S-109A Introduction to Data Science:

Homework 5: Logistic Regression, High Dimensionality and PCA, LDA/QDA

Harvard University
Summer 2018
Instructors: Pavlos Protopapas, Kevin Rader

INSTRUCTIONS

- To submit your assignment follow the instructions given in canvas.
- Restart the kernel and run the whole notebook again before you submit.
- If you submit individually and you have worked with someone, please include the name of your [one] partner below.

Names of people you have worked with goes here:

Nikita Roy

In [1]:
```
import numpy as np
import pandas as pd

import statsmodels.api as sm
from statsmodels.api import OLS

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 cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold

import math
from scipy.special import gamma

import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline

import seaborn as sns
sns.set()

alpha = 0.5
```

Cancer Classification from Gene Expressions

In this problem, we will build a classification model to distinguish between two related classes of cancer, acute lymphoblastic leukemia (ALL) and acute myeloid leukemia (AML), using gene expression measurements. The data set is provided in the file `dataset_hw5_1.csv`. Each row in this file corresponds to a tumor tissue sample from a patient with one of the two forms of Leukemia. The first column contains the cancer type, with 0 indicating the ALL class and 1 indicating the AML class. Columns 2-7130 contain expression levels of 7129 genes recorded from each tissue sample.

In the following questions, we will use linear and logistic regression to build a classification models for this data set. We will also use Principal Components Analysis (PCA) to visualize the data and to reduce its dimensions.

Question 1: Data Exploration

1. First step is to split the observations into an approximate 50-50 train-test split. Below is some code to do this for you (we want to make sure everyone has the same splits).
2. Take a peek at your training set: you should notice the severe differences in the measurements from one gene to the next (some are negative, some hover around zero, and some are well into the thousands). To account for these differences in scale and variability, normalize each predictor to vary between 0 and 1.
3. Notice that the results training set contains more predictors than observations. Do you foresee a problem in fitting a classification model to such a data set?
4. Lets explore a few of the genes and see how well they discriminate between cancer classes. Create a single figure with four subplots arranged in a 2x2 grid. Consider the following four genes: `D29963_at`, `M23161_at`, `hum_alu_at`, and `AFFX-PheX-5_at`. For each gene overlay two histograms of the gene expression values on one of the subplots, one histogram for each cancer type. Does it appear that any of these genes discriminate between the two classes well? How are you able to tell?
5. Since our data has dimensions that are not easily visualizable, we want to reduce the dimensionality of the data to make it easier to visualize. Using PCA, find the top two principal components for the gene expression data. Generate a scatter plot using these principal components, highlighting the two cancer types in different colors. How well do the top two principal components discriminate between the two classes? How much of the variance within the data do these two principal components explain?

Answers:

1.1: First step is to split the observations into an approximate 50-50 train-test split. Below is some code to do this for you (we want to make sure everyone has the same splits).

In [2]:
```
np.random.seed(9002)
df = pd.read_csv('data/dataset_hw5_1.csv')
msk = np.random.rand(len(df)) < 0.5
data_train = df[msk]
data_test = df[~msk]

# Create train and test prediction and response datasets
X_train = data_train.copy().drop(["Cancer_type"], axis = 1)
y_train = data_train.copy()["Cancer_type"]
X_test = data_test.copy().drop(["Cancer_type"], axis = 1)
y_test = data_test.copy()["Cancer_type"]
```

In [3]:
```
data_train.head()
```

Out[3]:

	Cancer_type	AFFX-BioB-5_at	AFFX-BioB-M_at	AFFX-BioB-3_at	AFFX-BioC-5_at	AFFX-BioC-3_at	AFFX-BioDn-5_at	AFFX-BioDn-3_at	AFFX-CreX-5_at	AFFX-CreX-3_at	...
0	0	-214	-153	-58	88	-295	-558	199	-176	252	...
2	0	-106	-125	-76	168	-230	-284	4	-122	70	...
5	0	-67	-93	84	25	-179	-323	-135	-127	-2	...
9	0	-476	-213	-18	301	-403	-394	-42	-144	98	...
10	0	-81	-150	-119	78	-152	-340	-36	-141	96	...

5 rows × 7130 columns

1.2: Take a peek at your training set: you should notice the severe differences in the measurements from one gene to the next (some are negative, some hover around zero, and some are well into the thousands). To account for these differences in scale and variability, normalize each predictor to vary between 0 and 1.

In [4]:
```
# Look at training set
data_train.head()

# Create a normalize function
def normalize(df: pd.DataFrame) -> pd.DataFrame:
    """ Normalizes all columns in a DataFrame to vary between 0 and 1.

    Params:
        df -> the pandas DataFrame to be normalized.

    Returns:
        returns the normalized DataFrame.
        """
    # Create a copy of the dataframe
    df_normal = df.copy()

    return (df_normal - df_normal.min()) / (df_normal.max() - df_normal.min())

# Normalize the predictors between 0 and 1
X_train_normal = X_train.copy()
X_train_normal = normalize(X_train_normal)

X_train_normal.head()
```

Out[4]:

	AFFX-BioB-5_at	AFFX-BioB-M_at	AFFX-BioB-3_at	AFFX-BioC-5_at	AFFX-BioC-3_at	AFFX-BioDn-5_at	AFFX-BioDn-3_at	AFFX-CreX-5_at	AI C
0	0.466192	0.739726	0.255814	0.246154	0.433190	0.240418	0.880427	0.625850	0.928
2	0.658363	0.794521	0.213953	0.421978	0.573276	0.717770	0.741637	0.748299	0.505
5	0.727758	0.857143	0.586047	0.107692	0.683190	0.649826	0.642705	0.736961	0.338
9	0.000000	0.622309	0.348837	0.714286	0.200431	0.526132	0.708897	0.698413	0.570
10	0.702847	0.745597	0.113953	0.224176	0.741379	0.620209	0.713167	0.705215	0.566

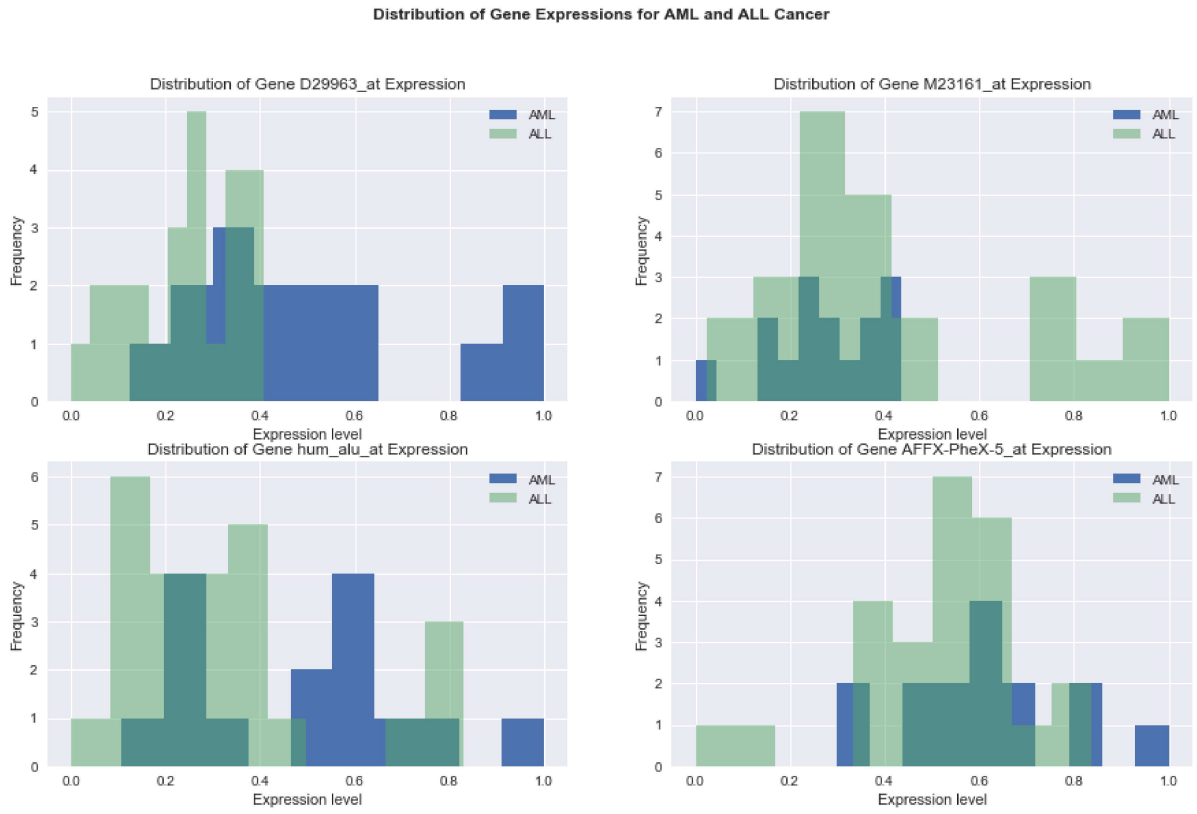
5 rows × 7129 columns

1.3: Notice that the results training set contains more predictors than observations. Do you foresee a problem in fitting a classification model to such a data set?

With more predictors than observations, the classification model will be overfitted to the dataset. One predictor can account for each observation and lead to a training R^2 of 1, incorrectly fitting to error epsilon.

1.4: Lets explore a few of the genes and see how well they discriminate between cancer classes. Create a single figure with four subplots arranged in a 2x2 grid. Consider the following four genes: `D29963_at`, `M23161_at`, `hum_alu_at`, and `AFFX-PheX-5_at`. For each gene overlay two histograms of the gene expression values on one of the subplots, one histogram for each cancer type. Does it appear that any of these genes discriminate between the two classes well? How are you able to tell?

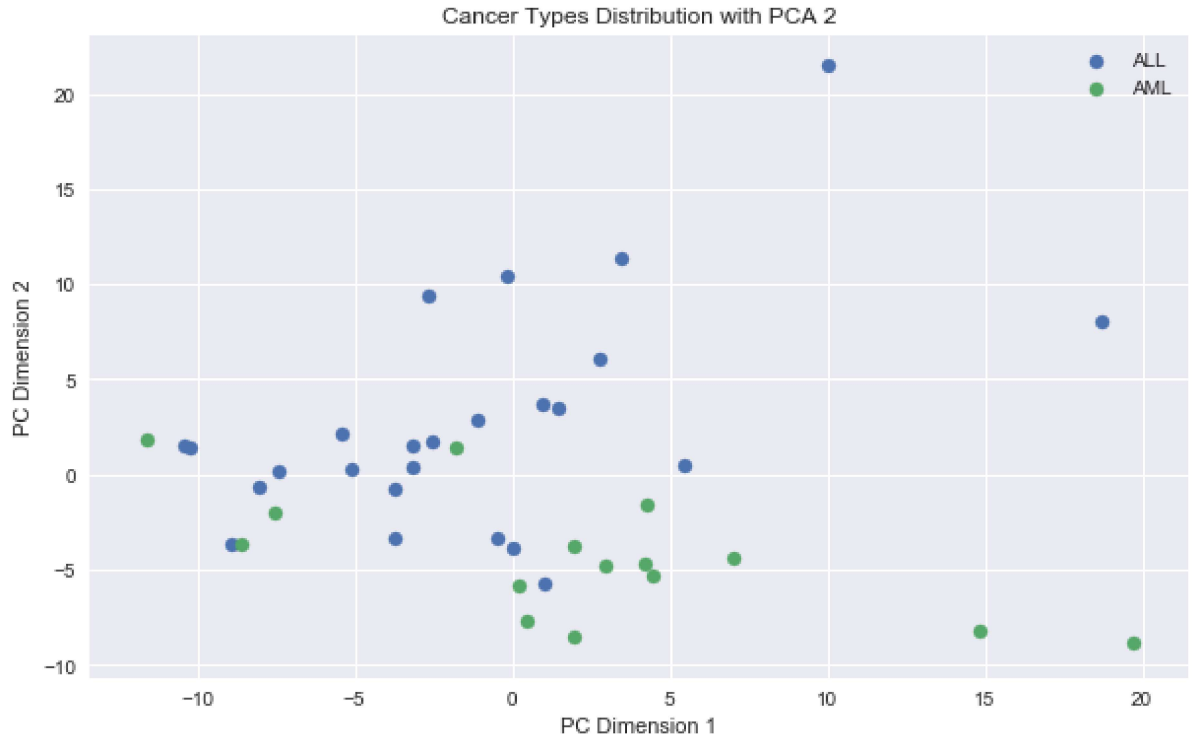
In [5]:
```
# initialize figure
fig, ax = plt.subplots(2,2, figsize=(15,9))

# For each gene create a subplot
for index, gene in enumerate(["D29963_at", "M23161_at", "hum_alu_at", "AFFX-PheX-5_at"]):
    # Initialize row, col
    row= int(index / 2)
    col = index % 2

    # Plot the histogram of the gene expression values on AML cancer type
    ax[int(index / 2)][index % 2].hist(X_train_normal[y_train == 1][gene], label="AML")

    # Plot the histogram of the gene expression values on ALL cancer type
    ax[int(index / 2)][index % 2].hist(X_train_normal[y_train == 0][gene], alpha=0.5, label="ALL")

    # Labels
    ax[row][col].legend()
    ax[row][col].set_title("Distribution of Gene {0} Expression".format(gene))
    ax[row][col].set_xlabel("Expression level")
    ax[row][col].set_ylabel("Frequency")

# Set suptitle
plt.suptitle("Distribution of Gene Expressions for AML and ALL Cancer", fontweight="bold");
```



Gene D22963_at and hum_alu_at discriminates between the two types of cancer the best, as the medians of gene expression appear to differ significantly between the two types of cancer, and there is less overlap between the two distributions. For the other two genes, the centers and spread for the two types of cancer are both similar.

1.5: Since our data has dimensions that are not easily visualizable, we want to reduce the dimensionality of the data to make it easier to visualize. Using PCA, find the top two principal components for the gene expression data. Generate a scatter plot using these principal components, highlighting the two cancer types in different colors. How well do the top two principal components discriminate between the two classes? How much of the variance within the data do these two principal components explain?

In [6]:
```
# Initialize normalized test data
X_test_normal = normalize(X_test)

# Find the top two principal components for the training data
pca_transformer = PCA(2).fit(X_train_normal)
X_train_2d = pca_transformer.transform(X_train_normal)

# Initialize figure and subplots
fig, ax = plt.subplots(1,1, figsize=(10,6))

# Plot the data for each type of cancer
for index, type_Cancer in enumerate(["ALL","AML"]):
    cur_df = X_train_2d[y_train==index]
    ax.scatter(cur_df[:,0], cur_df[:,1], label = type_Cancer)

# Add Labels
ax.set_ylabel("PC Dimension 2")
ax.set_xlabel("PC Dimension 1")
ax.set_title("Cancer Types Distribution with PCA 2")
ax.legend();
```



In [7]:
```
# Store explained variance
var_explained = pca_transformer.explained_variance_ratio_

# Print results
print("\033[1mVariance explained by each PCA component:\033[0m", var_explained
)
print("\033[1mTotal Variance Explained:\033[0m", np.sum(var_explained))
print("\nBy visual demonstration, the top two principal components differentia
te between the cancer types to a significant degree.\nALL resides in the upper
 portion of the plot, while AML resides more in the lower portion.")
```

```
Variance explained by each PCA component: [0.15889035 0.11428795]
Total Variance Explained: 0.2731782945208865

By visual demonstration, the top two principal components differentiate betwe
en the cancer types to a significant degree.
ALL resides in the upper portion of the plot, while AML resides more in the l
ower portion.
```

Question 2: Linear Regression vs. Logistic Regression

In class we discussed how to use both linear regression and logistic regression for classification. For this question, you will work with a single gene predictor, `D29963_at`, to explore these two methods.

1. Fit a simple linear regression model to the training set using the single gene predictor `D29963_at`. We could interpret the scores predicted by the regression model interpreted for a patient as an estimate of the probability that the patient has `Cancer_type`=1. Is there a problem with this interpretation?
2. The fitted linear regression model can be converted to a classification model (i.e. a model that predicts one of two binary labels 0 or 1) by classifying patients with predicted score greater than 0.5 into `Cancer_type`=1, and the others into the `Cancer_type`=0. Evaluate the classification accuracy (1 - misclassification rate) of the obtained classification model on both the training and test sets.
3. Next, fit a simple logistic regression model to the training set. How do the training and test classification accuracies of this model compare with the linear regression model? Remember, you need to set the regularization parameter for sklearn's logistic regression function to be a very large value in order to not regularize (use 'C=100000').
4. Plot the quantitative output from the linear regression model and the probabilistic output from the logistic regression model (on the training set points) as a function of the gene predictor. Also, display the true binary response for the training set points in the same plot. Based on these plots, does one of the models appear better suited for binary classification than the other? Explain.

Answers:

2.1: Fit a simple linear regression model to the training set using the single gene predictor `D29963_at`. We could interpret the scores predicted by the regression model interpreted for a patient as an estimate of the probability that the patient has `Cancer_type`=1. Is there a problem with this interpretation?

In [8]:
```
# Fit a simple linear model to the training data
simple_linear_model = OLS(y_train, sm.add_constant(X_train_normal['D29963_at'
]))
fitted_model = simple_linear_model.fit()

# Store the predictions
predictions_test = fitted_model.predict(sm.add_constant(X_test_normal['D29963_
at']))
```

There is definitely a problem with this interpretation. The Linear Model predictions are comtinious and numerical with no bound, so there are negative predictions and predictions greater than one. The values are not probabilities.

2.2: The fitted linear regression model can be converted to a classification model (i.e. a model that predicts one of two binary labels 0 or 1) by classifying patients with predicted score greater than 0.5 into `Cancer_type=1`, and the others into the `Cancer_type=0`. Evaluate the classification accuracy (1 - misclassification rate) of the obtained classification model on both the training and test sets.

In [9]:
```
# Calculate Predictions
predictions_train = fitted_model.predict(sm.add_constant(X_train_normal['D2996
3_at']))
sm_binary_predictions_train = predictions_train >= .5
# Note: test predictions have already been stored
sm_binary_predictions_test  = predictions_test >= .5

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0}".format(accuracy_score(y_train, sm_binary_p
redictions_train)))
print("\033[1mTest:\033[0m {0: .4f}".format(accuracy_score(y_test, sm_binary_p
redictions_test)))
```

```
Accuracy Scores
Training: 0.8
Test:  0.7879
```

2.3 Next, fit a simple logistic regression model to the training set. How do the training and test classification accuracies of this model compare with the linear regression model? Remember, you need to set the regularization parameter for sklearn's logistic regression function to be a very large value in order to not regularize (use 'C=100000').

In [10]:
```
# Fit the Logistic Regression Model to the training set
fitted_lr = LogisticRegression(C=100000, solver='newton-cg', max_iter=250).fit
(X_train_normal['D29963_at'].values.reshape(-1,1),y_train)

# Store the predictions
lr_predictions_train = fitted_lr.predict(X_train_normal['D29963_at'].values.re
shape(-1,1))
lr_predictions_test = fitted_lr.predict(X_test_normal['D29963_at'].values.resh
ape(-1,1))

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(accuracy_score(y_train, lr_pre
dictions_train)))
print("\033[1mTest:\033[0m {0: .3f}".format(accuracy_score(y_test, lr_predicti
ons_test)))
```

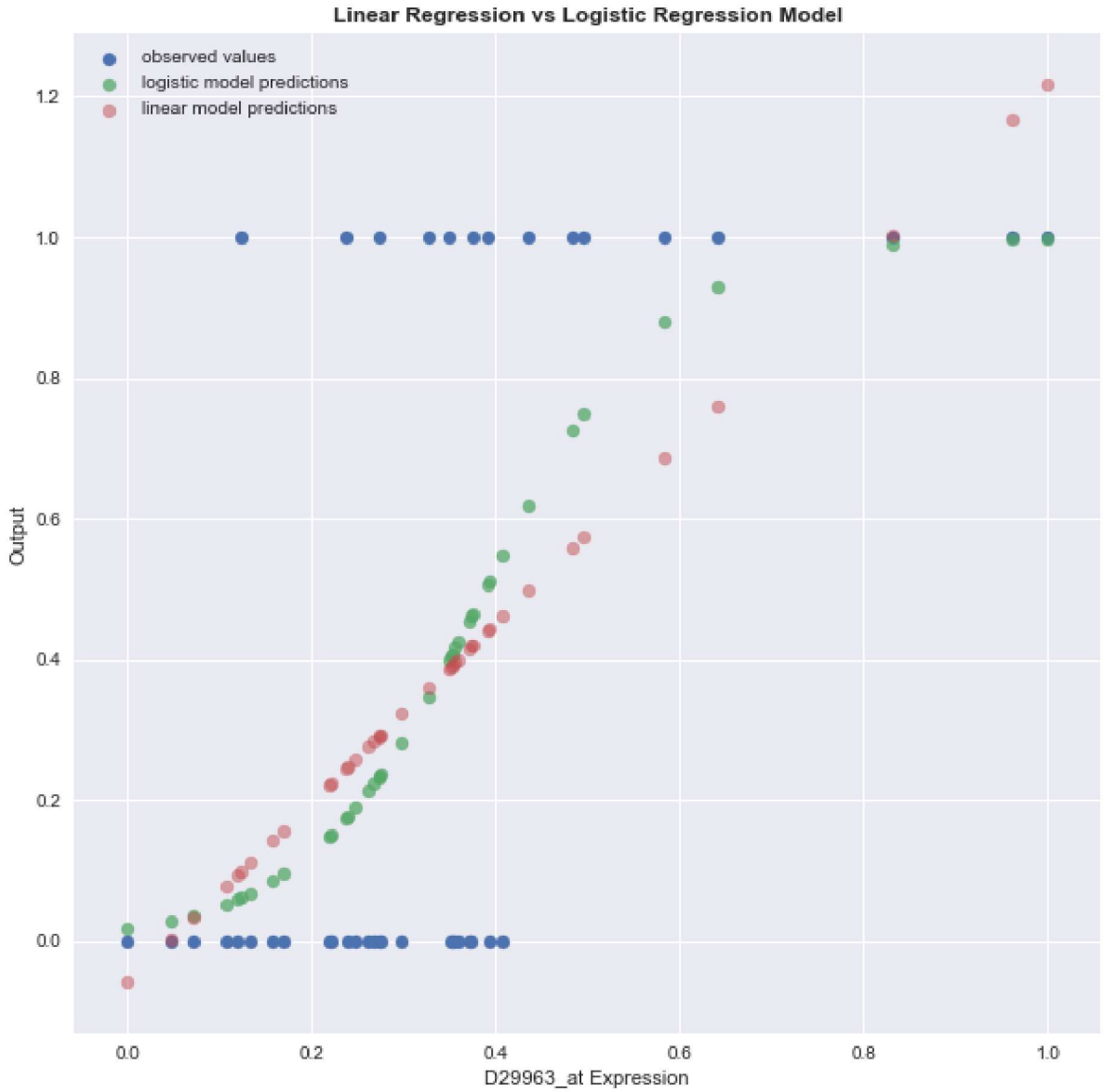
```
Accuracy Scores
Training:  0.800
Test:  0.697
```

Compared to the linear regression model, the logistic regression model performs the same in comparison for training but scores are worse for the test data.

2.4: Plot the quantitative output from the linear regression model and the probabilistic output from the logistic regression model (on the training set points) as a function of the gene predictor. Also, display the true binary response for the training set points in the same plot. Based on these plots, does one of the models appear better suited for binary classification than the other? Explain.

In [11]:
```
# Store the probabilistic output from the logistic regression model
probs = fitted_lr.predict_proba(X_train_normal['D29963_at'].values.reshape(-1,
1))
```

In [12]:
```
# Initialize figure and subplot
fig, ax = plt.subplots(1,1, figsize = (10,10))

# Plot the Observed values on the scatter plot
ax.scatter(X_train_normal['D29963_at'], y_train, label = 'observed values')

# Probability of beng classified as 1
probs_1 = []
for i in range(len(probs)):
    probs_1.append(probs[i][1])

# Plot predictions for the two models
ax.scatter(X_train_normal['D29963_at'], probs_1, label = 'logistic model predi
ctions', alpha = 0.75)
ax.scatter(X_train_normal['D29963_at'], predictions_train, label = 'linear mod
el predictions', alpha = 0.5)

# Add Labels
ax.set_xlabel("D29963_at Expression")
ax.set_ylabel("Output")
ax.set_title("Linear Regression vs Logistic Regression Model", fontweight="bol
d")
ax.legend();
```



Logistic regression appear to be better suited for binary classification than linear regression, as it lies in the same range of y-values [0,1] as the observed values and the curved shape appears to display a more binary nature than that of the linear model.

Question 3: Multiple Logistic Regression

1. Next, fit a multiple logistic regression model with all the gene predictors from the data set. How does the classification accuracy of this model compare with the models fitted in question 2 with a single gene (on both the training and test sets)?
2. Use the `visualize_prob` function provided below to visualize the probabilties predicted by the fitted multiple logistic regression model on both the training and test data sets. The function creates a visualization that places the data points on a vertical line based on the predicted probabilities, with the different cancer classes shown in different colors, and with the 0.5 threshold highlighted using a dotted horizontal line. Is there a difference in the spread of probabilities in the training and test plots? Are there data points for which the predicted probability is close to 0.5? If so, what can you say about these points?

In [13]:
```
#--------  visualize_prob
# A function to visualize the probabilities predicted by a Logistic Regression
 model
# Input:
#      model (Logistic regression model)
#      x (n x d array of predictors in training data)
#      y (n x 1 array of response variable vals in training data: 0 or 1)
#      ax (an axis object to generate the plot)

def visualize_prob(model, x, y, ax):
    # Use the model to predict probabilities for
    y_pred = model.predict_proba(x)

    # Separate the predictions on the label 1 and label 0 points
    ypos = y_pred[y==1]
    yneg = y_pred[y==0]

    # Count the number of label 1 and label 0 points
    npos = ypos.shape[0]
    nneg = yneg.shape[0]

    # Plot the probabilities on a vertical line at x = 0,
    # with the positive points in blue and negative points in red
    pos_handle = ax.plot(np.zeros((npos,1)), ypos[:,1], 'bo', label = 'Cancer
 Type 1')
    neg_handle = ax.plot(np.zeros((nneg,1)), yneg[:,1], 'ro', label = 'Cancer
 Type 0')

    # Line to mark prob 0.5
    ax.axhline(y = 0.5, color = 'k', linestyle = '--')

    # Add y-label and legend, do not display x-axis, set y-axis limit
    ax.set_ylabel('Probability of AML class')
    ax.legend(loc = 'best')
    ax.get_xaxis().set_visible(False)
    ax.set_ylim([0,1])
```

Answers

3.1: Next, fit a multiple logistic regression model with all the gene predictors from the data set. How does the classification accuracy of this model compare with the models fitted in question 2 with a single gene (on both the training and test sets)?

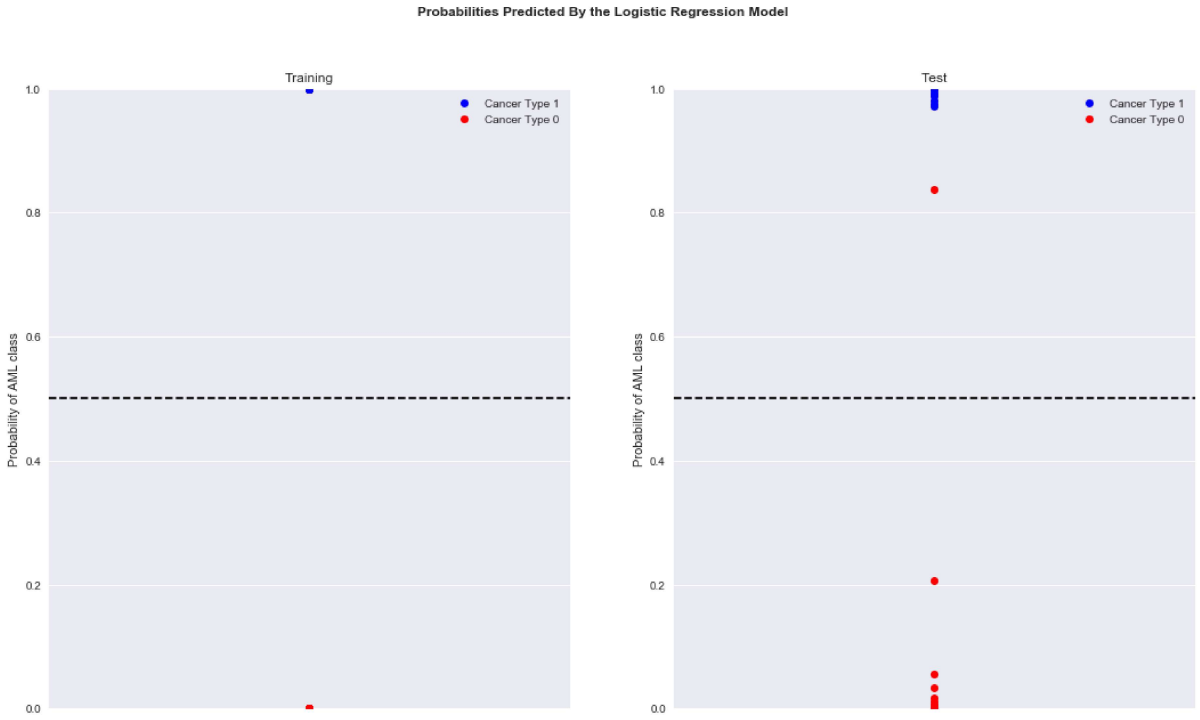
In [14]:
```
# Fit the multiple logistic regression model to the training data
fitted_multiple_lr = LogisticRegression(C=100000, solver='newton-cg', max_iter
=250).fit(X_train_normal,y_train)

# Store the predictions
mlr_predictions_train = fitted_multiple_lr.predict(X_train_normal)
mlr_predictions_test = fitted_multiple_lr.predict(X_test_normal)


# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(accuracy_score(y_train, mlr_pr
edictions_train)))
print("\033[1mTest:\033[0m {0: .3f}".format(accuracy_score(y_test, mlr_predict
ions_test)))
```

```
Accuracy Scores
Training:  1.000
Test:  0.970
```

The classification accuracy is much higher. The model fits perfectly on the training set and has a higher score on test compared to the modesl fitted in question 2.

3.2: Use the `visualize_prob` function provided below to visualize the probabilties predicted by the fitted multiple logistic regression model on both the training and test data sets. The function creates a visualization that places the data points on a vertical line based on the predicted probabilities, with the different cancer classes shown in different colors, and with the 0.5 threshold highlighted using a dotted horizontal line. Is there a difference in the spread of probabilities in the training and test plots? Are there data points for which the predicted probability is close to 0.5? If so, what can you say about these points?

In [15]:
```
# Initialize figure and subplots
fig, ax = plt.subplots(1,2,figsize = (18,10))

# Plot the visualized probabilities
visualize_prob(fitted_multiple_lr, X_train_normal, y_train, ax[0])
visualize_prob(fitted_multiple_lr, X_test_normal, y_test, ax[1]);

# Add labels
ax[0].set_title("Training")
ax[1].set_title("Test")

fig.suptitle("Probabilities Predicted By the Logistic Regression Model", fontw
eight="bold");
```



There is a difference in the spread. The training set is focused around probabilities of 1 and 0, while the test set has more of a spread points that span 0.2 on each side of the 0.5 horizontal line. One of the data points closer to 0.5 (located at ~0.83) is incorrectly predicted. The model appears to be overfitted on the training data.

Question 4: Analyzing Significance of Coefficients

How many of the coefficients estimated by the multiple logistic regression in the previous problem are significantly different from zero at a *significance level of 95%*?

Hint: To answer this question, use *bootstrapping* with 1000 boostrap samples/iterations.

Answer:

In [16]:
```
## accepts dataset inputs as numpy arrays
def make_bootstrap_sample(dataset_X, dataset_y, size = None):
    """Returns a bootstrap sample of the given data and size

    Args:
        dataset_X: the Pandas dataframe of feature variables and values
        dataset_y: the column vector of response values
        size: the size of the Bootstrap sample with None as the default

    Returns:
        Returns a tuple containing a dataframe containing the Bootstrap X-valu
es and a list of Bootstrap y-values
    """

    # by default return a bootstrap sample of the same size as the original da
taset
    if not size: size = len(dataset_X)

    # if the X and y datasets aren't the same size, raise an exception
    if len(dataset_X) != len(dataset_y):
        raise Exception("Data size must match between dataset_X and dataset_y"
)

    # Store an ndarray of random indices
    inds_to_sample = np.random.choice(dataset_X.shape[0], size, replace = True
)

    # Store the observations located at these indices in the Dataframe as the
 bootstrap sample data
    bootstrap_dataset_X = dataset_X.iloc[inds_to_sample]
    bootstrap_dataset_y = dataset_y[inds_to_sample]

    # return as a tuple the bootstrap samples of dataset_X as a pandas datafra
me
    # and the bootstrap samples of dataset y as a numpy column vector
    return (bootstrap_dataset_X, bootstrap_dataset_y)

def calculate_coefficients(dataset_X, dataset_y, model):
    """Returns the Beta-coefficients of the given data fitted to the given mod
el

    Args:
        dataset_X: the Pandas dataframe of feature variables and values
        dataset_y: the column vector of response values
        model: the model object to be fitted with the given data

    Returns:
        coefficients_dictionary: a dictionary containing the feature names as
 keys and the
        coefficients as values."""

    # Fit the data to the given model parameter
    model.fit(dataset_X, dataset_y)

    # Initalize dictionary of coefficents
    coefficients_dictionary = {}
```

```
    # Iterate through dataset coefficients and store them in the dictionary
    # Since the response variable is binary, just access the first (and only)
 element in the tuple
    for index, val in enumerate(model.coef_[0]):
        coefficients_dictionary[dataset_X.columns[index]] = val

    # return coefficients  in the variable  coefficients_dictionary as a dicti
onary
    # with the key being the name of the feature as a string
    # the value being the value of the coefficients
    # do not return the intercept as part of this
    return coefficients_dictionary

def get_significant_predictors(regression_coefficients, significance_level):
    """Returns the significant predictors from a model

    Args:
        regression_coefficients: a list of dictionaries containing the regress
ion coefficients
        of models made from different samples
        significance_level: the alpha-level used to determine feature signific
ance

    Returns:
        significant_coefficients: a list of the significant feature names
        """

    # initialize a DataFrame to contain a list of the column names
    coefficient_values = pd.DataFrame(index=list(regression_coefficients[0].ke
ys()))

    # Store all of the values from each bootstrap sample in a column
    for row, cur_sample in enumerate(regression_coefficients):
        coefficient_values[row] = cur_sample.values()

    # Transpose the DataFrame so that each row now represents one Bootstrap sa
mple
    coefficient_values = coefficient_values.T

    # Initialize significant_coefficients to an empty list
    significant_coefficients = []

    # Store the upper and lower percentile bounds
    lower_bound = significance_level/2 * 100
    upper_bound = 100 - lower_bound

    # Iterate through all of coefficient_values and determine which coefficien
ts are significant
    for index_column, col in enumerate(coefficient_values):
        if not(((np.percentile(coefficient_values[col], lower_bound ) < 0)) &
((np.percentile(coefficient_values[col], upper_bound) > 0))):
            significant_coefficients.append(coefficient_values.columns[index_c
olumn])

    # return significant_coefficients
    return significant_coefficients
```

```
# Initialize the list of coefficients
boot_coefs = [np.nan]*1000

# Create 1000 Bootstrap samples
for i in range(1000):
    boot_x, boot_y = make_bootstrap_sample(X_train_normal, y_train.values)
    boot_coefs[i] = calculate_coefficients(boot_x, boot_y, LogisticRegression(
C=100000, solver='newton-cg', max_iter=250))

# Store the significant predictors
CancerLog_significant_bootstrap = get_significant_predictors(boot_coefs, 0.05)

# Print the significant predictors
print("\033[1mNumber of Significant Predictors:\033[0m\n{0}".format(len(Cancer
Log_significant_bootstrap)))
```

```
Number of Significant Predictors:
1931
```

Question 5: High Dimensionality

One of the issues you may run into when dealing with high dimensional data is that your 2D and 3D intuition may fail breakdown. For example, distance metrics in high dimensions can have properties that may feel counterintuitive.

Consider the following: You have a hypersphere with a radius of 1, inside of a hypercube centered at 0, with edges of length 2.

1. As a function of d, the number of dimensions, how much of the hypercube's volume is contained within the hypersphere?
2. What happens as d gets very large?
3. Using the functions provided below, create a plot of how the volume ratio changes as a function of d.
4. What does this tell you about where the majority of the volume of the hypercube resides in higher dimensions?

HINTS:

- The volume of a hypercube with edges of length 2 is $V_c(d) = 2^d$.
- The volume of a hyperphere with a radius of 1 is $V_s(d) = \frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)}$, where Γ is Euler's Gamma Function.
- Γ is increasing for all $d \geq 1$.

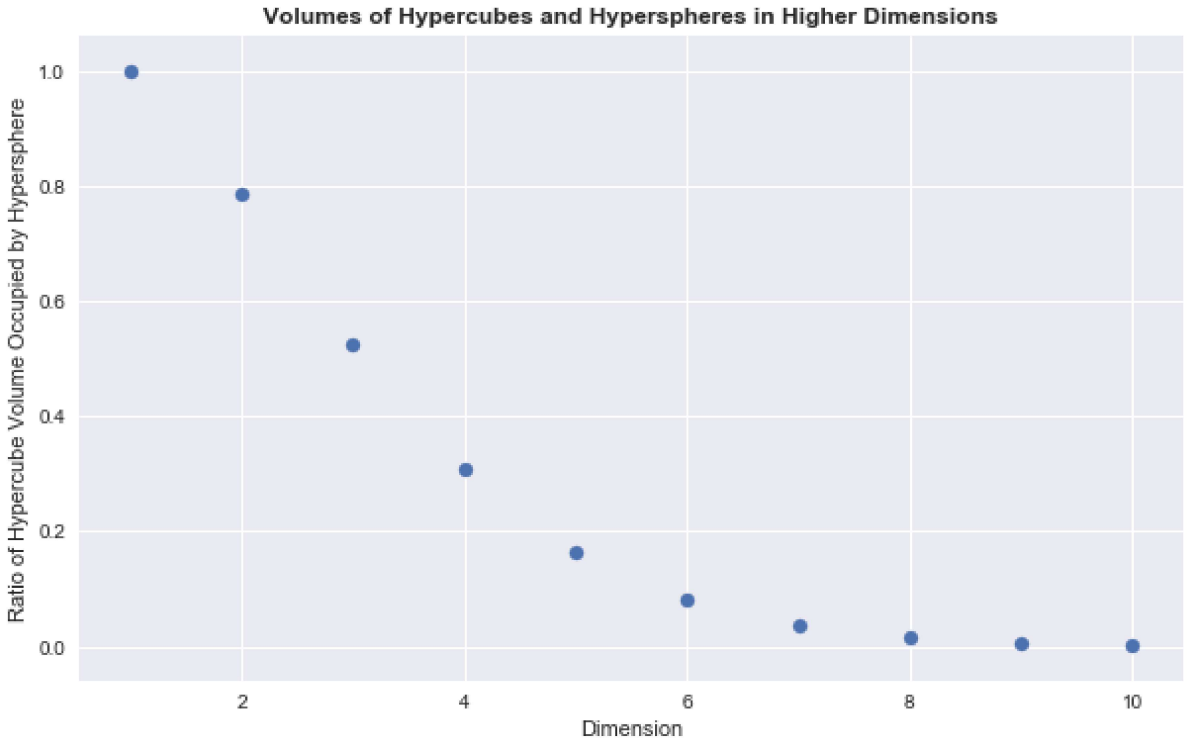
In [17]:
```
def V_c(d):
    """
    Calculate the volume of a hypercube of dimension d.
    """
    return 2**d

def V_s(d):
    """
    Calculate the volume of a hypersphere of dimension d.
    """
    return math.pi**(d/2)/gamma((d/2)+1)
```

Answers:

1). $\frac{\pi^{\frac{d}{2}}}{\Gamma(\frac{d}{2}+1)*2^d}$ **of the hypercube's volume is contained within the hypersphere.**

2). **As d gets very large, the proportion of the hypercube's volume contained within the hypersphere decreases and approaches 0.**

In [18]:
```
# 3).

# Initialize figure and subplot
fig, ax = plt.subplots(1,1, figsize=(10,6))

# Initialize number of dimensions to plot and the lists of dimensions and volu
mes
max_dimensions = 10
dimensions = np.arange(1, max_dimensions + 1)
vol_ratios = [np.nan] * max_dimensions

# Calculate the ratios utilizing the functions
for d in dimensions:
    vol_ratios[d-1] = V_s(d)/V_c(d)

# Plot the results
ax.scatter(dimensions, vol_ratios)

# Add labels
ax.set_title("Volumes of Hypercubes and Hyperspheres in Higher Dimensions", fo
ntweight = "bold")
ax.set_xlabel("Dimension")
ax.set_ylabel("Ratio of Hypercube Volume Occupied by Hypersphere");
```



4). The majority of the volume of the hypercube resides outside of the hypersphere in higher dimensions and approaches 100% as the dimensions increase.

Question 6: PCA and Dimensionality Reduction

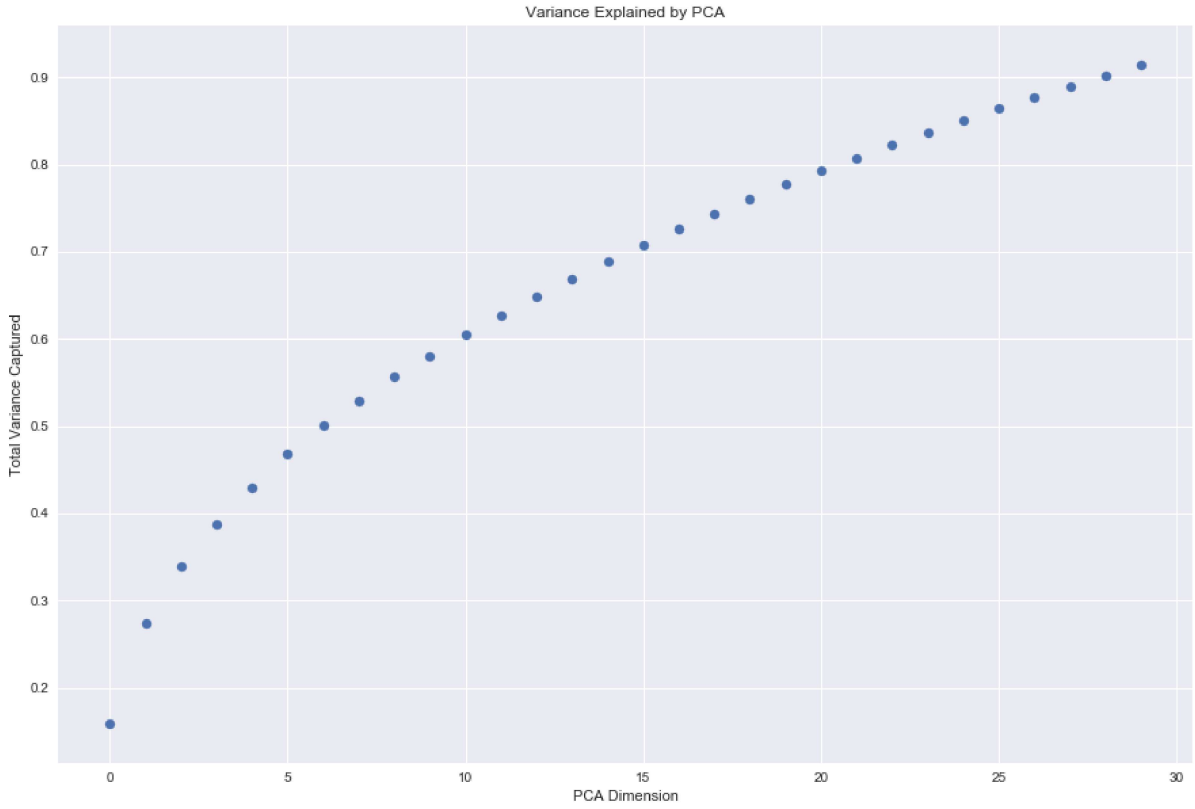
As we saw above, high dimensional problems can have counterintuitive behavior, thus we often want to try to reduce the dimensionality of our problems. A reasonable approach to reduce the dimensionality of the data is to use PCA and fit a logistic regression model on the smallest set of principal components that explain at least 90% of the variance in the predictors.

1. Using the gene data from Problem 1, how many principal components do we need to capture at least 90% of the variance? How much of the variance do they actually capture? Fit a Logistic Regression model using these principal components. How do the classification accuracy values on both the training and tests sets compare with the models fit in question 3.1?
2. Use the code provided in question 3 to visualize the probabilities predicted by the fitted model on both the training and test sets. How does the spread of probabilities in these plots compare to those for the model in question 3.2? If the lower dimensional representation yields comparable predictive power, what advantage does the lower dimensional representation provide?

Answers:

6.1: Using the gene data from Problem 1, how many principal components do we need to capture at least 90% of the variance? How much of the variance do they actually capture? Fit a Logistic Regression model using these principal components. How do the classification accuracy values on both the training and tests sets compare with the models fit in question 3.1?

In [19]:
```
# Transform the training data
pca_transformer30 = PCA(30).fit(X_train_normal)

# initialize figure and plots
fig, ax = plt.subplots(1,1,figsize = (15,10))

# Plot the Explained Variance Ratio
plt.scatter(range(0,30),np.cumsum(pca_transformer30.explained_variance_ratio_
))

# Add Labels
plt.xlabel("PCA Dimension")
plt.ylabel("Total Variance Captured")
plt.title("Variance Explained by PCA");
```



In [20]:
```
# Transform the training data
pca_transformer = PCA(29).fit(X_train_normal)
X_train_2d = pca_transformer.transform(X_train_normal)
X_test_2d =  pca_transformer.transform(X_test_normal)

# Fit the PCA training data to a logistic regression Model
fitted_pca = LogisticRegression(C=100000, solver='newton-cg', max_iter=250).fi
t(X_train_2d, y_train)

# Store the predictions
pca_predictions_train = fitted_pca.predict(X_train_2d)
pca_predictions_test = fitted_pca.predict(X_test_2d)

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(accuracy_score(y_train, pca_pr
edictions_train)))
print("\033[1mTest:\033[0m {0: .3f}".format(accuracy_score(y_test, pca_predict
ions_test)))
```

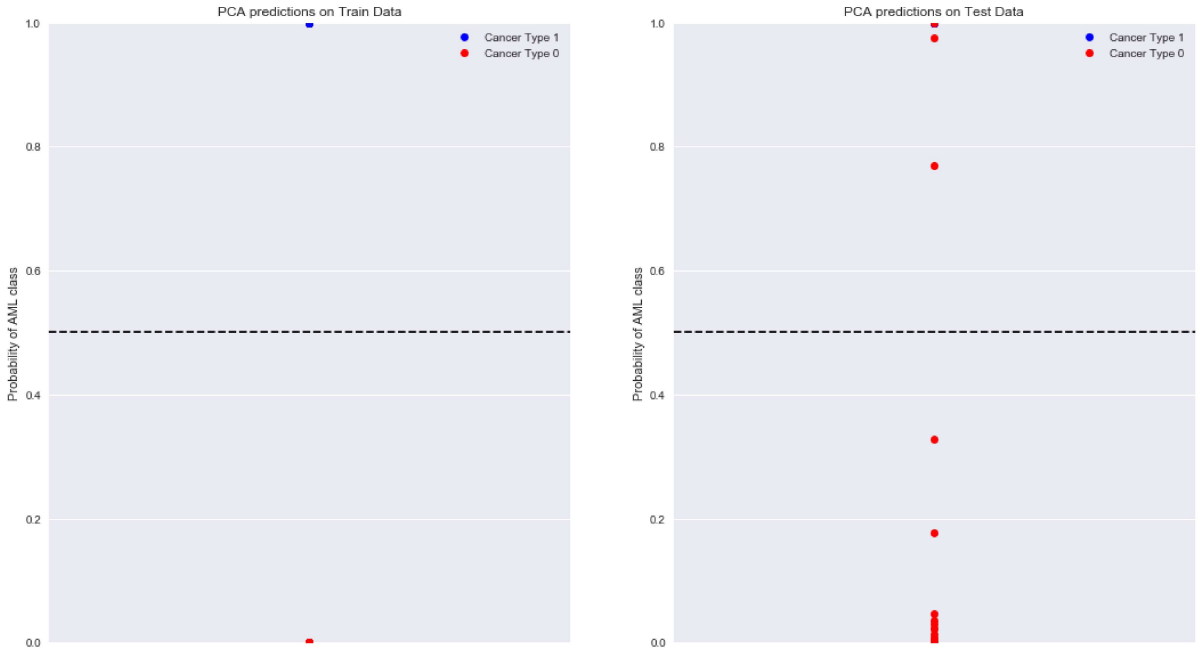
```
Accuracy Scores
Training:  1.000
Test:  0.909
```

We need 29 PCAs to capture 90% of the variance. There was no clearly defined elbow visible in the plot. We chose 10 PCAs which captured 60% of the variance in our model.

Compared to the model in 3.1, our current model still fits perfectly on the training set, however it has a lower accuracy score in comparison.

6.2: Use the code provided in question 3 to visualize the probabilities predicted by the fitted model on both the training and test sets. How does the spread of probabilities in these plots compare to those for the model in question 3.2? If the lower dimensional representation yields comparable predictive power, what advantage does the lower dimensional representation provide?

In [21]:
```
# Initialize figure and plot
fig, ax = plt.subplots(1,2,figsize = (18,10))

# Plot the predictions
visualize_prob(fitted_pca, X_train_2d, y_train, ax[0]);
visualize_prob(fitted_pca, X_test_2d, y_test, ax[1]);

# Add labels
ax[0].set_title('PCA predictions on Train Data')
ax[1].set_title('PCA predictions on Test Data');
```



The training data is fit perfectly, perhaps suggesting overfitting. The test data probablilities are spread throughout 0 and 1, and several points are wongly to predicted to be type 1. If the lower dimensional representation yields comparable predictive power, it will yield similar results to the higher dimensional representation while performing at a much faster rate.

Multiclass Thyroid Classification

In this problem, 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_hw5_2.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.

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

First task: split the data using the code provided below.

Question 7: Fit Classification Models

1. Generate a 2D scatter plot of the training set, denoting each class with a different color. Does it appear that the data points can be separated well by a linear classifier?
2. Briefly explain the difference between multinomial logistic regression and one-vs-rest (OvR) logistic regression methods for fitting a multiclass classifier (in 2-3 sentences).
3. Fit linear classification models on the thyroid data set using both the methods. You should use L_2 regularization in both cases, tuning the regularization parameter using cross-validation. Is there a difference in the overall classification accuracy of the two methods on the test set?
4. Also, compare the training and test accuracies of these models with the following classification methods:

    - Multiclass Logistic Regression with quadratic terms
    - Linear Discriminant Analysis
    - Quadratic Discriminant Analysis
    - k-Nearest Neighbors
      Note: you may use either the OvR or multinomial variant for the multiclass logistic regression (with L_2 regularization). Do not forget to use cross-validation to choose the regularization parameter, and also the number of neighbors in k-NN.
5. Does the inclusion of the polynomial terms in logistic regression yield better test accuracy compared to the model with only linear terms?

Hint: You may use the `KNeighborsClassifier` class to fit a k-NN classification model.

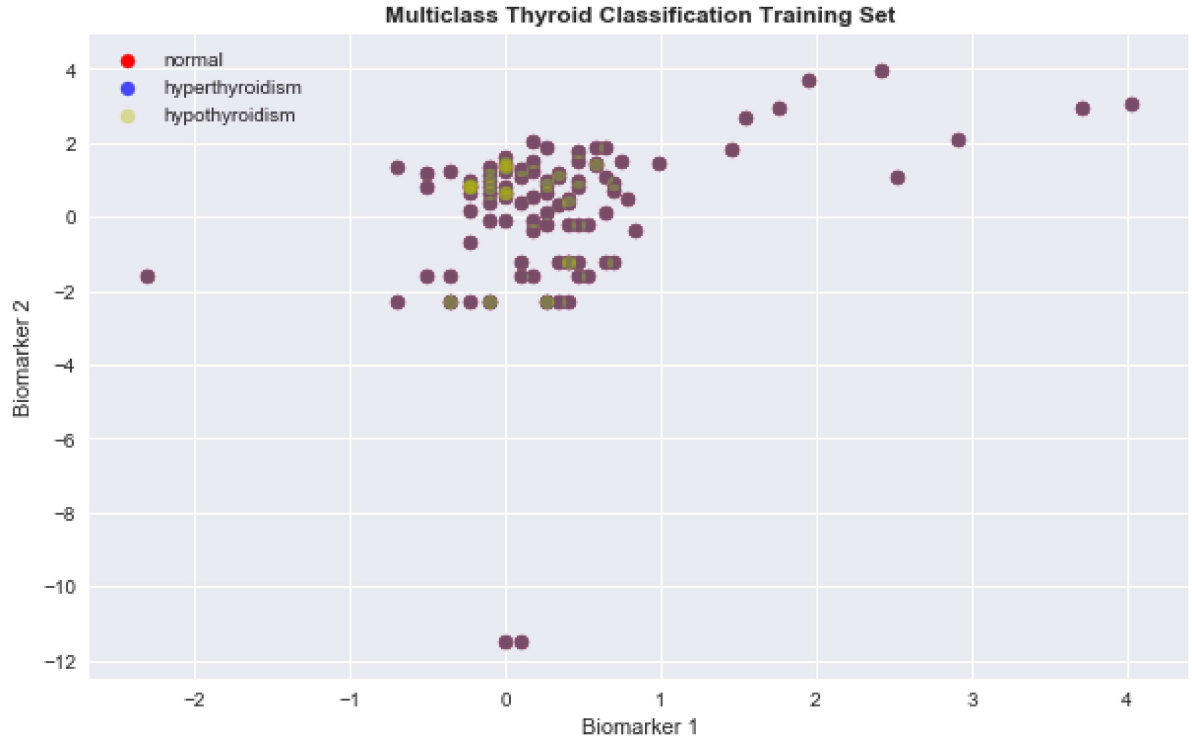
Answers:

7.0: First task: split the data using the code provided below.

In [22]:
```
np.random.seed(9001)
df = pd.read_csv('data/dataset_hw5_2.csv')
msk = np.random.rand(len(df)) < 0.5
data_train = df[msk]
data_test = df[~msk]

# Create X_train, y_train
X_train = data_train.copy().drop(columns="Diagnosis")
y_train = data_train.copy()["Diagnosis"]

# Create X_test, y_test
X_test = data_test.copy().drop(columns="Diagnosis")
y_test = data_test.copy()["Diagnosis"]
```

7.1: Generate a 2D scatter plot of the training set, denoting each class with a different color. Does it appear that the data points can be separated well by a linear classifier?

In [23]:
```
# initialize figure and subplots
fig, ax = plt.subplots(1, 1, figsize=(10,6))

# initialize an array of colors
colors = ["r", "b", "y"]

# Iterate through each class
for index, thyroid_class in enumerate(["normal", "hyperthyroidism", "hypothyroidism"]):
    # plot each class of data
    class_data = data_train[data_train["Diagnosis"] == (index + 1)]
    ax.scatter(data_train["Biomarker 1"].values, data_train["Biomarker 2"].values, label= thyroid_class, color =colors[index], alpha=(1.0 - 0.3*index))

# add labels
ax.legend()
ax.set_xlabel("Biomarker 1")
ax.set_ylabel("Biomarker 2")
ax.set_title("Multiclass Thyroid Classification Training Set", fontweight = "bold");
```



It does not appear that the data points can be separated well by a linear classifier, as the data are all clustered in a similar region of the scatter plot.

7.2: Briefly explain the difference between multinomial logistic regression and one-vs-rest (OvR) logistic regression methods for fitting a multiclass classifier (in 2-3 sentences).

Multinomial logistic regression and one-vs-rest logistic regression differ in their reference group for each model in the regression. Multinomial logistic regression references the same baseline class for each model and compares each class individually to that baseline class, while one-vs-rest regression references each different class to the combination of all other classes for all models.

7.3: Fit linear classification models on the thyroid data set using both the methods. You should use L_2 regularization in both cases, tuning the regularization parameter using cross-validation. Is there a difference in the overall classification accuracy of the two methods on the test set?

In [24]:
```
# List of potential regularization parameters
lambdas = [0.001, 0.005, 0.01, 0.05, 0.1, 1, 5, 10, 50, 100, 500, 1000]

# OvR Model
kfold = KFold(5, shuffle=True)
OvRCV_object = LogisticRegressionCV(Cs=lambdas, cv=kfold, multi_class='ovr', p
enalty= 'l2')
OvRModel = OvRCV_object.fit(X_train, y_train)
cvscores_ovr = accuracy_score(y_test, OvRModel.predict(X_test))

# Multinomial Model
kfold = KFold(5, shuffle=True)
MultinomialCV_object = LogisticRegressionCV(Cs=lambdas, cv=kfold, multi_class=
'multinomial', penalty= 'l2')
Multinomial_Model = MultinomialCV_object.fit(X_train, y_train)
cvscores_multinomial = accuracy_score(y_test, Multinomial_Model.predict(X_test
))

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mOvR:\033[0m {0: .3f}".format(cvscores_ovr))
print("\033[1mMultinomial:\033[0m {0: .3f}".format(cvscores_multinomial))
```

```
Accuracy Scores
OvR:  0.858
Multinomial:  0.885
```

The multinomial regression model has a higher accuracy score for the test set than the OVR model.

7.4: Also, compare the training and test accuracies of these models with the following classification methods:

- Multiclass Logistic Regression with quadratic terms
- Linear Discriminant Analysis
- Quadratic Discriminant Analysis
- k-Nearest Neighbors

Note: you may use either the OvR or multinomial variant for the multiclass logistic regression (with L_2 regularization). Do not forget to use cross-validation to choose the regularization parameter, and also the number of neighbors in k-NN.

In [25]:
```
# Multiclass Logistic Regression with quadratic terms

# Add the polynomial features
poly_transform = PolynomialFeatures(2, include_bias = False)
X_train_poly = poly_transform.fit_transform(X_train)
X_test_poly = poly_transform.fit_transform(X_test)

# Fit a multiclass logistic regression to the data with polynomial features
# Use lbfgs solver by default
poly_model = LogisticRegressionCV(Cs = lambdas, cv = kfold, multi_class = 'ov
r', penalty = 'l2')
poly_model.fit(X_train_poly, y_train)

# Store the accuracy scores
poly_score_train = accuracy_score(y_train, poly_model.predict(X_train_poly))
poly_score_test = accuracy_score(y_test, poly_model.predict(X_test_poly))

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(poly_score_train))
print("\033[1mTest:\033[0m {0: .3f}".format(poly_score_test))
```

```
Accuracy Scores
Training:  0.882
Test:  0.885
```

In [26]:
```
# Linear Discriminant Analysis

# fit
thyroid_lda = LinearDiscriminantAnalysis().fit(X_train, y_train)

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(thyroid_lda.score(X_train, y_t
rain)))
print("\033[1mTest:\033[0m {0: .3f}".format(thyroid_lda.score(X_test, y_test
)))
```

```
Accuracy Scores
Training:  0.873
Test:  0.832
```

In [27]:
```
# Quadratic Disciminant Analysis

# fit
thyroid_qda = QuadraticDiscriminantAnalysis().fit(X_train, y_train)

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(thyroid_qda.score(X_train, y_train)))
print("\033[1mTest:\033[0m {0: .3f}".format(thyroid_qda.score(X_test, y_test)))
```

Accuracy Scores
Training: 0.873
Test: 0.850

In [28]:
```
# kNN

# initialize array
results = np.zeros((10,3))

# Loop through the first 10 nearest neighbours and compute the cross val score
 for each neighbour
for i,n in enumerate(range(1,11)):
    model = KNeighborsClassifier(n_neighbors = n)
    results[i,:] = cross_val_score(model, X_train, y_train, cv=3)

# Create Data frame with CV scores for choosing the number of neighbours in kNN
results_df = pd.DataFrame(results, index=list(range(1,11)), columns= ["CV1","CV2","CV3"])
results_df['meanCV'] = np.mean(results, axis=1)
results_df
```

Out[28]:

	CV1	CV2	CV3	meanCV
1	0.800000	0.852941	0.939394	0.864112
2	0.742857	0.882353	0.909091	0.844767
3	0.885714	0.941176	0.969697	0.932196
4	0.828571	0.941176	0.909091	0.892946
5	0.828571	0.882353	0.909091	0.873338
6	0.828571	0.852941	0.878788	0.853433
7	0.857143	0.852941	0.939394	0.883159
8	0.828571	0.852941	0.909091	0.863535
9	0.885714	0.852941	0.909091	0.882582
10	0.857143	0.794118	0.909091	0.853450

In [29]:
```
#Creating kNN model with optimal nearest neighbours
optimal_knn = KNeighborsClassifier(n_neighbors = 3).fit(X_train, y_train)

# Print the accuracy scores:
print("\033[1mAccuracy Scores\033[0m")
print("\033[1mTraining:\033[0m {0: .3f}".format(optimal_knn.score(X_train, y_t
rain)))
print("\033[1mTest:\033[0m {0: .3f}".format(optimal_knn.score(X_test, y_test
)))
```

```
Accuracy Scores
Training:  0.931
Test:  0.867
```

7.5: Does the inclusion of the polynomial terms in logistic regression yield better test accuracy compared to the model with only linear terms?

The inclusion of polynomial terms does not yield better test accuracy compared to the model with only linear terms. In fact, it's slightly worse.

Question 8: Visualize Decision Boundaries

The following code will allow you to visualize the decision boundaries of a given classification model.

In [30]:
```
#--------  plot_decision_boundary
# A function that visualizes the data and the decision boundaries
# Input:
#      x (predictors)
#      y (labels)
#      model (the classifier you want to visualize)
#      title (title for plot)
#      ax (a set of axes to plot on)
#      poly_degree (highest degree of polynomial terms included in the model;
 None by default)

def plot_decision_boundary(x, y, model, title, ax, poly_degree=None):
    # Create mesh
    # Interval of points for biomarker 1
    min0 = x[:,0].min()
    max0 = x[:,0].max()
    interval0 = np.arange(min0, max0, (max0-min0)/100)
    n0 = np.size(interval0)

    # Interval of points for biomarker 2
    min1 = x[:,1].min()
    max1 = x[:,1].max()
    interval1 = np.arange(min1, max1, (max1-min1)/100)
    n1 = np.size(interval1)

    # Create mesh grid of points
    x1, x2 = np.meshgrid(interval0, interval1)
    x1 = x1.reshape(-1,1)
    x2 = x2.reshape(-1,1)
    xx = np.concatenate((x1, x2), axis=1)

    # Predict on mesh of points
    # Check if polynomial terms need to be included
    if(poly_degree!=None):
        # Use PolynomialFeatures to generate polynomial terms
        poly = PolynomialFeatures(poly_degree,include_bias = False)
        xx_ = poly.fit_transform(xx)
        yy = model.predict(xx_)

    else:
        yy = model.predict(xx)

    yy = yy.reshape((n0, n1))

    # Plot decision surface
    x1 = x1.reshape(n0, n1)
    x2 = x2.reshape(n0, n1)
    ax.contourf(x1, x2, yy, cmap=plt.cm.coolwarm, alpha=0.8)

    # Plot scatter plot of data
    yy = y.reshape(-1,)
    ax.scatter(x[yy==1,0], x[yy==1,1], c='blue', label='Normal', cmap=plt.cm.c
oolwarm)
    ax.scatter(x[yy==2,0], x[yy==2,1], c='cyan', label='Hyper', cmap=plt.cm.co
olwarm)
    ax.scatter(x[yy==3,0], x[yy==3,1], c='red', label='Hypo', cmap=plt.cm.cool
```

```
warm)

    # Label axis, title
    ax.set_title(title)
    ax.set_xlabel('Biomarker 1')
    ax.set_ylabel('Biomarker 2')
```

Note: The provided code uses `sklearn`'s `PolynomialFeatures` to generate higher-order polynomial terms, with degree `poly_degree`. Also, if you have loaded the data sets into `pandas` data frames, you may use the `as_matrix` function to obtain a `numpy` array from the data frame objects.

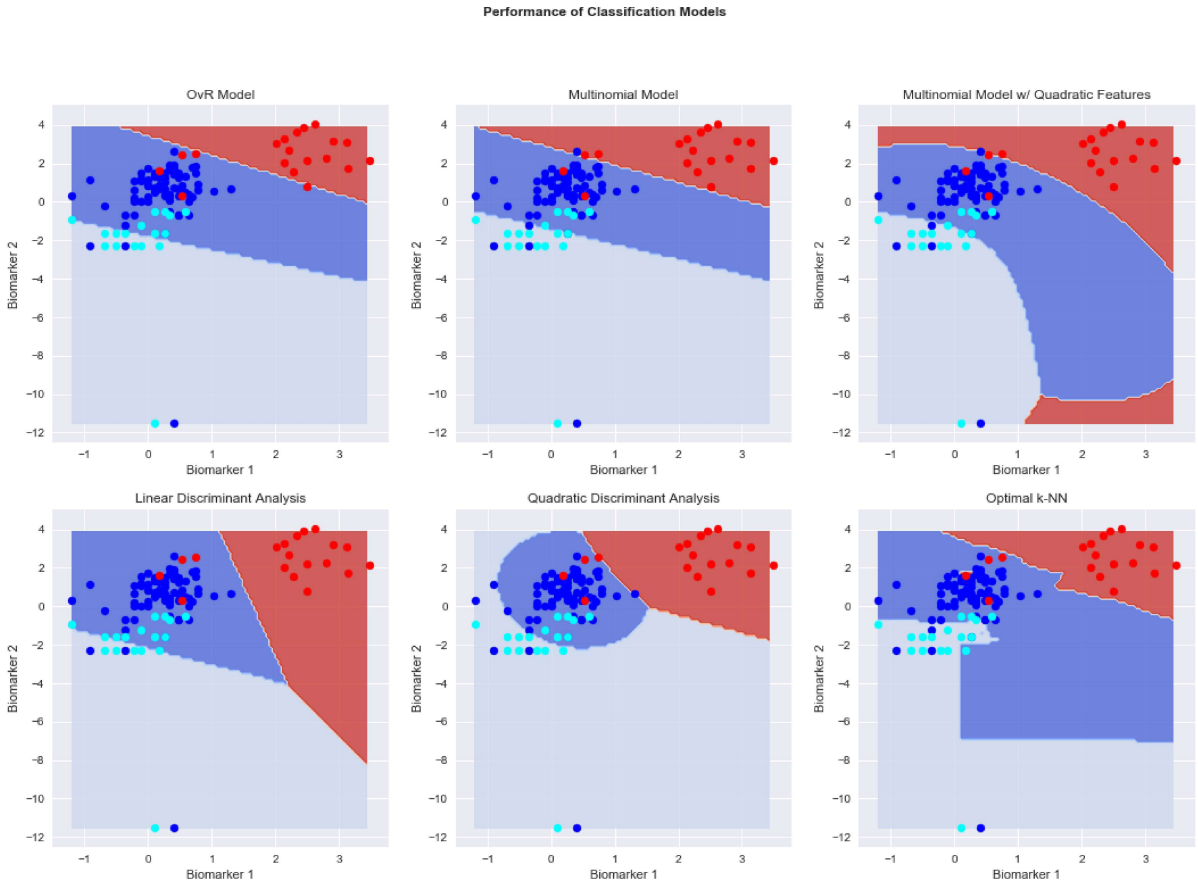
1. Use the above code to visualize the decision boundaries for each of the model fitted in the previous question.
2. Comment on the difference in the decision boundaries (if any) for the OvR and multinomial logistic regression models. Is there a difference between the decision boundaries for the linear logistic regression models and LDA. What about the decision boundaries for the quadratic logistic regression and QDA? Give an explanation for your answer.
3. QDA is a generalization of the LDA model. What's the primary difference that makes QDA more general? How does that manifest in the plots you generated?

Answers:

8.1: Use the above code to visualize the decision boundaries for each of the model fitted in the previous question.

In [33]:
```
# Change dataframes to arrays as told in directions despite warning
X_test_array = X_test.as_matrix()
y_test_array = y_test.as_matrix()

# Initialize figure and subplots
fig, ax = plt.subplots(2,3, figsize=(18,12))

# OVR
plot_decision_boundary(X_test_array, y_test_array, OvRModel, "OvR Model", ax[0
][0])

# Multinomial_
plot_decision_boundary(X_test_array, y_test_array, Multinomial_Model, "Multino
mial Model", ax[0][1])

# Multinomial with Polynomial Features
plot_decision_boundary(X_test_array, y_test_array, poly_model, "Multinomial Mo
del w/ Quadratic Features", ax[0][2], 2)

# Linear Discriminant Analysis
plot_decision_boundary(X_test_array, y_test_array, thyroid_lda, "Linear Discri
minant Analysis", ax[1][0])

# Quadratic Discriminant Analysis
plot_decision_boundary(X_test_array, y_test_array, thyroid_qda, "Quadratic Dis
criminant Analysis", ax[1][1])

# Optimal k-NN
plot_decision_boundary(X_test_array, y_test_array, optimal_knn, "Optimal k-NN"
, ax[1][2])

# Plot labels
fig.suptitle("Performance of Classification Models", fontweight="bold");
```

```
C:\Users\mkolo\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWar
ning: Method .as_matrix will be removed in a future version. Use .values inst
ead.

C:\Users\mkolo\Anaconda3\lib\site-packages\ipykernel_launcher.py:3: FutureWar
ning: Method .as_matrix will be removed in a future version. Use .values inst
ead.
  This is separate from the ipykernel package so we can avoid doing imports u
ntil
```



8.2: Comment on the difference in the decision boundaries (if any) for the OvR and multinomial logistic regression models. Is there a difference between the decision boundaries for the linear logistic regression models and LDA. What about the decision boundaries for the quadratic logistic regression and QDA? Give an explanation for your answer.

The decision boundaries better capture the classes for the multinomial regression models than for the OvR. The boundary lines for the multinomial model appear to show that the class that is dark blue is the baseline model, as both boundary lines run parallel on opposite sides of the dark blue region. Meanwhile, the OvR boundary lines intersect at several points, indicating that each boundary was made using a different baseline class with comparison to all other classes.

The LDA and logsitic models have different decision boundaries. The logistic model has decision boundaries that are parallel to the baseline class. While the LDA assumes that the covariance is the same for all classes, and therefore the decision boundaries are created such that it goes through the middle point between the centroids of each class.

The quadratic discriminant analysis does not assume that the covariance is the same for all classes, leading to different nonlinear boundaries between each class' centroid that mark the threshold for which class is more likely. Meanwhile, for multinomial quadratic logistic regression, a baseline class is still assumed, leading to decision boundaries being drawn with respect to that class.

8.3: QDA is a generalization of the LDA model. What's the primary difference that makes QDA more general? How does that manifest in the plots you generated?

QDA does not assume the covariance matrix is the same between all classes so the decision boundaries are determined by a quadratic function. In LDA it is assumed that every class follows the Gaussian distribution, but with equal variance and covariance, so the boundary between classes becomes linear.