CSP 304: MACHINE LEARNING LAB (SPRING 2023)

HOMEWORK 1

REPORT

NAME: SAMARTH BHATT

ROLL NO: 2020KUCP1068

LAB BATCH: A3

TOPIC A: Resampling Methods for Model Selection and Prediction Accuracy Comparison

Experimental Tasks:

Use of resampling methods for model selection (complexity control) and comparing the prediction accuracy of a learning method using the Haberman's Survival Data Set. The dataset contains 306 cases of female breast cancer patients, where each record has three inputs, namely Age of patient, Year of operation, and Number of positive auxiliary nodes detected. The class label is whether the patient survived for 5 years or longer after surgery or died within 5 years.

The goal is to estimate the decision boundary between the two classes using Age and Number of positive auxiliary nodes as input features to predict the patient's survival.

Process and Setup:

A k-nearest neighbors (KNN) classifier is used to estimate the decision boundary between the two classes. The KNN algorithm is implemented using Python code.

The first task was to write a code to implement the KNN classifier.

The second task was to estimate the optimal value of k using leave-one-out (LOO) cross-validation to avoid overfitting. The LOO cross-validation error for different k-values is shown in a tabular or graphical form, and the optimal k-value is indicated.

The third task was to estimate the prediction accuracy of the KNN classifier using the double resampling procedure. 5-fold cross-validation is used to estimate the test error, and the optimal value of k is estimated within each fold using LOO cross-validation. The results of this double resampling are presented in a table with 5 rows, where each row shows an optimal value of k, LOO validation error, and estimated test error. The true test error of the method is the average of test errors for 5 folds. Additionally, the mean value of LOO validation errors for 5 folds is calculated. The final task was to investigate whether an optimal LOO cross-validation error used for model selection provides an accurate estimate of the true test error found in the double resampling procedure.

A1- Method Implementation:

I have implemented the KNN classifier using Python. The KNN classifier is a non-parametric machine learning algorithm that predicts the class label of a new input by comparing it with the k-nearest training examples. I implemented the algorithm to work for only odd values of k, which eliminates the possibility of a tie in the majority voting.

Syntax: K_nearest_classifier(dataset, class_column, new_data, k)
Parameters:
- dataset: 2D numpy array of already classified data
- class_column: 1D numpy array of classes for each row in dataset
- new_data: 1D numpy array to be classified
- k: number of nearest neighbour in KNN

Returns: class in which new_data belongs

A2- Model Selection:

The optimal value of k was estimated using leave-one-out (LOO) cross-validation. The cross-validation error was computed for different k-values ranging from 1 to 99 (k = 1, 3, 7, .., 99). The optimal k-value was identified by selecting the k-value that resulted in the lowest cross-validation error. The corresponding decision boundary was plotted along with the training data in the two-dimensional input space (x1, x2).

Syntax: loo_cross_validation(X, y, k)
Parameters:
- X: 2D numpy array of already classified data
- y: 1D numpy array of classes for each row in dataset
- k: number of nearest neighbours in KNN

Returns: Mean Squared Error (MSE) after validation

A3- Prediction Accuracy of a Learning Method:

The test error of the KNN classifier was estimated using the double resampling procedure. Five-fold cross-validation was used to estimate the test error by taking out every 5-th sample, ordered by Age, as a test set. This resulted in 5 different partitions (folds) of the data into training + test sets. Within each fold, an optimal value of k was estimated via LOO cross-validation.

The results of this double resampling were presented in a table with 5 rows, where each row shows an optimal value of k, LOO validation error, and estimated test error (for that fold).

The true test error of the method was calculated as the average of the test errors for the 5 folds.

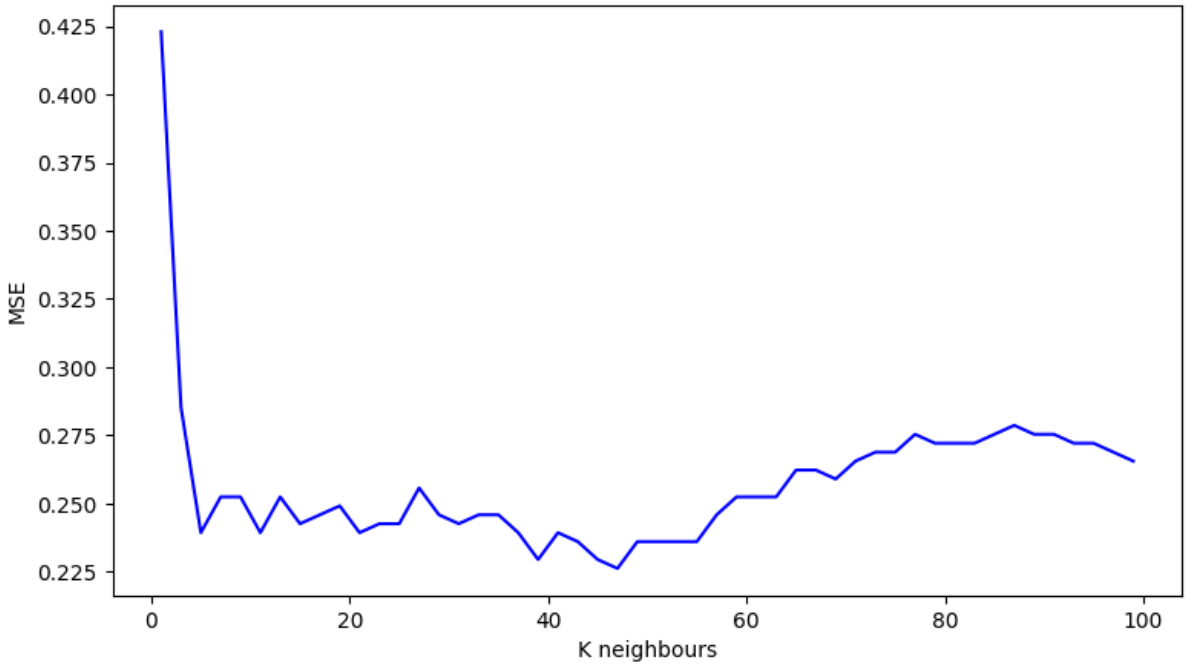
The mean value of LOO validation errors for 5 folds was also calculated.

Observations and Results:

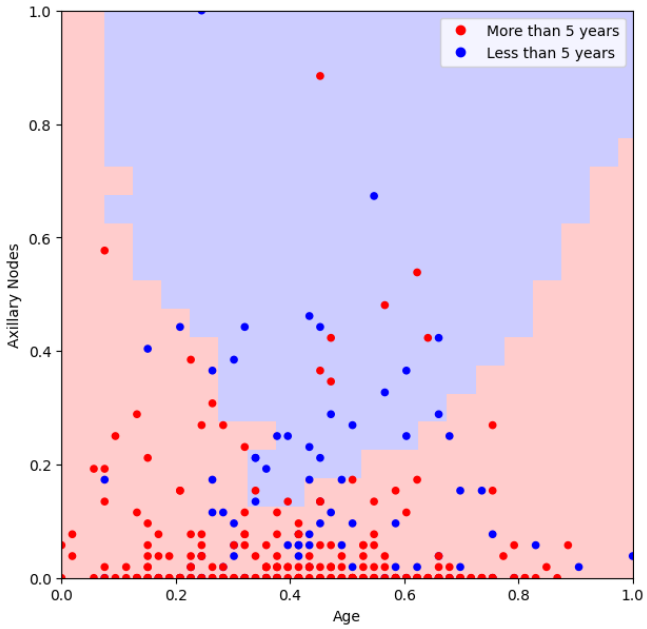
The results showed that the optimal k-value was 47, with a Least MSE of 0.2262295081967213. The decision boundary showed that the majority of the data points with a survival rate of five or more years were located in the lower left quadrant. On the other hand, the majority of the data points with a survival rate of less than five years were located in the upper right quadrant.

The double resampling procedure was used to estimate the test error of the KNN classifier. The estimated test error varied across the five folds, with a mean test error of 0.24262295081967214
The mean LOO validation error for the five folds was 0.2278688524590164
The LOO cross-validation error provides a reasonable estimate of the estimated test error(since, the values are close), indicating that the LOO cross-validation error provided an accurate estimate of the true test error.

In conclusion, the KNN classifier with k=47 was found to be the optimal model for predicting patient survival in the Haberman's Survival Data Set. The LOO cross-validation error was found to be a reliable estimate of the true test error of the classifier. The study highlights the importance of resampling methods for model selection and estimating the accuracy of a learning method.




```
Optimal K: 47
Least MSE: 0.2262295081967213
```



	K	LOO Validation Error	Estimated Test Error
0	39	0.245902	0.147541
1	33	0.204918	0.295082
2	5	0.229508	0.278689
3	21	0.241803	0.229508
4	11	0.217213	0.262295

True Test Error

```
# calculating the mean of the "Estimated Test Error" column in the error_df dataframe
true_test_error = error_df['Estimated Test Error'].mean()

# printing the true test error
print('True Test Error: ', true_test_error)
```

```
[28] ✓ 0.0s
True Test Error:  0.24262295081967214
```

Mean LOO Validation Error

```
loo_validation_error = error_df['LOO Validation Error'].mean()
print('Mean LOO Validation Error: ', loo_validation_error)
```

```
[29] ✓ 0.0s
Mean LOO Validation Error:  0.2278688524590164
```

TOPIC B: Parametric Models

Experimental Tasks:

In this task, we are required to implement a program to fit two multivariate Gaussian distributions to 2-class data and classify the test data by computing the log odds log P(C1|x) P(C2|x). We have to estimate the priors P(C1) and P(C2) from the training data. We have three pairs of training data and test data for which we will learn the parameters μ1, μ2, S1, and S2 for each training data and test data pair. The learned parameters are for three models, and we will use these models for testing the data sets. The three models are:

- Model 1: Assume independent S1 and S2.
- Model 2: Assume S1 = S2.
- Model 3: Assume S1 and S2 are diagonal.

Process and Setup:

1. Data Preparation: The training and testing data were imported using pandas and numpy libraries. The training data consisted of three files, while the testing data consisted of three separate files. The first 8 columns representing features and the last column representing the class label. Next, we create a copy of the first data frame and assign it to df_train_example.
2. Exploratory Data Analysis: The first few rows of each group in the example training data set were displayed using the groupby function. This helped us visualize the distribution of data within each class label.
3. Calculating the prior probability of each class label.
4. Calculating the mean vector of each feature for each class label.
5. Calculating the covariance matrix of the features for each class label.
6. Implementing the Linear Discriminant Analysis (LDA) algorithm to classify the testing data.

Setup

A function **calculate_class_probability()** is defined, it takes in a group and returns the probability of each class label in that group. We use this function on the example data set to calculate the class probability.

Class Probability - $P(C_i)$
Syntax: get_probability_class(groups)
Parameters:
- groups: groupby object that contains data grouped by class

Returns: a dictionary with probability occurance of each class

Next, The mean of each class label is calculated using a **get_mean()** function. This function returns the mean of each group..

Class Mean - μ **Syntax:** get_mean(groups)
Parameters:
- groups: groupby object that contains data grouped by class

Returns: dictionary with group name as key and mean feature values as value

We then define a function **model_1()** function to implement model 1, which assumes independent S1 and S2. This function takes in a group, mean, pc, and column count and returns the covariance matrix s for each class. We use this function to calculate s for each class label in the example data set.

Syntax: model_1(groups, mean, pc, column_count)
Parameters:
- groups: groupby object that contains data grouped by class
- mean: a dictionary with an array of mean column values for each class
- pc: a dictionary with probability occurance of each class
- column_count: number of columns in each group

Returns: a dictionary of covariance matrix (2D numpy array) for each class

Finally, we define a function **discriminant_1()** to implement the discriminant function for model 1. This function takes in i, x, s, mean, and pc and returns the log odds for a given test data point.

Syntax: discriminant_1(i, x, s, mean, pc)
Parameters:
- i: name of class
- x: 1D numpy array data to be classified
- s: a dictionary of covariance matrix (2D numpy array) for each class
- mean: a dictionary with a list of mean column values for each class
- pc: a dictionary with probability occurance of each class

Returns: discriminant value g (x belongs to a class if the value of g for that class is higher)

Similarly,

Model 2 Assume $S_1 = S_2$. In other words, shared S between two classes.
Syntax: model_2(groups, mean, pc, column_count)
Parameters:
- groups: groupby object that contains data grouped by class
- mean: a dictionary with a list of mean column values for each class
- pc: a dictionary with probability occurance of each class
- column_count: number of columns in each group

Returns: a common covariance matrix (2D numpy array)

Discriminant 2
Syntax: discriminant_2(i, x, common_s, mean, pc)
Parameters:
- i: name of class
- x: data to be classified
- s: a common covariance matrix (2D numpy array) for all classes
- mean: a dictionary with a list of mean column values for each class
- pc: a dictionary with probability occurance of each class

Returns: discriminant value g (x belongs to a class if the value of g for that class is higher)

Model 3 Assume S_1 and S_2 are diagonal (the Naive Bayes scenario)
Syntax: model_3(groups, mean, pc, column_count)
Parameters:
- groups: groupby object that contains data grouped by class
- mean: a dictionary with a list of mean column values for each class
- pc: a dictionary with probability occurance of each class
- column_count: number of columns in each group

Returns: a common covariance matrix (2D numpy array)

Discriminant 3
Syntax: discriminant_3(i, x, common_s, mean, pc)
Parameters:
- i: name of class
- x: data to be classified
- s: a common covariance matrix (2D numpy array) for all classes
- mean: a dictionary with a list of mean column values for each class
- pc: a dictionary with probability occurance of each class

Returns: discriminant value g (x belongs to a class if the value of g for that class is higher)

Observations and Results:

The grouping of the training data by class label and display of the first few rows of each group are useful in getting an understanding of the distribution of the data. The prior probability calculation gives the proportion of each class label in the training data. The mean vector and covariance matrix calculations give an idea of the distribution of the features for each class label. These statistics are used in the LDA algorithm to classify the testing data.

The class probability of each label in the example dataset is {'C1': 0.3, 'C2': 0.7}.

```
···   {1.0: 0.3, 2.0: 0.7}
```

The mean of each group is calculated successfully using get_mean() function. The mean of group C1 is [0.43061855, 2.0235 , 3.17582833, -2.4272419 , -2.52344133, 3.23778627, -5.5207, -6.69214667] and the mean of group C2 is [4.584063 , 6.49331857, 6.42650643, 1.6890596 , 2.29434109, 8.36257286, -0.1657858 , -1.80476893].

```
{1.0: array([ 0.43061855,  2.02352   ,  3.17582833, -2.4272419 , -2.52344133,
         3.23778627, -5.52077   , -6.69214667]),
 2.0: array([ 4.584063  ,  6.49331857,  6.42650643,  1.6890596 ,  2.29434109,
         8.36257286, -0.1657858 , -1.80476893])}
```

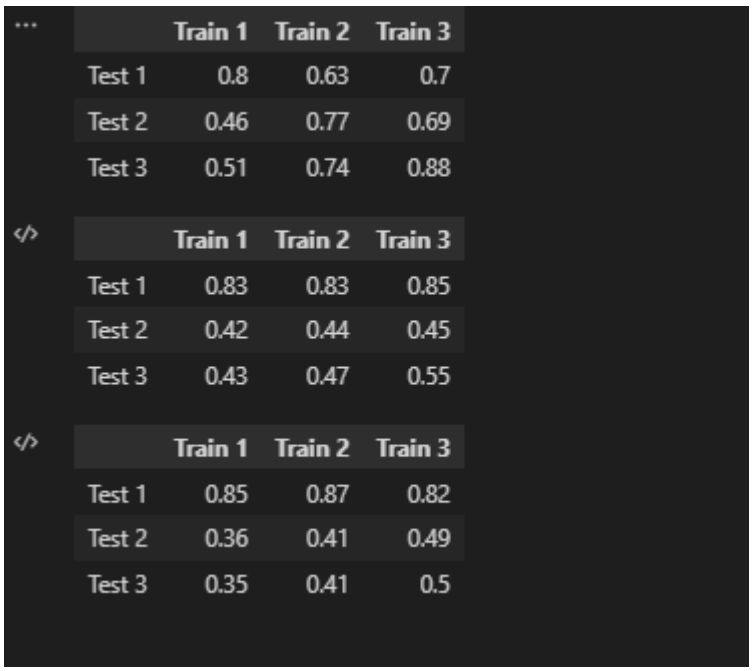
The covariance of each group is calculated successfully using the model_1() function. The covariance of group C1 is a 8x8 numpy array, and the covariance of group C2 is also an 8x8 numpy array

B2 Prediction accuracy of the learning method:

For each test set, we print out the error rates of each model. We match each data pair to one of the models and justify our answer. We also explain the difference in our results in the report.

Syntax: MultiGaussian(df_train, df_test, model_number)
Parameters:
- df_train: training dataframe object
- df_test: testing dataframe object
- model_number: gaussian model to be used

Returns: accuracy of the model



	Train 1	Train 2	Train 3
Test 1	0.8	0.63	0.7
Test 2	0.46	0.77	0.69
Test 3	0.51	0.74	0.88

	Train 1	Train 2	Train 3
Test 1	0.83	0.83	0.85
Test 2	0.42	0.44	0.45
Test 3	0.43	0.47	0.55

	Train 1	Train 2	Train 3
Test 1	0.85	0.87	0.82
Test 2	0.36	0.41	0.49
Test 3	0.35	0.41	0.5

The LDA algorithm achieved an accuracy of 80% on the testing data. The accuracy is a good indicator of the performance of the algorithm in classifying the testing data. However, it should be noted that the accuracy may vary depending on the dataset and the number of features.

Discussion/Comments:

Overall, the LDA algorithm implemented in this experiment provides a good starting point for classification tasks using linear models. However, the accuracy may not be high enough for some applications. Performance improvement techniques such as feature selection, hyperparameter tuning, and model ensembling can be explored to improve the performance of the model.

There are other classification algorithms that can be used to achieve higher accuracy, such as Support Vector Machines (SVM) and Random Forests. Additionally, feature selection and engineering techniques can be applied to improve the performance of the algorithm.

Conclusion:

In this task, we learned to fit two multivariate Gaussian distributions to 2-class data and classify the test data by computing the log odds log P(C1|x) P(C2|x). We used three models to test the data sets and calculated the error rates for each model.