker 2'],
               color=color[i-1], label=classes[i-1])
ax.set_xlabel('Biomarker 1', fontsize=12)
ax.set_ylabel('Biomarker 2', fontsize=12)
ax.legend(loc='lower right', fontsize=12)
ax.set_title('Predictor Values for Different Classes', fontsize=14)
```

Out[9]: Text(0.5,1,'Predictor Values for Different Classes')



1.5

In [10]:
```
# your code here
train_gbdf = data_train.groupby(by='Diagnosis')
baseline_accuracy = train_gbdf['Diagnosis'].agg('count').max()/train_gbdf['Diagn
print('Baseline accuracy if we always predict most common class: %s'%baseline_ac
```

```
Baseline accuracy if we always predict most common class: 0.7009345794392523
```

1.6

In [11]:
```
def overlay_decision_boundary(ax, model, colors=None, nx=200, ny=200, desaturate=
    """
    A function that visualizes the decision boundaries of a classifier.

    ax: Matplotlib Axes to plot on
    model: Classifier to use.
     - if `model` has a `.predict` method, like an sklearn classifier, we call `m
     - otherwise, we simply call `model(X)`
    colors: list or dict of colors to use. Use color `colors[i]` for class i.
     - If colors is not provided, uses the current color cycle
    nx, ny: number of mesh points to evaluated the classifier on
    desaturate: how much to desaturate each of the colors (for better contrast wi
    xlim, ylim: range to plot on. (If the default, None, is passed, the limits wi
    """
    # Create mesh.
    xmin, xmax = ax.get_xlim() if xlim is None else xlim
    ymin, ymax = ax.get_ylim() if ylim is None else ylim
    xx, yy = np.meshgrid(
        np.linspace(xmin, xmax, nx),
        np.linspace(ymin, ymax, ny))
    X = np.c_[xx.flatten(), yy.flatten()]

    # Predict on mesh of points.
    model = getattr(model, 'predict', model)
    y = model(X)
    y = y.reshape((nx, ny))

    # Generate colormap.
    if colors is None:
        # If colors not provided, use the current color cycle.
        # Shift the indices so that the lowest class actually predicted gets the
        # ^ This is a bit magic, consider removing for next year.
        colors = (['white'] * np.min(y.astype(int)) + sns.utils.get_color_cycle(

    if isinstance(colors, dict):
        missing_colors = [idx for idx in np.unique(y) if idx not in colors]
        assert len(missing_colors) == 0, f"Color not specified for predictions {m

        # Make a list of colors, filling in items from the dict.
        color_list = ['white'] * (np.max(y.astype(int)) + 1)
        for idx, val in colors.items():
            color_list[idx] = val
    else:
        assert len(colors) >= np.max(y) + 1, "Insufficient colors passed for all
        color_list = colors
    color_list = [sns.utils.desaturate(color, desaturate) for color in color_list
    cmap = matplotlib.colors.ListedColormap(color_list)

    # Plot decision surface
    ax.pcolormesh(xx, yy, y, zorder=-2, cmap=cmap, norm=matplotlib.colors.NoNorm
    xx = xx.reshape(nx, ny)
    yy = yy.reshape(nx, ny)
    if len(np.unique(y)) > 1:
        ax.contour(xx, yy, y, colors="black", linewidths=1, zorder=-1)
    else:
        print("Warning: only one class predicted, so not plotting contour lines.
```

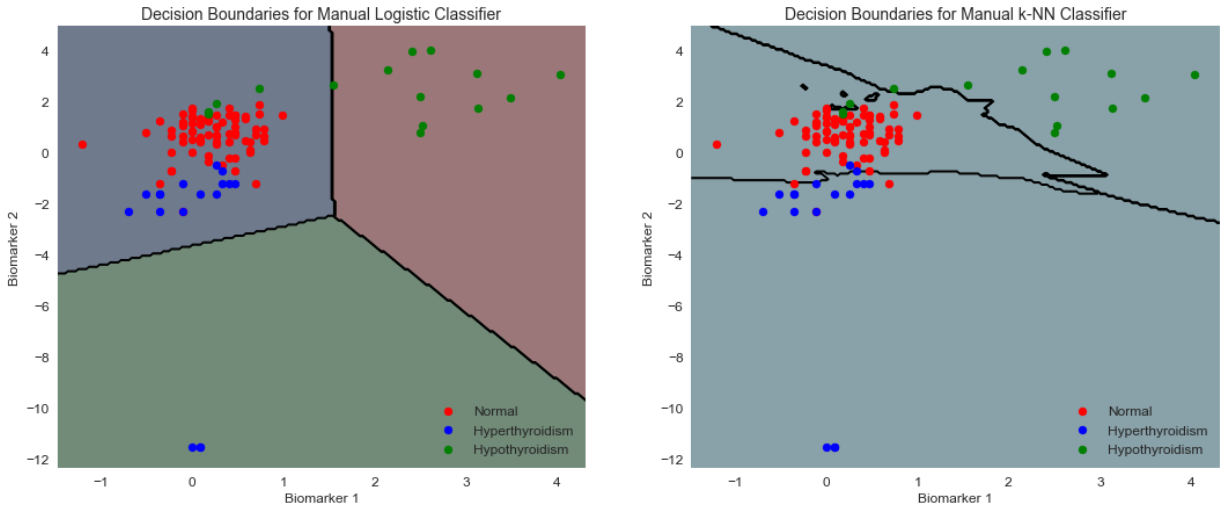
In [12]:
```
# scatter plot function
def make_scatter(ax, X, y, color, title):
    for l in range(1, len(color)+1):
        ax.scatter(X.loc[y==l,'Biomarker 1'],
                   X.loc[y==l,'Biomarker 2'],
                   color=color[l-1], label=classes[l-1])
    ax.set_xlabel('Biomarker 1', fontsize=12)
    ax.set_ylabel('Biomarker 2', fontsize=12)
    ax.set_title(title, fontsize=14)
    ax.legend(loc='lower right', fontsize=12)
    ax.tick_params(labelsize=12)
```

In [13]:
```
class Classifier_log:
    def __init__(self, y):
        self.slopes = {}
        self.intercept = {}
        self.y_copy = y.copy()
    def fit(self, X, y):

        classes = y.unique()
        for k in classes:
            self.y_copy[y==k] = 1
            self.y_copy[y!=k] = 0
            y_bar = np.mean(self.y_copy)
            x_bar = np.mean(X)
            #slope beta1
            beta = []
            for i in X.columns:
                numerator = np.sum( (X - x_bar)[i]*(self.y_copy - y_bar) )
                denominator = np.sum((X - x_bar)[i]**2)
                beta.append(numerator/denominator)
            self.slopes[k] = np.array([beta]).reshape(-1,)
            self.intercept[k] = y_bar - np.dot(self.slopes[k], x_bar.T)

    def predict(self, X):
        sigmoid = {}
        for k in self.slopes.keys():
            best_fit = self.intercept[k] + np.dot(X, self.slopes[k].T)
            y_pred = best_fit
            sigmoid[k] = 1/(1+np.exp(-y_pred))
        sigmoids = pd.DataFrame(sigmoid
#                                 , index=self.y_copy.index
                               )
        y_pred_class = []
        for i in sigmoids.index:
            y_pred_class.append(sigmoids.loc[i,:].idxmax())
        return(np.array(y_pred_class))
```

In [14]:
```
import operator
class Classifier_knn:
    def __init__(self, data_train, n):
        self.trainSet = data_train
        self.n_neighbors = n

    def euclideanDistance(self, instance1, instance2, length):
        distance = 0
        for x in range(length):
            distance += pow((instance1[x] - instance2[x]), 2)
        return np.sqrt(distance)
    def getNeighbors(self, trainingSet, testInstance, k):
        distances = []
        length = len(testInstance)-1
        for x in trainingSet:
            dist = self.euclideanDistance(testInstance, x, 2)
            distances.append((x, dist))
        distances.sort(key=operator.itemgetter(1))
        neighbors = []
        for x in range(k):
            neighbors.append(distances[x][0])
        return neighbors
    def getResponse(self, neighbors):
        classVotes = {}
        for x in range(len(neighbors)):
            response = neighbors[x][-1]
            if response in classVotes:
                classVotes[response] += 1
            else:
                classVotes[response] = 1
        sortedVotes = sorted(classVotes.items(), key=operator.itemgetter(1), reve
        return sortedVotes[0][0]

    def predict(self, testInstance):
        result = []
        for i in testInstance:
            neighbors = self.getNeighbors(self.trainSet, i, self.n_neighbors)
            response = self.getResponse(neighbors)
            result.append(response)
        return(np.array(result))
```

In [15]:
```
X_train = data_train.drop(['Diagnosis'], axis=1)
y_train = data_train['Diagnosis']
X_test = data_test.drop(['Diagnosis'], axis=1)
y_test = data_test['Diagnosis']

color = ['red', 'blue', 'green']
classes = ['Normal', 'Hyperthyroidism','Hypothyroidism']
```

In [16]:
```
c_log = Classifier_log(y=y_train)
fit_log = c_log.fit(X=X_train, y=y_train)

c_knn = Classifier_knn(data_train.values, 5)
predict_knn = c_knn.predict(data_train.values)

fig, ax = plt.subplots(1,2, figsize=(18,7))
make_scatter(ax[0], X_train, y_train, color, 'Decision Boundaries for Manual Log
overlay_decision_boundary(ax[0], c_log, desaturate=.3)

make_scatter(ax[1], X_train, y_train, color, 'Decision Boundaries for Manual k-N
overlay_decision_boundary(ax[1], c_knn, desaturate=.3)
```



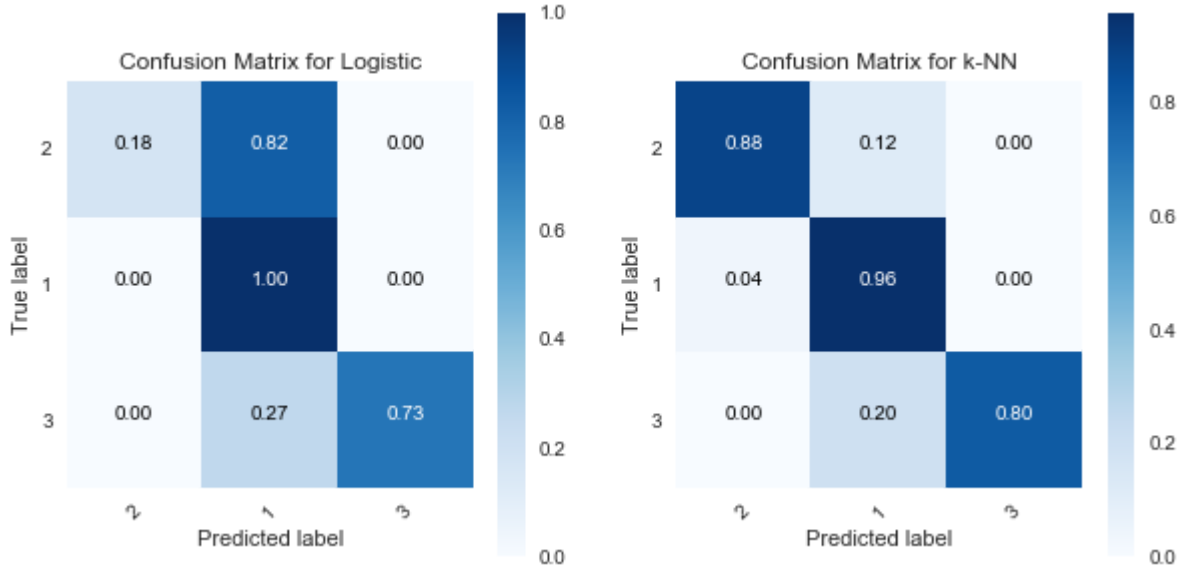
In [17]:
```
manual_predictions_log = c_log.predict(X_train)
accuracy_log = accuracy_score(y_train, manual_predictions_log)
print("Accuracy with Manual Logistic Regression:", accuracy_log)

manual_predictions_knn = c_knn.predict(data_train.values)
accuracy_knn = accuracy_score(y_train, manual_predictions_knn)
print("Accuracy with Manual k-NN:", accuracy_knn)
```

```
Accuracy with Manual Logistic Regression: 0.8317757009345794
Accuracy with Manual k-NN: 0.9252336448598131
```

In [18]:
```
assert accuracy_log >= (baseline_accuracy * 1.10), "Accuracy too low"
assert all(np.sum(manual_predictions_log == i) > 0 for i in [1, 2, 3]), "Should

assert accuracy_knn >= (baseline_accuracy * 1.10), "Accuracy too low"
assert all(np.sum(manual_predictions_knn == i) > 0 for i in [1, 2, 3]), "Should
```

In [19]:
```
def plot_confusion_matrix(ax, cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
#         print("Normalized confusion matrix")
#     else:
#         print('Confusion matrix, without normalization')
    img = ax.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.colorbar(img, ax=ax)
    tick_marks = np.arange(len(classes))
    ax.set(xticks=tick_marks, yticks=tick_marks)
    ax.set_xticklabels(classes, rotation=45)
    ax.set_yticklabels(classes)
    ax.grid(b=False)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        ax.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    ax.set(title=title, ylabel='True label', xlabel='Predicted label')
```

In [20]:
```
fig, ax = plt.subplots(1,2, figsize=(10,5))
cnf_matrix_log = confusion_matrix(y_train, manual_predictions_log, labels=y_trai
plot_confusion_matrix(ax[0], cnf_matrix_log, classes=y_train.unique(), normalize
cnf_matrix_knn = confusion_matrix(y_train, manual_predictions_knn, labels=y_trai
plot_confusion_matrix(ax[1], cnf_matrix_knn, classes=y_train.unique(), normalize
```



Answer:

A linear classifier can achieve an overall accuracy of 85% or higher is the decision boundaries between `Class 1` and `Class 2` was shifted a bit towards `Class 1`. Also, based on the plots and the confusion matrices, the linear classifier gives us an overall performance (~83%) that is greater than the 10% above our `baseline_accuracy`. However, it performs really poor between `Class 1` and `Class 2`. We know that a k-NN is always going to perform well on the training data with small 'k', however, we see that it does well in identifying the boundary between `Class 1` and `Class 2`. We can also see that the distributions (mean and standard deviation) of 'X' is different for different classes, which means a non-linear model might be better choice for this data.

Question 2 [20 pts]: Multiclass Logistic Regression

2.1 Fit two one-vs-rest logistic regression models using sklearn. For the first model, use the train dataset as-is (so the decision boundaries will be linear); for the second model, also include quadratic and interaction terms. For both models, use L_2 regularization, tuning the regularization parameter using 5-fold cross-validation.

For each model, make a plot of the training data with the decision boundaries overlayed.

2.2 Interpret the decision boundaries:

- Do these decision boundaries make sense?
- What does adding quadratic and interaction features do to the shape of the decision boundaries? Why?
- How do the different models treat regions where there are few samples? How do they classify such samples?

2.3 Compare the performance of the two logistic regression models above using 5-fold cross-validation. Which model performs best? How confident are you about this conclusion? Does the inclusion of the polynomial terms in logistic regression yield better accuracy compared to the model with only linear terms? Why do you suspect it is better or worse?

Hint: You may use the `cross_val_score` function for cross-validation.

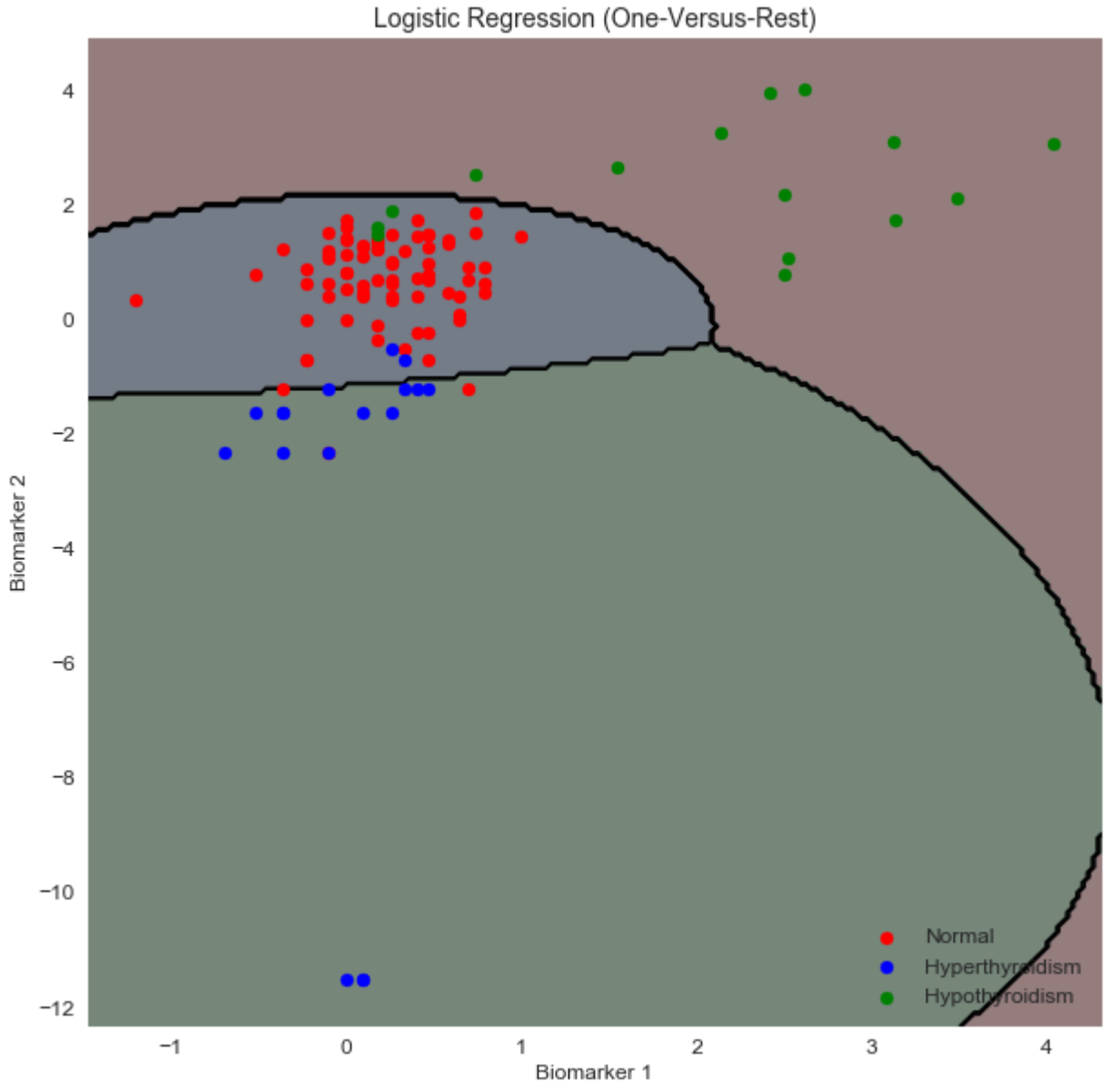
2.1

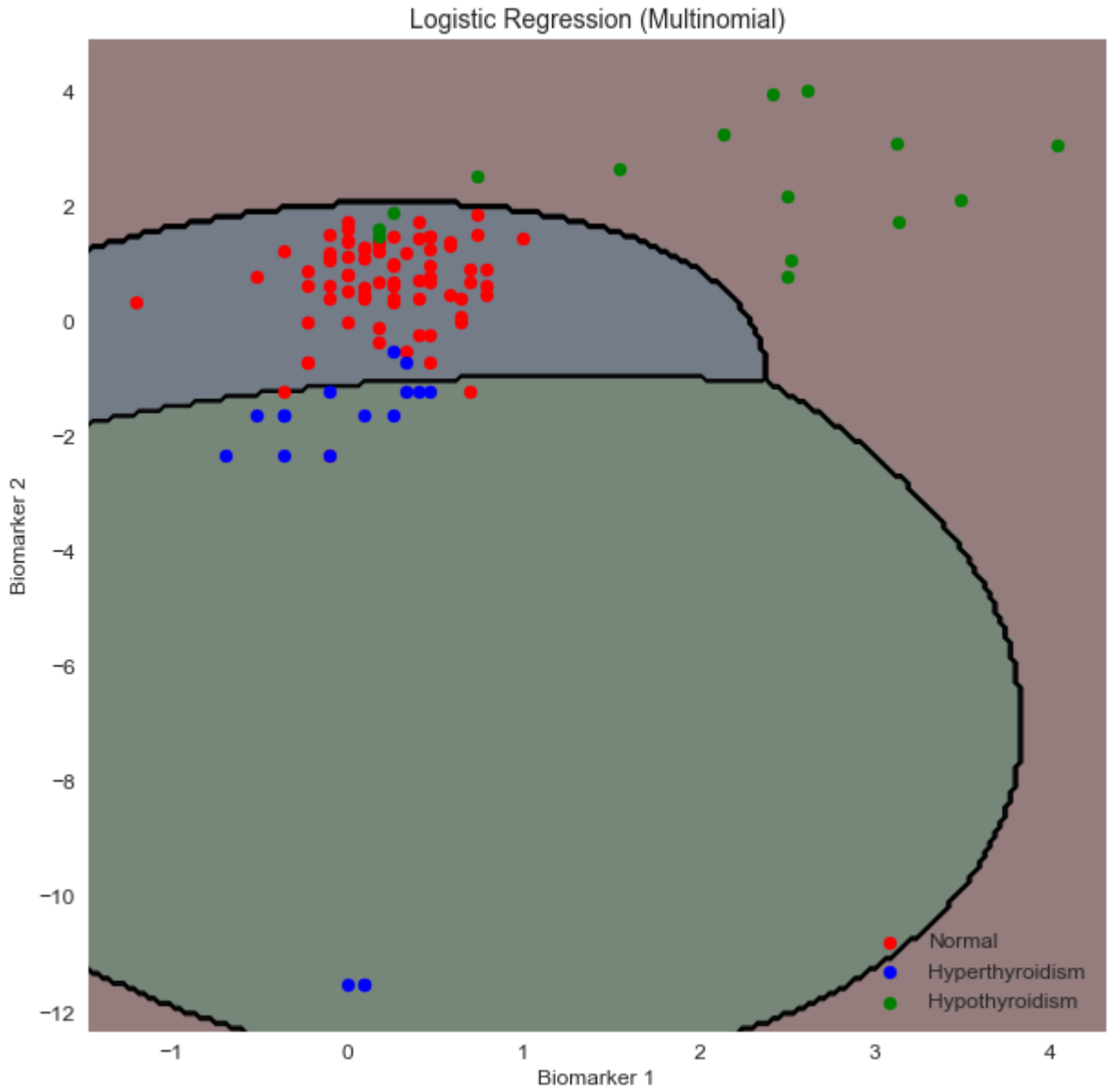
Hint: You should use `LogisticRegressionCV`. For the model with quadratic and interaction terms, use the following Pipeline:

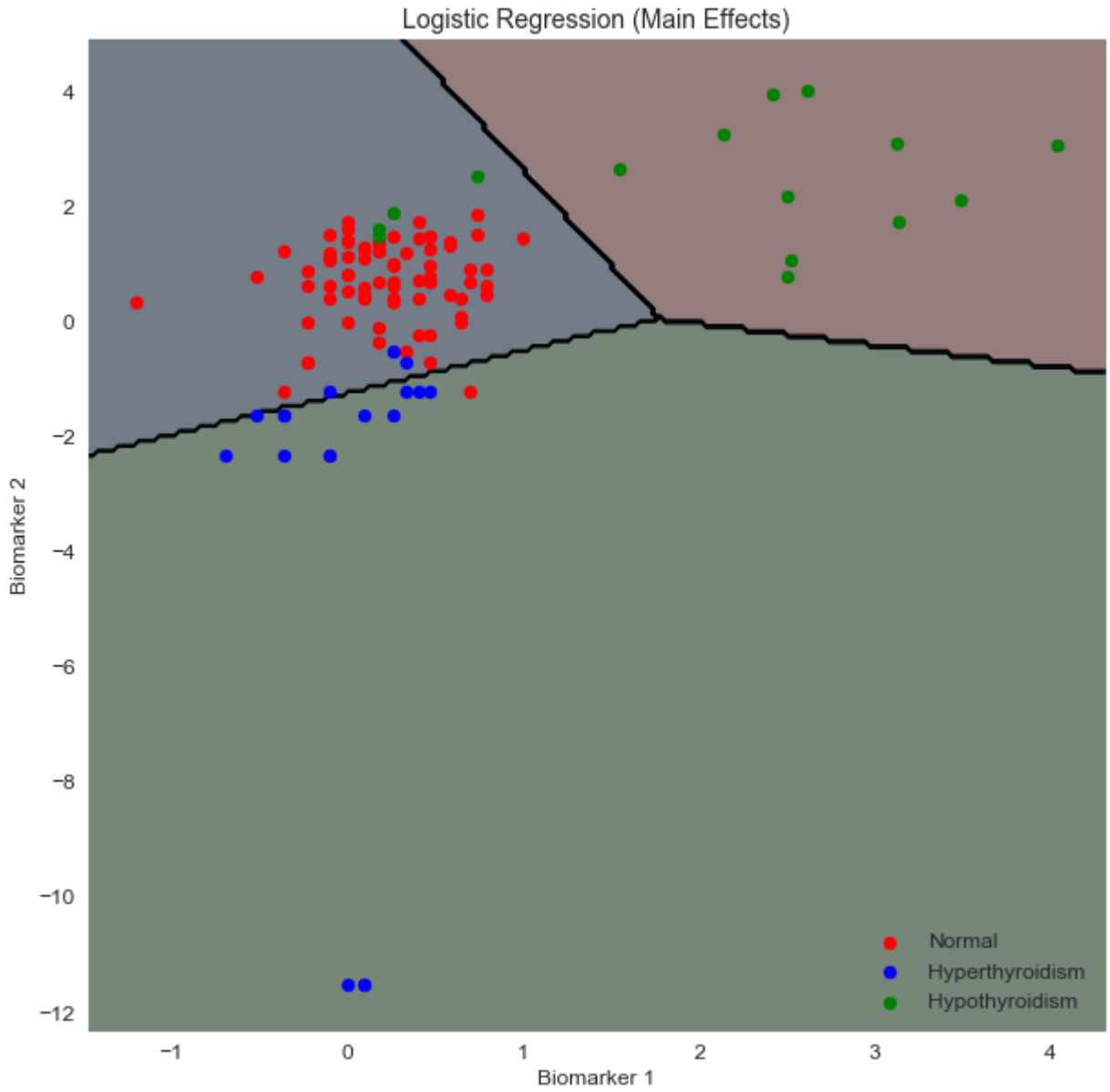
In [21]:
```
# estimators
polynomial_logreg_estimator_ovr = make_pipeline(
    PolynomialFeatures(degree=2, include_bias=False),
    LogisticRegressionCV(cv=5, penalty='l2', multi_class="ovr"))

polynomial_logreg_estimator_multi = make_pipeline(
    PolynomialFeatures(degree=2, include_bias=False),
    LogisticRegressionCV(cv=5, penalty='l2', multi_class="multinomial"))

# fit, test models
logreg_ovr = polynomial_logreg_estimator_ovr.fit(X_train, y_train)
display(logreg_ovr.score(X_train, y_train))

logreg_multi = polynomial_logreg_estimator_multi.fit(X_train, y_train)
display(logreg_multi.score(X_train, y_train))

logreg_main = LogisticRegressionCV(cv=5, penalty='l2', multi_class="ovr").fit(X_
display(logreg_main.score(X_train, y_train))

# Note that you can access the logistic regression classifier itself by
# polynomial_logreg_estimator.named_steps['logisticregressioncv']
```

```
0.9345794392523364

0.9252336448598131

0.9158878504672897
```

In [22]:
```
# your code here

## plot decision boundaries
# polynomial using 'one-versus-rest'
fig1, ax1 = plt.subplots(1, 1, figsize=(10,10))
make_scatter(ax1, X_train, y_train, color, title='Logistic Regression (One-Versus
overlay_decision_boundary(ax1, model=logreg_ovr, desaturate=.2)

# polynomial using 'multinomial'
fig2, ax2 = plt.subplots(1, 1, figsize=(10,10))
make_scatter(ax2, X_train, y_train, color, title='Logistic Regression (Multinomia
overlay_decision_boundary(ax2, model=logreg_multi, desaturate=.2)

# main effects
fig3, ax3 = plt.subplots(1, 1, figsize=(10,10))
make_scatter(ax3, X_train, y_train, color, title='Logistic Regression (Main Effe
overlay_decision_boundary(ax3, model=logreg_main, desaturate=.2)
```




Logistic Regression (Multinomial)
Biomarker 2
Biomarker 1
4
2
0
−2
−4
−6
−8
−10
−12
−1
0
1
2
3
4
Normal
Hyperthyroidism
Hypothyroidism



2.2

Answer:

1. Yes, based on the true observations and the effects we are including in each of the model the decision boundaries are looking to be what we were expecting.
2. Quadratic and interaction terms make the decision boundaries curvilinear. The reason being that, logistic regression gives us the log-odds of the response being in a particular class. Formula for simple linear **log-odds** below:

$$ln\Big[\frac{P(Y=1)}{(1-P(Y=1)}\Big] = \beta_0 + \beta_1 . X$$

If we plot the sigmoid function for K=2 w.r.t. any given x, we will get the decision boundary at P(Y=1) = 0.5 which will fall on a straight line if our log-odds are a linear function of X.

However, if we include the polynomial and interaction terms, P(Y=1) = 0.5 will not occur on a straight line becasue the log-odds won't be linear.

3. Since the model with just the main effects can only make linear decision boundaries, it draws straight lines from the point where we see the three boundaries intersect, to minimize the overall mis-classification assuming P(Y=k) = 0.5 falls on a straight line. However, we can see that that leads to higher mis-classifications within Class 2 (blue) and 3 (green). The models incuding polynomial effects changes the direction of decision boundary for different values of the X's and gives us a better classification for classes with fewer samples. There is very little difference between `OVR` and `multinomial`. We can see that the decision boundary between Class 1 (red) and Class 2 (blue) is slightly towards the x-axis for `OVR` and in the opposite way for `mutlinomial`.

2.3

In [23]:
```
# your code here
cvscore_main = cross_val_score(logreg_main, X_train, y_train, cv=5)
cvscore_ovr = cross_val_score(logreg_ovr, X_train, y_train, cv=5)
cvscore_multi = cross_val_score(logreg_multi, X_train, y_train, cv=5)

cv_scores = {'Main Effects with OVR': cvscore_main,
             'Polynomial with OVR': cvscore_ovr,
             'Polynomial with Multinomial': cvscore_multi}
```

In [24]:
```
# your code here
cv_df = pd.DataFrame(cv_scores, index=np.arange(1,6))
avg = pd.DataFrame(np.array([np.mean(cvscore_main), np.mean(cvscore_ovr), np.mea
                   columns=cv_df.columns, index=['Mean'])
std = pd.DataFrame(np.array([np.std(cvscore_main), np.std(cvscore_ovr), np.std(c
                   columns=cv_df.columns, index=['Standard Deviation'])
pd.concat([cv_df, avg, std])
```

Out[24]:

	Main Effects with OVR	Polynomial with OVR	Polynomial with Multinomial
1	0.863636	0.909091	0.909091
2	0.863636	0.863636	0.863636
3	0.952381	0.904762	0.904762
4	0.952381	0.904762	0.904762
5	0.952381	0.952381	0.904762
Mean	0.916883	0.906926	0.897403
Standard Deviation	0.043476	0.028122	0.016966

Answer:

We get an overall better performance for the model with main effects, however, the standard deviation for that model is higher compared to that of the polynomial and interaction effects so we are less confident about this model's performance compared to the models with polynomial effects. Also, we saw in the above plots that the model with polynomial effects has a better performance

within classes with fewer samples. The inclusion of polynomial and interaction terms yeilds us a more consistent model without drastically affecting the overall performance compared to the one with main effects - thus we consider polynomial model to be better than the linear one.

Question 3 [20 pts]: Discriminant Analysis

3.1 Consider the following synthetic dataset with two classes. A green star marks a test observation; which class do you think it belongs to? How would LDA classify that observation? How would QDA? Explain your reasoning.

3.2 Now let's return to the thyroid dataset. Make a table of the total variance of each class for each biomarker.

3.3 Fit LDA and QDA on the thyroid data, and plot the decision boundaries. Comment on how the decision boundaries differ. How does the difference in decision boundaries relate to characteristics of the data, such as the table you computed above?

3.1

In [25]:
```
X_blobs, y_blobs = make_blobs(centers=[[0., 0.], [1., 0.]], cluster_std=[[.4, .1
plt.scatter(X_blobs[y_blobs==0][:,0], X_blobs[y_blobs==0][:,1], label="Class 0")
plt.scatter(X_blobs[y_blobs==1][:,0], X_blobs[y_blobs==1][:,1], label="Class 1")
plt.scatter([.75], [0.], color="green", marker="*", s=150, label="Test observati
plt.legend();
```

In [26]:
```
# Your code here
display(np.var(X_blobs[1]))
display(np.var(X_blobs[0]))
```

```
515.3740207601871
```

```
0.00014595923913538785
```

Answer:

LDA assumes that the variance (or covariances for multiple predictors) of predictor values is equal for all the classes, while QDA assumes predictors have different variances for each of the classes. We can see in the plot and the output above that the standard deviation for `Class 0` and `Class 1` is not the same. Also, the spread for `Class 0` is along x-axis and the spread for `Class 1` is along y-axis. Hence, LDA will most likely draw a decision boundary further to the left of 'green star' and classify it as `Class 1` as the decision boundary will be at the average of the means for two classes. QDA, however, will most likely classify the 'green star' as `Class 0` since it will calculate the standard deviations of X individually for each of the classes.

3.2

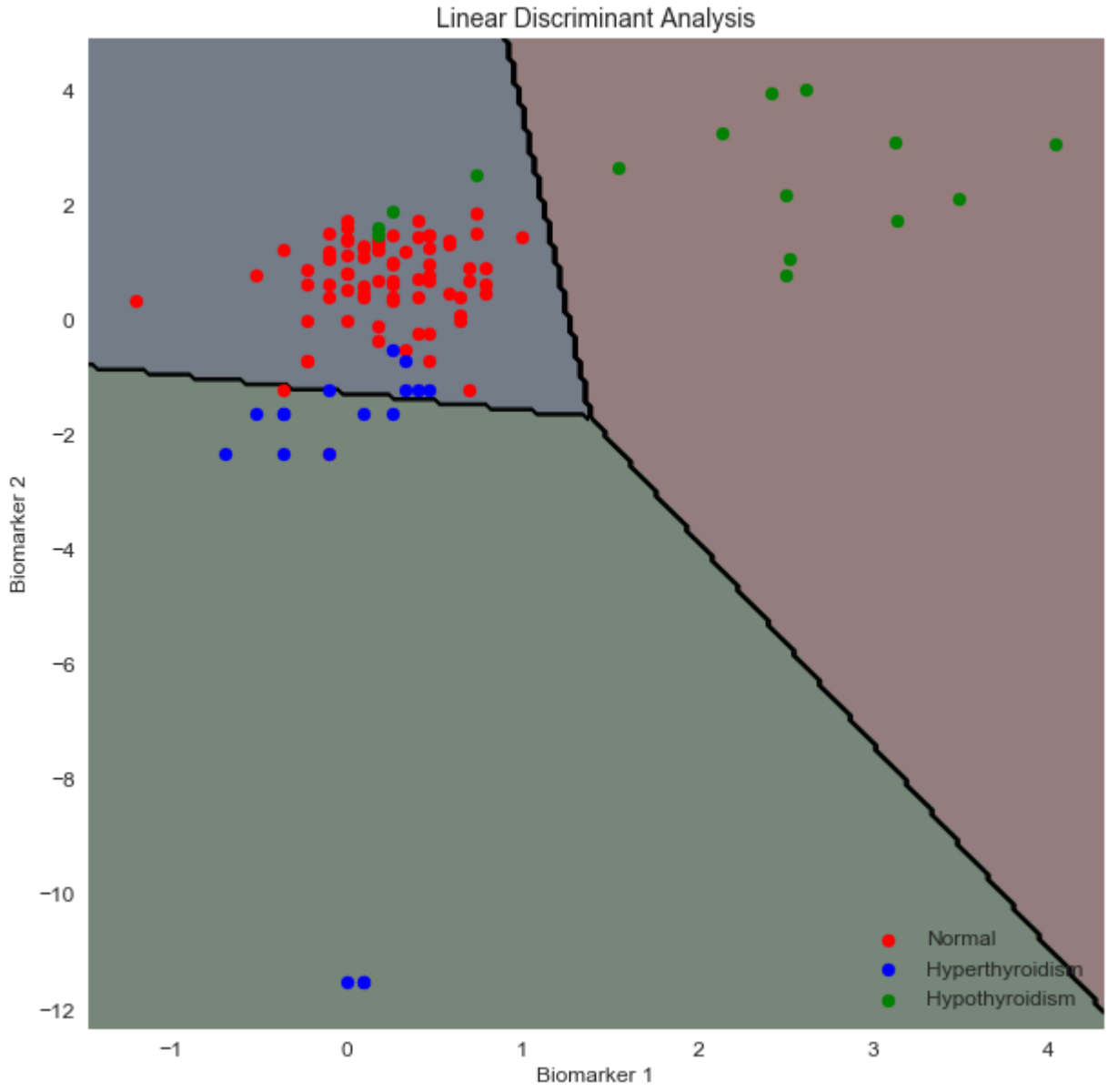
In [27]:
```
# your code here
variance = {}
for i in X_train.columns:
    var = []
    for j in y_train.sort_values(ascending=True).unique():
        var.append(np.var(X_train.loc[y_train==j, i]))
    variance['%s'%i] = var

pd.DataFrame(variance, index=y_train.sort_values(ascending=True).unique())
```

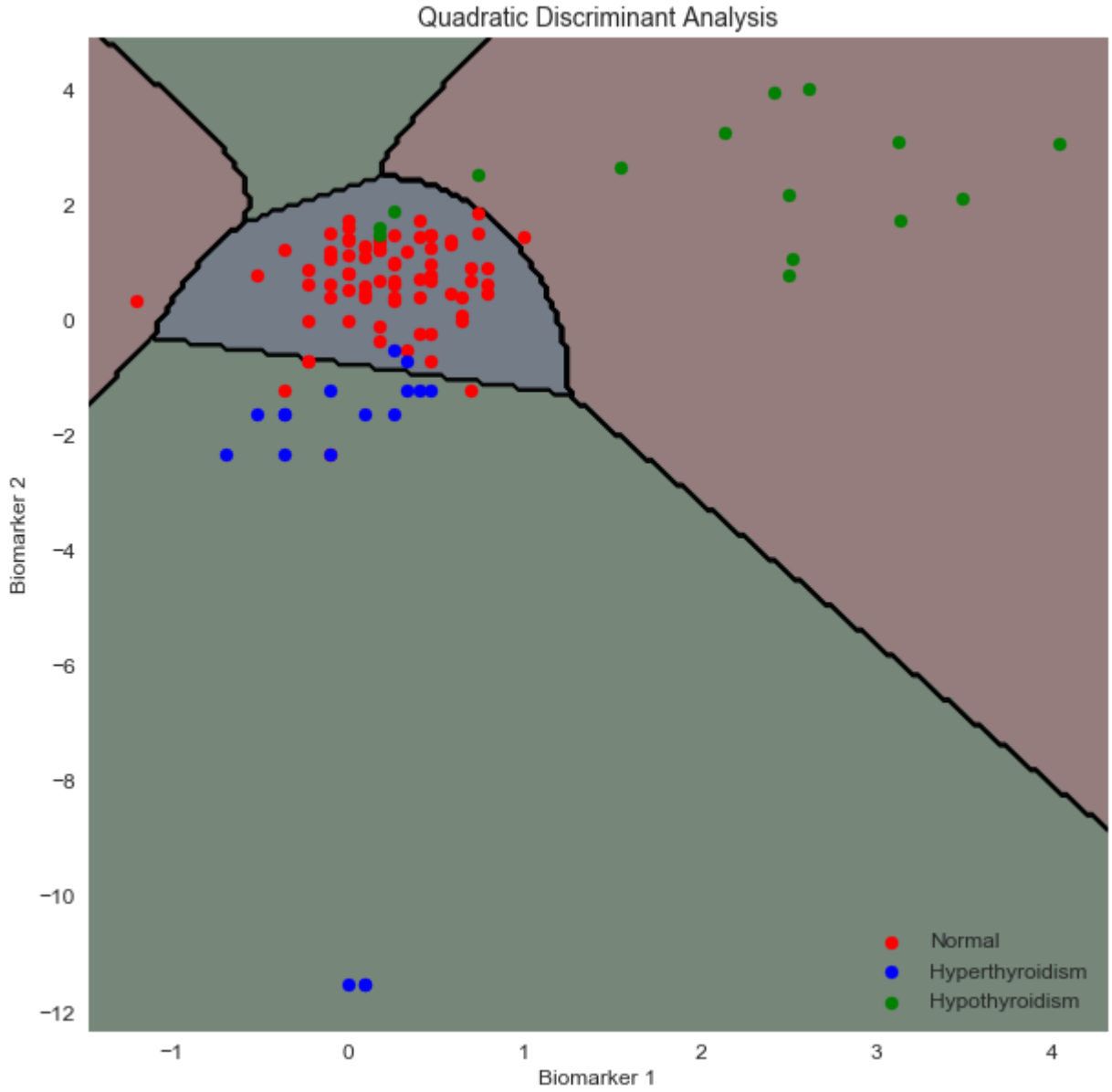
Out[27]:

	Biomarker 1	Biomarker 2
1	0.132315	0.598927
2	0.112996	14.809360
3	1.435367	0.899186

3.3

In [28]:
```
# your code here
lda_model = LinearDiscriminantAnalysis(priors=[1/3,1/3,1/3]).fit(X_train, y_trai
qda_model = QuadraticDiscriminantAnalysis(priors=[1/3,1/3,1/3]).fit(X_train, y_t

fig1, ax1 = plt.subplots(1, 1, figsize=(10,10))
make_scatter(ax1, X_train, y_train, color, title='Linear Discriminant Analysis')
overlay_decision_boundary(ax1, model=lda_model, desaturate=.2)

fig2, ax2 = plt.subplots(1, 1, figsize=(10,10))
make_scatter(ax2, X_train, y_train, color, title='Quadratic Discriminant Analysis
overlay_decision_boundary(ax2, model=qda_model, desaturate=.2)
```





Answer:

The decision boundaries are formed by intersecting distributions with same height (standard deviations) for `LDA` but different heights for `QDA` (if the standard deviations are different across the classes). So in cases where the standard deviation of the predictors are the same across classes, `LDA` and `QDA` will give us same decision boundaries. The fact that we get linear boundaries for `LDA` and curved for `QDA` means that the standard deviations are indeed different for our predictors across classes. We can attest this by looking at the scatter plot and the variance table in 3.2. Notice variances for predictors, specifically for `Biomarker 2` for `Class 2`.

Question 4 [20 pts]: Fit Decision Trees

We next try out decision trees for thyroid classification. For the following questions, you should use the *Gini* index as the splitting criterion while fitting the decision tree.

Hint: You should use the `DecisionTreeClassifier` class to fit a decision tree classifier and the `max_depth` attribute to set the tree depth.

4.1. Fit a decision tree model to the thyroid data set with (maximum) tree depths 2, 3, ..., 10. Make plots of the accuracy as a function of the maximum tree depth, on the training set and the mean score on the validation sets for 5-fold CV. Is there a depth at which the fitted decision tree model achieves near-perfect classification on the training set? If so, what can you say about how this tree will generalize? Which hyperparameter setting gives the best cross-validation performance?

4.2: Visualize the decision boundaries of the best decision tree you just fit. How are the shapes of the decision boundaries for this model different from the other methods we have seen so far? Given an explanation for your observation.

4.3 Explain *in words* how the best fitted model diagnoses 'hypothyroidism' for a new patient. You can use the code below to examine the structure of the best decision tree.

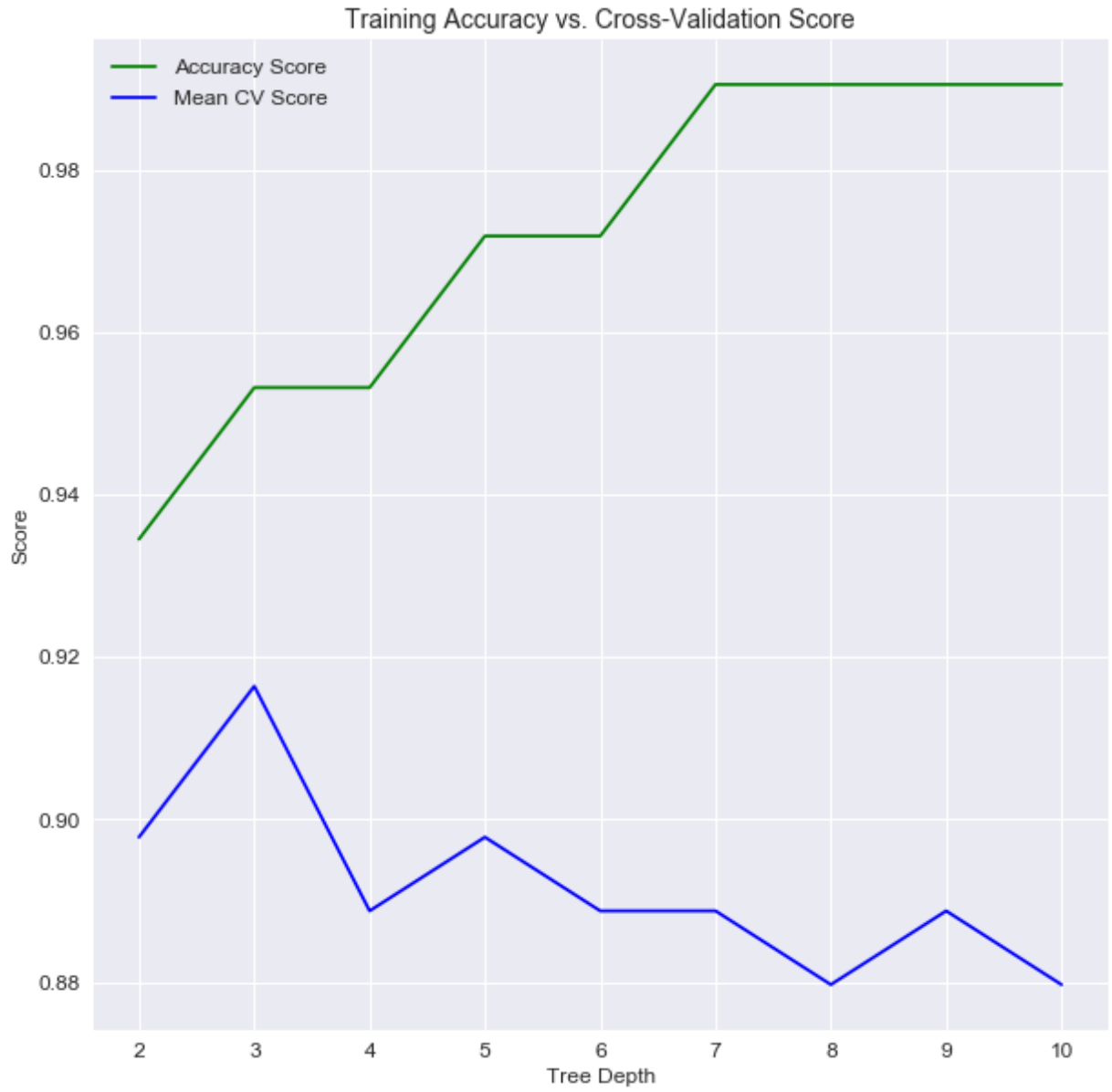
4.1

In [29]:
```
# your code here
tree_depth = np.arange(2,11)
tree_models = {}
tree_acc = {}
cv_tree_acc = {}
for i in tree_depth:
    tree_models['%s'%i] = DecisionTreeClassifier(max_depth=i).fit(X_train, y_tra
    tree_acc['%s'%i] = np.array([tree_models['%s'%i].score(X_train, y_train),
                                 np.mean(cross_val_score(tree_models['%s'%i], X_
acc_df = pd.DataFrame(tree_acc, index=['Accuracy Score', 'Mean CV Score']).T
display(acc_df)
```

	Accuracy Score	Mean CV Score
2	0.934579	0.897835
3	0.953271	0.916450
4	0.953271	0.888745
5	0.971963	0.897835
6	0.971963	0.888745
7	0.990654	0.888745
8	0.990654	0.879654
9	0.990654	0.888745
10	0.990654	0.879654

In [30]:
```
# your code here
fig, ax = plt.subplots(1,1, figsize = (10,10))
ax.plot(acc_df['Accuracy Score'], color='green', label='Accuracy Score')
ax.plot(acc_df['Mean CV Score'], color='blue', label='Mean CV Score')
ax.set_xlabel('Tree Depth', fontsize=12)
ax.set_ylabel('Score', fontsize=12)
ax.set_title('Training Accuracy vs. Cross-Validation Score', fontsize=14)
ax.tick_params(labelsize=12)
ax.legend(loc='best', fontsize=12)
```

Out[30]: <matplotlib.legend.Legend at 0x9357a54da0>



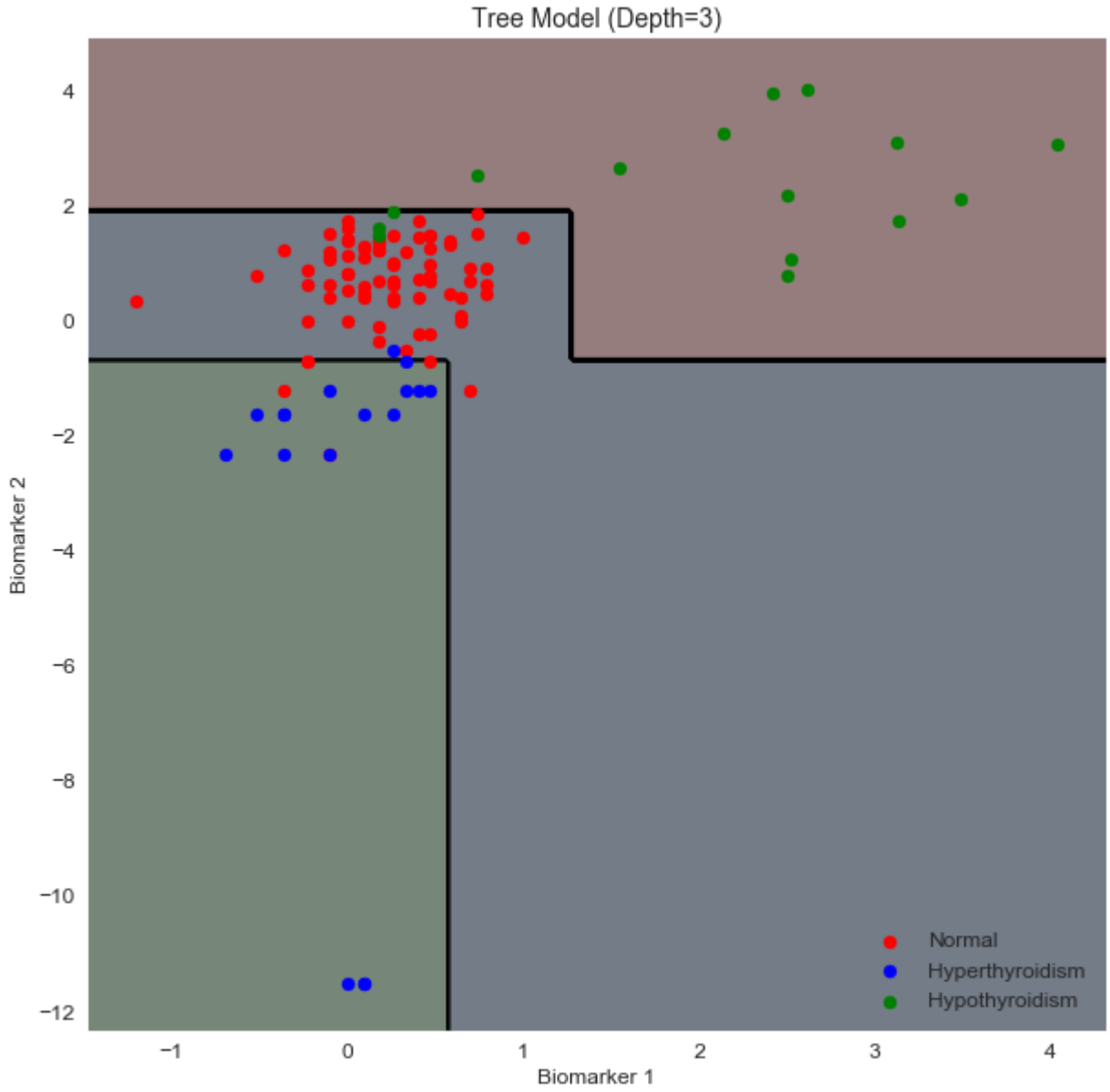
In [31]:
```
# your code here
tree_best = tree_models['3']
```

Answer:

Trees with the depth of 7 and above give us a near perfect accuracy score, however, their mean CV score is lower than those of other models with lower depth. Which means we are overfitting our training data if we go above the depth of 7 and our model perfomrance will be poor on gernalized/test data. Based on the table above we get the highest CV score from tree with a depth of 3, which means it is a better model overall to be generalized compared to other depths.

4.2

In [32]:
```
# your code here
fig2, ax2 = plt.subplots(1, 1, figsize=(10,10))
make_scatter(ax2, X_train, y_train, color, title='Tree Model (Depth=3)')
overlay_decision_boundary(ax2, model=tree_best, desaturate=.2)
```



Answer:

The decision boundaries for the tree models are always perpendicular to one of the dimensions (or predictor axis) in our feature set, while this was not the case for other classification models (`k-NN` , `Logistic` , `LDA` , `QDA`) we have seen so far. This is because of the difference in the approach for these model, while creating the boundaries. The other models create the boundaries along the combined effect of the entire feature set at each partition, while the tree models consider only the best feature (one along which we get the lowest error metric) at each partition.

4.3

Entirely optional note: You can also generate a visual representation using the `export_graphviz` . However, viewing the generated GraphViz file requires additional steps. One approach is to paste the generated graphviz file in the text box at http://www.webgraphviz.com/ (http://www.webgraphviz.com/). Alternatively, you can run GraphViz on your own computer, but you may need to install some additional software. Refer to the Decision Tree section of the sklearn user guide (http://scikit-learn.org/stable/modules/tree.html#classification) for more information.

In [33]:
```
# This code is adapted from
# http://scikit-learn.org/stable/auto_examples/tree/plot_unveil_tree_structure.ht
def show_tree_structure(clf):
    tree = clf.tree_

    n_nodes = tree.node_count
    children_left = tree.children_left
    children_right = tree.children_right
    feature = tree.feature
    threshold = tree.threshold

    # The tree structure can be traversed to compute various properties such
    # as the depth of each node and whether or not it is a leaf.
    node_depth = np.zeros(shape=n_nodes, dtype=np.int64)
    is_leaves = np.zeros(shape=n_nodes, dtype=bool)
    stack = [(0, -1)]  # seed is the root node id and its parent depth
    while len(stack) > 0:
        node_id, parent_depth = stack.pop()
        node_depth[node_id] = parent_depth + 1

        # If we have a test node
        if (children_left[node_id] != children_right[node_id]):
            stack.append((children_left[node_id], parent_depth + 1))
            stack.append((children_right[node_id], parent_depth + 1))
        else:
            is_leaves[node_id] = True

    print(f"The binary tree structure has {n_nodes} nodes:\n")

    for i in range(n_nodes):
        indent = node_depth[i] * "  "
        if is_leaves[i]:
            prediction = clf.classes_[np.argmax(tree.value[i])]
            print(f"{indent}node {i}: predict class {prediction}")
        else:
            print("{}node {}: if X[:, {}] <= {:.3f} then go to node {}, else go t
                indent, i, feature[i], threshold[i], children_left[i], children_
```

In [34]:
```
# your code here
show_tree_structure(tree_best)
```

```
The binary tree structure has 11 nodes:

node 0: if X[:, 1] <= -0.693 then go to node 1, else go to node 6
  node 1: if X[:, 0] <= 0.582 then go to node 2, else go to node 5
    node 2: if X[:, 0] <= -0.053 then go to node 3, else go to node 4
      node 3: predict class 2
      node 4: predict class 2
    node 5: predict class 1
  node 6: if X[:, 0] <= 1.270 then go to node 7, else go to node 10
    node 7: if X[:, 1] <= 1.879 then go to node 8, else go to node 9
      node 8: predict class 1
      node 9: predict class 3
    node 10: predict class 3
```

Answer:

Based on the tree structure, the best tree model predicts `Hypothyriodism` if the `Biomarker 2` is > 1.879 or if `Biomarker 2` > -0.693 and `Biomarker 1` > 1.27. We do, however, see some mis-classified observations with `Biomarker 2` < 1.879. Thus, if `Biomarker 2` is < 1.879 but very close to 1.879, the person might require additional tests to be sure of the true state.

Question 5 [18 pts]: k-NN and Model comparison

We have now seen six different ways of fitting a classification model: **linear logistic regression**, **logistic regression with polynomial terms**, **LDA**, **QDA**, **decision trees**, and in this problem we'll add **k-NN**. Which of these methods should we use in practice for this problem? To answer this question, we now compare and contrast these methods.

5.1 Fit a k-NN classifier with uniform weighting to the training set. Use 5-fold CV to pick the best k.

Hint: Use `KNeighborsClassifier` and `cross_val_score`.

5.2 Plot the decision boundaries for each of the following models that you fit above. For models with hyperparameters, use the values you chose using cross-validation.

- Logistic Regression (linear)
- Logistic Regression (polynomial)
- Linear Discriminant Analysis
- Quadratic Discriminant Analysis
- Decision Tree
- k-NN

Comment on the difference in the decision boundaries between the following pairs of models. Why does this difference make sense given how the model works?

- Linear logistic regression; LDA
- Quadratic logistic regression; QDA.
- k-NN and whichever other model has the most complex decision boundaries

5.3 Describe how each model classifies an observation from the test set in one short sentence for each (assume that the model is already fit). For example, for the linear regression classifier you critiqued in hw5, you might write: "It classifies the observation as class 1 if the dot product of the feature vector with the the model coefficients (with constant added) exceeds 0.5."

- Logistic Regression (One-vs-Rest)
- Linear Discriminant Analysis
- Quadratic Discriminant Analysis
- k-Nearest-Neighbors Classifier
- Decision Tree

5.4 Estimate the validation accuracy for each of the models. Summarize your results in a graph or table. (Note: for some models you have already run these computations; it's ok to redo them here if it makes your code cleaner.)

5.5 Based on everything you've found in this question so far, which model would you expect to perform best on our test data?

Now evaluate each fitted model's performance on the test set. Also, plot the same decision boundaries as above, but now showing the test set. How did the overall performance compare with your performance estimates above? Which model actually performed best? Why do you think this is the case?

5.6. Compare and contrast the six models based on each of the following criteria (a supporting table to summarize your thoughts can be helpful):

- Classification performance
- Complexity of decision boundary
- Memory storage
- Interpretability

If you were a clinician who had to use the classifier to diagnose thyroid disorders in patients, which among the six methods would you be most comfortable in using? Justify your choice in terms of at least 3 different aspects.

5.1

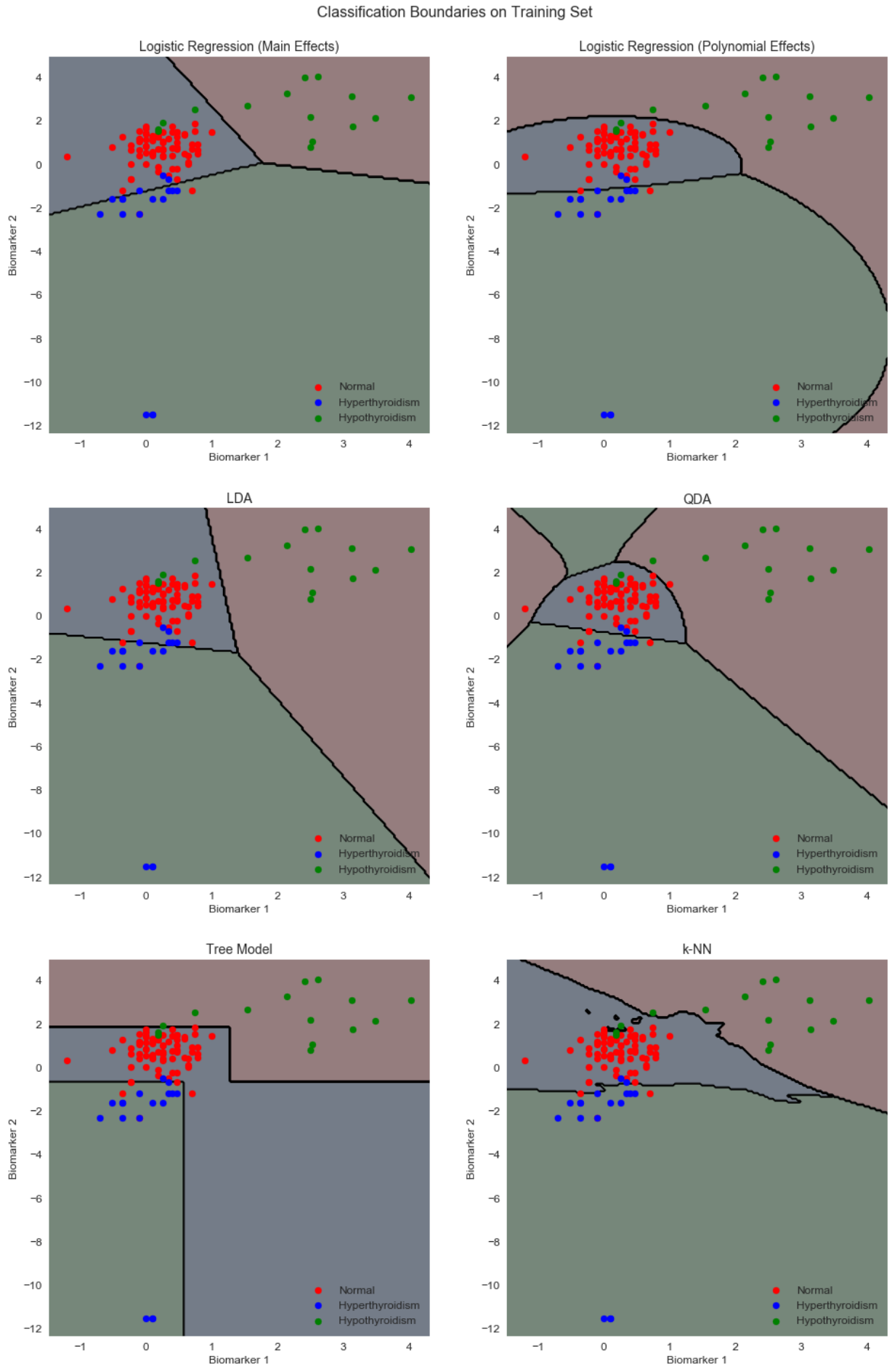
In [35]:
```
# your code here
k = [2, 5, 10, 15, 20, 50]
KNeighborsClassifier()
knn_models = {}
knn_acc = {}
for i in k:
    knn_models['%s'%i] = KNeighborsClassifier(n_neighbors=i).fit(X_train, y_trai
    knn_acc['%s'%i] = np.array([knn_models['%s'%i].score(X_train, y_train),
                                np.mean(cross_val_score(knn_models['%s'%i], X_t
acc_df = pd.DataFrame(knn_acc, index=['Accuracy Score', 'Mean CV Score']).T
display(acc_df)
knn_best = knn_models['5']
```

	Accuracy Score	Mean CV Score
2	0.925234	0.869697
5	0.925234	0.916450
10	0.897196	0.888312
15	0.887850	0.897835
20	0.897196	0.822511
50	0.700935	0.701299

5.2

In [36]:
```
# Your code here
models = {'Logistic Regression (Main Effects)': logreg_main, 'Logistic Regressio
          'LDA': lda_model, 'QDA': qda_model, 'Tree Model': tree_best, 'k-NN': k
color = ['red', 'blue', 'green']
classes = ['Normal', 'Hyperthyroidism','Hypothyroidism']

figure, axes = plt.subplots(3, 2, figsize=(16,25))
i=0
j=0
k=0
for model in models:
    #axes[i,j].set_xlim(left=np.min(X_train['Biomarker 1'])-1, right=np.max(X_tr
    #axes[i,j].set_ylim(bottom=np.min(X_train['Biomarker 2'])-1, top=np.max(X_tr
    make_scatter(axes[i,j], X_train, y_train, color, title=model)
    overlay_decision_boundary(axes[i,j], model=models[model], desaturate=.2)
    if k in [1,3,5]:
        i=i+1
        j=0
    else:
        j=j+1
    k=k+1
figure.suptitle('Classification Boundaries on Training Set', y=0.91, fontsize=16
```

Out[36]:
```
Text(0.5,0.91,'Classification Boundaries on Training Set')
```


Classification Boundaries on Training Set
Logistic Regression (Main Effects)
Logistic Regression (Polynomial Effects)
LDA
QDA
Tree Model
k-NN
Biomarker 1
Biomarker 2
Normal
Hyperthyroidism
Hypothyroidism

Answer:

1. `Linear Logistic Regression` and `LDA` have similar looking decision bounadries, but the decision boundary for `LDA` seems to be rotated by certain degree in clockwise direction. This is mainly because the `Linear Logistic Regression` models the `log-odds` as a linear function of X's where the βs are calculated using the eucledian distances. So it somewhat accounts for the differences in the standard deviations of the X's in a linear fashion. That is why we see such a huge area for Class 2, as the two points at the bottom of the plot with very low values of `Biomarker 2` are adding to the distances. `LDA` , however, assumes equal standard deviations so the decision boundary between Class2 and Class 3 is where the means of the two respective distributions are equidistant from the boundary. Class 1 has relative less spread across `Biomarker 1` and `Biomarker 2` and its mean is relatively closer to other two classes, so its decision boundaries with other two classes are at an acute angle with each other.
2. `Quadratic Logistic Regression` gives us three distinct regions with the boundaries being curved while the regions in `QDA` appear more like patches (especially the "island" for Class 1). `Quadratic Logistic Regression` again models the log-odds, but now its modeling it as a quadratic function of the X's. `QDA` models the distribution of X's and calculates the standard deviation individually for each class. Since Class 1 has relatively small standard deviation along both the predictors, it is contained in its own region.
3. Compraing `k-NN` and `tree` models. The boundaries for `k-NN` follow no specific pattern while those for `tree` are parallel to either one of the axes. This is because k-NN is looking at the points in the neighborhood of specified size in its vicinity and predicting the class with highest occurence, while the tree model considers only the best variable (that gives the lowest error rate) at each partitioning step.

5.3

Answer:

1. `Logistic Regression (One-vs-Rest)` : Calculates log-odds for each of the K classes as a function of X's and classifies an observation to a particular class for which the log-odds are highest.
2. `Linear Discriminant Analysis` : Models distribution of X's assuming same standard deviation for each of the K classes and assigns the class that has the higest probability density for the given values of X's..
3. `Quadratic Discriminant Analysis` : Models distribution of X's assuming different standard deviations for each of the K classes and assigns the class that has the higest probability density for the given values of X's.
4. `k-Nearest-Neighbors Classifier` : Looks at the neighborhood of specified size and assigns class that has the highest occurence in that neighborhood.
5. `Decision Tree` : Creates a flow chart logic by partitioning our data along one variable at a time and classifies an observation to a class to which it falls closest to in that tree structure.

5.4

In [37]:
```
# your code here
acc_train = {}

for i in models:
    acc_train['%s'%i] = np.array([models[i].score(X_train, y_train),
                                  np.mean(cross_val_score(models[i], X_train, y_t
acc_train_df = pd.DataFrame(acc_train, index=['Accuracy Score', 'Mean CV Score']
display(acc_train_df)
```

	Accuracy Score	Mean CV Score
Logistic Regression (Main Effects)	0.915888	0.916883
Logistic Regression (Polynomial Effects)	0.934579	0.906926
LDA	0.887850	0.906494
QDA	0.887850	0.888312
Tree Model	0.953271	0.916450
k-NN	0.925234	0.916450

Answer:

We can see that the `Linear Logistic regresion` , `Tree Model` and `k-NN` give us the highest CV scores.

5.5

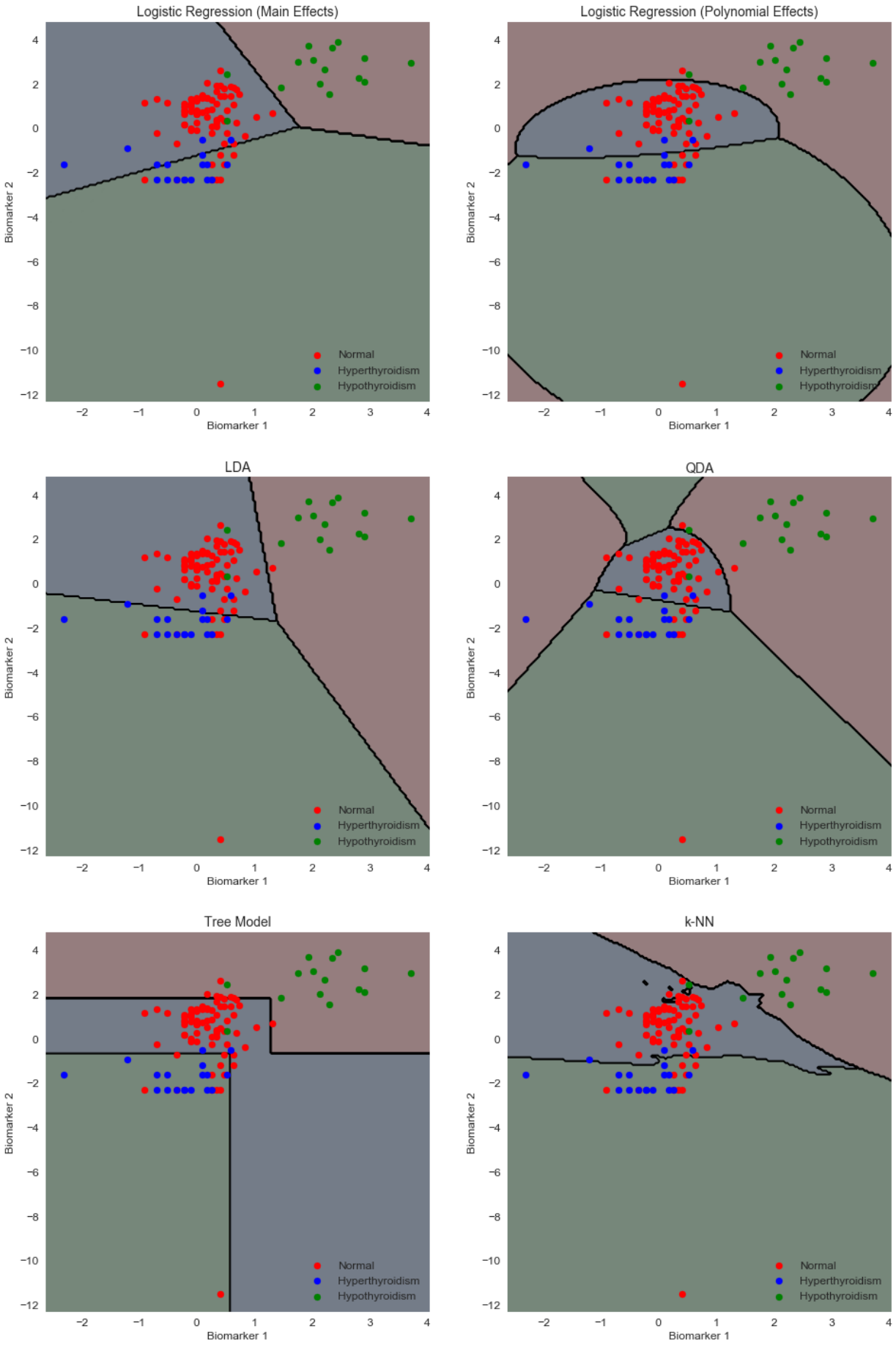
In [38]:
```
# your code here
acc_test = {}

for i in models:
    acc_test['%s'%i] = models[i].score(X_test, y_test)
acc_test_df = pd.DataFrame(acc_test, index=['Test Accuracy Score']).T
display(acc_test_df)
```

	Test Accuracy Score
Logistic Regression (Main Effects)	0.842593
Logistic Regression (Polynomial Effects)	0.861111
LDA	0.870370
QDA	0.833333
Tree Model	0.833333
k-NN	0.851852

In [39]:
```
# your code here
models = {'Logistic Regression (Main Effects)': logreg_main, 'Logistic Regressio
          'LDA': lda_model, 'QDA': qda_model, 'Tree Model': tree_best, 'k-NN': k
color = ['red', 'blue', 'green']
classes = ['Normal', 'Hyperthyroidism','Hypothyroidism']

figure, axes = plt.subplots(3, 2, figsize=(16,25))
i=0
j=0
k=0
for model in models:
    #axes[i,j].set_xlim(left=np.min(X_train['Biomarker 1'])-1, right=np.max(X_tr
    #axes[i,j].set_ylim(bottom=np.min(X_train['Biomarker 2'])-1, top=np.max(X_tr
    make_scatter(axes[i,j], X_test, y_test, color, title=model)
    overlay_decision_boundary(axes[i,j], model=models[model], desaturate=.2)
    if k in [1,3,5]:
        i=i+1
        j=0
    else:
        j=j+1
    k=k+1
figure.suptitle('Classification Boundaries on Test Set', y=0.91, fontsize=16)
```

Out[39]:
```
Text(0.5,0.91,'Classification Boundaries on Test Set')
```

Classification Boundaries on Test Set



Answer:

Based on the findings upto 5.4, we expect the `Logistic Regression (Main Effects)`, `Tree` and `k-NN` to perform the best as they have the highest CV scores.

However, after we evaluate the models on test set, we find that `LDA` outperforms all the other models while `Logistic Regression (Polynomial Effects)` being the second best. `QDA` and `Tree` perform worse than any other models. One of the reasons this could be the case is because we can see that the standard deviations in the test set are not actually the same as the train set, especially for Class 1 and Class 2 where we can see some sort of overlap in the true values. Thus, most of the complex models were probably on the training set.

5.6

5.6. Compare and contrast the six models based on each of the following criteria (a supporting table to summarize your thoughts can be helpful):

- Classification performance
- Complexity of decision boundary
- Memory storage
- Interpretability

If you were a clinician who had to use the classifier to diagnose thyroid disorders in patients, which among the six methods would you be most comfortable in using? Justify your choice in terms of at least 3 different aspects.

Answer:

1. Classification performance: The classification performance of the models on the test set in the descneding order is as follows: `LDA`, `Logistic Regression (Polynomial Effects)`, `k-NN`, `Logistic Regression (Main Effects)` and `Decision Tree` and `QDA`.
2. Complexity of decision boundary: `k-NN` and `QDA` have relatively the most complex decision boundaries since they are both non-linear, then comes `Logistic Regression (Polynomial Effects)` and then the remaining fall somewhat in the same category. However, `Decision Tree` have the least complex boundaries since we have limited the depth to 3.
3. Memory storage: `LDA` uses the least memory since there are far less number of calculations that are needed to be stored in this algorithm compared to others, then comes `QDA`. `Logistic Regression (Main Effects)` and `Logistic Regression (Polynomial Effects)` come after that in that order - polynomial will take less time though since it will converge faster to find the minimum loss compared to linear. Memory usage for `k-NN` and `Decision Tree` depend on the hyper parameters, but in general are the most expensive than others.
4. Interpretability: `Logistic Regression (Main Effects)` is the most interpretable model since it gives us log-odds in terms of linear function of the X's. Then comes `Logistic Regression (Polynomial Effects)`, `LDA` and `QDA`. `Decision Tree` and `k-NN` are least interpretable.

For the given dataset and based on the findings, `LDA` gives us the highest accuracy on the test data. Also, its the least expensive in terms of the memory usage and has a moderately complex decision boundary and interpretability. Hence, we would be most comfortable with `LDA` and that would be the first choice.

Question 6: [2 pts] Including an 'abstain' option

**Note this question is only worth 2 pts. **

One of the reasons a hospital might be hesitant to use your thyroid classification model is that a misdiagnosis by the model on a patient can sometimes prove to be very costly (e.g. if the patient were to file a law suit seeking a compensation for damages). One way to mitigate this concern is to allow the model to 'abstain' from making a prediction: whenever it is uncertain about the diagnosis for a patient. However, when the model abstains from making a prediction, the hospital will have to forward the patient to a thyroid specialist (i.e. an endocrinologist), which would incur additional cost. How could one design a thyroid classification model with an abstain option, such that the cost to the hospital is minimized?

Hint: Think of ways to build on top of the logistic regression model and have it abstain on patients who are difficult to classify.

6.1 More specifically, suppose the cost incurred by a hospital when a model mis-predicts on a patient is $5000, and the cost incurred when the model abstains from making a prediction is 1000$. What is the average cost per patient for the OvR logistic regression model (without quadratic or interaction terms) from Question 2? Note that this needs to be evaluated on the patients in the test set.

6.2 Design a classification strategy (into the 3 groups plus the *abstain* group) that has as low cost as possible per patient (certainly lower cost per patient than the logistic regression model). Give a justification for your approach.

6.1

In [40]:
```
# your code here
pred = pd.DataFrame(logreg_main.predict(X_test), columns=['y_pred'], index=X_tes
misclassify = pred.shape[0] - accuracy_score(y_test, pred, normalize=False)
cost_noabstain = misclassify * 5000 / pred.shape[0]
print('Average Cost per Person without Abstaining: %s'%cost_noabstain)
```

```
Average Cost per Person without Abstaining: 787.0370370370371
```

In [41]:
```
# your code here
prob = pd.DataFrame(logreg_main.predict_proba(X_test), columns=logreg_main.classe
pred_vs_true = pd.concat([prob, pred, y_test], axis=1)

#initialize to 0 and update based on the condition
pred_vs_true['Abstain'] = 0
mask = (pred_vs_true[1]<0.6) & (pred_vs_true[2]<0.5) & (pred_vs_true[3]<0.5)
pred_vs_true.loc[mask, 'Abstain'] = 1

pred_vs_true['Cost'] = 0
mask_abstain = pred_vs_true['Abstain']==1
pred_vs_true.loc[mask_abstain, 'Cost'] = 1000
mask_misclass = (pred_vs_true['Abstain']==0) & (pred_vs_true['y_pred']!=pred_vs_t
pred_vs_true.loc[mask_misclass, 'Cost'] = 5000

display(pred_vs_true.head(5))

cost_abstain = np.sum(pred_vs_true['Cost'])/pred_vs_true.shape[0]
print('Average Cost per Person with Abstaining: %s'%cost_abstain)
```

	1	2	3	y_pred	Diagnosis	Abstain	Cost
208	0.394350	0.605648	0.000002	2	2	0	0
97	0.912484	0.087265	0.000251	1	1	0	0
211	0.768323	0.000451	0.231226	1	1	0	0
24	0.498639	0.501352	0.000009	2	2	0	0
99	0.374726	0.625031	0.000243	2	1	0	5000

```
Average Cost per Person with Abstaining: 694.4444444444445
```

Answer:

To create a classification strategy we need to determine how confident are we in the predictions we make. Hence, we are going to create a dataframe that gives us probabilities for each of the classes.

6.2

Answer:

Since we used OVR for logisitic regression with main effects, we will get probabilities that don't add up to 1. So we will need to set a higher threshold compared to if we used `multinomial` . Also, since Class 1 has more prevalence and is well identifiable compared to other classes we establish a higher threshold for this class compared to others.

Classification Strategy: If the predicted probabilities for Class 1 < 0.6, for Class 2 < 0.5 and for Class 5 < 0.5 then we abstain from predicting. With this logic we get the average cost per person $= 694.44$.

In [ ]: