NOTRE DAME UNIVERSITY BANGLADESH

Machine Learning Lab Report-06

Course Code: CSE4214

Course Title: Machine Learning Lab

Lab Task Topic: Logistic Regression & Support Vector Machine

Submitted by:

Name: Istiak Alam

ID: 0692230005101005

Batch: CSE-20

Submission Date: April 26, 2026

Submitted to:

A. H. M. Saiful Islam

Chairman, Dept of CSE

Notre Dame University Bangladesh

Table of Contents

1 Objective

The objective of this lab is to implement and analyze two supervised machine learning algorithms: Logistic Regression and Support Vector Machine (SVM). The goal is to understand how these classification techniques work on real-world datasets and to evaluate their performance in predicting categorical outcomes.

Specifically, this lab aims to:

- Apply Logistic Regression to model the probability of a binary outcome based on input features.
- Implement Support Vector Machine (SVM) to classify data by finding the optimal hyperplane.
- Compare the effectiveness of both models in terms of accuracy and decision boundaries.
- Gain practical experience in data preprocessing, model training, and evaluation.

Both models successfully classify the given datasets, allowing comparison between Logistic Regression and SVM in terms of prediction accuracy and performance.

2 Dataset Description

In this lab, three datasets are used to demonstrate and evaluate the performance of Logistic Regression and Support Vector Machine (SVM):

- **marital status.csv:** This dataset contains information related to individuals' marital status along with several independent features such as age, income, or other demographic attributes. It is used for general data exploration and preprocessing.
- **marital status_Log_Reg.csv:** This dataset is specifically prepared for Logistic Regression. It contains input features and a binary target variable representing marital status (e.g., Married or Unmarried), making it suitable for binary classification.
- **svm.csv:** This dataset is used for applying the Support Vector Machine algorithm. It typically includes feature variables and corresponding class labels, allowing SVM to determine the optimal separating hyperplane between classes.

These datasets provide a practical basis for understanding how different classification algorithms perform on structured data. The datasets are successfully loaded and utilized for training and testing both Logistic Regression and SVM models, enabling proper evaluation and comparison of results.

3 Logistic Regression

3.1 Importing Libraries and Loading Dataset

Explanation

In this step, essential Python libraries are imported to perform data analysis and visualization. The `numpy` library is used for numerical operations, while `pandas` is used for handling and manipulating the dataset. The `matplotlib.pyplot` and `seaborn` libraries are used for data visualization. The `train_test_split` function from `sklearn.model_selection` is imported to divide the dataset into training and testing sets.

The dataset `marital status.csv` is then loaded into a DataFrame using `pandas.read_csv()`, and the DataFrame is displayed to observe the structure and contents of the dataset.

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import seaborn
%matplotlib inline
```

[2]:
```
df = pd.read_csv('marital status.csv')
```

[3]:
```
df
```

Output

The output shows the loaded dataset in tabular form, including rows and columns with feature values and the target variable. It provides an initial view of the data, helping to understand its structure, data types, and possible preprocessing requirements.

[3]:

	age	marital_status
0	25	Single
1	30	Married
2	NaN	Single
3	45	Divorced
4	35	Married
5	28	Single
6	40	NaN
7	55	Divorced
8	,	Married
9	27	Single

3.2 Checking Null/NaN Values in Dataset

Explanation

This code is used to identify missing (Null/NaN) values in the dataset. The `isnull().sum()` function checks each column and returns the total number of missing values. Initially, the check is performed on the existing dataframe, and then the dataset `marital status_Log_Reg.csv` is loaded using `read_csv()`, followed by another check for missing values. This helps ensure data quality before applying machine learning models.

[4]:
```
df.isnull().sum()
```

[4]:
```
age              1
marital_status   1
dtype: int64
```

[5]:
```
df = pd.read_csv('marital status_Log_Reg.csv')
```

[6]:
```
df
```

Output

The output displays the number of null values for each column in the dataset. If all values are zero, it indicates that there are no missing values and the dataset is clean. Otherwise, it shows which

columns contain missing data that may require preprocessing.

[6]:

	age	marital_status
0	25	0
1	30	1
2	60	0
3	45	1
4	35	1
5	28	0
6	40	1
7	55	1
8	37	1
9	27	0
10	26	0
11	32	1
12	35	0
13	47	1
14	37	1
15	29	0
16	41	1
17	53	1
18	45	1
19	25	0

[7]:
```
df.isnull().sum()
```

[7]:
```
age              0
marital_status   0
dtype: int64
```

3.3 Handling Missing Values and Basic Statistics

Explanation

In this code, missing values in the `age` column are handled by replacing them with the mean value of that column using the `fillna()` function. This ensures that the dataset remains consistent and no rows are dropped due to null values.

The `info()` function is then used to display a summary of the dataset, including the number of non-null values and data types of each column.

Finally, the median of the `marital_status` column is calculated and stored in a variable, which can be useful for understanding the central tendency of the target variable.

[8]:
```
updated_df = df
updated_df['age']=updated_df['age'].fillna(updated_df['age'].mean())
updated_df.info()
```

Output

The output shows that all missing values in the `age` column have been successfully filled, resulting in no null entries in that column. The dataset summary confirms the updated non-null counts and data types. Additionally, the computed median value of the `marital_status` column is displayed as a numerical result.

```
<class 'pandas.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 2 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   age             20 non-null     int64
 1   marital_status  20 non-null     int64
dtypes: int64(2)
memory usage: 452.0 bytes
```

[9]:
```
handle = df['marital_status'].median()
```

[10]:
```
handle
```

[10]:
```
np.float64(1.0)
```

3.4 Handling Missing Values in Dataset

Explanation

In this step, missing values in the `marital_status` column are handled by replacing them with the median value stored in the variable `handle`. This ensures that the dataset does not contain null values, which could negatively affect model performance.

After filling the missing values, the dataset is displayed to verify the changes. Then, the `isnull().sum()` function is used to confirm that there are no remaining null values in any column. Finally, `value_counts()` is applied to the `marital_status` column to observe the distribution of different classes.

[11]:
```
df.marital_status= df.marital_status.fillna(handle)
```

[12]:
```
df
```

Output

The output shows that all missing values in the `marital_status` column have been successfully replaced. The null value summary confirms zero missing values across the dataset. Additionally, the value counts display the frequency of each category in the `marital_status` column, helping to understand class distribution.

[12]:

	age	marital_status
0	25	0
1	30	1
2	60	0
3	45	1
4	35	1
5	28	0
6	40	1
7	55	1
8	37	1
9	27	0
10	26	0
11	32	1

```
12   35              0
13   47              1
14   37              1
15   29              0
16   41              1
17   53              1
18   45              1
19   25              0
```

[13]:
```
df.isnull().sum()
```

[13]:
```
age              0
marital_status   0
dtype: int64
```

[14]:
```
df.marital_status.value_counts()
```

[14]:
```
marital_status
1    12
0     8
Name: count, dtype: int64
```

3.5 Feature and Target Variable Selection

Explanation

In this step, the independent feature and dependent target variable are separated from the dataset. The variable `x` is assigned the `age` column as the input feature, while `y` is assigned the `marital_status` column as the target variable. This separation is necessary for training machine learning models, where `x` represents the input data and `y` represents the output labels.

[15]:
```
x= df[['age']]
```

[16]:
```
x
```

Output

The variable `x` displays a DataFrame containing the `age` values of all records, while `y` displays a Series containing the corresponding `marital_status` labels (such as Married or Unmarried). These variables are now ready to be used for model training.

[16]:
```
    age
0    25
1    30
2    60
3    45
4    35
5    28
6    40
7    55
8    37
9    27
```

```
10   26
11   32
12   35
13   47
14   37
15   29
16   41
17   53
18   45
19   25
```

[17]:
```
y= df['marital_status']
```

[18]:
```
y
```

[18]:
```
0     0
1     1
2     0
3     1
4     1
5     0
6     1
7     1
8     1
9     0
10    0
11    1
12    0
13    1
14    1
15    0
16    1
17    1
18    1
19    0
Name: marital_status, dtype: int64
```

3.6 Dataset Splitting using train_test_split

Explanation

This code splits the dataset into training and testing subsets using the `train_test_split` function from `sklearn.model_selection`. The input features (`x`) and target variable (`y`) are divided such that 70% of the data is used for training and 30% is used for testing (`test_size = 0.30`).

The parameter `random_state = 1` ensures reproducibility, meaning the same split will be generated every time the code is executed. The resulting variables are:

- `xtrain`: Training feature set
- `xtest`: Testing feature set
- `ytrain`: Training labels

- `ytest`: Testing labels

Finally, `xtest` and `ytest` are displayed to observe the testing data samples and their corresponding labels.

[19]:
```
from sklearn.model_selection import train_test_split
```

[20]:
```
#xtrain,ytrain,xtest,ytest = train_test_split(x,y,test_size= .20,␣
↪random_state= 1)
xtrain, xtest, ytrain, ytest = train_test_split(x, y, test_size = 0.30,␣
↪random_state =1)
```

Output

The output shows a portion of the testing dataset:

- `xtest`: Displays feature values of the test samples.
- `ytest`: Displays the actual class labels corresponding to those test samples.

This test data is later used to evaluate the performance of the trained machine learning model.

[21]:
```
xtest
```

[21]:
```
    age
3    45
16   41
6    40
10   26
2    60
14   37
```

[22]:
```
ytest
```

[22]:
```
3     1
16    1
6     1
10    0
2     0
14    1
Name: marital_status, dtype: int64
```

3.7 Logistic Regression Model Initialization

Explanation

This code imports the Logistic Regression class from the `sklearn.linear_model` module and initializes a Logistic Regression model. Logistic Regression is a supervised learning algorithm used for binary classification problems. The statement `model = LogisticRegression()` creates an instance of the model with default parameters, which can later be trained using training data. At this stage, no learning or fitting has occurred; the model is only prepared for training.

[23]:
```
from sklearn.linear_model import LogisticRegression
model = LogisticRegression ()
```

3.8 Model Training and Evaluation using Logistic Regression/SVM

Explanation

In this step, the machine learning model is trained using the training dataset and then evaluated using the test dataset. The 'fit()' function is used to train the model by learning patterns from the input features ('xtrain') and corresponding labels ('ytrain'). After training, the 'predict()' function is used to generate predictions on unseen test data ('xtest'). Finally, the 'score()' function is used to measure the model's performance by calculating its accuracy based on the comparison between predicted values and actual test labels ('ytest').

Output

The model is successfully trained on the training dataset and generates predictions for the test dataset. The final output shows the accuracy score of the model, which indicates how well the model performs in classifying unseen data. A higher score represents better model performance and better generalization capability.

[25]:
```
model.fit(xtrain, ytrain)
```

[25]:
```
LogisticRegression()
```

[26]:
```
model.predict(xtest)
```

[26]:
```
array([1, 1, 1, 0, 1, 1])
```

[27]:
```
#Finding out accuracy
model.score(xtest,ytest)
```

[27]:
```
0.8333333333333334
```

[28]:
```
#predicting performance using sigmoid function
model.predict_proba(xtest)
# here first values in the array is for no, second value is for yes
```

[28]:
```
array([[4.28664393e-03, 9.95713356e-01],
       [2.17640342e-02, 9.78235966e-01],
       [3.24559767e-02, 9.67544023e-01],
       [9.13231426e-01, 8.67685743e-02],
       [9.10031684e-06, 9.99990900e-01],
       [1.03119980e-01, 8.96880020e-01]])
```

3.9 Model Training and Prediction using Logistic Regression

Explanation

In this section, the Logistic Regression model is trained using the training dataset `xtrain` and `ytrain`. The `fit()` function is used to learn the relationship between input features and the target variable.

After training, the model is tested using `xtest` to generate predictions. The `predict()` function is used to classify the test data into discrete classes (e.g., Married or Not Married).

To evaluate the probability of each class, the `predict_proba()` function is used. This function returns probability values for both classes, where the first value represents the probability of class "No" and the second value represents the probability of class "Yes".

Additionally, a custom prediction function is implemented to predict marital status based on a single input feature (age). The input age is reshaped into a 2D array and passed into the trained model. The output is then converted into a human-readable format.

[30]:
```
# Predicting status of anyother single value providing age (one of the ways)
# Assuming you have a trained logistic regression model called 'model'
# model = LogisticRegression() # Your trained model here
def predict_marriage_status(model, age):

    # Prepare the input as a 2D array
    age_input = np.array([[age]])

    # Get the prediction: 1 for married (Yes), 0 for not married (No)
    prediction = model.predict(age_input)[0]

    # Convert prediction to readable format
    return "Yes" if prediction == 1 else "No"

# Example usage
age = 42 # Replace with any age you want to predict
status = predict_marriage_status(model, age)

print(f"Prediction: {status}")
```

Output

The model successfully learns from the training data and predicts marital status on the test dataset. The `predict_proba()` output shows probability distributions for each class, helping to understand model confidence.

For a sample input age (e.g., 42), the model returns a prediction such as: **Prediction: Yes** or **Prediction: No**, indicating whether the person is likely married based on the trained logistic regression model.

```
Prediction: Yes
```

[31]:
```
# Predicting status of anyother single value providing age (second another␣
 ↪way)
age = 30 # Replace with any age you want to predict
status = "Yes" if model.predict([[age]])[0] == 1 else "NO"

print(f"Prediction: {status}")
```

```
Prediction: NO
```

[34]:
```
# Predicting status of anyother single value providing age (third another␣
 ↪way)
# Assuming you have a trained logistic regression model called 'model'
# model = LogisticRegression() # Your trained model here
```

```
# Predict marriage status for a given age
age = 30 # Replace with any age you want to predict
status = ["No", "Yes"] [int(model.predict([[age]])[0])]

print(f"Prediction: {status}")
```

```
Prediction: No
```

[35]:
```
age = 30 # Replace with any age you want to predict
status = ["No", "Yes"][int(model.predict([[age]])[0])]
print(f"Prediction: {status}")
```

```
Prediction: No
```

4 SVM Classifier

4.1 Data Loading

Explanation

In this section, the Support Vector Machine (SVM) classification process is implemented. First, all the necessary libraries are imported, including pandas for data handling, train_test_split for splitting the dataset, StandardScaler for feature scaling, LabelEncoder for encoding categorical variables, SVC for the SVM model, and evaluation metrics such as accuracy_score and classification_report.

After importing the libraries, the dataset `svm.csv` is loaded using `pandas.read_csv()`. This dataset is then stored in a DataFrame named `data`, which is used for further preprocessing and model training.

[1]:
```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
```

[2]:
```
# Load the dataset
data = pd.read_csv('svm.csv')
data
```

Output

The dataset is successfully loaded into the system and displayed as a structured table (DataFrame). This confirms that the data is ready for preprocessing steps such as encoding, scaling, and splitting before training the SVM classifier.

[2]:

	Length	Weight	Snout Width	Habitat	Species
0	3.4	400	20	Saltwater	Crocodile
1	4.2	500	22	Freshwater	Alligator
2	3.9	450	19	Saltwater	Crocodile
3	4.5	600	24	Freshwater	Alligator
4	2.8	300	21	Saltwater	Crocodile
5	3.6	480	23	Freshwater	Alligator
6	5.0	700	20	Saltwater	Crocodile
7	4.1	550	25	Freshwater	Alligator
8	3.3	420	18	Saltwater	Crocodile
9	4.3	530	24	Freshwater	Alligator

4.2 Data Preprocessing using Label Encoding for SVM

Explanation

In this section, categorical variables in the dataset are converted into numerical format using `LabelEncoder`, which is necessary for applying the Support Vector Machine (SVM) algorithm.

The columns `Habitat` and `Species` contain categorical text values, which cannot be directly processed by the SVM model. Therefore, `LabelEncoder` is used to transform these categorical values

into numeric labels.

First, the `Habitat` column is encoded into numerical values, and then the same encoder is used to convert the `Species` column into numeric form. This ensures that all input features are in a machine-readable format suitable for training the SVM classifier.

[3]:
```
# Encode the categorical 'Habitat' and 'Species' columns
label_encoder = LabelEncoder()
data['Habitat'] = label_encoder.fit_transform(data['Habitat'])

# Convert habitat to numeric
data['Species'] = label_encoder.fit_transform(data['Species'])

# Convert species to numeric
data
```

Output

After applying label encoding, both `Habitat` and `Species` columns are successfully converted into numeric values. The dataset is now fully prepared for training the SVM model, enabling it to perform classification based on the transformed features.

[3]:

	Length	Weight	Snout Width	Habitat	Species
0	3.4	400	20	1	1
1	4.2	500	22	0	0
2	3.9	450	19	1	1
3	4.5	600	24	0	0
4	2.8	300	21	1	1
5	3.6	480	23	0	0
6	5.0	700	20	1	1
7	4.1	550	25	0	0
8	3.3	420	18	1	1
9	4.3	530	24	0	0

4.3 SVM Classifier Training and Evaluation

Explanation

In this section, a Support Vector Machine (SVM) classifier is implemented to classify species based on physical and environmental features. First, the dataset is divided into input features `X` (Length, Weight, Snout Width, Habitat) and target variable `y` (Species).

The dataset is then split into training and testing sets using an 70-30 ratio to evaluate model performance on unseen data. To improve model efficiency and ensure all features contribute equally, feature scaling is applied using `StandardScaler`.

An SVM model with a linear kernel is initialized using `SVC(kernel='linear', C=1.0)`. The linear kernel is suitable for linearly separable data and helps in finding an optimal hyperplane that separates the classes.

After training the model using `fit()`, predictions are made on the test dataset using `predict()`. The model performance is evaluated using accuracy score and a classification report, which includes precision, recall, and F1-score for each class.

[4]:
```
# Separate features and target
X = data[['Length', 'Weight', 'Snout Width', 'Habitat']]
y = data['Species']
```

[5]:
```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

[6]:
```
# Standardize the feature values
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[7]:
```
# Initialize and train the SVM model
svm_model = SVC(kernel='linear', C=1.0)
# Linear kernel SVM for binary classification
svm_model.fit(X_train, y_train)

SVC(kernel='linear')
```

[7]:
```
SVC(kernel='linear')
```

[8]:
```
# Make predictions on the test set
y_pred = svm_model.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
report = classification_report(y_test, y_pred,
target_names=['Alligator', 'Crocodile'])
```

[9]:
```
# Output the results
print("Accuracy:", accuracy)
print("Classification Report:\n", report)
```

Output

The trained SVM model produces classification results on the test dataset with a computed accuracy value (e.g., 0.85 or 85% depending on data split). The classification report shows detailed performance metrics for both classes: **Alligator** and **Crocodile**, including precision, recall, and F1-score, indicating how well the model distinguishes between the two species.

```
Accuracy: 1.0
Classification Report:
               precision    recall  f1-score   support

   Alligator       1.00      1.00      1.00         2
   Crocodile       1.00      1.00      1.00         1

    accuracy                           1.00         3
   macro avg       1.00      1.00      1.00         3
weighted avg       1.00      1.00      1.00         3
```

4.4 Habitat and Species Encoding using Mapping Dictionaries

Explanation

In this section, categorical data is converted into numerical format using mapping dictionaries, which is a necessary preprocessing step for machine learning models like Support Vector Machine (SVM).

The `habitat_mapping` dictionary is used to convert habitat types into numerical values, where:

- *Freshwater* is encoded as 0
- *Saltwater* is encoded as 1

Similarly, the `species_mapping` dictionary is used to convert predicted numeric class labels into meaningful species names:

- 0 represents *Alligator*
- 1 represents *Crocodile*

This encoding helps the SVM model process categorical inputs efficiently and also improves interpretability of the output results.

[10]:
```
# Define the habitat mapping outside the function
habitat_mapping = {
 'Freshwater': 0,
 'Saltwater': 1
}
```

[11]:
```
# Define species mapping as well, for completeness
species_mapping = {0: 'Alligator', 1: 'Crocodile'}
```

4.5 SVM Classification and Species Prediction

Explanation

In this section, a Support Vector Machine (SVM) model is used to classify an input sample into one of two species: Crocodile or Alligator. A custom prediction function `predict_species()` is defined to handle new input data.

First, the categorical feature `habitat` is converted into a numerical value using a predefined mapping (`habitat_mapping`). If an invalid habitat is provided, the function displays an error message and stops execution.

Next, the input features (length, weight, snout width, and encoded habitat) are combined into a DataFrame to ensure consistency with the training dataset structure. This data is then scaled using the previously fitted scaler (`scaler`) to match the preprocessing applied during training.

The processed input is passed to the trained SVM model (`svm_model`) to generate a prediction. The numerical output is then converted back into a meaningful class label (Crocodile or Alligator) using `species_mapping`.

[12]:
```
# Function to predict whether a new input is a Crocodile or Alligator
def predict_species(length, weight, snout_width, habitat):
     # Convert habitat to numeric using the previously saved mapping
     if habitat in habitat_mapping:
      habitat_num = habitat_mapping[habitat]
```

```
    else:
     print("Invalid habitat. Please use 'Freshwater' or 'Saltwater'.")
     return
    # Prepare the feature vector and scale it
    # Convert the new data into a DataFrame with the same columns as X_train
    new_data_df = pd.DataFrame([[length, weight, snout_width, habitat_num]],
                               columns=['Length', 'Weight', 'Snout Width',␣
↪'Habitat'])
    new_data_scaled = scaler.transform(new_data_df)
    # Predict using the SVM model
    prediction = svm_model.predict(new_data_scaled)

    # Map the numeric prediction back to species name
    species_name = species_mapping[prediction[0]]
    print(f"The predicted species is: {species_name}")
   # Example input to check the species
   # Example: Length=4.0 meters, Weight=500 kg, Snout Width=22 cm,
Habitat='Freshwater'
predict_species(4.0, 500, 22, 'Freshwater')
```

Output

The model successfully predicts the species based on the given input features. For example, for input values:

`Length = 4.0 meters, Weight = 500 kg, Snout Width = 22 cm, Habitat = Freshwater`

the model outputs:

The predicted species is: Crocodile (or Alligator depending on model decision boundary).

This demonstrates that the SVM model can effectively classify unseen data based on learned patterns from the training dataset.

```
The predicted species is: Alligator
```

[13]:
```
predict_species(3.4, 400, 20, 'Saltwater')
```

```
The predicted species is: Crocodile
```