POLITECNICO
DI TORINO

Nearest Neighbors, Linear SVM, SVM with RBF Kernel

Homework 1

Ludovisi Linda
s278221

May 2020

Academic Year 2019/2020

Contents

1 Homework Description

The goal of this homework is to understand how different classification algorithms perform over the same dataset.
In this project the *Wine* dataset of the scikit library will be used. Some different classifiers will be tried out in order to perform the classification task: *KNN*, *Linear SVM* and *SVM with RBF Kernel.* For simplicity, the classification task is performed on the first two attributes of the dataset.

2 Data preparation

2.1 Loading the Dataset

The *Wine* dataset from the scikit library is first loaded, then converted into a Dataframe, using a custom function *sklearn_to_dataframe*, which takes as input the original dataset and retrieves a pandas Dataframe.
Then the first two attributes are selected: the experiments conducted, from now on, are referred to *alcohol* and *malic_acid* features. Also the *target* column was selected and stored in a different dataframe: these are the values we are going to predict.

2.2 Creating Train, Test and Validation sets

Next step was splitting the dataset into *train, validation* and *test* in proportion 5:2:3. The *train_test_split* function of the scikit library was used in order to perform the split and a *random_state*=42 was set in order to produce replicable results.
Then the data were scaled using StandardScaler: it centers the data by removing the mean value of each feature and then scale it by dividing features by their standard deviation. Standardization in machine learning is important because some algorithms (such as the RBF kernel of Support Vector Machines that we'll use later) often assume that all features are centered around zero and have variance in the same order. So an instance of StandardScaler was created; it was trained on the training set and then train, validation and test set were transformed.

3 K Nearest Neighbors

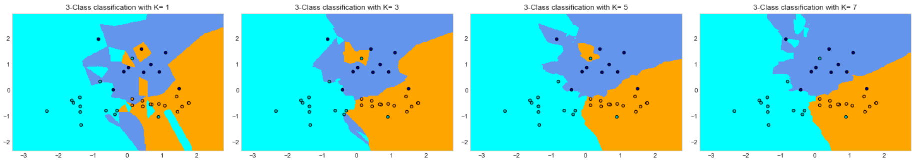
Next thing to do was performing KNN on the two selected features, using as parameter K = [1, 3, 5, 7].

3.1 Training: KNN

An iteration over these possible values of K was done in order to perform hyperparameter selection. At each iteration the K-Nearest Neighbors model was trained on the training set and evaluated on the validation set, then the decision boundaries were plotted, using a custom function:

plot_classifier(clf , X_val , y_val)

Here, the *clf* is the instantiated model trained on the training set; *X_val* is the 2D validation dataframe and *y_val* is the true label of the validation set.
This function plots the decision boundaries generated by the training model in a way that we can see which points (we are in a 2D representation) are classified correctly.
Finally the model with the best accuracy was saved: this is the model that we will use for evaluating the model on the test set.
Using these values of K the following plots were obtained (shown results refer to the validation set):



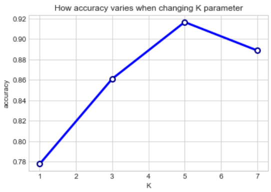
(a) K = 1 (b) K = 3 (c) K = 5 (d) K = 7

K	1	3	5	7
accuracy	0.7777	0.8611	0.9166	0.8888

Table 1: Validation accuracy with KNN

3.2 Accuracy on the validation set: KNN

As mentioned before, the best results refer to the hyperparameter configuration: K=5, with a validation accuracy=0.9166.



Also, if we take a look at the figures (a), (b), (c), (d) we can notice that the boundaries change along with the parameter K; in fact we can think of K as controlling the shape of the decision boundary.

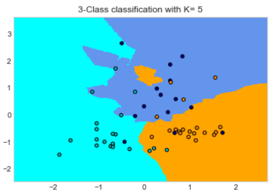
'K' in KNN is a parameter that refers to the number of nearest neighbors to include in the majority of the voting process. Choosing smaller values for K can be noisy and will have a higher influence on the result. Larger values of K will have smoother decision boundaries which mean lower variance but increased bias; also, computationally expensive. For KNN models, complexity is determined by the value of K (lower value = more complex).

In figure (a), the one with K=1, we can see that the boundaries are not well-defined: this is because, using K=1, each point is assigned to their 1-nearest neighbor. The resulting plot is the most jagged. As we expected, the predictions seem to get more simple in some sense as k gets larger (i.e. the first plot has the most complicated looking predictions, and (c), (d) plots seem to be the simplest). The last plot , the one with K=7, is the most simple: the boundaries are well-defined.

3.3 Testing: KNN

In training phase, the model with the best accuracy on the validation set was saved: this is the model that is used for testing.

The accuracy score performed on the test set is **0.7037**; which is a little bit smaller, compared to the one obtained on the validation set. This difference is mainly due to the cardinality of the dataset: the Wine dataset, being so small, becomes very dependent to the data used to perform validation, so it can not generalize well to new data.



4 Linear SVM

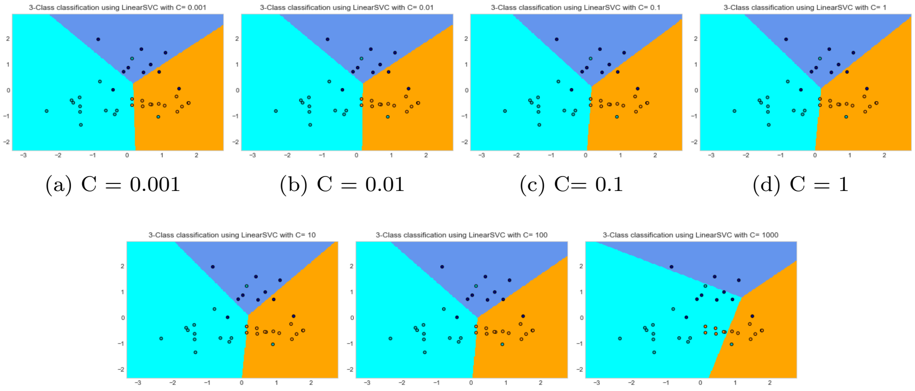
Next thing to do was performing Linear SVM (in our case, SVC because we are performing a classification task) on the two selected features, using as parameter C = [0.001, 0.01, 0.1, 1, 10, 100, 1000].

4.1 Training: Linear SVC

An iteration over these possible values of C was done in order to perform hyperparameter selection. At each iteration the model was created, using:

svm.LinearSVC(C=c)

Then the model was trained on the training set and evaluated on the validation set.
The decision boundaries were plotted, calling the *plot_classifier* function mentioned in the last section, passing as *clf* parameter the instantiated *linearSVC()* model trained on the training set. As a result, the following 2D plots were generated (shown results refer to the validation set):



(a) C = 0.001 (b) C = 0.01 (c) C= 0.1 (d) C = 1

(e) C= 10 (f) C = 100 (g) C = 1000

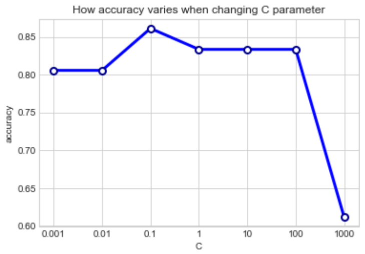
C	0.001	0.01	0.1	1	10	100	1000
accuracy	0.8055	0.8055	0.8611	0.8333	0.8333	0.8333	0.6111

Table 2: Validation accuracy with LinearSVC

Finally the model with the best accuracy was saved: this is the model that we will use for evaluating on the test set.

4.2 Accuracy on the validation set: Linear SVC

The best results refer to the hyperparameter configuration: C=0.1 with a validation accuracy=0.8611.



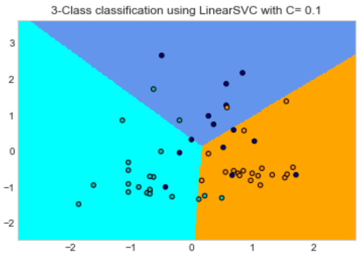
In the figures above, we displayed the decision boundaries found by LinearSVC as straight lines, separating the space in 3 areas (each area is corresponding to a different predicted value). The parameter that determines the strength of the regularization is *C*, and higher values of C correspond lo *less regularization*.

- with higher values for C (fig. (e), (f), (g)) , the model will try to fit the training set as best as possible: this is because when C is large, the classifier is heavily penalized for misclassified data and therefore will try to avoid any misclassified data points (even outliers!). This leads to low bias, high variance.
  For example, in figure (g) the margins are so large (*C=1000*) that they classify a lot of points in a region of the space that is incorrect. There, the model is clearly overfitting.

- with lower values for C (fig. (a), (b), (c)), the model tries to adjust the majority of data points; this means a lot of regularization, corresponding to a very small margin. This leads to high bias, low variance.
  For all the points which are far from the "dense" region of their class there is a misclassification, but the overall accuracy is quite good. In fact, we find out that the best hyperparameter configuration is the one showed in figure (c), corresponding to C=0.1

4.3 Testing: Linear SVC

The model with the best validation accuracy (the one corresponding to C=0.1) was saved and finally tested on the test set. The accuracy score performed on the test set is **0.7407**, which is a smaller value, compared to the one obtained in the validation set. The difference in accuracy could be explained like we did in the last section.



5 SVM with RBF kernel

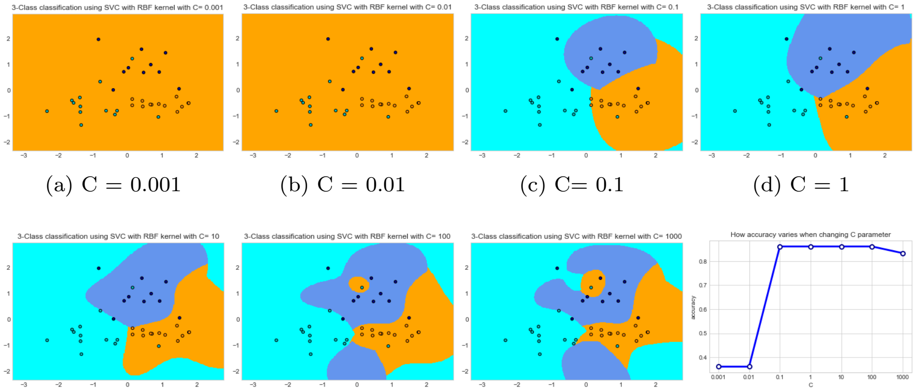
Next thing to do was performing SVM using a RBF kernel. Like before, the same values for C were tried out.

5.1 Training: SVC with RBF kernel

An iteration over the values C = [0.001, 0.01, 0.1, 1, 10, 100, 1000] was done in order to tune this hyperparameter of the model. At each iteration the model was created, using:

svm.SVC(kernel = 'rbf', C=c)

and then the generated boundaries were plotted, along with the points associated with the true label of the val dataset, using the custom *plot_classifier* function, passing as *clf* parameter the instantiated *SVC(kernel='rbf')* model trained on the training set. As a result, the following 2D plots were generated (shown results refer to the validation set):



(a) C = 0.001

(b) C = 0.01

(c) C= 0.1

(d) C = 1

(e) C= 10

(f) C = 100

(g) C = 1000

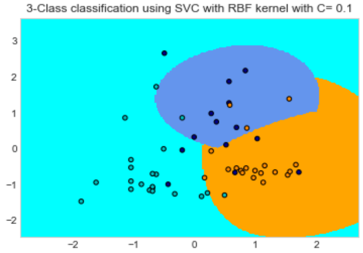
C	0.001	0.01	0.1	1	10	100	1000
accuracy	0.3611	0.3611	0.8611	0.8611	0.8611	0.8611	0.8333

Table 3: Validation accuracy with SVC, kernel RBF

Finally the model with the best accuracy (c) was saved: this is the model that we will use for evaluating on the test set.

5.2 Testing: SVC with RBF kernel

The accuracy score performed on the test set, using the best hyperparameter (C=0.1) is **0.7777**. We already expected it to be lower than the one obtained with the validation dataset, like we already explained.



5.3 Differences between Linear SVC and SVC with RBF kernel

First, the SVC model with rbf kernel has non-linear boundaries, unlike the LinearSVC model which was only able to generate linear ones. This behavior is made possible thanks to the use of kernels, a function that maps the data from one space to a new one, which is simpler for SVM to deal with.
Non-linear boundaries generated from SVC(kernel='rbf') vary according to two parameters: *C* and *gamma*. In fact, SVM with rbf kernel has a "new" parameter, gamma: it can be thought of as the 'spread' of the kernel and therefore the decision region. If not specified, default value of gamma is 'scale'. On the other hand, parameter C represents the penalty for misclassification: low values for C correspond to a model which is tolerant of misclassified data point; with high values for C the classifier becomes intolerant to misclassified data points.

5.4 Grid search for hyperparameter tuning

In this section we will try to tune both C and gamma at the same time. Possible values for C are [0.001, 0.01, 0.1, 1, 10, 100,1000] and possible values for gamma are [0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000].

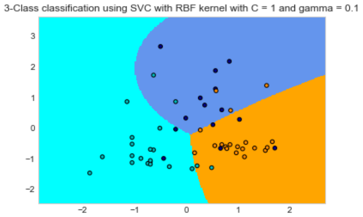
5.4.1 Parameter Grid

A grid search of the best parameters was performed iterating over all the possible tuples *(c-value, gamma-value)*. At each iteration, the model was trained on the training set and evaluated on the validation set.
The following results were obtained:

gamma / C	**0.001**	**0.01**	**0.1**	**1**	**10**	**100**	**1000**
0.0001	0.3611	0.3611	0.3611	0.3611	0.3611	0.75	0.8333
0.001	0.3611	0.3611	0.3611	0.3611	0.7500	0.8333	0.8611
0.01	0.3611	0.3611	0.3611	0.7500	0.8333	0.8611	0.8055
0.1	0.3611	0.3611	0.5833	**0.8888**	0.8888	0.8611	0.8333
1	0.3611	0.3611	0.8055	0.8611	0.8611	0.7777	0.6666
10	0.3611	0.3611	0.3611	0.7777	0.7222	0.6111	0.6111
100	0.3611	0.3611	0.3611	0.6111	0.6388	0.6388	0.6388
1000	0.3611	0.3611	0.3611	0.3888	0.3888	0.3888	0.3888

Table 4: Validation accuracy with SVC, kernel RBF

Then, the model reaching the best validation accuracy was saved in order to perform the evaluation on the test set. C=1 and gamma=0.1 were the best hyperparameters, and the resulting accuracy on the test set was: **0.7592**.



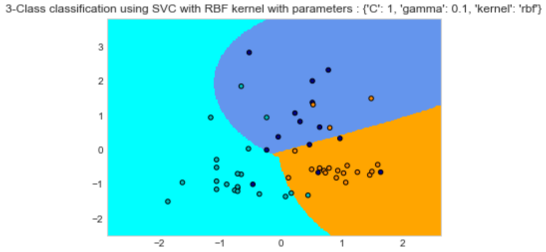
5.4.2 Grid Search Cross Validation: K-Fold

Finally, another grid search for C and gamma was performed, using 5-fold validation. In order to do this, training and validation set were merged and finally scaled again, along with the test set.
The grid search cross validation was performed, using the function:

gsCV = GridSearchCV(svm.SVC(), grid_svc, cv=5, verbose=3, n_jobs=-1)

provided by Scikit-Learn library. An important thing to underline is that grid_svc parameter is a dictionary containing all the hyperparameters which are involved in the grid search process.
After the training process, the best hyperparameter configuration found was 'C': 1, 'gamma': 0.1, 'kernel': 'rbf' that is exactly the same as before.

Finally the test set was evaluated using the *gsCV.best_estimator_* and the resulting accuracy was **0.7962**, which is bigger than the one obtained with the previous method.



This (positive) variation in accuracy could be explained taking in considerations two factors.
First, the cardinality of the training dataset: in the gsCV scenario we were using the 5-fold validation; this means that, at each iteration, we had a training dataset which was bigger compared to the one obtained with the proportion train:val = 5:2 .
Second, gsCV generates a defined number of folds over which different hyperparameter configurations are tested, generating a more robust model.

6 Differences between KNN and SVM

7 Final considerations

Due to the limited amount of data (Wine dataset is a very small dataset, only 178 entries!), results may change along with different values of *random_split*. A possible thing we could do to obtain reliable results is performing *cross-validation* over all methods.
Another factor that affects results is the attribute *test_size* in the functions performing the split of the dataset. In fact, having a proportion **5:2:3** among train, val and test datasets is not feasible, because of the math (dataset length is 178...) . Splitting the dataset first into train/test , and then into train/val or viceversa produces very different results.
Also, results of LinearSVC() are run-dependent, so they are not replicable unless a seed is set.