CO 544- Machine Learning and Data Mining

Project Report

GROUP 14

ISHANTHI D.S. – E/15/139

PAMODA W.A.D. – E/15/249

RANUSHKA L.M. - E/15/299

Introduction

Machine learning and Data mining can aid in solving problems, which may include a large amount of heterogeneous data. In this project, machine learning and data mining is used to solve a classification problem with 15 features.

Technologies Used

Python 3.7.3 along with Anaconda and Jupyter Notebook was used to train the model and predict results. By using Jupyter Notebook we could run short blocks of code and see the results quickly, making it easy to test and debug the code.

We used libraries such as Scikit-learn, NumPy and Pandas. Scikit-learn is one of the best and most documented machine learning libraries for Python. NumPy is a python library, which adds support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays while Pandas is a python library for data manipulation and analysis.

Final Approach to Predictions

Read datasets

First, we read the train dataset and test dataset separately from the two-csv files. Then we add a column: ['train'] to identify whether each row of data belongs to train or test data. After that, we concatenate train and test datasets making it a single dataset.

Handle missing values

Since there were some missing values in the dataset, we replace these missing values (question marks in this dataset) as NaN by using the replace() function. However, values with a NaN value are ignored from operations such as sum, count, etc. Therefore, we had to handle these missing values.

For handling numerical missing values (in A2 and A14 columns), we used mean imputation method. Mean imputation replaces missing values with the mean value of that feature/variable. Mean imputation is one of the simplest and commonly used imputation methods.

```
# Create an imputer object that looks for 'Nan' values, then replaces them with the mean value of the feature by columns (axis=0)

mean_imputer = Imputer(missing_values='NaN', strategy='mean', axis=0)

# Train the imputor on the df dataset
```

```
mean_imputer = mean_imputer.fit(df)


# Apply the imputer to the df dataset
imputed_df = mean_imputer.transform(df.values)
```

Then to handle non-numeric missing values, we replace them using forward fill. We use forward-fill or back-fill to propagate the next values backward or previous value forward. This 'forward fill' will propagate the last valid observation forward over index axis.

```
df=df.ffill(axis = 0)
```

<u>One hot encode</u>

We can see in our data set, there are some columns that have categorical data. Categorical variables can take on only a limited and usually fixed number of possible values.

One hot encoding allows the representation of categorical data to be more expressive.

In these cases, we would like to give the network more expressive power to learn a probability-like number for each possible label value. This makes the categorical variables easier to quantify and compare. When one hot encoding is used for the output variable, it may offer a more characterized set of predictions than a single label.

Therefore we one hot encode columns that have categorical data (A1, A3, A4, A6, A8, A9, A11, A13, A15 columns).

We used the get_dummies function in pandas library for categorical variables. pd.get_dummies creates a new data frame which consists of zeros and ones. The data frame will have a one depending on the value the variable has.

In creating dummy variables, we essentially created new columns for our original dataset. Then we concatenate these new columns with the other columns.

We use pd.concat (the concatenation function) for this. As they are columns, we concatenate them on axis=1.

	A15_g	A15_p	A15_s
0	1	0	0
1	1	0	0
2	1	0	0
3	1	0	0
4	1	0	0
..	...	...	...
133	1	0	0
134	1	0	0
135	1	0	0
136	1	0	0
137	1	0	0

	A8_False	A8_True
0	0	1
1	0	1
2	1	0
3	0	1
4	0	1
..	...	...
133	0	1
134	0	1
135	0	1
136	1	0
137	1	0

Figure 1: Dummy columns after one hot encode

Label encode

Most of the algorithms work better with numerical inputs. Therefore, we should convert text/categorical data into numerical data. (Like we did in one hot encoding). This approach is very simple and it involves converting each value in a column to a number. We use label encode instead of one hot encode to encode the class variable (A16) because if we use one hot encode it will create two columns related to class variable and it will create problems when fitting into the model.

```
# Import label encoder
from sklearn import preprocessing


# label_encoder object knows how to understand word labels.
label_encoder = preprocessing.LabelEncoder()


# Encode labels in column 'species'.
df['species']= label_encoder.fit_transform(df['species'])
```

After that, we concatenate label encoded A16 and ‘train’ column with other (one hot encoded and other) columns.

Then we divide the whole data set again into train and test sets.

```
#devide to train and test again
train_df=final[final['train']==1]
test_df=final[final['train']==0]
```

This concatenation and division was done to avoid the mismatch of the number of columns in the train and test datasets. Since the number of columns we get after one hot encode depends on the values that a feature has, it definitely depends on the dataset we consider. Therefore, if we one hot encode train and test datasets separately, number of columns of train and test sets would mismatch. To avoid this mismatching, first we consider the both datasets together and one hot encode is done and then they are separated again. In that way we can ensure that both datasets have the same number of feature columns (46 columns) after one hot encode is done.

After that the 'train' column which we add to identify the each row of data is dropped.

```
#drop last column in both
train_df_new=train_df.drop(['train'],axis=1)
test_df_new=test_df.drop(['train'],axis=1)
```

Then, the full set of features (without class column) of train set is taken as X.

```
#take full attributes set
X=train_df_new.iloc[:, 0:46]
```

Split train data

To evaluate how well a classifier is performing, we should always test the model on unseen data. Therefore, before building a model, we split data into two parts: a training set and a test set. We use the training set to train and evaluate the model during the development stage. Then we use the trained model to make predictions on the unseen test set. This approach improves the model's performance and robustness. sklearn has a function called train_test_split(), which divides data into these sets.

We import the function and then use it to split the data. This function randomly splits the data using the test_size parameter. Here we have a test set (test) that represents 20% of the original dataset. The remaining data (train) is considered as the training data. We also add the respective labels for both the train/test variables, I.e. y_train, and y_test.

```
x_train, x_test, y_train, y_test = train_test_split(X,train_df_new['A16'], test_size=0.2,random_state = 5)
```

Here x_train is the values of features and y_train is the class column that are going to train the model. And x_test is the values of features that we are going to input to test the model. y_test is the values of class column that we are going to compare with the output of the model to calculate the accuracy.

Here, every time we run this without specifying `random_state`, we get a different result. Output changes. On the other hand if we use `random_state=some_number`, then we can guarantee that the output of Run 1 will be equal to the output of Run 2, i.e. split will always be the same. It does not matter what the actual `random_state` number is. In practice, it is said that we should set the `random_state` to some fixed number while we test stuff.

Building the Model

There are many models for machine learning, and each model has its own strengths and weaknesses. What model we should choose depends on the data set. Here we tried several algorithms and checked the accuracy as alternative approaches. In the final approach, we selected Random Forest classifier.

To create a Random Forest Classifier:

```
model = RandomForestClassifier(n_estimators=2000, criterion='entropy',random_state=1,max_features="sqrt",max_depth=100,min_samples_leaf=4,min_samples_split=10)
```

Hyperparameter Tuning

To increase the accuracy of the random forest classifier we used hyperparameter tuning.

Parameters which define the model architecture are referred to as hyperparameters and the process of searching for the ideal model architecture is referred to as hyperparameter tuning.

We can find out things like, what should be the maximum depth allowed for our decision tree model or how many trees should be included in the random forest model.

Here we use Grid search which is the most basic hyperparameter tuning method.

We import another library for this.

```
from sklearn.model_selection import GridSearchCV
```

Here we define a list of values to try for max_depth, max_features, min_samples_leaf, min_samples_split and n_estimators and a grid search builds a model for each possible combination.

Performing grid search over the defined hyperparameter space.:

```
param_grid = {'bootstrap': [True, False],

'max_depth': [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, None],

 'max_features': ['auto', 'sqrt'],

'min_samples_leaf': [1, 2, 4],

 'min_samples_split': [2, 5, 10],

'n_estimators': [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]}

n_estimators = [10, 50, 100, 200]

max_depth = [3, 10, 20, 40]
```

The grid of parameter values that were specified are shown in the code above. More parameter values can be specified, but the lesser values that are given, the faster the search will be.

Then each model would be fit to the training data and evaluated on the validation data.

```
clf = RandomForestClassifier(criterion='entropy')
CV_rfc = GridSearchCV(estimator=clf, param_grid=param_grid, cv= 5, verbose=10)
CV_rfc.fit(x_train, y_train)
```

Then the best parameter values and the accuracy are printed:

```
print(CV_rfc.best_params_)
print("score={}".format(CV_rfc.score(x_test,y_test)))
```

Parameters we considered:

n_estimators: This is the number of trees in the forest. A higher number of trees can be better for learning the data, but higher numbers can increase the time it takes for the training process.

max_depth: This is the maximum tree depth. The depth of the tree relates to how much information of the data is captured, so a deeper depth captures more information.

min_samples_leaf: This is the minimum required number of samples to be at a leaf node.

min_samples_split: This is the minimum required number of samples to split an internal node.

max_features: This is the size of the random subsets of features to consider when splitting a node.

Training the Model

To train the model using the training sets:

```
model.fit(x_train,y_train)
```

After we train the model, we can then use the trained model to make predictions on our test set, which we do using the predict() function with the test set.

```
y_pred = model.predict(x_test)
```

The predict() function returns an array of predictions for each data instance in the test set and it is stored in y_pred.

We can then print our predictions to get a sense of what the model determined.

Confusion matrix.

cm=confusion_matrix(y_test,y_pred)

A confusion matrix is a matrix (table) that can be used to measure the performance of a machine learning algorithm, usually a supervised learning one. Each row of the confusion matrix represents the instances of an actual class and each column represents the instances of a predicted class.

```
[[62  4]
 [ 1 44]]
0.954954954954955
```

Figure 2: Confusion matrix and the accuracy of the final approach

Evaluating the Model's Accuracy

We evaluate the accuracy of our model's predicted values by comparing the two arrays (x_test vs. y_test). We will use the sklearn function accuracy_score() to determine the accuracy of our machine learning classifier.

print(model.score(x_test,y_test))

Predictions

After successfully training the model, we use it to predict the output of test set given. (which was previously one hot encoded.) All the features of that data set are taken and used as input to the model we trained before.

Xnew=test_df_new.iloc[:, 0:46]

#predict for the data
y_prednew = model.predict(Xnew)

These final predictions are in ones and zeroes. We need to convert them into ‘Success ’ and ‘Failure’ accordingly. Therefore, we iterate through the array with predicted results and assign ‘Success ’ and ‘Failure’ accordingly.

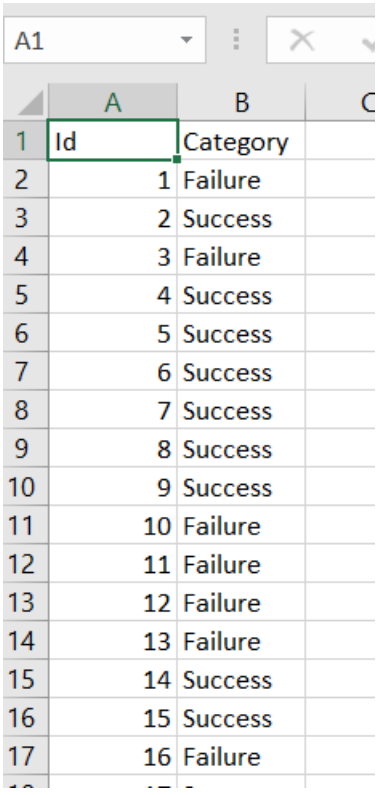
```
Final Predictions
['Failure', 'Success', 'Failure', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Failure', 'Failure', 'Fail
ure', 'Failure', 'Success', 'Success', 'Failure', 'Success', 'Success', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure',
'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failu
re', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure',
'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failu
re', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure',
'Failure', 'Success', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failu
re', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure',
'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failure', 'Failu
re', 'Failure', 'Failure', 'Failure', 'Success', 'Failure', 'Success', 'Success', 'Failure', 'Success', 'Failure', 'Success',
'Success', 'Failure', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Succe
ss', 'Failure', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success', 'Success']
```

Figure 3: Final Predictions

Writing to Excel File

As the final step, the given test data id and predicted category is written to an excel file sheet using xlsxwriter python library.

```
workbook = xlsxwriter.Workbook('submit6.xlsx')
sheet1 = workbook.add_worksheet()
sheet1.write('A1', 'Id')
sheet1.write('B1', 'Category')

row = 1
column = 1
number=1

for item in final :

    # write operation perform
    sheet1.write(row, 0, number)
    sheet1.write(row, column, item)

    # incrementing the value of row by one
    # with each iteratons.
    row += 1
    number=number+1


workbook.close()
```



	A	B	C
1	Id	Category	
2	1	Failure	
3	2	Success	
4	3	Failure	
5	4	Success	
6	5	Success	
7	6	Success	
8	7	Success	
9	8	Success	
10	9	Success	
11	10	Failure	
12	11	Failure	
13	12	Failure	
14	13	Failure	
15	14	Success	
16	15	Success	
17	16	Failure	

Figure 4: Final Predictions in excel sheet

Alternative Methods Used

Alternative 1 – Used Label Encode

In this alternative method:

- Missing values were handled using mean and forward filling.
- Label Encode was used.

Since many machine learning algorithms cannot work with categorical data directly, categories must be converted into numbers. In this method, label encoding was used to convert categorical data into numbers. Model Accuracy gained in this alternative was **0.8818181818181818**. Therefore, this alternative was not selected as the final approach since it gives a low accuracy.

When label encoding is used, numeric values can be misinterpreted by algorithms as having some sort of hierarchy or order in them. This might be the reason behind this low accuracy.

Alternative 2- Removed Missing Values

In this alternative method:

- Data records with missing values were removed from the train data set.
- Label Encode

In this alternative, rows with missing values were removed from the train data set assuming that taking mean and forward filling is not accurate. The data set of 547 rows was transformed into a data set of 524 rows after dropping these rows.

But, this returned a low model accuracy of **0.8476190476190476**. Therefore alternative was not selected as the final approach.

The reason for this low accuracy might be that this data sample is not large enough to drop data without substantial loss of statistical power.

Alternative 3- Used One Hot Encode

In this alternative method:

- Missing values were handled using mean and forward filling
- One Hot Encode

By using one hot encode instead of label encode, the accuracy of the model was increased. One hot encoding allows the representation of categorical data to be more expressive. This is better than label encoding since its result is binary rather than ordinal and that everything sits in an orthogonal vector space. This returned a higher model accuracy of **0.8909090909090909**. Therefore instead of label encoding, One hot encode was used in final approach.

However, after doing one hot encode separately for train and test data sets, the number of columns in the training and test set data were unequal. To avoid this confliction, initially both test and train sets were concatenated, then the one hot encoding was done and finally the data was again divided into train and test sets.

Alternative 4, 5, 6, 7- Different Classification Methods

In this alternative method:

- Missing values were handled using mean and forward filling
- One Hot Encode
- Different Classification Methods

Different classification methods were considered to get the maximum model accuracy.

- Alternative 4- Random Forest Classifier Model

Random forests is a supervised learning algorithm. It can be used both for classification and regression. It is also the most flexible and easy to use algorithm. A forest consists of trees. It is said that the more trees it has, the more robust a forest is. Random forests create decision trees on randomly selected data samples, get prediction from each tree and select the best solution by means of voting. In other words each tree votes and the most popular class is chosen as the final result.

This method is highly accurate and robust method because of the number of decision trees participating in the process. Also it does not suffer from the overfitting problem. The main reason is that it takes the average of all the predictions, which cancels out the biases. But the whole process is time-consuming because it has multiple decision trees.

```
model = DecisionTreeClassifier()
model.fit(x_train,y_train)
```

- Alternative 5-Naive Bayes Classification

In this approach we focused on a simple algorithm that usually performs well in binary classification tasks, namely Naive Bayes (NB). For this first, we import the GaussianNB module. Then the model is initialized and trained by fitting it to the data using following.

```
model= GaussianNB()

model.fit(x_train,y_train)
```

- Alternative 6 -Decision Tree Classification

Decision Trees can be used as classifier or regression models. Here a tree structure is constructed that breaks the dataset down into smaller subsets eventually resulting in a prediction. There are decision nodes that partition the data and leaf nodes that give the prediction that can be followed by traversing simple IF..AND..AND….THEN logic down the nodes.

The root node (the first decision node) partitions the data based on the most influential feature partitioning. There are 2 measures for this, Gini Impurity and Entropy.

First we load the required libraries.(*Import Decision Tree Classifier from sklearn*)

```
from sklearn.tree import DecisionTreeClassifier
```

And then we create Decision Tree classifier object

```
model = DecisionTreeClassifier()
```

Then we Train Decision Tree Classifier

```
model.fit(x_train,y_train)
```

- Alternative 7-Logistic Regression

Logistic regression is a fundamental classification algorithm. Logistic regression is fast and relatively uncomplicated, and it's convenient to interpret the results. Basically what happens in this technique is finding the logistic regression function such that the predicted responses are as close as possible to the actual response for each observation.

Once we have the logistic regression function, we can use it to predict the outputs for new and unseen inputs, assuming that the underlying mathematical dependence is unchanged.

For this we import the LogisticRegression library from scikit-learn.

```
from sklearn.linear_model import LogisticRegression
```

Like shown below we define the classification model and we represent it with an instance of the class LogisticRegression:

```
model= LogisticRegression()

model.fit(x_train,y_train)
```

The above statement creates an instance of LogisticRegression and binds its references to the variable model. LogisticRegression has several optional parameters that define the behavior of the model and approach:

Result accuracies gained from different classification methods are shown below.

Classification Method	Model Accuracy
Random Forest Classification	0.9099099099099099
Naive Bayes Classification	0.8727272727272727
Decision Tree Classification	0.7818181818181819
Logistic Regression	0.8818181818181818

According to the above results, Random Forest Classification was the best model for this dataset. Therefore, Random Forest Classification was used in the final approach.

Alternative 8- Hyperparameter Tuning (Final approach)

In this alternative method:

- Missing values were handled using mean and forward filling
- One Hot Encode
- Random Forest Classification
- Hyperparameter tuning

Hyperparameters are important because they directly control the behavior of the training algorithm and have a significant impact on the performance of the model is being trained. Proving this, model accuracy reached a maximum value of **0.954954954954955** when hyperparameters are tuned.

Alternative 9- Filled missing non numerical values with most frequent value

In this alternative method:

- Hyperparameter tuning
- Missing values were handled using mean and most frequent value
- One Hot Encode
-

Here, while numeric missing values were filed using the mean of the specific column, non numeric missing values were filled using most frequently recorded value of that specific column.

This returned a model accuracy of **0.954954954954955**. But, the model accuracy was neither increased nor decreased.

Alternative 10- Different Train –Validation split percentages

In this alternative method:

- Hyperparameter tuning
- Missing values were handled using mean and forward filling method
- One Hot Encode
- Different Train –Validation split percentages

In this code, different ratios between training set and validation set were used to find the ideal ratio which gives the maximum model accuracy. Result accuracies gained from different ratios are shown below.

Training Set Percentage	Validation Set Percentage	Model Accuracy
85%	15%	0.9397590361445783
80%	20%	0.954954954954955
70%	30%	0.9337349397590361
60%	40%	0.918552036199095

According to these results, when using 80% of training set and 20% of validation set, maximum model accuracy was gained. Therefore, it was used in the final approach.

Conclusion

After checking the accuracies of the different alternatives, the best approach was selected depending on the highest model accuracy. Using this final approach, a maximum model accuracy of **0.954954954954955** could achieved for this classification problem.