COMP534

Report

Assignment 1: Supervised learning methods for binary classification

Alice Varley (ID: 201685457)

March 19, 2023

1 Introduction

I chose to write an `.ipynb` file as opposed to a plain `.py` file because it makes visualisation more modular and visually easier to understand. For this, `%matplotlib inline` was a necessary command to enable the notebook to work with `matplotlib`.

1.1 Libraries

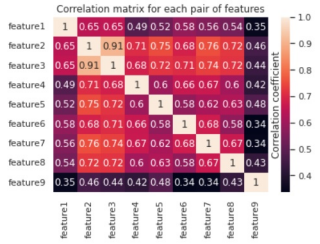


Figure 1: Correlation matrix of features

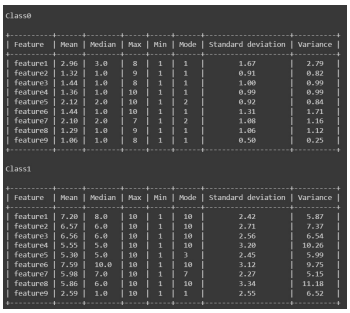
I chose to use `seaborn` and `matplotlib.pyplot` as my main visualisation toolkits. To access the algorithms and to generate the confusion matrices, I imported various modules from `sklearn` (SciKit-Learn). Obviously, `numpy` was also very important for accessing the array data structure and `numpy.random` was important for shuffling the data to ensure better outcomes. I also chose to use the `tabulate` package for the presentation of some small tables (see figs.2 and 5).

1.2 The dataset

The dataset consists of 700 samples each belonging to either class 0 or class 1. Each data sample has 9 feature values. As is shown in fig.2, class 0 generally has lower mean values for all features and a lower maximum value for all features, indicating that the values in class 0 tend to be smaller than in class 1. The standard deviation and variance measures also show that the feature values for class 0 are more centred around the mean than in class 1, which has a wider distribution. The correlation matrix shows that features 2 and 3 are closely correlation (see fig.1).

1.3 Classification methods

Figure 2: Statistical description of each feature, per class



Class0

Feature	Mean	Median	Max	Min	Mode	Standard deviation	Variance
feature1	2.90	3.0	8	1	1	1.67	2.79
feature2	1.32	1.0	9	1	1	0.91	0.82
feature3	1.44	1.0	8	1	1	1.00	0.99
feature4	1.36	1.0	10	1	1	0.99	0.99
feature5	2.12	2.0	10	1	2	0.92	0.84
feature6	1.44	1.0	10	1	1	1.31	1.71
feature7	2.10	2.0	7	1	2	1.08	1.16
feature8	1.29	1.0	9	1	1	1.06	1.12
feature9	1.06	1.0	8	1	1	0.50	0.25

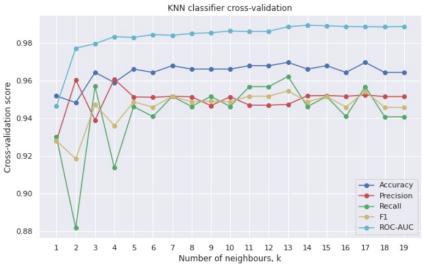
Class1

Feature	Mean	Median	Max	Min	Mode	Standard deviation	Variance
feature1	7.20	8.0	10	1	10	2.42	5.87
feature2	6.57	6.0	10	1	10	2.71	7.37
feature3	6.56	6.0	10	1	10	2.56	6.54
feature4	5.55	5.0	10	1	10	3.20	10.26
feature5	5.30	5.0	10	1	3	2.45	5.99
feature6	7.59	10.0	10	1	10	3.12	9.75
feature7	5.98	7.0	10	1	7	2.27	5.15
feature8	5.86	6.0	10	1	10	3.34	11.18
feature9	2.59	1.0	10	1	1	2.55	6.52

The three classification methods chosen were **K-Nearest Neighbours** (KNN), **Naïve Bayes** (NB), and **Logistic Regression** (LR). The hyperparameters were the number of neighbours, smoothing values and the regularisation term respectively.
In tuning the hyperparameters, for each classification method I set up a loop to iterate through a list of values to test, in which a classifier was created for each term and the classifier and term were then put through my `cross_validate` function which uses K-fold validation with 5 folds (creating a validation set inside) and the `sklearn.model_selectionn.cross_val_score` method, which takes the classifier, training objects and labels, number of folds and the type of evaluation method as parameters. This function calculates the accuracy, precision, recall, f-score and ROC-AUC score for each term. Each of these list of scores as well as the list of values to test are then put through my `tune_param` function, which returns a list of the optimal parameters according to each evaluation measure, as well as the scores for each evaluation method.

For KNN, I tested $k = 1, \ldots, 19$. The best k was 13 according to accuracy, recall and f-score, but according to precision was 4 and according to ROC-AUC it was 14. After inspecting the graph, I decided to use $k = 7$ as this was somewhat of a turning point (elbow) on the graph (see fig.3), and didn't want to choose too high of a k to avoid overfitting the data.

Figure 3: Elbow graph of K-NN cross-validation

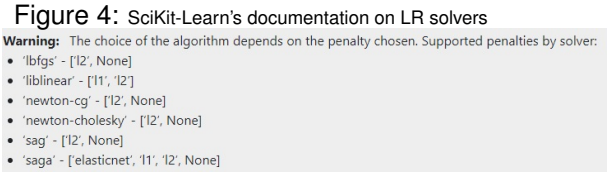


For NB, I tested along a logarithmic scale of base 10, namely $\alpha = 1 \times 10^{-}7$,$1 \times 10^{-}6$, $1 \times 10^{-}5$, $1 \times 10^{-}4$,$0.001, 0.01, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10$. Predictably, all evaluation measures favoured the smallest α, as this lead to the least disruption in the data. All scores dropped after $\alpha = 1$. Whilst it is common to have an α between $1 \times 10^{-}7$

and $1 \times 10^{-}4$ [1], for this relatively small set of data, there wasn't enough of a difference in the evaluation scores to justify having such a small α, therefore I chose to use 0.001.

Finally, for LR, accuracy, precision and f-score favoured ridge regression (l2 regularisation) whereas recall and ROC-AUC favoured lasso regression (l1 regularisation). Both terms produced similar results, so I chose l2 as it was favoured in more of the evaluation measures. It should be noted here that in the API for `sklearn.linear_model.LogisticRegression`, there isn't a solver which can compare all possible penalty terms at once (`None`, l1, l2, elasticnet(both)), as is demonstrated in fig.4.

Figure 4: SciKit-Learn's documentation on LR solvers



1.4 Training and testing process

To prepare the data for training and testing, I first converted the dataframe into an array and shuffled using `random`. Then I created a list of feature values for each object in the dataset and a list of labels. This made sure that the data was in the right format for `sklearn.model_selection.train_test_split`. I used this method to split the data 80/20 into training and testing data, respectively, using the optional parameters `train_size` and `text_size` to do so. This method also provides a stratified split, meaning the proportion of the two classes in equal in both the train and test data and equivalent to the proportion found in the original data. The function provides four outputs: `X_train`, `X_test`, `y_train`, and `y_test`, where `X` and `y` represent objects (i.e. their feature values) and labels.

Figure 5: Evaluation measures across the methods

Evaluation and comparison across all classifiers

	KNN	Naïve Bayes	Logistic Regression
Accuracy	0.96	0.92	0.92
Precision	0.98	0.96	0.96
Recall	0.93	0.84	0.84
F1-score	0.95	0.91	0.9
ROC-AUC	0.96	0.9	0.91

I wrote several reusable functions for use in training and testing:

- **`compute_scores`**, which calculates the evaluation scores given a list of true labels and labels predicted by a model
- **`create_confusion_matrix`**, which creates a confusion matrix object, and then confusion matrix display object, using `sklearn.model_selection.confusion_matrix` and `.ConfusionMatrixDisplay`
- **`create_scores_table`**, which uses `tabulate` to produce and print a tabular visualisation of the various evaluation measures for a given classifier
- **`overall_table`**, which creates and prints a tabulate object to compare the scores for each evaluation measure across each classifier (see fig.5)
- **`cross_validate`**, as aforementioned
- **`tune_param`**, as aforementioned

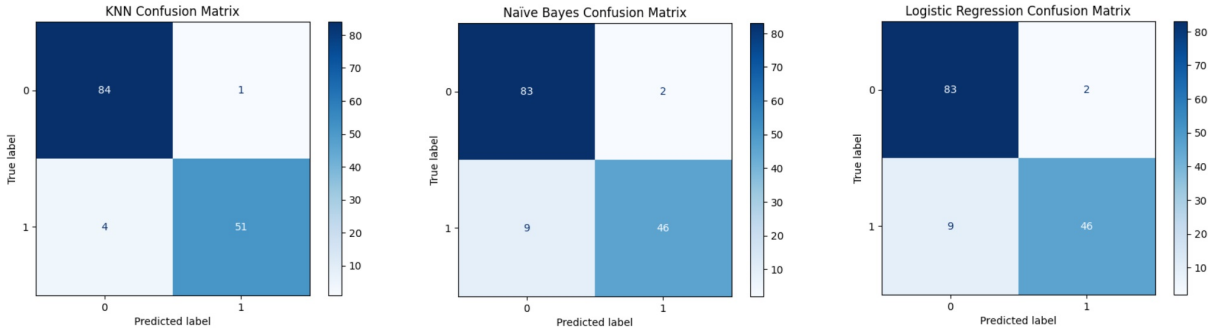
Because I used `sklearn` to construct each classifier object, the training and testing was done in the same way each time. Namely, after creating the classifier instance, <classifier> `.fit (X_train, y_train)` to train the classifier then `y_pred=list(`<classifier>`.predict(X_test))` for testing. The code then calls **`create_confusion_matrix`** and **`create_scores_table`** to generate and print the confusion matrix and table of evaluation scores for the classifier in question.

2 Evaluation of and interpretation of results

By comparing model-predicted labels with true labels, confusion matrices provide a way to calculate and display **True Positives** (TP), where predicted = 1 and true = 1), **False Positives** (FP), where predicted = 1 but true = 0 (negative), **True Negatives** (TN), where predicted = 0 and true = 0, and finally **False Negatives** (FN), which were predicted = 0 but true = 1 (positive). As fig.6 shows, all models had more FNs than FPs, which could suggest that the models were just better at not

misclassifying negatives, or the training data was skewed to include more negatives. It would seem that the latter is the case, as each classifier has 30-40 more TN than TP. KNN was slightly better at predicting positives than NB or LR, which is backed up by the higher precision (0.98) compared to NB and LR (0.96). That said, all had a similar number of TNs so were equally good at correctly classifying negatives. With such low FNs and FPs, it would seem that the models are fairly accurate, meaning that training was effective with a good selection of hyperparameter.

Figure 6: The confusion matrices for each classification method



From the evaluation measures (see fig.5), it becomes clearer that KNN is the best model for this set of data. It has higher scores across all measures, though it should be noted that some of the scores are very similar (for example, precision). Furthermore, NB and LR have nearly identical scores across the first three measures and then very similar ones for the latter two. This makes sense when you take into account the fact that they are both probabilistic methods that make assumptions about the distribution of the data, and are both linear models that try to fit a decision boundary to separate classes, while KNN is distance-based. KNN made more correct predictions (accuracy=0.96) compared to NB and LR (0.92). The lowest evaluation score is the NB and LR recall (0.84), which means that they aren't as good (whilst still pretty good) at identifying positive instances. F1-score provides the harmonic mean between precision and recall, and each classifier has reasonably high (over 0.9) f1. Finally, all classifiers have reasonable ability to distinguish between the positive (1) and negative (0) classes, as shown by the ROC-AUC scores.

3 Concluding remarks

Ultimately, this was an interesting exercise. It is interesting to remark that, despite the difference in the functioning of the three models chosen, they all had similar evaluation scores, above 0.9, (see fig.5) apart from Recall, which differed slightly. As aforementioned, KNN performed the best according to all evaluation metrics, which could suggest that the data isn't entirely linearly separable. It is also notable that NB and LR have almost exactly the same scores; whilst they are both probabilistic models, they make different assumptions about the data.

One main advantage of KNN are that it can handle more complex decision boundaries, which seems like it may be the case here. It is also more robust when dealing with outliers and noisy data. However, it is very important (and sometimes difficult, as we can see in fig.3) to choose the right value of k as this can have a massive impact on the accuracy of the classification. NB, on the other hand, is a simple algorithm that can handle large datasets with high-dimensional features (which wasn't the case here). However, the assumptions that it makes about the data, such as the features being conditionally independent, may not be the case (and wasn't the case for at least one pair of features: see fig.1) and mean that it may not perform as well on more complicated data. It also might suffer when the dataset isn't well balanced. Finally, LR is another simple algorithm, but it assumes that there is a linear relationship between the features and target variable (which may not always be the case), is sensitive

to outliers and multi-co-linearity [2] (which was the case for this data) and may not perform well when the dataset is imbalanced. Here we can see that NB and LR are fairly similar in many respects and quite different to KNN. That said, non-parametric models such as K-NN can actually be used to learn a probability model [3], so perhaps the three models have more in common than it would seem on first inspection.

If I were to do this again, I would chose to compare more of a variety of models, rather than having two probabilistic models. Comparing Decision Tree, Support Vector Machines and KNN would be a good mix, as these three algorithms are quite different from each other in terms of their approach to modelling the data and predicting labels. I would also try using `.StratifiedShuffleSplit` from `sklearn.model_selection`, as it provides a more sophisticated and efficient code where it both splits the data and provides cross-validation in the same method. Additionally, it would be good to find a way to test all regularisation options, and it would be interesting to experiment with using different distributions for NB to see which works the best for the data. Finally, I would like to improve my proficiency with plotting libraries; the cross-validation graph that I provided for LR in particular was not a very effective visualisation.

References

[1] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*. O'Reilly Media, Inc., 2022.

[2] A. Alin, "Multicollinearity," *Wiley interdisciplinary reviews: computational statistics*, Journal 2, no.3, pages 370–374, 2010.

[3] S. J. Russell and P. Norvig, *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010