Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

School of Computer Science and Statistics

CS7CS4 Machine Learning Week 4 Assignment

Boris Flesch
20300025

November 6, 2020

MSc Computer Science, Intelligent Systems

1 Downloaded Dataset

id:23-46–23-0

id:23–23-23-0

2 Questions

2.1 Part (i)

(a) Before training a Logistic Regression classifier, it can be useful to visualise the data. We can plot it on a graph with features X1 and X2 respectively on x- and y-axis, while distinguishing the output class -1 or +1 with a specific color. The first training dataset (id:23-46–23-0) gave the following Figure 1:

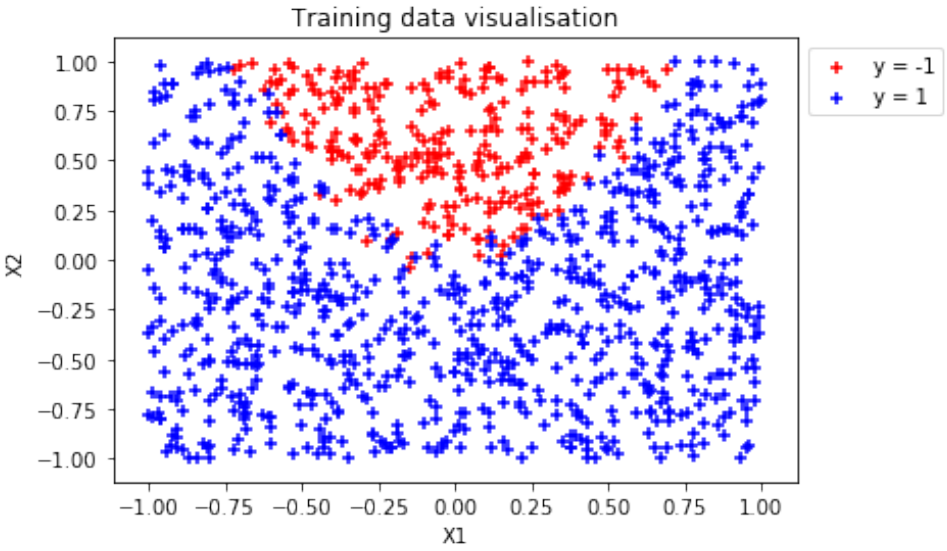

Figure 1

(i) To determine the maximum order of polynomial to use for our Logistic Regression classifier, we can choose a range of values to test regarding the visualisation of our training data. Considering q as the degree of polynomial features to use, we can argue that using $q < 2$ would lead to under-fitting as it would lead to a linear model that is not able to fit the quadratic shape of our training data as observed on Figure 1. As the data does not seem particularly noisy or adopting a behaviour that is not quadratic at some point, we could also exclude the use of any $q > 2$ to avoid over-fitting.

Therefore $q = 2$ seems to be the most appropriate value. To confirm this hypothesis, we can plot the F1 Score against a wider range of values for q (i.e. $q \in \{1, 2, 3\}$). F1 Score is interesting in this case as it measures the effects of both false positives and false negatives. Therefore, choosing a value for q that maximises the F1 Score will allow us to improve the accuracy of our model.

Finally, our Logistic Regression should use a L2 penalty which requires a value for the hyperparameter C. As we will focus on tuning C later, we can evaluate our

range of q values for another wide range of C values (i.e. $C \in \{0.001, 1, 1000\}$)) to have a rough idea of the behaviour of our model regarding C.

This function has been created to evaluate a range of q and C values for our Logistic Regression model while plotting their respective F1 Scores:

```
def plotRangeQandC(X, y, q_range, C_range):
    plt.figure(num=None, figsize=(8, 6), dpi=80)
    for Ci in C_range:
        model = LogisticRegression(penalty="l2", C=Ci,
        ↪ max_iter=1000)
        mean_error, std_error = [], []
        for qi in q_range:
            Xpoly = PolynomialFeatures(qi).fit_transform(X)
            scores = cross_val_score(model, Xpoly, y, cv=5,
            ↪ scoring='f1')
            mean_error.append(np.array(scores).mean())
            std_error.append(np.array(scores).std())
        plt.errorbar(q_range, mean_error, yerr=std_error,
        ↪ linewidth=3, label="C = %.3f"%Ci)
    # [...]
    plt.show()
```

We can use this function to plot the graph for our values; shown on Figure 2 below:

```
plotRangeQandC(X, y, q_range=[1,2,3], C_range=[0.001, 1, 1000])
```

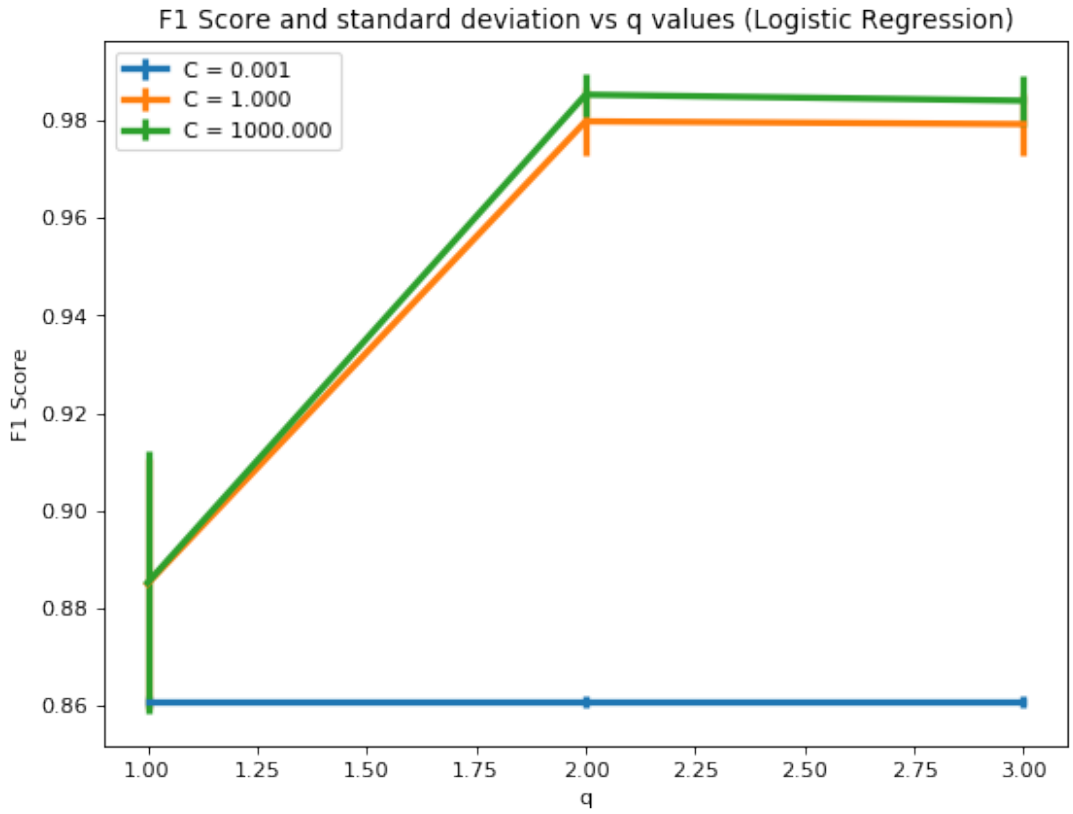

Figure 2

For $C = 0.001$ the graph not relevant. However, for $C = 1$ and $C = 1000$, we can see that $q = 2$ maximises F1 Score (as expected). Using $q = 1$ would obviously lead to under-fitting as we can see on the graph with the small F1 Score high standard deviation. $q > 2$ would tend to over-fitting: the F1 Score slightly begins to decrease between $q = 2$ and $q = 3$; even if the difference is not huge.

To confirm the choice of $q = 2$ for our model, we can also take a look at the predictions it gives. To do that, we can split our training data to obtain a test dataset (i.e. 80/20 split with *train_test_split* sklearn function) and plot the prediction surface of our trained model against the real outputs of the test dataset. Instead of trying to plot a decision boundary, we can plot a decision surface using a *meshgrid*. This will also be useful when discussing kNN later as there is not a "single decision boundary" in this case.

The code below has been adapted from an example of classifier plot from sklearn documentation for this use case:

```
def plotPredictions(X, y, q, model, title="Prediction surface on test data"):
    Xpoly = PolynomialFeatures(q).fit_transform(X)
    Xtrain, Xtest, ytrain, ytest = train_test_split(Xpoly,y,test_size=0.2)
    Xtest_m1 = Xtest[np.where(ytest == -1)]
    Xtest_p1 = Xtest[np.where(ytest == 1)]

    model.fit(Xtrain, ytrain)
    cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
    meshStep = .01

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    x_min, x_max = Xtest[:, 1].min() - 1, Xtest[:, 1].max() + 1
    y_min, y_max = Xtest[:, 2].min() - 1, Xtest[:, 2].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, meshStep), np.arange(y_min, y_max, meshStep))
    Xtest = np.c_[xx.ravel(), yy.ravel()]
    Xtest = PolynomialFeatures(q).fit_transform(Xtest)
    Z = model.predict(Xtest).reshape(xx.shape)
    plt.figure(num=None, figsize=(8, 6), dpi=80)
    plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

    plt.scatter(Xtest_m1[:, 1], Xtest_m1[:, 2], c='r', marker='+', label="y = -1")
    plt.scatter(Xtest_p1[:, 1], Xtest_p1[:, 2], c='b', marker='+', label="y = 1")

    plt.gca().set(title=title, xlabel='X1', ylabel="X2")

```

```
    handles, labels = plt.gca().get_legend_handles_labels()
    handles.append(mpatches.Patch(color='#FFAAAA', label='y_pred = -1'))
    handles.append(mpatches.Patch(color='#AAAAFF', label='y_pred = 1'))
    plt.legend(handles=handles)
    plt.show()
```

The following plotting of predictions therefore give us Figures 3 and 4:

```
model = LogisticRegression(penalty="l2", C=1)
plotPredictions(X, y, q=1, model=model, title="Prediction surface on test data (LogisticRegression, q=1, C=1)")
plotPredictions(X, y, q=2, model=model, title="Prediction surface on test data (LogisticRegression, q=2, C=1)")
```

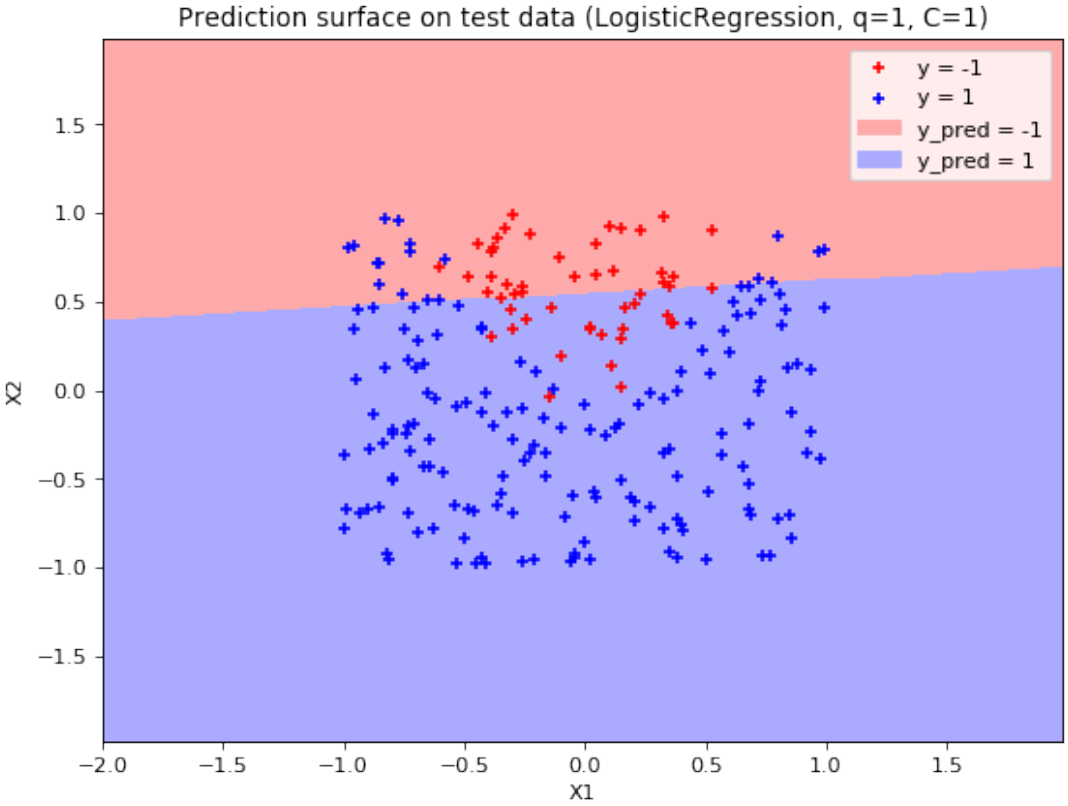

Figure 3

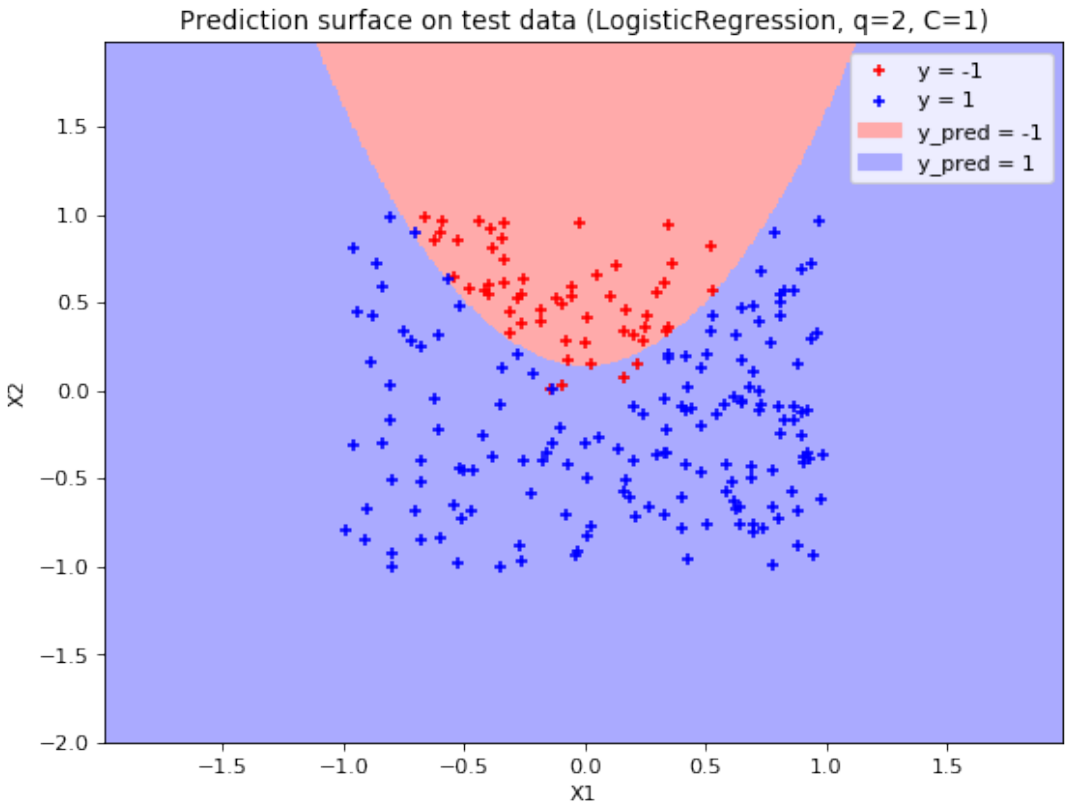

Figure 4

By observing how our model behaves and follows the quadratic behaviour of our data on Figure 4, we can definitely argue that $q = 2$ is the most appropriate value to use.

(ii) The L2 penalty encourages small (but non-zero) parameters values. The weight C is an hyperparameter which can be adjusted to modify the intensity of this penalty. Bigger is C, smaller will be the penalty applied to parameter values and vice versa.

In this case, we can choose a wide range of values for C and plot the F1 Score of the trained model against its C value. As seen on Figure 4, our model already fits very well the data for $q = 2$ and $C = 1$. Altering the value of C should not have much impact on the accuracy of the model. We can use $C \in \{1, 10, 100, 1000\}$ and the *cross_val_score* function to cross-validate F1 with 5-fold cross-validation, which will allow us to keep 20% of our original data as test data (i.e. 80/20 split):

```
def plotRangeC(X, y, q, C_range, printParameters=False):
    Xpoly = PolynomialFeatures(q).fit_transform(X)
    mean_error, std_error = [], []
    for Ci in C_range:
        model = LogisticRegression(penalty="l2", C=Ci,
        ↪ max_iter=1000)
        scores = cross_val_score(model, Xpoly, y, cv=5,
        ↪ scoring='f1')
        mean_error.append(np.array(scores).mean())
        std_error.append(np.array(scores).std())
```

```
9
10          if (printParameters):
11              model.fit(Xpoly, y)
12              theta = np.insert(model.coef_, 0, model.intercept_)
13              print("C = %.1f"%Ci)
14              print(" =", theta)
15
16      fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
17      plt.errorbar(C_range, mean_error, yerr=std_error,
        ↪  linewidth=3)
18      plt.title("F1 Score and standard deviation vs C values
        ↪  (Logistic Regression)")
19      plt.gca().set(xlabel='C', ylabel="F1 Score")
20      plt.show()
21
22  plotRangeC(X, y, q=2, C_range=[1, 5, 10, 50, 100, 500, 1000])
```

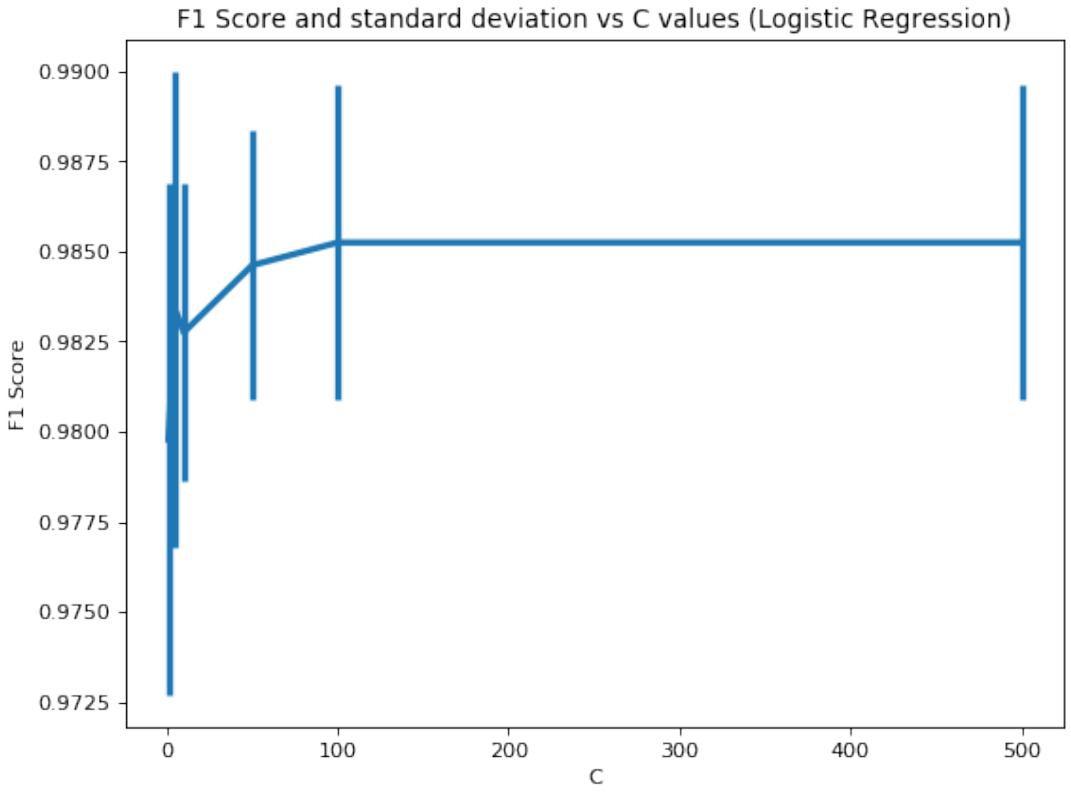

Figure 5

Given the result obtained in Figure 5, we can see as expected that the delta in the F1 Scores is very small; from roughly 0.9825 to 0.9850. To maximise this score, we could use any value of $C \geq 100$. To avoid over-fitting, the best option is to opt for the simplest model (i.e. the smallest value for C). Hence $C = 100$ is the most appropriate value for this Logistic Regression classifier.

(b) kNN classifiers are instance-based models; they base their predictions on the training data. In other words, there is no notion of "following the behaviour of the data" as we would

have for the decision boundaries of Logistic Regression classifiers. Hence augmenting the features with polynomial features for a kNN would have only a very small impact as features would remain proportional and therefore create drastic change in the neighborhood of some specific data points. We can easily check that by plotting different values of q when cross-validating the choice of k.

Generally speaking, increasing k will smooth out the function and can potentially lead to under-fitting. Decreasing k will make the function more complex by tracking data points more closely; which can also mean fitting noise and therefore lead to over-fitting. A straightforward approach for choosing k would be to compute the F1 Score of a range of values for k and observe the its evolution for each trained model. If nothing relevant is obtained from this first observation (i.e. peak or pit in F1 Score, repeating pattern, etc.), we could extend our range of values for k.

The following code will allow us to plot F1 Scores (again using *cross_val_score*) for a range of k, and also q to ensure that adding polynomial features will not impact neither our classifier nor our decision:

```
def plotRangeKandQ(X, y, q_range, k_range):
    fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
    q_range = [1,2,3]
    for qi in q_range:
        Xpoly = PolynomialFeatures(qi).fit_transform(X)
        mean_f1, std_f1 = [], []
        for ki in k_range:
            model =
            ↪ KNeighborsClassifier(n_neighbors=ki,weights='uniform')
            scores = cross_val_score(model, Xpoly, y, cv=5,
            ↪ scoring='f1') # cv -> KFold
            mean_f1.append(np.array(scores).mean())
            std_f1.append(np.array(scores).std())
        plt.errorbar(k_range, mean_f1, yerr=std_f1, linewidth=3,
        ↪ label='q = %d'%qi)

    # [...]
    plt.show()
```

We can use this method with $k \in \{2, 3, 5, 7, 10, 13, 15, 17, 20\}$ and $q \in \{1, 2, 3\}$:

```
plotRangeKandQ(X, y, q_range=[1,2,3],
↪ k_range=[2,3,5,7,10,13,15,17,20])
```

Figure 6

First of all, results from Figure 6 above show us that augmenting features with polynomial features can introduce slight variations in the classifier. However, its behaviour (i.e. evolution of F1 Score) remains the same whatever the value of q.

In addition to that, we can see that F1 Score tends to increase until $k = 5$ and then decrease. That means that $k = 5$ maximises F1 Score and therefore would be the most appropriate value for our kNN.

We can plot the prediction surface of this kNN classifier with the following code (using the *plotPredictions* function introduced earlier):

```
model = KNeighborsClassifier(n_neighbors=5,weights='uniform')
plotPredictions(X, y, q=2, model=model, title="Prediction surface on
↪ test data (kNN, k=5)")
```

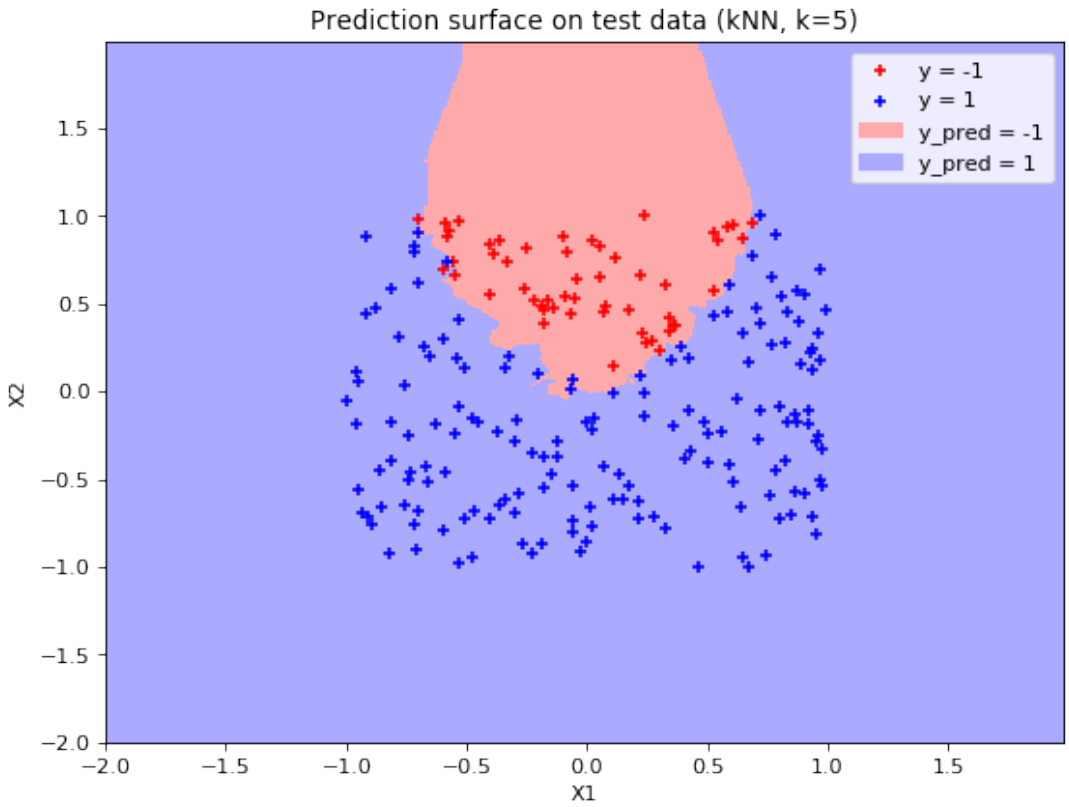

Figure 7

Figure 7 above shows us that our kNN classifier provides accurate predictions on test data, although it does not seem to follow the quadratic behaviour of the data as much as the previous Logistic Regression classifier, especially outside the scope of the training data.

(c) The confusion matrices of each model can be generated using the *plot_confusion_matrix* from sklearn. We can use the following generic function to plot the matrix of a list of given models and a given value for q (i.e. polynomial features):

```
def confusionMatrixes(X, y, q, models):
    Xpoly = PolynomialFeatures(q).fit_transform(X)
    Xtrain, Xtest, ytrain, ytest =
    ↪ train_test_split(Xpoly,y,test_size=0.2)
    for model, title in models:
        model.fit(Xtrain, ytrain)
        disp = plot_confusion_matrix(model, Xtest, ytest,
        ↪ display_labels=["y = -1", "y = 1"], cmap=plt.cm.Blues,
        ↪ values_format = 'd')
        disp.ax_.set_title("Confusion matrix for " + title)
        plt.show()
```

This function also splits the original data so that confusion matrices are created based on a sample of 20% of the original data (i.e. 80/20 split similar to the split used in 5-fold cross-validation earlier). Passing all models at once as a parameter of this function also ensures that the confusion matrices are generated using the same test data for each

model.

We can use this function to plot confusion matrices for the Logistic Regression classifier, the kNN classifier and two dummy classifiers with "most_frequent" and "uniform" (i.e. random at uniform) predictions. Having two baseline classifiers (i.e. the dummy ones) is interesting as depending on the real use case we could want to find the best trade-off between true positives and false negatives: hence a "most frequent" baseline classifier will give us both a true positive and false positive rate of 1, while a "random" classifier will basically provide random rates, accuracy and precision.

```
models = [[LogisticRegression(penalty="l2", C=100), 'Logistic Regression...'],
          [KNeighborsClassifier(n_neighbors=5,weights='uniform'), '...'],
          [DummyClassifier(strategy="most_frequent"), '...'],
          [DummyClassifier(strategy="uniform"), '...']
confusionMatrices(X, y, q=2, models)
```

This snippet gives us the following confusion matrices (Figures 8 to 11):

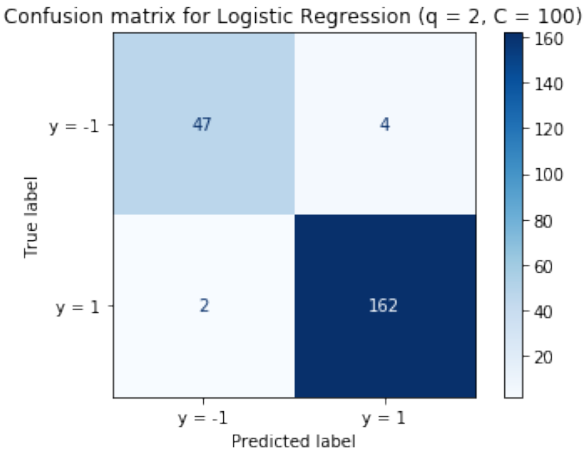

Figure 8

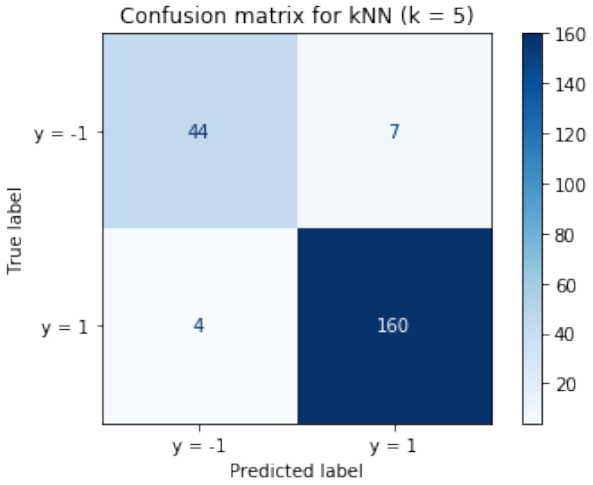

Figure 9

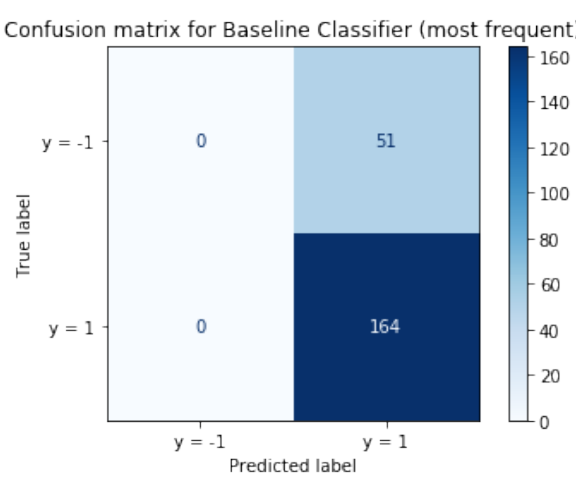

Figure 10

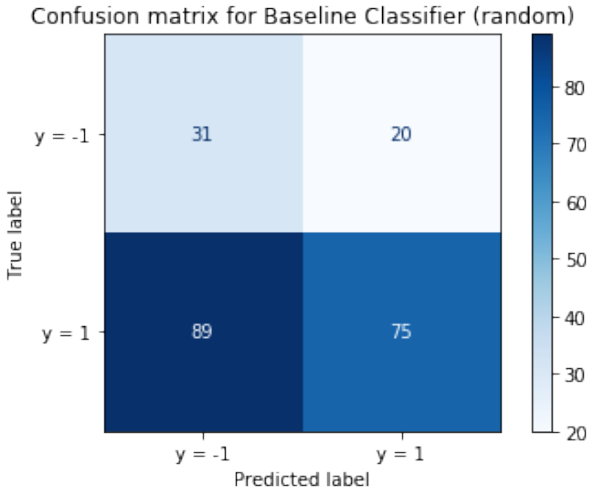

Figure 11

(d) *roc_curve* function from sklearn allows us to easily plot the ROC curve of our different models. This function takes the true y labels and the probability estimates of the positive class as input parameters. Therefore, the approach with LogisticRegression is straightforward as it provides a *decision_function* method. For other classifiers (i.e. kNN, DummyClassifier), we can use the *predict_proba* function and isolate the column containing only probabilities for positive class.

The following function plots ROC curves for each classifier mentioned above (i.e. the same as for confusion matrices) for given q, C and k values:

```
1  def plotRocCurves(X, y, q, C, k):
2      fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
3
4      Xpoly = PolynomialFeatures(q).fit_transform(X)
5      Xtrain, Xtest, ytrain, ytest =
       ↪  train_test_split(Xpoly,y,test_size=0.2)
6
7      model = LogisticRegression(penalty="l2", C=100).fit(Xtrain,
       ↪  ytrain)
8      fpr, tpr, _ = roc_curve(ytest,model.decision_function(Xtest))
9      plt.plot(fpr,tpr, label='Logistic Regression (q = %d, C =
       ↪  %.3f)'%(q,C))
10
11     model =
       ↪  KNeighborsClassifier(n_neighbors=k,weights='uniform').fit(Xtrain,
       ↪  ytrain)
12     fpr, tpr, _ = roc_curve(ytest,model.predict_proba(Xtest)[:,1])
13     plt.plot(fpr,tpr, label='kNN (k = %d)'%k)
14
15     #[...] (repeat for dummy classifiers)
16
17     plt.show()
```

Using this function *plotRocCurves* with the following parameters gives Figure 12:

```
plotRocCurves(X, y, q=2, C=100, k=5)
```

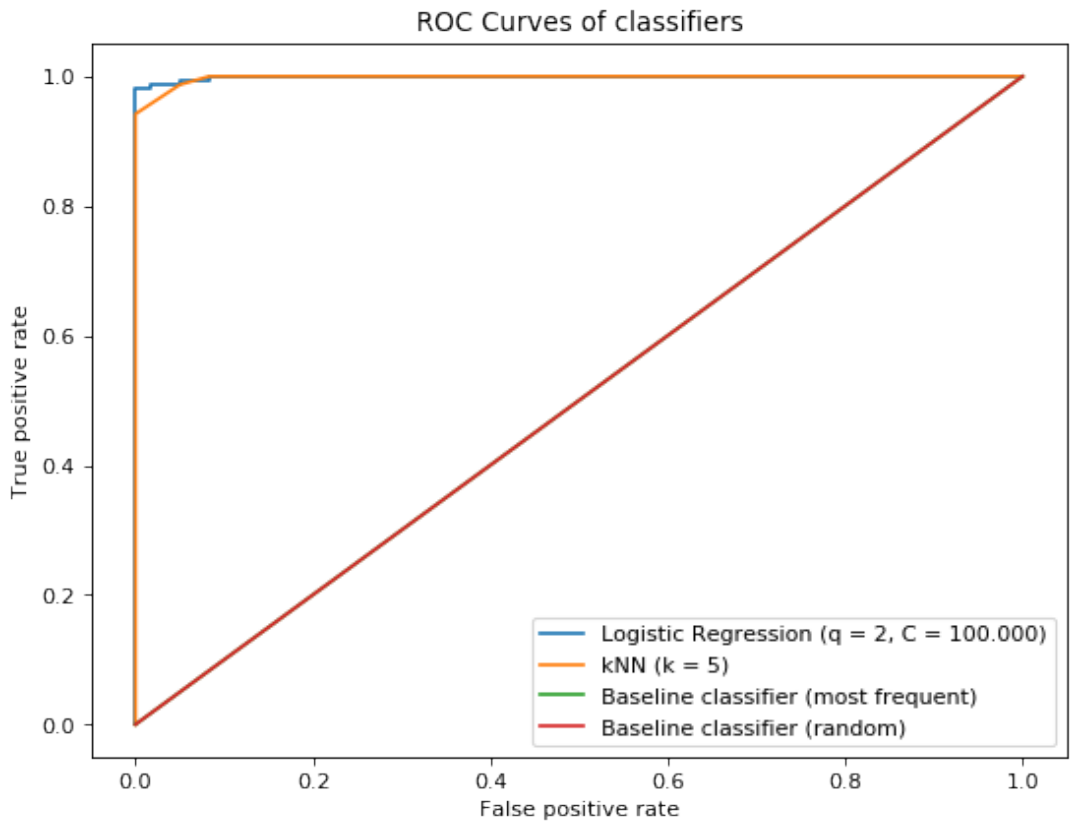

Figure 12

(e) First, we can calculate the following measurements for each confusion matrix (i.e. for each model):

- Logistic Regression ($q = 2$, $C = 100$):
  - $Accuracy = \frac{TN+TP}{TN+TP+FN+FP} = \frac{162+47}{4+162+2+47} = 0.97$
  - $True\ positive\ rate = \frac{TP}{TP+FN} = \frac{162}{162+2} = 0.98$
  - $False\ positive\ rate = \frac{FP}{TN+FP} = \frac{4}{47+4} = 0.07$
  - $Precision = \frac{TP}{TP+FP} = \frac{162}{162+4} = 0.97$
- kNN ($k = 5$):
  - $Accuracy = \frac{160+44}{7+160+4+44} = 0.94$
  - $True\ positive\ rate = \frac{160}{160+4} = 0.97$
  - $False\ positive\ rate = \frac{7}{44+7} = 0.13$
  - $Precision = \frac{160}{160+7} = 0.95$
- Baseline Classifier (most frequent):

  - *Accuracy* $= \frac{164+0}{51+164+0+0} = 0.76$
  - *True positive rate* $= \frac{164}{164+0} = 1$
  - *False positive rate* $= \frac{51}{0+51} = 1$
  - *Precision* $= \frac{164}{164+51} = 0.76$
- Baseline Classifier (random):
  - *Accuracy* $= \frac{75+31}{20+75+89+31} = 0.49$
  - *True positive rate* $= \frac{75}{75+89} = 0.45$
  - *False positive rate* $= \frac{20}{31+20} = 0.39$
  - *Precision* $= \frac{75}{75+20} = 0.78$

The ideal model would have a true positive rate of 1 and a false positive rate of 0. In our case, we can see that both Logistic Regression and kNN classifiers are performing better than the two baseline models. In addition to that, their respective precision is greater than the one of the baseline models. Finally, as our data is quite well balanced, using the accuracy is also a good indicator of the performance of our classifiers. We can therefore argue that both our models are suitable for this data.

Comparing only kNN to Logistic Regression, we can see that the Logistic Regression model performs slightly better: better accuracy, better precision and smaller false positive rate. We can also visualise that with the true positive rate of the Logistic Regression classifier on the ROC plot (Figure 12): the ideal classifier gives points in the top-left corner of this plot. It is also worth noting that some tests have shown kNN performing exactly as well as Logistic Regression. Also, we can see that the ROC curves of both baseline classifiers are overlapping on the 45° line.

In conclusion, although both models would be reasonable, I would recommend using the Logistic Regression model which performs slightly better and seems to better follow the quadratic curve of the data (Figure 4) outside the scope of the test data.

2.2 Part (ii)

Note: this Part (ii) uses functions of the code that have already been introduced in Part (i) and are therefore not mentioned/explained in this section again.

(a) The second training dataset (id:23–23-23-0) gives the following Figure 13:

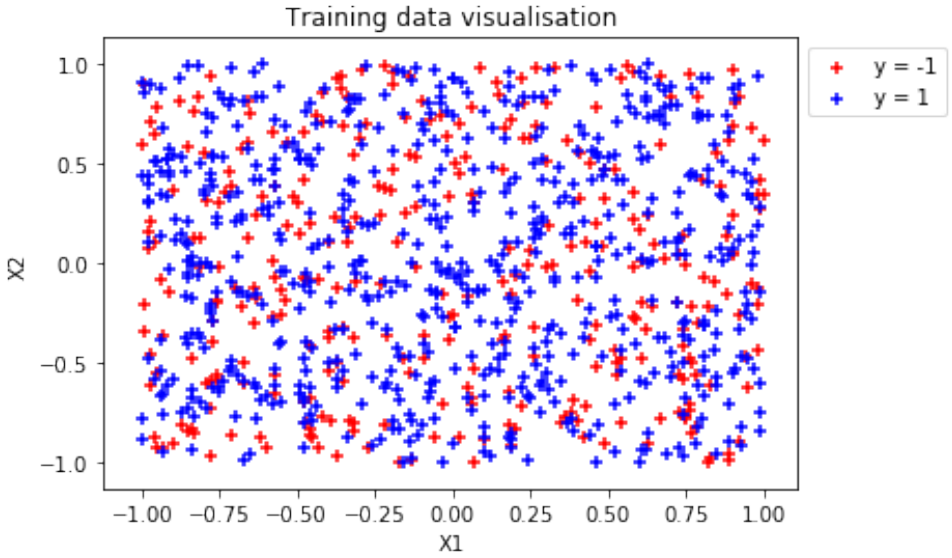

Figure 13

(i) This second dataset is extremely noisy and does not seem to follow any behaviour or pattern that may help classifying the data. Hence adding polynomial feature would only lead to capture noise (i.e. follow some points of the dataset). We could therefore assume that $q = 1$ is the best option. However, to have a rough idea of the behaviour of our model for $q > 1$, we can test a bigger range of values such as $q \in \{1, 2, 3, 4, 5\}$ and plot the F1 Score of each value. As done in Part (i), we can do this evaluation for a wide range of C values that we will focus on later (i.e. $C \in \{0.001, 1, 1000\}$):

```
plotRangeQandC(X, y, q_range=[1,2,3,4,5], C_range=[0.001, 1,
↪  1000])
```

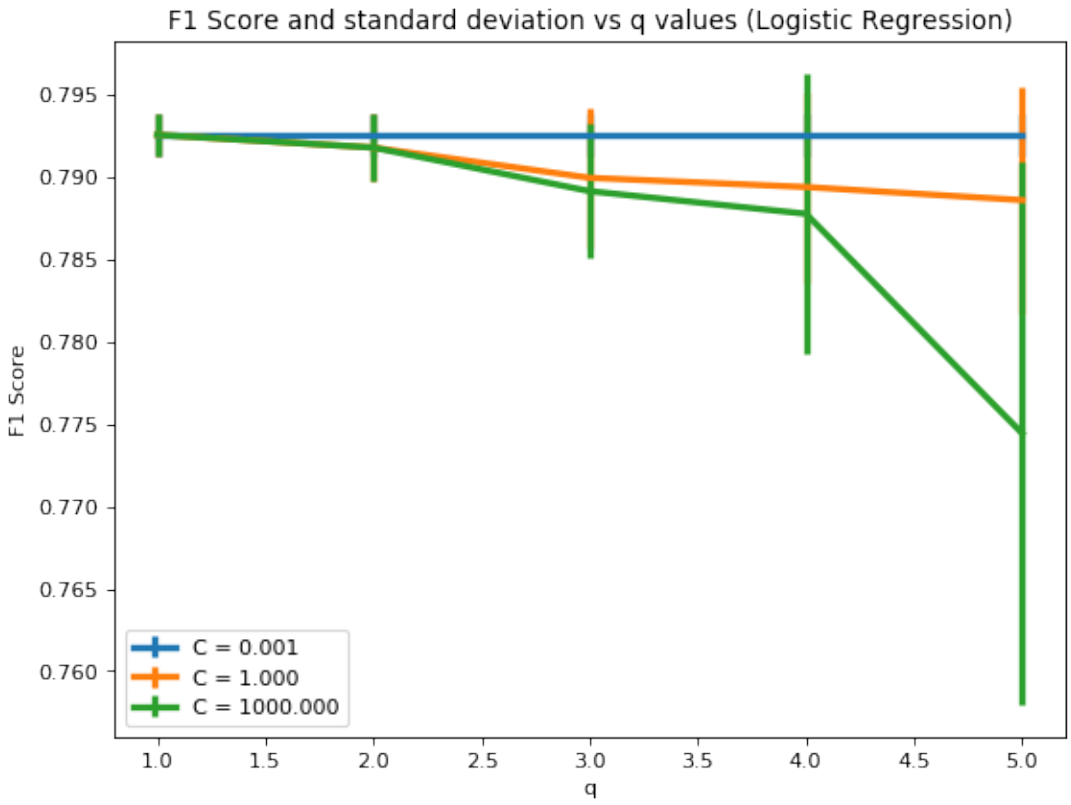

Figure 14

Figure 14 above shows us that the F1 Score of our classifier decreases when q increases. We can therefore confirm our hypothesis of keeping $q = 1$ for this classifier to maximise F1 Score.

As the data is very noisy, it is also interesting to visualise what happens for predictions given by the classifier. Figures 15 and 16 show us predictions for $q = 1$ and $q = 2$ using the following code:

```
model = LogisticRegression(penalty="l2", C=1)
plotPredictions(X, y, q=1, model=model, title="Prediction surface
↪  on test data (LogisticRegression, q=1, C=1)")
plotPredictions(X, y, q=2, model=model, title="Prediction surface
↪  on test data (LogisticRegression, q=2, C=1)")
```

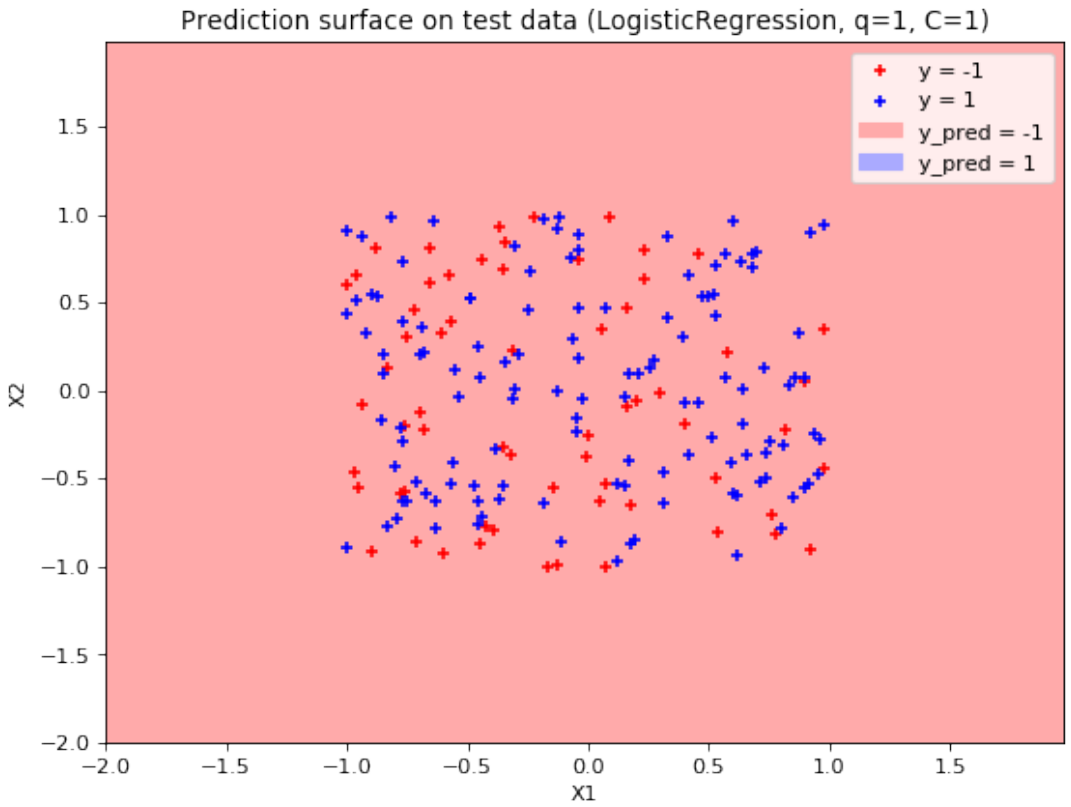

Figure 15

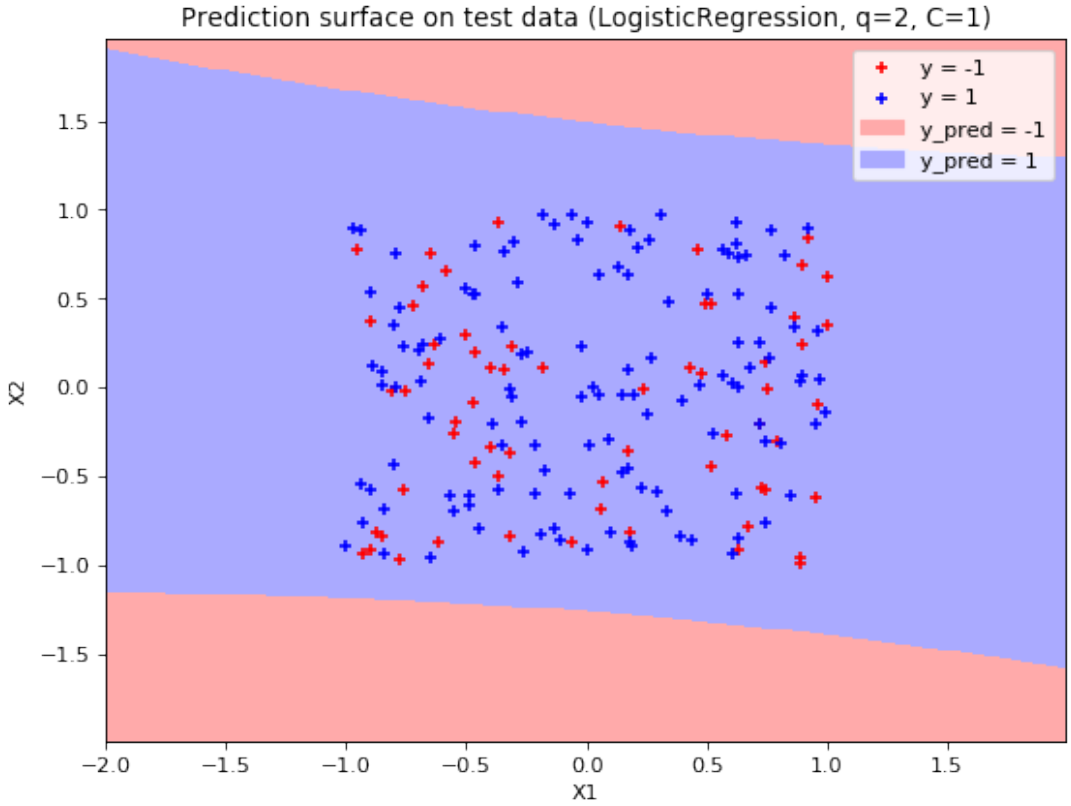

Figure 16

It is interesting to note that both classifiers are predicting a single class in the scope of the training data. This confirms the noisiness of the data and the difficulty for our classifiers to find any behaviour to follow.

(ii) The hyperparameter C is supposed to add penalty to the parameter values of our model. However, due to the noise of the original data, it is hard to conceive "where to apply penalty". Hence we can choose a wide range of C: $C \in \{1, 10, 100, 1000\}$ and plot the F1 Scores using the following code (output Figure 17):

```
plotRangeC(X, y, q=1, C_range=[0.1, 1, 10, 100],
↪  printParameters=True)
```

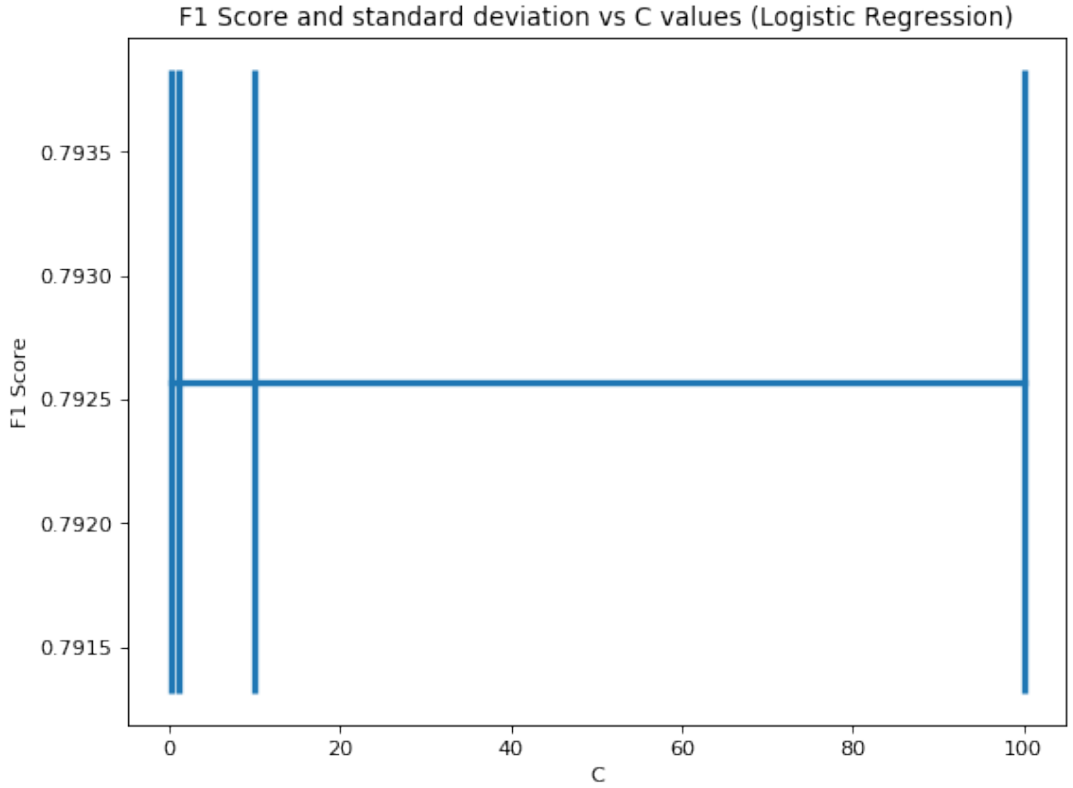

Figure 17

As we can see on the Figure 17 above, changing the value of C does not seem to have any impact on our classifier. It is interesting to note that the parameters of the different models are indeed not much changing even for a wide range of C:

- $C = 0.1$: $\theta = \begin{bmatrix} 6.48638046\text{e}{-1} & -1.00520948\text{e}{-5} \\ -1.15050255\text{e}{-2} & 1.26980135\text{e}{-1} \end{bmatrix}$
- $C = 1.0$: $\theta = \begin{bmatrix} 6.48902503\text{e}{-1} & 6.94262229\text{e}{-6} \\ -1.27662173\text{e}{-2} & 1.43048460\text{e}{-1} \end{bmatrix}$
- $C = 10$: $\theta = \begin{bmatrix} 6.49055930\text{e}{-1} & -1.16094044\text{e}{-4} \\ -1.29037472\text{e}{-2} & 1.44909831\text{e}{-1} \end{bmatrix}$
- $C = 100$: $\theta = \begin{bmatrix} 0.32454461 & 0.32439524 \\ -0.01292577 & 0.14506562 \end{bmatrix}$

The penalty should penalise values that are close from the decision boundary of our classifier. But as the data is extremely noisy, it is difficult to penalise any parameter rather than another. Hence tuning C has almost no impact on our classifier: we can keep $C = 1$.

(b) As well as in Part (i), we can first try a small range of values for k to try to identify any interesting behaviour in the evolution of the F1 Score of our classifier. Hence we can start with $k \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ (see Figure 18). Again, we can evaluate k along with a small range of q to ensure that adding polynomial features does not impact our classifier:

```
plotRangeKandQ(X, y, q_range=[1,2,3], k_range=[1,2,3,4,5,6,7,8,9,10])
```

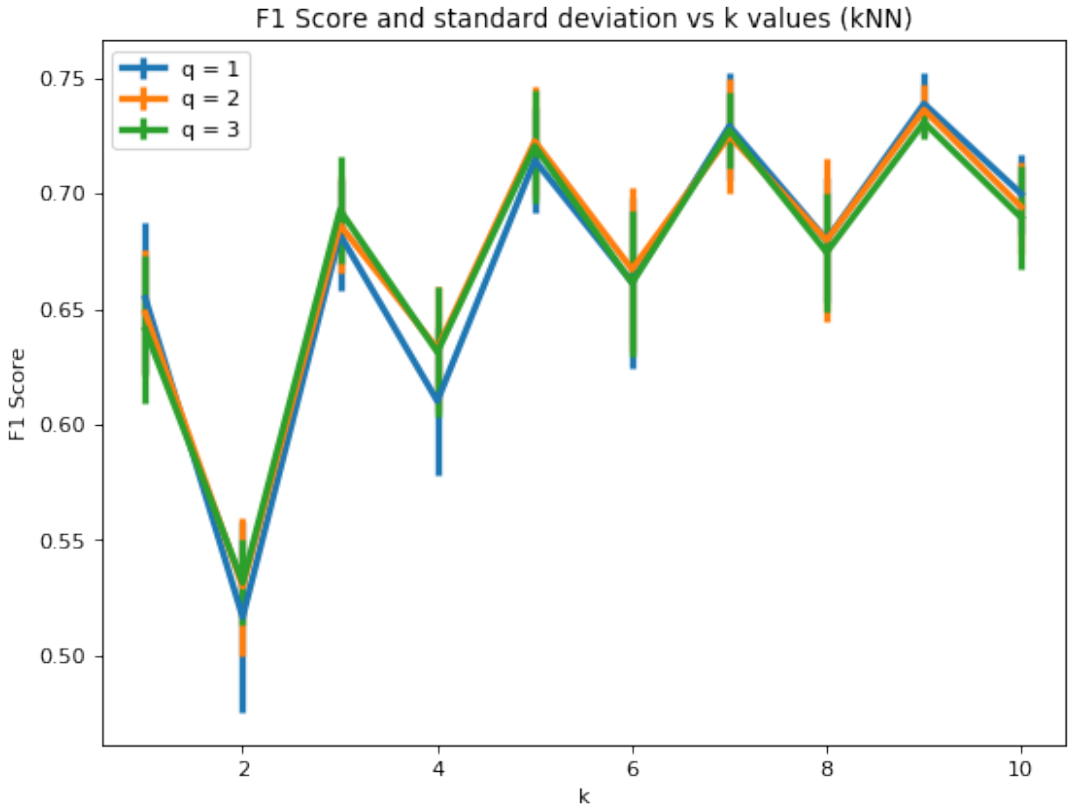

Figure 18

As we can see, the pattern seems to repeat and the F1 Score seems to slowly increase along with the value of k. We can go further by exploring a much wider range of values to see what happens, for example with $k \in \{1, 10, 100, 500\}$ (see Figure 19):

```
plotRangeKandQ(X, y, q_range=[1,2,3], k_range=[1,10,100,500])
```

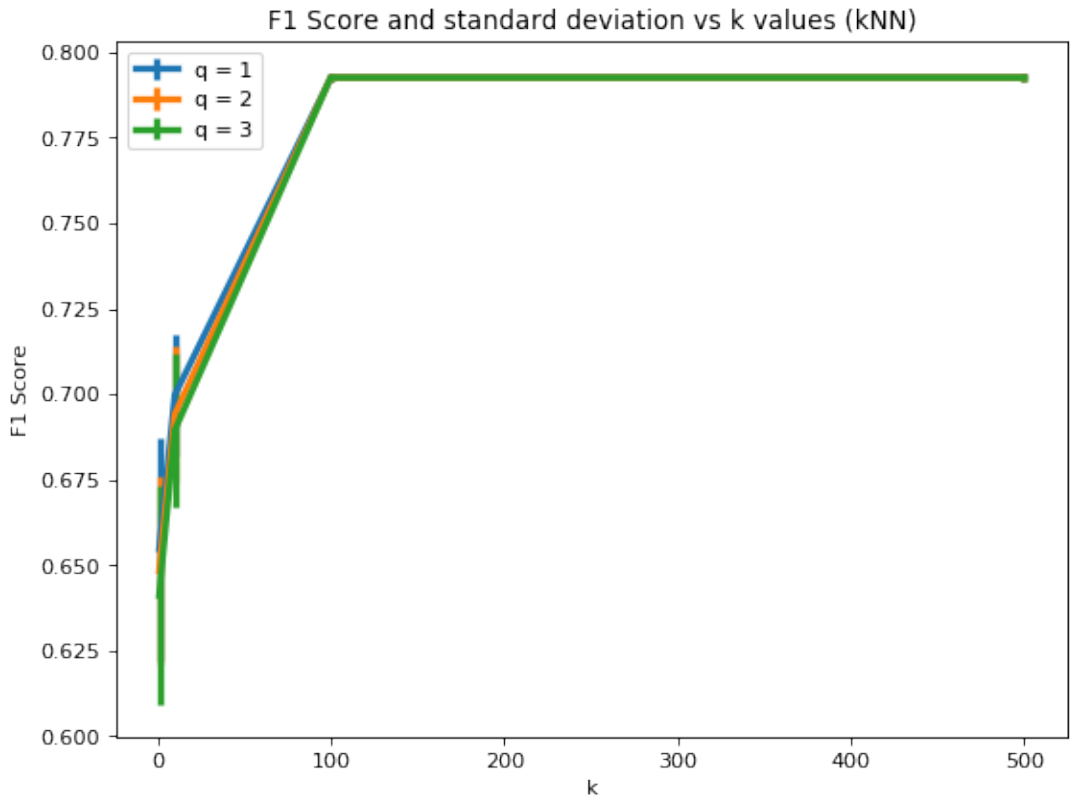

Figure 19

The graph shows us that $k = 100$ seems to be a good value as it maximises F1 Score. Normally, increasing k could lead to under-fitting. But due to the extreme noise of this dataset, it could be an interesting solution and seems more appropriate than relying on only a few neighbors which are approximately equally distributed (see Figure 13).

The predictions given by this classifier are shown on the following Figure 20:

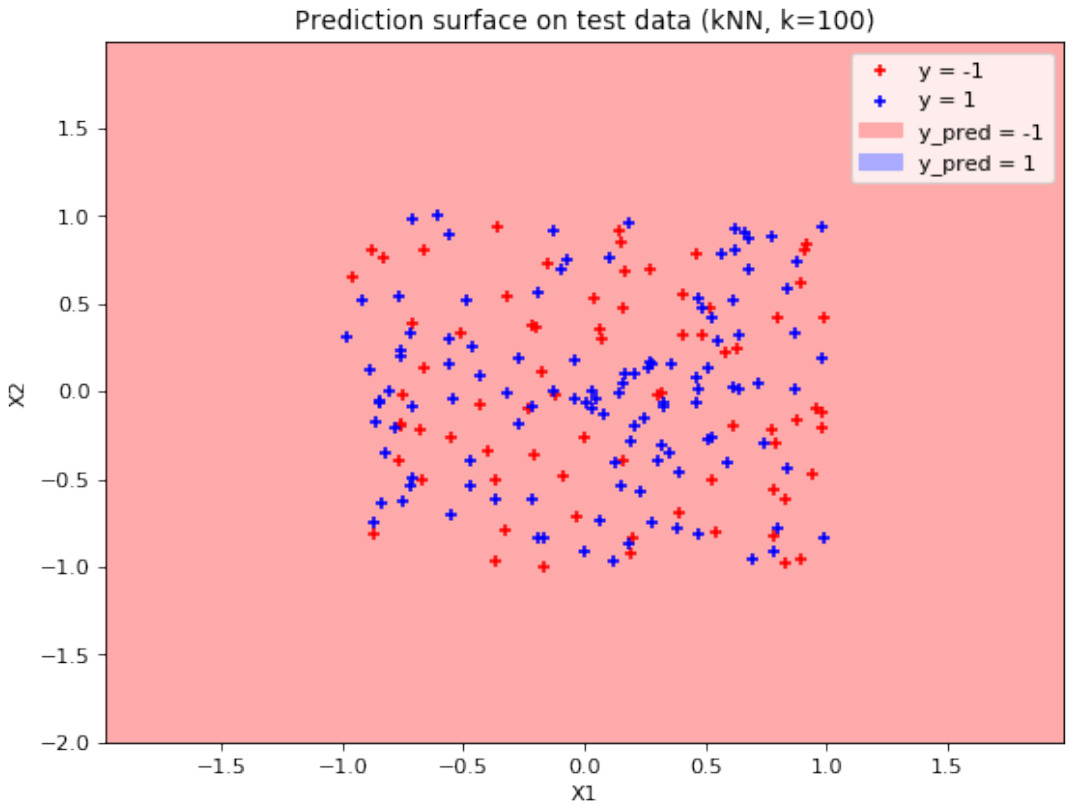

Figure 20

Again, as for the Logistic Regression, the prediction basically gives a single output class.

(c) Confusion matrices have been generated the same way as in Part (i) for the Logistic Regression classifier, the kNN classifier and two dummy classifiers with "most_frequent" and "uniform" (i.e. random at uniform) predictions:

```
models = [[LogisticRegression(penalty="l2", C=1), 'Logistic Regression...'],
          [KNeighborsClassifier(n_neighbors=100,weights='uniform'), '...'],
          [DummyClassifier(strategy="most_frequent"), '...'],
          [DummyClassifier(strategy="uniform"), '...']
confusionMatrices(X, y, q=1, models)
```

This snippet gives us the following confusion matrices (Figures 21 to 24):

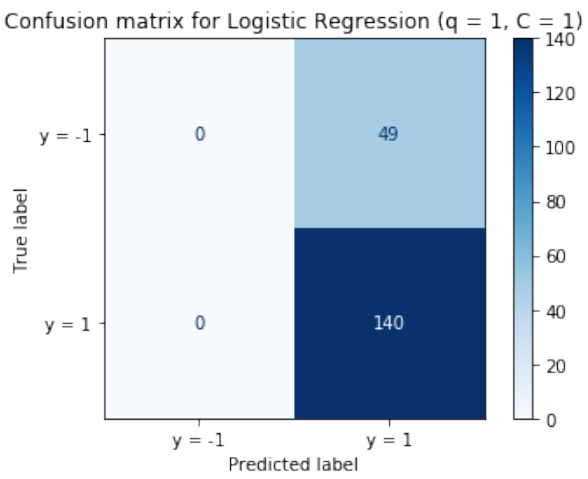

Figure 21

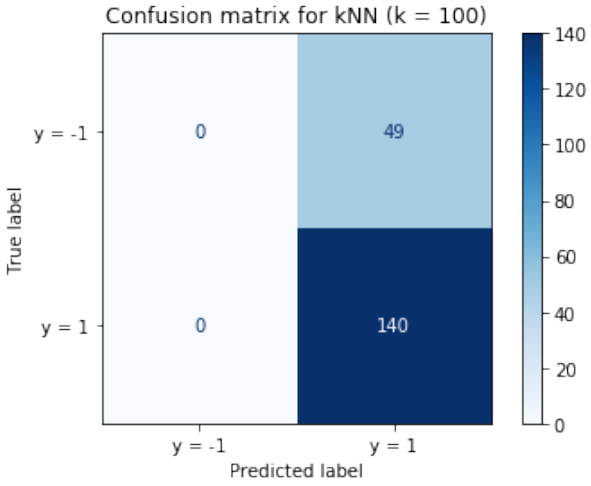

Figure 22

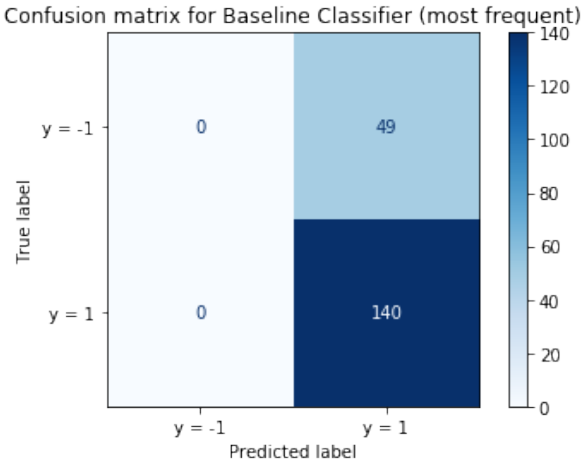

Figure 23

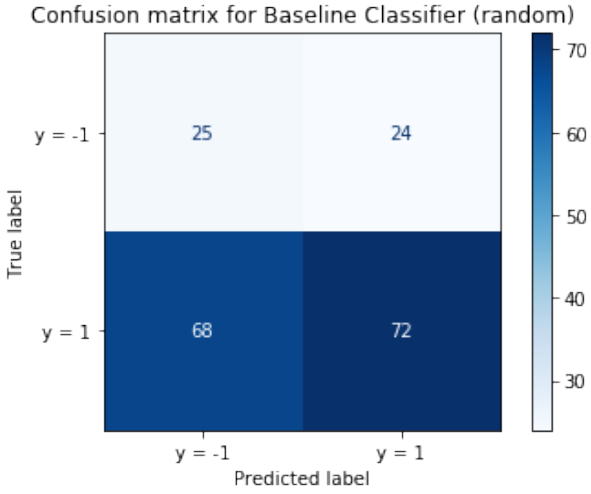

Figure 24

(d) Using the function *plotRocCurves* with the following parameters gives Figure 25:

```
1 plotRocCurves(X, y, q=2, C=100, k=5)
```

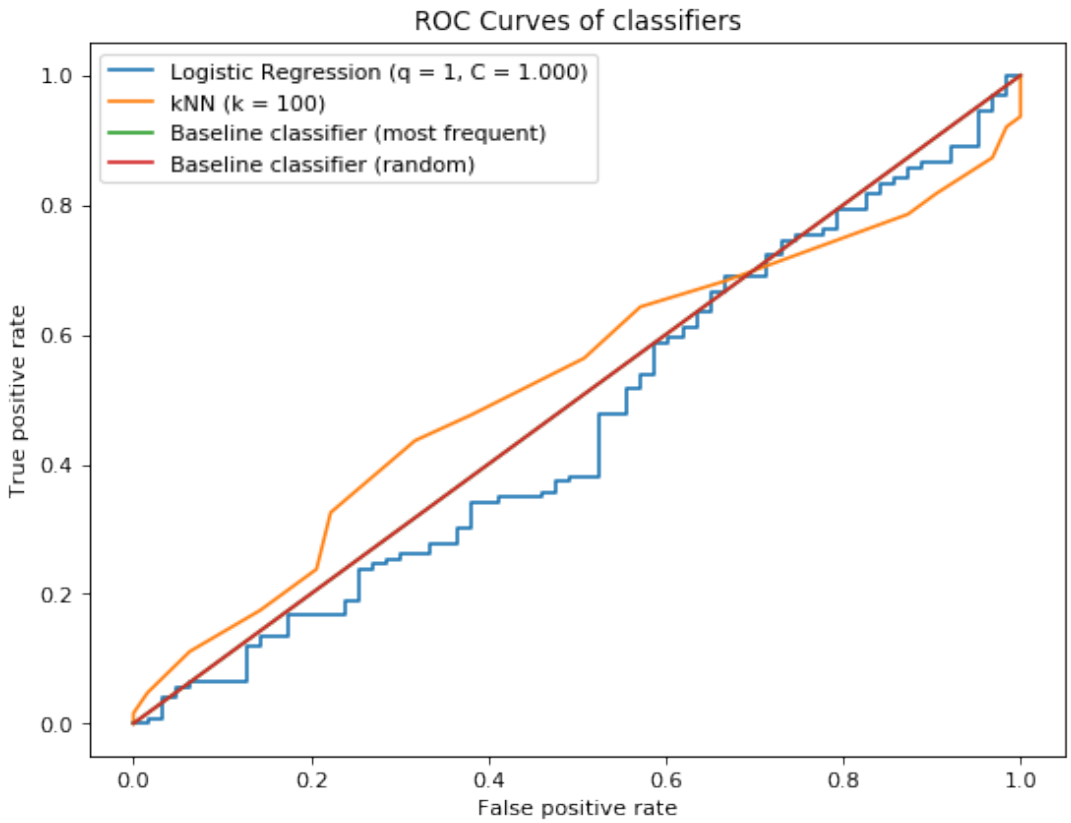

Figure 25

(e) We can calculate the following measurements for each confusion matrix:

- Logistic Regression ($q = 1,\ C = 1$):
  - $Accuracy = \frac{TN+TP}{TN+TP+FN+FP} = \frac{0+140}{0+140+0+49} = 0.74$
  - $True\ positive\ rate = \frac{TP}{TP+FN} = \frac{140}{140+0} = 1$
  - $False\ positive\ rate = \frac{FP}{TN+FP} = \frac{49}{0+49} = 1$
  - $Precision = \frac{TP}{TP+FP} = \frac{140}{140+49} = 0.74$
- kNN ($k = 100$):
  - $Accuracy = \frac{0+140}{0+140+0+49} = 0.74$
  - $True\ positive\ rate = \frac{140}{140+0} = 1$
  - $False\ positive\ rate = \frac{49}{0+49} = 1$
  - $Precision = \frac{140}{140+49} = 0.74$
- Baseline Classifier (most frequent):
  - $Accuracy = \frac{0+140}{0+140+0+49} = 0.74$
  - $True\ positive\ rate = \frac{140}{140+0} = 1$
  - $False\ positive\ rate = \frac{49}{0+49} = 1$

  - *Precision* $= \frac{140}{140+49} = 0.74$

- Baseline Classifier (random):
  - *Accuracy* $= \frac{25+72}{25+140+68+24} = 0.37$
  - *True positive rate* $= \frac{72}{72+68} = 0.51$
  - *False positive rate* $= \frac{24}{25+24} = 0.48$
  - *Precision* $= \frac{72}{72+24} = 0.75$

It is definitely interesting to note that Logistic Regression, kNN and most frequent classifiers have the same accuracy, true positive rate, false positive rate and precision. That clearly indicates the inability of any of these models to perform better than the baseline classifier; in other words: their inability to classify our original dataset. The ROC Curves also confirm that: as we can see on Figure 25, all classifiers are roughly following the baseline line.

In conclusion, I could hardly recommend using a classifier rather than another. I would rather recommend searching for less noisy data, additional features, etc. before trying to use any classifier.

A Appendix

A.1 Python Code

```
#!/usr/bin/env python
# coding: utf-8

# CS7CS4/CSU44061 Machine Learning
# Week 4 Assignment
# Boris Flesch (20300025)
#
# Downloaded dataset
# id:23-46--23-0
# id:23--23-23-0

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import PolynomialFeatures
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import plot_confusion_matrix
from sklearn.dummy import DummyClassifier
from sklearn.metrics import roc_curve
from matplotlib.colors import ListedColormap
import matplotlib.patches as mpatches

# (a)
def readData(filepath):
    df = pd.read_csv(filepath, comment="#")
    X1 = df.iloc[:,0]
    X2 = df.iloc[:,1]
    X = np.column_stack((X1,X2))
    y = df.iloc[:,2]
    return X,y

def plotData(X,y):
    X_m1 = X[np.where(y == -1)]
    X_p1 = X[np.where(y == 1)]
    plt.scatter(X_m1[:, 0], X_m1[:, 1], c='r', marker='+', label="y =
    ↪  -1")
    plt.scatter(X_p1[:, 0], X_p1[:, 1], c='b', marker='+', label="y = 1")
    plt.gca().set(title="Training data visualisation", xlabel="X1",
    ↪  ylabel="X2")
```

```
    plt.legend(bbox_to_anchor=(1, 1), loc='upper left')
    plt.show()

# (a)(i)
def plotRangeQandC(X, y, q_range, C_range):
    plt.figure(num=None, figsize=(8, 6), dpi=80)

    for Ci in C_range:
        model = LogisticRegression(penalty="l2", C=Ci, max_iter=1000)
        mean_error, std_error = [], []
        for qi in q_range:
            Xpoly = PolynomialFeatures(qi).fit_transform(X)
            scores = cross_val_score(model, Xpoly, y, cv=5, scoring='f1')
            mean_error.append(np.array(scores).mean())
            std_error.append(np.array(scores).std())

        plt.errorbar(q_range, mean_error, yerr=std_error, linewidth=3,
        ↪  label="C = %.3f"%Ci)

    plt.title("F1 Score and standard deviation vs q values (Logistic
    ↪  Regression)")
    plt.gca().set(xlabel='q', ylabel="F1 Score")
    plt.legend()
    plt.show()

# (a)(ii)
def plotRangeC(X, y, q, C_range, printParameters=False):
    Xpoly = PolynomialFeatures(q).fit_transform(X)
    mean_error, std_error = [], []
    for Ci in C_range:
        model = LogisticRegression(penalty="l2", C=Ci, max_iter=1000)
        scores = cross_val_score(model, Xpoly, y, cv=5, scoring='f1')
        mean_error.append(np.array(scores).mean())
        std_error.append(np.array(scores).std())

        if (printParameters):
            model.fit(Xpoly, y)
            theta = np.insert(model.coef_, 0, model.intercept_)
            print("C = %.1f"%Ci)
            print("theta =", theta)

    fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
    plt.errorbar(C_range, mean_error, yerr=std_error, linewidth=3)
    plt.title("F1 Score and standard deviation vs C values (Logistic
    ↪  Regression)")
    plt.gca().set(xlabel='C', ylabel="F1 Score")
    plt.show()

```

```
88  def plotPredictions(X, y, q, model, title="Prediction surface on test
    ↪  data"):
89      Xpoly = PolynomialFeatures(q).fit_transform(X)
90      Xtrain, Xtest, ytrain, ytest =
        ↪  train_test_split(Xpoly,y,test_size=0.2)
91      Xtest_m1 = Xtest[np.where(ytest == -1)]
92      Xtest_p1 = Xtest[np.where(ytest == 1)]
93
94      model.fit(Xtrain, ytrain)
95      cmap_light = ListedColormap(['#FFAAAA', '#AAAAFF'])
96      meshStep = .01
97
98      # Plot the decision boundary. For that, we will assign a color to
        ↪  each
99      # point in the mesh [x_min, x_max]x[y_min, y_max].
100     x_min, x_max = Xtest[:, 1].min() - 1, Xtest[:, 1].max() + 1
101     y_min, y_max = Xtest[:, 2].min() - 1, Xtest[:, 2].max() + 1
102     xx, yy = np.meshgrid(np.arange(x_min, x_max, meshStep),
        ↪  np.arange(y_min, y_max, meshStep))
103     Xtest = np.c_[xx.ravel(), yy.ravel()]
104     Xtest = PolynomialFeatures(q).fit_transform(Xtest)
105     Z = model.predict(Xtest).reshape(xx.shape)
106     plt.figure(num=None, figsize=(8, 6), dpi=80)
107     plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
108
109     plt.scatter(Xtest_m1[:, 1], Xtest_m1[:, 2], c='r', marker='+',
        ↪  label="y = -1")
110     plt.scatter(Xtest_p1[:, 1], Xtest_p1[:, 2], c='b', marker='+',
        ↪  label="y = 1")
111
112     plt.gca().set(title=title, xlabel='X1', ylabel="X2")
113
114     handles, labels = plt.gca().get_legend_handles_labels()
115     handles.append(mpatches.Patch(color='#FFAAAA', label='y_pred = -1'))
116     handles.append(mpatches.Patch(color='#AAAAFF', label='y_pred = 1'))
117     plt.legend(handles=handles)
118     plt.show()
119
120 # (b)
121 def plotRangeKandQ(X, y, q_range, k_range):
122     fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
123     q_range = [1,2,3]
124     for qi in q_range:
125         Xpoly = PolynomialFeatures(qi).fit_transform(X)
126
127         # k for kNN cross-validation
128         mean_f1, std_f1 = [], []
129         for ki in k_range:
```

```
130             model =
                ↪  KNeighborsClassifier(n_neighbors=ki,weights='uniform')
131             scores = cross_val_score(model, Xpoly, y, cv=5, scoring='f1')
                ↪  # cv -> KFold
132             mean_f1.append(np.array(scores).mean())
133             std_f1.append(np.array(scores).std())
134
135         plt.errorbar(k_range, mean_f1, yerr=std_f1, linewidth=3, label='q
            ↪  = %d'%qi)
136
137     plt.title("F1 Score and standard deviation vs k values (kNN)")
138     plt.gca().set(xlabel='k', ylabel="F1 Score")
139     plt.legend()
140     plt.show()
141
142 # (c)
143 def confusionMatrices(X, y, q, models):
144     Xpoly = PolynomialFeatures(q).fit_transform(X)
145     # Use the same split for each model tested
146     Xtrain, Xtest, ytrain, ytest =
        ↪  train_test_split(Xpoly,y,test_size=0.2)
147     # cross_val_predict
148
149     for model, title in models:
150         model.fit(Xtrain, ytrain)
151         disp = plot_confusion_matrix(model, Xtest, ytest,
            ↪  display_labels=["y = -1", "y = 1"], cmap=plt.cm.Blues,
            ↪  values_format = 'd')
152         disp.ax_.set_title("Confusion matrix for " + title)
153         plt.show()
154
155 # (d)
156 def plotRocCurves(X, y, q, C, k):
157     fig = plt.figure(num=None, figsize=(8, 6), dpi=80)
158
159     Xpoly = PolynomialFeatures(q).fit_transform(X)
160     Xtrain, Xtest, ytrain, ytest =
        ↪  train_test_split(Xpoly,y,test_size=0.2)
161
162     model = LogisticRegression(penalty="l2", C=100).fit(Xtrain, ytrain)
163     # decision_function: Predict confidence scores for samples.
164     fpr, tpr, _ = roc_curve(ytest,model.decision_function(Xtest))
165     plt.plot(fpr,tpr, label='Logistic Regression (q = %d, C =
        ↪  %.3f)'%(q,C))
166
167     model =
        ↪  KNeighborsClassifier(n_neighbors=k,weights='uniform').fit(Xtrain,
        ↪  ytrain)
```

```
168     fpr, tpr, _ = roc_curve(ytest,model.predict_proba(Xtest)[:,1])
169     plt.plot(fpr,tpr, label='kNN (k = %d)'%k)
170 
171     model = DummyClassifier(strategy='most_frequent').fit(Xtrain, ytrain)
172     fpr, tpr, _ = roc_curve(ytest,model.predict_proba(Xtest)[:,1])
173     plt.plot(fpr,tpr, label='Baseline classifier (most frequent)')
174 
175     model = DummyClassifier(strategy="uniform").fit(Xtrain, ytrain)
176     fpr, tpr, _ = roc_curve(ytest,model.predict_proba(Xtest)[:,1])
177     plt.plot(fpr,tpr, label='Baseline classifier (random)')
178 
179     plt.title("ROC Curves of classifiers")
180     plt.gca().set(xlabel='False positive rate', ylabel="True positive
        ↪  rate")
181     plt.legend()
182     plt.show()
183 
184 
185 #######
186 # (i) #
187 #######
188 
189 X,y = readData("week4-1.csv")
190 plotData(X,y)
191 
192 # (a)(i) Choose value of q
193 plotRangeQandC(X, y, q_range=[1,2,3], C_range=[0.001, 1, 1000])
194 
195 # Check with predictions:
196 model = LogisticRegression(penalty="l2", C=1)
197 plotPredictions(X, y, q=1, model=model, title="Prediction surface on test
    ↪  data (LogisticRegression, q=1, C=1)")
198 plotPredictions(X, y, q=2, model=model, title="Prediction surface on test
    ↪  data (LogisticRegression, q=2, C=1)")
199 # plotPredictions(X, y, q=10, model=model, title="Prediction surface on
    ↪  test data (LogisticRegression, q=3, C=1)")
200 
201 # (a)(ii)
202 plotRangeC(X, y, q=2, C_range=[1, 5, 10, 50, 100, 500, 1000])
203 
204 model = LogisticRegression(penalty="l2", C=1)
205 plotPredictions(X, y, q=2, model=model, title="Prediction surface on test
    ↪  data (LogisticRegression, q=2, C=1)")
206 model = LogisticRegression(penalty="l2", C=100)
207 plotPredictions(X, y, q=2, model=model, title="Prediction surface on test
    ↪  data (LogisticRegression, q=2, C=100)")
208 
209 # (b)
```

```
plotRangeKandQ(X, y, q_range=[1,2,3], k_range=[2,3,5,7,10,13,15,17,20])

model = KNeighborsClassifier(n_neighbors=5,weights='uniform')
plotPredictions(X, y, q=2, model=model, title="Prediction surface on test data (kNN, k=5)")

# (c)
# print(np.unique(y, return_counts=True))
models = [
            [LogisticRegression(penalty="l2", C=100), 'Logistic Regression (q = 2, C = 100)'],
            [KNeighborsClassifier(n_neighbors=5,weights='uniform'), 'kNN (k = 5)'],
            [DummyClassifier(strategy="most_frequent"), 'Baseline Classifier (most frequent)'],
            [DummyClassifier(strategy="uniform"), 'Baseline Classifier (random)']
        ]
confusionMatrices(X, y, 2, models)

# (d)
plotRocCurves(X, y, q=2, C=100, k=5)


########
# (ii) #
########
X,y = readData("week4-2.csv")
plotData(X,y)

# (a)(i)
plotRangeQandC(X, y, q_range=[1,2,3,4,5], C_range=[0.001, 1, 1000])

model = LogisticRegression(penalty="l2", C=1)
plotPredictions(X, y, q=1, model=model, title="Prediction surface on test data (LogisticRegression, q=1, C=1)")
plotPredictions(X, y, q=2, model=model, title="Prediction surface on test data (LogisticRegression, q=2, C=1)")

# (a)(ii)
plotRangeC(X, y, q=1, C_range=[0.1, 1, 10, 100], printParameters=True)

model = LogisticRegression(penalty="l2", C=1)
plotPredictions(X, y, q=1, model=model, title="Prediction surface on test data (LogisticRegression, q=1, C=1)")
model = LogisticRegression(penalty="l2", C=100)
plotPredictions(X, y, q=1, model=model, title="Prediction surface on test data (LogisticRegression, q=1, C=100)")
```

```
249
250 # (b)
251 plotRangeKandQ(X, y, q_range=[1,2,3], k_range=[1,2,3,4,5,6,7,8,9,10])
252 plotRangeKandQ(X, y, q_range=[1,2,3], k_range=[1,10,100,500]) # max
    ↪ nsamples = 754
253
254 model = KNeighborsClassifier(n_neighbors=100,weights='uniform')
255 plotPredictions(X, y, q=1, model=model, title="Prediction surface on test
    ↪ data (kNN, k=100)")
256
257 # (c)
258 models = [
259             [LogisticRegression(penalty="l2", C=1), 'Logistic Regression
                ↪ (q = 1, C = 1)'],
260             [KNeighborsClassifier(n_neighbors=100,weights='uniform'),
                ↪ 'kNN (k = 100)'],
261             [DummyClassifier(strategy="most_frequent"), 'Baseline
                ↪ Classifier (most frequent)'],
262             [DummyClassifier(strategy="uniform"), 'Baseline Classifier
                ↪ (random)']
263         ]
264 confusionMatrices(X, y, q=1, models=models)
265
266 # (d)
267 plotRocCurves(X, y, q=1, C=1, k=100)
268
269 # Purely for testing purposes (seeing the impact of a very small value
    ↪ for k)
270 # model = KNeighborsClassifier(n_neighbors=1,weights='uniform')
271 # plotPredictions(X, y, q=1, model=model, title="Prediction surface on
    ↪ test data (kNN, k=100)")
272 # plotRocCurves(X, y, q=1, C=1, k=3)
```