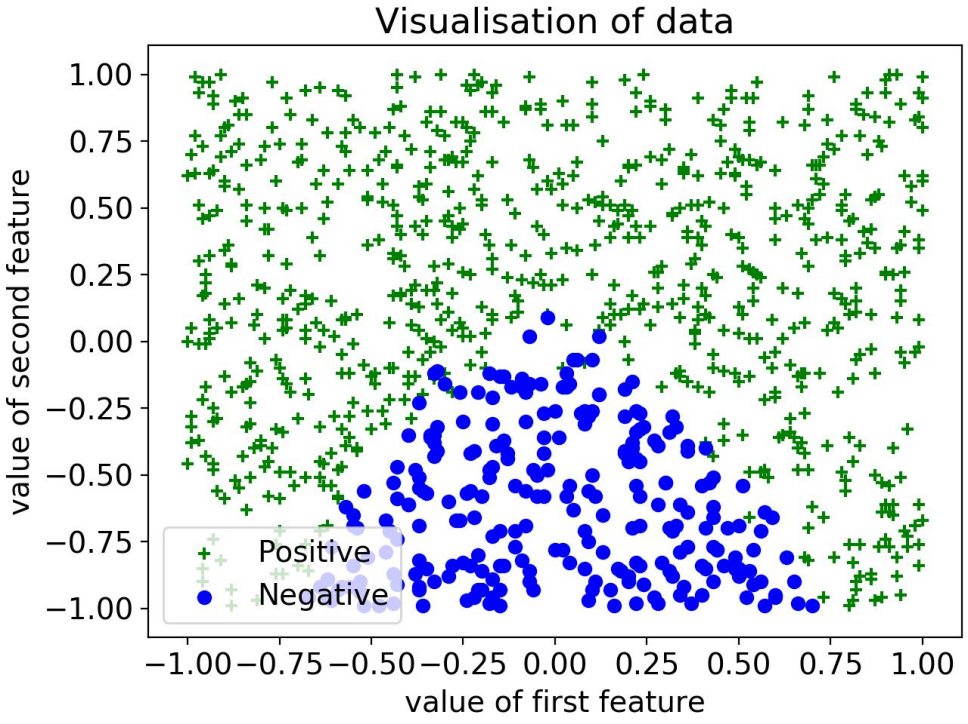
Machine Learning | Week 2 Assignment
17324263 | Stephen Byrne

a (i)
To visualize the data I first split the data into positive and negative data point arrays. I wrote a function called split to do this *(Appendix a.1)*. The split function iterates over the values and using the passed in result value (positive/negative) it then adds the features to a positive or negative feature array. These feature arrays are then combined to return two arrays, the positive and negative data point arrays. Using the two returned arrays I then plotted them, giving the positive data point pairs a marker of '+' and a color of green, and the negative point pairs a marker of 'o' and a color of blue *(Appendix a.1.2)*.



Explanation of the above graph: As we can see the value of the first feature is plotted on the x-axis and the value of the second feature is plotted on the y-axis. The negative data resembles a quadratic or triangular shape.

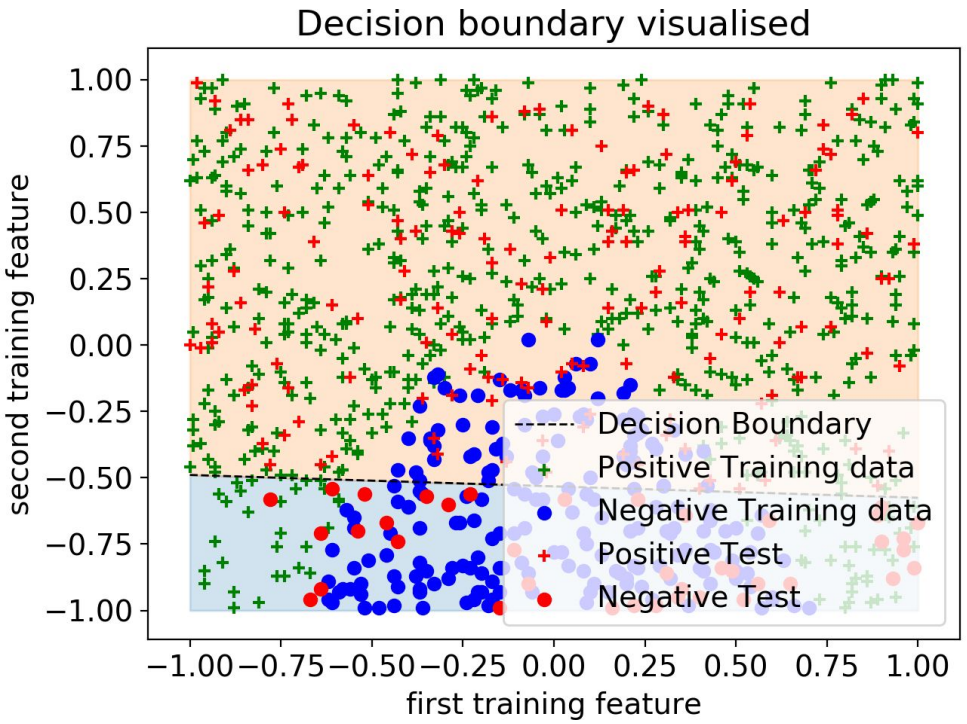
a (ii)

I first split my data into training and testing data (80% training, 20% test) using sklearn.model_selection train_test_split function, this number was chosen to have adequate data for testing and still to leave enough data to test against. I then trained the model using sklearn LogisticRegression model. The logistic regression model is trained on the two features X1 and X2 which can be seen on the graph above on their respective axis and on the training y actual values (positive/negative). After training the model the intercept was 2.06805739 and the coefficient values were 0.16684665 and 3.88594923 these were accessed from the model parameters at *(Appendix a.2)*.

As the formula shows prediction ŷ = sign(2.06805739 + 0.16684665 x_1 + 3.88584923 x_2) so from this we can form out that Y = +1 when 2.06805739 + 0.16684665 x_1 + 3.88584923 x_2 > 0 and Y = -1 when 2.06805739 + 0.16684665 x_1 + 3.88584923 x_2 < 0. I think this looks reasonable based on plotted data. When we take a datapoint we can then test if it agrees with the models prediction. Taking the point (0,-1) we can substitute into the equation and we get 2.06805739 + 0.16684665 (0) + 3.88584923 (-1) which equals to -1.81779. As we know Y = -1 when 2.06805739 + 0.16684665 x_1 + 3.88584923 x_2 < 0 this would lead us to predict -1. This holds true and can be seen graphically in the graph in a (iii) where the decision boundary is visible.

a (iii)

Using the trained model, we use the predict() function and pass our target values (test) as the parameter. This returns a predicted outcome, positive or negative, for each of the target values *(Appendix a.3.1)*. I then used the same function as before to split the data into positive and negative paired data points *(Appendix a.1).* Upon completion of this, they can be added to the graph seen below. To show the decision boundary I used an online resource *(Appendix r.2)* which I will now explain below the graph being referred to on the next page.



Our parameter values are w1,w2 and our bias the intercept of the model which relate a given input feature to the predicted log odd z. With two features we end up with the following formula for the relationship between z and the component features: $z = w_1 x_1 + w_2 x_2$ +b. We can think of the decision boundary as a line $x_2 = mx_1$ + b being defined for points which ŷ = 0.5 hence z =0. For x_1 = 0 we can then solve to find out our intercept. 0 = 0 + $w_2 x_2$ + b can be arranged to c = $-\frac{b}{w_2}$ *(Appendix a.3.3)*. For the gradient we can then further derive the formula to end up with m = $-\frac{w_1}{w_2}$ *(Appendix a.3.3)*. I wrote functions out for each of these parts and a function to add the decision boundary to all of my plots. This is used throughout the assignment and will be referenced as *(Appendix a.3.2).* It takes in the w1,w2 (parameters) and bias and derives line parameters as stated in the above formula and then plots it. The plotting function takes in the positive training data, negative training data, positive test and negative test. It can be seen at *(Appendix a.3.4)* and it is used throughout the assignment and will be referenced as such.

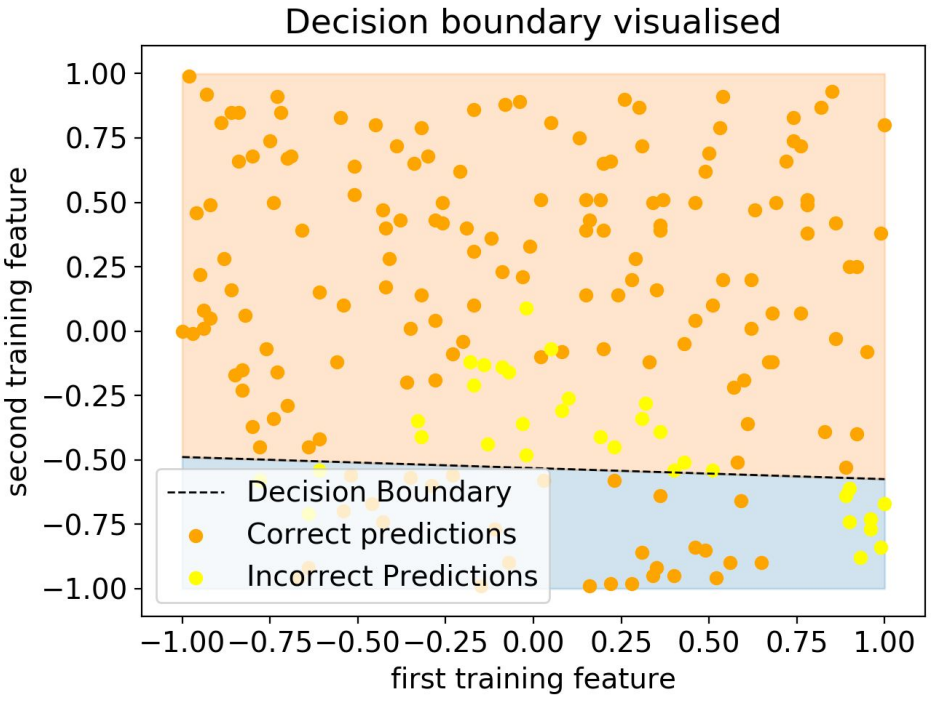
a (iv)

Visually in the plot above, we can see the comparisons between the training and test data and how the decision boundary of our model is working. The decision boundary is derived from the parameters *(Appendix a.3.2)* which give us an intercept and slope which represents the result the model will predict. Below the boundary, it predicts a negative, and above it predicts a positive. This doesn't give super accurate results as our negative training data has more of a quadratic shape however it is somewhat accurate. We can use a confusion matrix and compare it to the actual test data to show the true numbers for false positives, true positives, false negatives, and true

negatives. The sklearn confusion matrix *(Appendix a.4.2)* represents the actual target classes as columns and the predicted as rows It looks like this below;

True-negative: 28	False-positive: 22
False-negative: 11	True-positive: 139

As we can see, we have a lot of false negatives to the ratio of true negatives however our model seems quite accurate in regards to positives. I also wrote a function to split the test data into correct and incorrect predictions (*Appendix a.4.1*). Upon graphing this a slightly cleaner visual representation is shown of the accuracy of the model. This is shown below and the plotting function seen at (*Appendix.4.3).* The plotting function to show correct / incorrect is used throughout the report and will be referenced at *(Appendix 4.3).* The decision boundary is not always plotted alongside this as it is a separate function but it is included for some cases *(Appendix a.3.2)*.



As we can see below the decision boundary (highlighted in blue) the model will predict a negative value while above it (highlighted in orange) will predict a positive value. We can see clearly here that the quadratic negative shape is being “cut at the top.” Where the incorrect predictions (false positives) are in the positive prediction zone. We can also see the false negatives which are predicted close to the left and right edges of the graph inside the blue negative prediction zone.

b (i)

When C = 0.001

The coefficient (parameter) values were 3.01278330x 10^{-4} and 0.3.19746832x 10^{-1} and the intercept was 0.30527602. This is due to the fact the hyperparameter C was so small that theta transpose theta in the penalty will be large due to division by 0.001. The cost function, therefore, works to minimize the parameter values meaning they are small. I discuss the impact this has on the model in part b (ii) and b (iii).

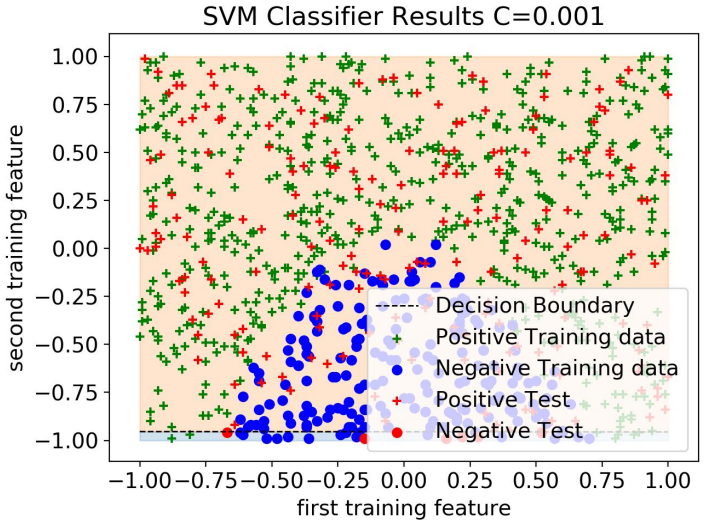
When C = 1

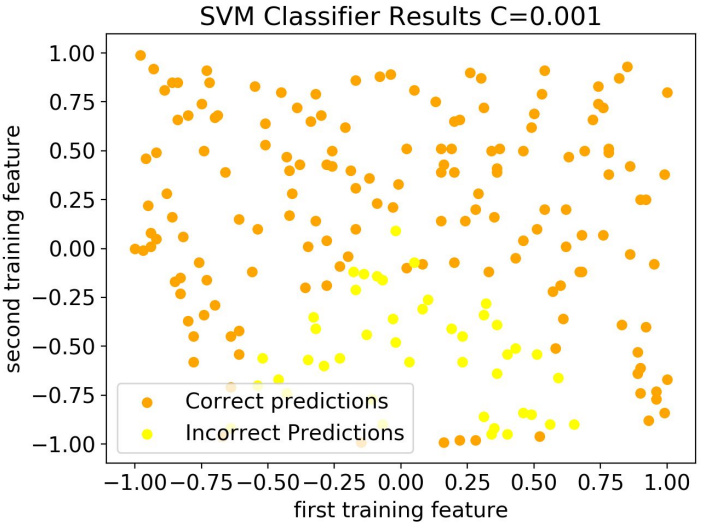
The coefficient values were 0.01971028 and 1.41197085 and the intercept was 0.73633749. This ends up leading to a moderate penalty as the penalty will just be theta transpose theta / 1 which is just theta transpose theta. This allows a good reduction of the parameter values in the cost function. The impact of this is discussed more in part b (ii) and b (iii).

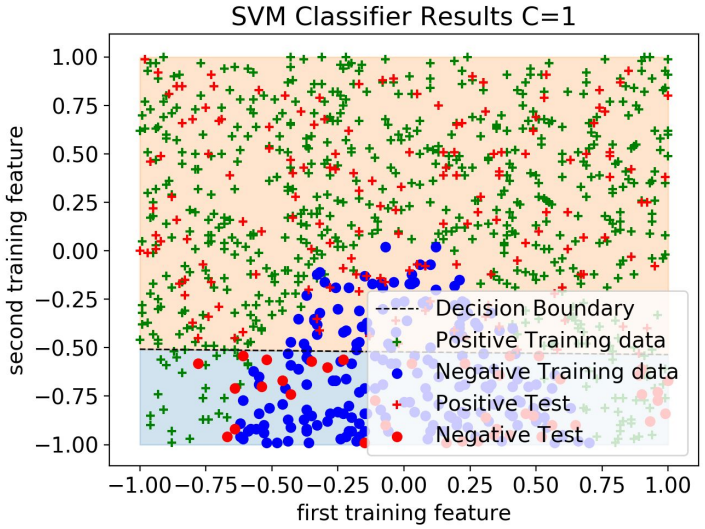
When C = 1000

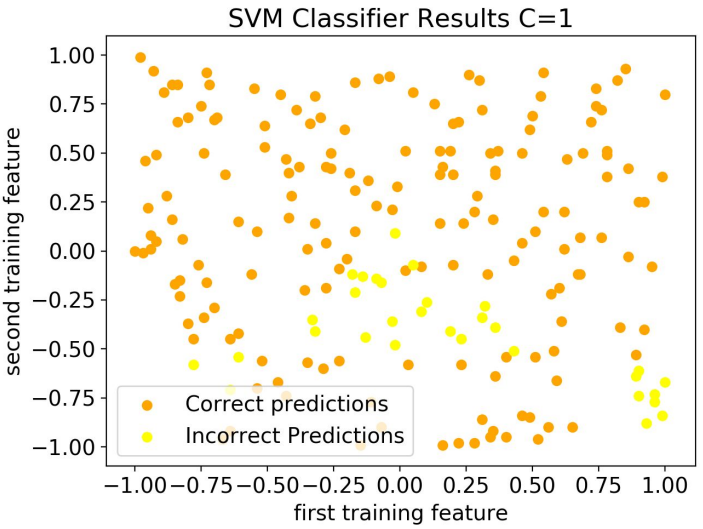
The coefficient values were -0.19649839 and 1.40041781 and intercept 0.19573446. The large value of 1000 lead to a very small penalty being added to the cost function. The impact of this is discussed more in part b (ii) and b (iii).

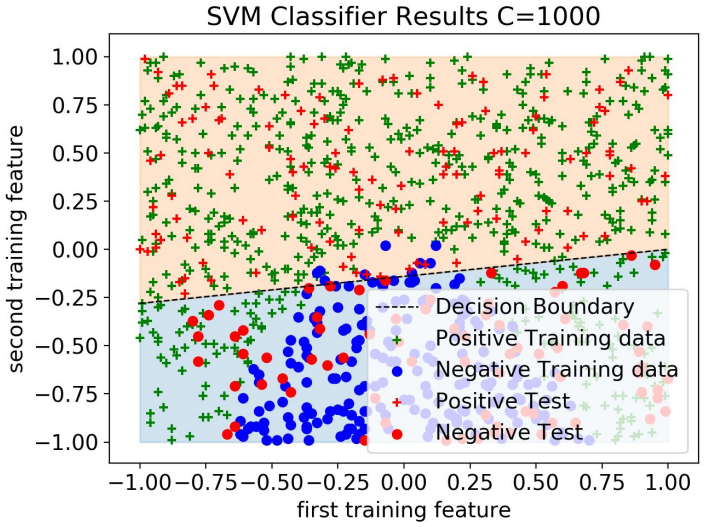
b (ii)

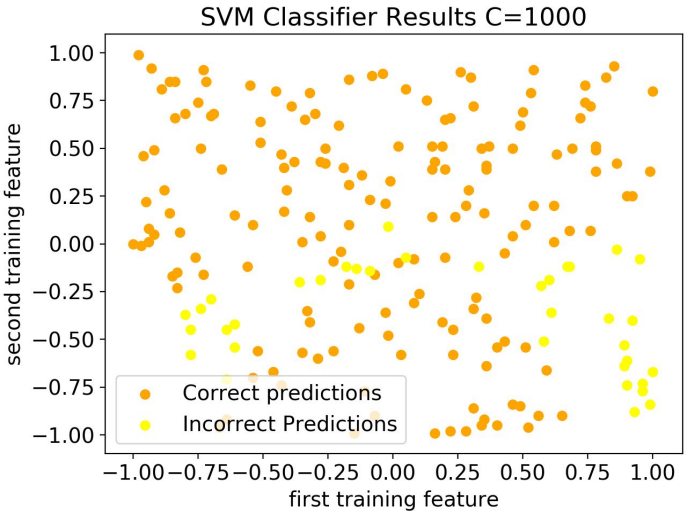

SVM Classifier Results C=0.001
second training feature
first training feature
Decision Boundary
Positive Training data
Negative Training data
Positive Test
Negative Test


SVM Classifier Results C=0.001
second training feature
first training feature
Correct predictions
Incorrect Predictions


SVM Classifier Results C=1
second training feature
first training feature
Decision Boundary
Positive Training data
Negative Training data
Positive Test
Negative Test


SVM Classifier Results C=1
second training feature
first training feature
Correct predictions
Incorrect Predictions


SVM Classifier Results C=1000
second training feature
first training feature
Decision Boundary
Positive Training data
Negative Training data
Positive Test
Negative Test


SVM Classifier Results C=1000
second training feature
first training feature
Correct predictions
Incorrect Predictions

In all of the above graphs the model parameters are taken the same way they are elsewhere *(Appendix a.2).* All graphs on the left with requested data exactly from assignment question are using *(Appendix a.3.4)*. The graphs on the right split the data by correct / incorrect prediction and are using *(Appendix a.4.3).* All three functions previously explained elsewhere.

When C = 0.001

When we take our coefficient values 3.01278330x 10^{-4} and 0.3.19746832x 10^{-1} and the intercept 0.30527602, if we use the same method as described in part a (iii) we can plot our decision boundary *(Appendix a.3.2) & (Appendix a.3.3).* We can see in the diagram that the decision boundary is predicting

The confusion matrix when comparing the model predictions for the test to actual test data is shown below:

True-negative: 6	False-positive: 44
False-negative: 0	True-positive: 150

When C = 1

The same method as above, However, now our decision boundary is placed much better due to the smaller penalty being added. Here it is pretty close to what our logistic regression model predicted and performs significantly better than C = 0.001 and C = 1000. In both of the other cases the penalty value is modified to be too large or small meaning the parameter values do not reach an optimal minimsation within the cost function.

The confusion matrix when comparing the model predictions for the test to actual test data is shown below:

True-negative: 30	False-positive: 20
False-negative: 11	True-positive: 139

When C = 1000

This one threw an error about convergence but also gave parameter values. The graph would indicate it predicts it relatively well however not as good as the previous

The confusion matrix when comparing the model predictions for the test to actual test data is shown below:

True-negative: 45	False-positive: 5
False-negative: 31	True-positive: 119

b (iii)

As the SVM uses a hinge loss function max(0, 1 − y θ^T x) we need to add a penalty to get sensible behavior. Sensible behavior would involve penalizing large values of θ. The penalty is the sum of the squares of the elements of the parameter vector. When we add the penalty we get to the final SVM cost function which is:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} \max(0, 1 - y^{(i)}\theta^T x^{(i)}) + \theta^T\theta/C$$

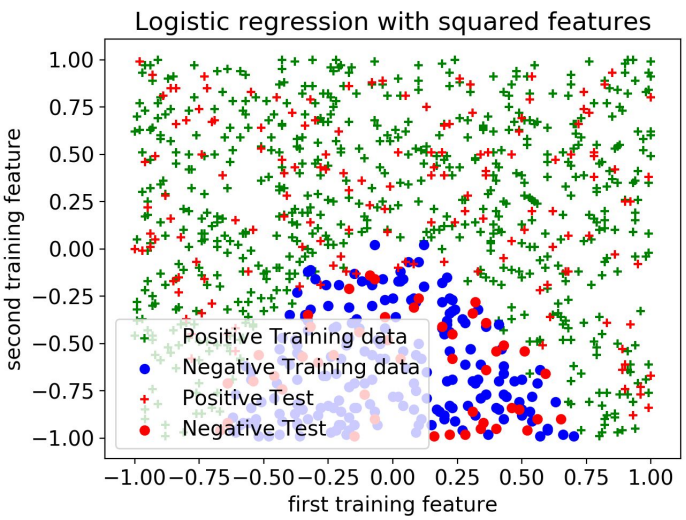
C here represents the hyperparameter we are changing to weigh the penalty, a hyperparameter is a parameter in the cost function rather than the model itself.

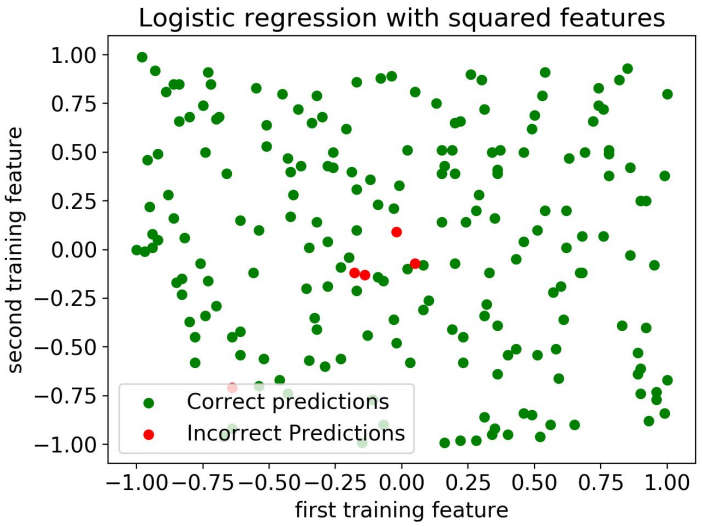
If we use a large C, theta transpose theta / C is very small which requires a lot of minimization of the first half of the cost function. If very small the theta transpose theta will be big which will focus on minimizing the penalty which will make theta values small. Looking at our data as we change the penalty we can see how the decision boundary shifts because of this.

c (i)

Additional features added by squaring each feature as seen in *(Appendix c.1)*. After training the model with the additional features the parameter values are as such: 0.03250029, 5.65256473, 8.22955266, -0.04247932 and the intercept 0.64184462. These values relate to our formula for logistic regression where $z = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4$ +b. As our data seems to have some form of quadratic function for the negative data it makes sense to use some form of feature engineering when constructing the model, which is what we do by adding the additional squared feature. The above formula could also be represented as $z = w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_2^2$ +b. We can then use the model and substitue into the parameter values we found so our prediction formula is: ŷ = sign(0.6418 + $0.0325\, x_1 + 5.652564\, x_2 + 8.22955\, x_1^2 - 0.042479\, x_2^2$)

c (ii)





As we can see when plotting the results of the test data, the model is very accurate. As seen visually in the graph with the correct/incorrect predictions, which was split using the same function as previously mentioned in *(Appendix a.4)*, there are only 5 misclassifications, this is a considerable improvement on previous models. When looking at the confusion matrix we see we have 4 false positives and only one false negative for our test data.

True-negative: 46	False-positive: 4
False-negative: 1	True-positive: 149

This would prove our feature engineering has been a success and that the addition of the squared features added a quadratic shape to our prediction boundary which resulted in the better predictions.

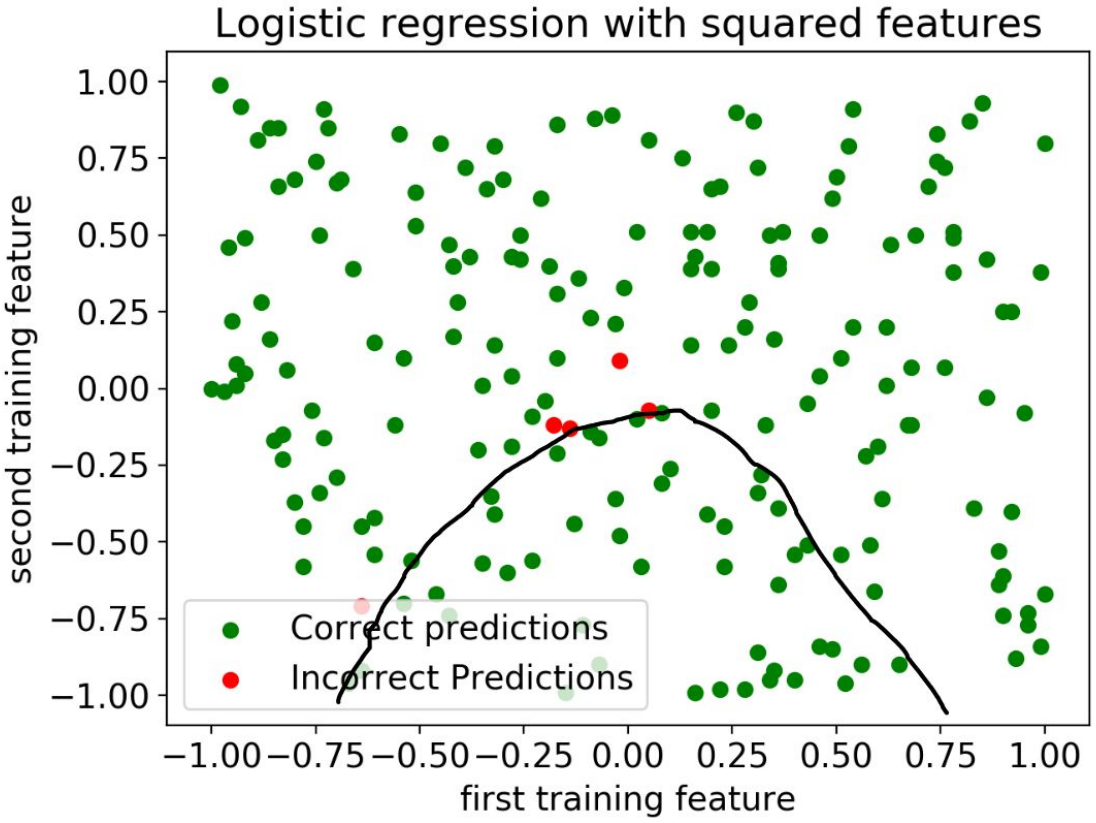
c (iii)

I choose the most common class as the baseline to compare my model against. I wrote a function to find the most common class by summing the actual values of the array and returning 1 as the most common class if the sum was greater than 0 and -1 if the sum of the array was less than 0, in the case of there being an even number it would return 0 indicating there is no common class and this baseline comparison would not work *(Appendix c.3)*. I think this is a reasonable baseline to compare to as if we cannot at least beat this figure the model does not have much use. This would be totally dependent on what we are trying to predict though as we do not know the nature of this data it is hard to pick a baseline. In the case of predicting medical tests, for example, giving everyone a positive result would not be of much use - in that case, it would make sense to not even test them. However, there are also other cases where false negatives may want to be avoided such as diagnosing cancer and we may be more comfortable giving a false positive. Without knowing the underlying nature of the data, I think a baseline model predicting the most common case is sufficient to compare to our model to give a general idea of how accurate it is. If the baseline model was to predict the most common class (positive) for the data we end up with 50 false positives and 150 true positives. The model, therefore, has an overall accuracy of predicting correct data points of 150/200 = 75% As we can see below in the confusion matrix see the false positives and true positives being predicted for every test value and no negatives.

True-negative: 0	False-positive: 50
False-negative: 0	True-positive: 150

c (iv)

Not sure how to do this but I've drawn what it would look like and attached it. It would be a quadratic function that looks similar to below. I will explain why in the attempt to gain some marks :). As the negative data seems to be quadratic in nature us engineering the features to add two quadratic parameters would allow our logistic regression model to perform better and curve. This is also only a two dimensional representation, while in reality there are numerous more plains occurring with the additional aded features we have engineered.



Appendix

(a.1)

```
4 references
def split(X1,X2,y):
    """ Splits data of two features (X1,X2) into positive and negative data point arrays
        using result of data (y). Returns two arrays (positive and negative data point pairs)

    """

    X1positive = []
    X2positive = []

    X1negative = []
    X2negative = []

    # iterate over both
    for (x1i, x2i, yi) in zip(X1, X2, y):


        if(yi == 1):
            # if positive add to positive array
            X1positive.append(x1i)
            X2positive.append(x2i)
        else:
            # if negative add to positive array
            X1negative.append(x1i)
            X2negative.append(x2i)

    Xpositive = np.column_stack((X1positive,X2positive))
    Xnegative = np.column_stack((X1negative,X2negative))

    return Xpositive, Xnegative
```

(a.1.2)

```
positive, negative = split(X1,X2,y)

# plot data
plt.title('Visualisation of data')
plt.xlabel('value of first feature')
plt.ylabel('value of second feature')
plt.scatter(positive[:,0],positive[:,1], label="Positive",color='g', marker='+')
plt.scatter(negative[:,0],negative[:,1], label="Negative",color='b', marker='o')
plt.legend()
plt.show()
```

(a.2)

```
# all attributes
classes = model.classes_
coef = model.coef_
intercept = model.intercept_

print('classes : ,' ,classes)
print('coef : ,' ,coef)
print('intercept : ,' ,intercept)
```

(a.3.1)

```
#train model
model = LogisticRegression()
model.fit(X, y)
y_pred = model.predict(X_test)
```

(a.3.2)

```
def add_decision_boundary_to_plot(b,w1,w2):
    """

    this code is derived from https://scipython.com/blog/plotting-the-decision-boundary-of-a-logistic-regression-m

    """

    # get intercept and gradient

    c,m = calculate_intercept_and_gradient_from_model_parameters(b,w1,w2)

    # Plot decision boundary.
    xmin, xmax = -1, 1
    ymin, ymax = -1, 1
    xd = np.array([xmin, xmax])
    yd = m*xd + c
    plt.plot(xd, yd, 'k', lw=1, ls='--', label="Decision Boundary")
    plt.fill_between(xd, yd, ymin, color='tab:blue', alpha=0.2)
    plt.fill_between(xd, yd, ymax, color='tab:orange', alpha=0.2)
```

(a.3.3)

```
1 reference
def calculate_intercept_and_gradient_from_model_parameters(b,w1,w2):
    """

    this code is derived from https://scipython.com/blog/plotting-the-decision-boundary-o

    """

    # Calculate the intercept and gradient of the decision boundary.
    c = -b/w2
    m = -w1/w2
    return c, m
```

(a.3.4)

```
6 references
def plot_training_and_test_data_graph(positive_train,negative_train,positive_test,negative_test,title):
    plt.title(title)
    plt.xlabel('first training feature')
    plt.ylabel('second training feature')
    plt.scatter(positive_train[:,0],positive_train[:,1], label="Positive Training data",color='g',marker='+')
    plt.scatter(negative_train[:,0],negative_train[:,1], label="Negative Training data",color='b')
    plt.scatter(positive_test[:,0],positive_test[:,1], label="Positive Test",color='r',marker='+')
    plt.scatter(negative_test[:,0],negative_test[:,1], label="Negative Test",color='r',marker='o')
    plt.legend()
    plt.show()
```

(a.4.1)

```
def split_by_correct_prediction(X1,X2,y,y_pred):
    """ Splits data by correct / incorrect prediction

    """

    X1correct = []
    X2correct = []

    X1incorrect = []
    X2incorrect = []

    # iterate over both
    for (x1i, x2i, yi, yi_pred) in zip(X1, X2, y, y_pred):

        if(yi == yi_pred):
            # if positive add to positive array
            X1correct.append(x1i)
            X2correct.append(x2i)
        else:
            # if negative add to positive array
            X1incorrect.append(x1i)
            X2incorrect.append(x2i)

    Xcorrect = np.column_stack((X1correct,X2correct))
    Xincorrect = np.column_stack((X1incorrect,X2incorrect))

    return Xcorrect, Xincorrect
```

(a.4.2)

```
# Show the Confusion Matrix
print(confusion_matrix(y_test, y_pred))
```

(a.4.3)

```
0 references
def plot_correct_incorrect_graph(Xcorrect,Xincorrect,title):
    plt.title(title)
    plt.xlabel('first training feature')
    plt.ylabel('second training feature')
    plt.scatter(Xcorrect[:,0],Xcorrect[:,1], label="Correct predictions",color='orange')
    plt.scatter(Xincorrect[:,0],Xincorrect[:,1], label="Incorrect Predictions",color='yellow')
    plt.legend()
    plt.show()
```

(c.1)

```
# Read in data
df = pd.read_csv('week2.csv')
print(df.head())
X1=df.iloc[: ,0]
X2=df.iloc[: ,1]
y=df.iloc[: , 2]

# add additional features by squaring
X1sq = X1 * X1
X2sq = X2 * X2
X=np.column_stack(( X1, X2, X1sq, X2sq ))

# split data for test/train
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

model = LogisticRegression()
model.fit(X, y)
```

(c.3)

```
    # Compare against baseline that always predicts most common case
    most_common_case = get_most_common_case(y)

    print('most common case : ', most_common_case)

    # fill with most common case
    baseline_prediction = np.ones(len(y_test))

    # compare predictions to actual to show results of model on test data
    positive_test,negative_test = split(X_test[:,0],X_test[:,1],baseline_prediction)

    plot_training_and_test_data_graph(positive_train,negative_train,positive_test,negative_test,'Baseline Comparison')

    Xcorrect, Xincorrect = split_by_correct_prediction(X_test[:,0],X_test[:,1],y_test, baseline_prediction)

    plot_correct_incorrect_graph(Xcorrect,Xincorrect,'Baseline Comparison')

    # Show the Confusion Matrix
    print(confusion_matrix(y_test, baseline_prediction))


    # Try to plot decision boundary

1 reference
def get_most_common_case(y):
    sum = np.sum(y)
    if sum > 0:
        return 1
    if sum < 0:
        return -1
    return 0
```

(r.1)

- https://scipython.com/blog/plotting-the-decision-boundary-of-a-logistic-regression-model/
-