Md Sen Bin Mustafiz
mbm52@njit.edu
NJIT ID: 31690921
24 Nov, 2024

Professor Yasser Abdullah
CS 634: Data Mining

Final Project Report

A machine learning classifier is an algorithm used to determine the category or class of a data point. It is a supervised learning technique where the model is trained on labeled data, consisting of input features and their corresponding output labels. The classifier identifies patterns in the training data and uses this understanding to classify new data.

Main Components of a Classifier: - Input Features: Characteristics or attributes of the data.
- Labeled Data: Data with known categories for training.
- Classification Model: The algorithm (e.g., Decision Tree, SVM, Neural Networks) that learns from the data.
- Output Class: The predicted category for the input data.

A machine learning classifier relies on structured data to make accurate predictions, with **input features**, **labeled data**, and **output classes** playing crucial roles in its functioning. In this project I use Car Evaluation Database. It is based on a hierarchical decision model for evaluating car acceptability. It simplifies the decision structure by linking car acceptability directly to six input attributes:

1. buying (v-high, high, med, low)
2. maint (v-high, high, med, low)
3. doors (2, 3, 4, 5-more)
4. persons (2, 4, more)
5. lug_boot (small, med, big)
6. safety (low, med, high)

The dataset contains 1,728 instances with no missing values and classifies the data into four categories:

1. unacceptable
2. acceptable
3. good
4. very good

This dataset is widely used for testing machine learning methods such as structure discovery and constructive induction.

Classification Model: In this project I used 3 different classification algorithms in Python. They are: 1. Random Forest 2. Naïve Bayes 3. Bidirectional-LSTM

In evaluating classification performance, I also used the 10-fold cross validation metho in every classification model.d

0.0.1 Importing the package

Remove the # and import the package when you run it.

[6]:
```
#pip install tensorflow
```

0.0.2 Importing the libraries that are required for the project

[8]:
```
# Import libraries
import pandas as pd
from sklearn.model_selection import KFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix, accuracy_score, brier_score_loss,␣
↪roc_auc_score
from sklearn.preprocessing import LabelEncoder
import numpy as np
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, LSTM, Bidirectional
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import Input
import warnings
```

0.0.3 Data reading

[10]:
```
# Load the dataset
data = pd.read_csv('car.csv')  # csv file

# Encode catagory
label_encoders = {}
for column in data.columns:
    le = LabelEncoder()
    data[column] = le.fit_transform(data[column])
    label_encoders[column] = le

# divide
X = data.drop(columns='class')
y = data['class']
```

0.0.4 10 fold cross validation

[12]:
```
# k = 10 fold
kfold = KFold(n_splits=10, shuffle=True, random_state=42)
```

0.1 1. Random Forest Classifier

[14]:
```python
# Random Forest Classifier
rf_mod = RandomForestClassifier(random_state=42)
```

Here I used Random Forest classifier to calculate values like Confusion matrix, Sensitivity, Specificity, False Positive Rate, False Negative Rate, precision, F1 score, Balanced Accuracy, True Skill Statistic, Heidke Skill Score and AUC. The results for each fold are stored for overall evaluation.

[16]:
```python
# empty list to store values for each fold
fold_values = []

for i, (train_index, test_index) in enumerate(kfold.split(X), start=1):
    # Splitting the data
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]     # Train
    rf_mod.fit(X_train, y_train)
    y_pred = rf_mod.predict(X_test)

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    tp = cm.diagonal()  # True Positives
    fn = cm.sum(axis=1) - tp  # False Negatives
    fp = cm.sum(axis=0) - tp  # False Positives
    tn = cm.sum() - (fp + fn + tp)  # True Negatives



    p = tp + fn
    n = tn + fp
    TPR = tp / (tp + fn)  # Sensitivity
    TNR = tn / (tn + fp)  # Specificity
    FPR = fp / (fp + tn)  # False Positive Rate
    FNR = fn / (fn + tp)  # False Negative Rate
    Precision = tp / (tp + fp)  # Precision
    F1_measure = 2 * (Precision * TPR) / (Precision + TPR)  # F1 Score
    Accuracy = accuracy_score(y_test, y_pred)
    Error_rate = 1 - Accuracy
    BACC = (TPR + TNR) / 2  # Balanced Accuracy
    TSS = TPR - FPR  # True Skill Statistic
    HSS = (2 * (tp * tn - fp * fn)) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp +␣
↪tn))  # Heidke Skill Score

    #  Brier Score
    y_proba = rf_mod.predict_proba(X_test)  # Probabilities
    brier_score = np.mean([(y_proba[:, i] - (y_test == i).astype(int)) ** 2 for␣
↪i in range(y_proba.shape[1])])

    #  AUC
```

```
    try:
        auc = roc_auc_score(y_test, y_proba, multi_class='ovr')
    except ValueError:
        auc = np.nan  #  NaN if calculation not meet

    # Store averaged values
    fold_values.append([
        tp.mean(), tn.mean(), fp.mean(), fn.mean(), p.mean(), n.mean(),
        TPR.mean(), TNR.mean(), FPR.mean(), FNR.mean(),
        Precision.mean(), F1_measure.mean(),
        Accuracy, Error_rate, BACC.mean(), TSS.mean(), HSS.mean(),
        brier_score, auc, Accuracy  # Acc_by_package_fn
    ])
```

0.1.1 Printing Output

[18]:
```
# values to DataFrame
values_df = pd.DataFrame(fold_values, columns=[
    "TP", "TN", "FP", "FN", "P", "N", "TPR", "TNR", "FPR", "FNR", "Precision",␣
 ↪"F1 measure",
    "Accuracy", "Error_rate", "BACC", "TSS", "HSS", "Brier score", "AUC",␣
 ↪"Acc_by_package_fn"
])

# Transpose
value_df_rf = values_df.T
value_df_rf.columns = [f"Fold : {i+1}" for i in range(value_df_rf.shape[1])]

# Display
value_df_rf
```

[18]:

	Fold : 1	Fold : 2	Fold : 3	Fold : 4	Fold : 5 \
TP	41.500000	42.750000	42.000000	42.500000	43.000000
TN	128.000000	129.250000	128.500000	129.000000	129.500000
FP	1.750000	0.500000	1.250000	0.750000	0.250000
FN	1.750000	0.500000	1.250000	0.750000	0.250000
P	43.250000	43.250000	43.250000	43.250000	43.250000
N	129.750000	129.750000	129.750000	129.750000	129.750000
TPR	0.940909	0.987179	0.921828	0.925000	0.937500
TNR	0.986434	0.993521	0.991248	0.994444	0.998252
FPR	0.013566	0.006479	0.008752	0.005556	0.001748
FNR	0.059091	0.012821	0.078172	0.075000	0.062500
Precision	0.887211	0.947984	0.937970	0.981707	0.991935
F1 measure	0.898857	0.964631	0.928447	0.946389	0.960187
Accuracy	0.959538	0.988439	0.971098	0.982659	0.994220
Error_rate	0.040462	0.011561	0.028902	0.017341	0.005780
BACC	0.963671	0.990350	0.956538	0.959722	0.967876

```
TSS                  0.927343    0.980700    0.913075    0.919444    0.935752
HSS                  0.885863    0.959523    0.916719    0.941380    0.958588
Brier score          0.023605    0.012539    0.015159    0.016942    0.010282
AUC                  0.994760    0.999785    0.999327    0.999321    0.999260
Acc_by_package_fn    0.959538    0.988439    0.971098    0.982659    0.994220

                     Fold : 6    Fold : 7    Fold : 8    Fold : 9   Fold : 10
TP                  42.750000   42.250000   43.000000   42.500000   42.000000
TN                 129.250000  128.750000  129.500000  128.500000  128.000000
FP                   0.500000    1.000000    0.250000    0.500000    1.000000
FN                   0.500000    1.000000    0.250000    0.500000    1.000000
P                   43.250000   43.250000   43.250000   43.000000   43.000000
N                  129.750000  129.750000  129.750000  129.000000  129.000000
TPR                  0.966684    0.926556    0.937500    0.991284    0.933333
TNR                  0.996296    0.989317    0.998106    0.993589    0.988126
FPR                  0.003704    0.010683    0.001894    0.006411    0.011874
FNR                  0.033316    0.073444    0.062500    0.008716    0.066667
Precision            0.987500    0.978247    0.994048    0.991284    0.984384
F1 measure           0.975886    0.947760    0.961274    0.991284    0.957204
Accuracy             0.988439    0.976879    0.994220    0.988372    0.976744
Error_rate           0.011561    0.023121    0.005780    0.011628    0.023256
BACC                 0.981490    0.957936    0.967803    0.992437    0.960730
TSS                  0.962980    0.915873    0.935606    0.984873    0.921460
HSS                  0.970889    0.935633    0.959601    0.984873    0.947713
Brier score          0.021500    0.014906    0.012831    0.016256    0.021362
AUC                  0.998276    0.999125    0.999815    0.999280    0.998830
Acc_by_package_fn    0.988439    0.976879    0.994220    0.988372    0.976744
```

0.2 2. Naive Bayes Model

Here I used Naive Bayes classifier to calculate values like Confusion matrix, Sensitivity, Specificity, False Positive Rate, False Negative Rate, precision, F1 score, Balanced Accuracy, True Skill Statistic, Heidke Skill Score and AUC. The results for each fold are stored for overall evaluation.

[21]:
```python
# Initialize Naive Bayes classifier

nb_model = GaussianNB()
```

[22]:
```python
#  empty list
fold_value = []

# Loop through each fold
for i, (train_index, test_index) in enumerate(kfold.split(X), start=1):
    # Splitting data
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]
```

```python
    # Train
    nb_model.fit(X_train, y_train)
    y_pred = nb_model.predict(X_test)

    # Confusion matrix
    cm = confusion_matrix(y_test, y_pred)
    tp = cm.diagonal()  # True Positives
    fn = cm.sum(axis=1) - tp  # False Negatives
    fp = cm.sum(axis=0) - tp  # False Positives
    tn = cm.sum() - (fp + fn + tp)  # True Negatives
    p = tp + fn
    n = tn + fp


    TPR = tp / (tp + fn)  # Sensitivity (Recall)
    TNR = tn / (tn + fp)  # Specificity
    FPR = fp / (fp + tn)  # False Positive Rate
    FNR = fn / (fn + tp)  # False Negative Rate

    #  Precision and F1_measure
    Precision = np.divide(tp, (tp + fp), out=np.zeros_like(tp, dtype=float),␣
↪where=(tp + fp) != 0)
    F1_measure = np.divide(2 * (Precision * TPR), (Precision + TPR), out=np.
↪zeros_like(TPR, dtype=float), where=(Precision + TPR) != 0)

    Accuracy = accuracy_score(y_test, y_pred)
    Error_rate = 1 - Accuracy
    BACC = (TPR + TNR) / 2  # Balanced Accuracy
    TSS = TPR - FPR  # True Skill Statistic
    HSS = (2 * (tp * tn - fp * fn)) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp +␣
↪tn))  # Heidke Skill Score

    #  Brier Score
    y_proba = nb_model.predict_proba(X_test)  # Probabilities
    brier_score = np.mean([(y_proba[:, i] - (y_test == i).astype(int)) ** 2 for␣
↪i in range(y_proba.shape[1])])

    # AUC
    try:
        auc = roc_auc_score(y_test, y_proba, multi_class='ovr')
    except ValueError:
        auc = np.nan  #  NaN

    # averaged values
    fold_value.append([
        tp.mean(), tn.mean(), fp.mean(), fn.mean(),p.mean(),n.mean(),
        TPR.mean(), TNR.mean(), FPR.mean(), FNR.mean(),
```

```
        Precision.mean(), F1_measure.mean(),
        Accuracy, Error_rate, BACC.mean(), TSS.mean(), HSS.mean(),
        brier_score, auc, Accuracy  # Acc_by_package_fn
    ])
```

0.2.1 Printing Output

[24]:
```
# values to DataFrame
value_df = pd.DataFrame(fold_value, columns=[
    "TP", "TN", "FP", "FN","P","N", "TPR", "TNR", "FPR", "FNR", "Precision",␣
 ↪"F1_measure",
    "Accuracy", "Error_rate", "BACC", "TSS", "HSS", "Brier_score", "AUC",␣
 ↪"Acc_by_package_fn"
])
 #transpose
value_df_nb = value_df.T
value_df_nb.columns = [f"Fold {i+1}" for i in range(value_df_nb.shape[1])]

# display
value_df_nb
```

[24]:

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5 \
TP	25.250000	27.750000	29.750000	26.500000	28.500000
TN	111.750000	114.250000	116.250000	113.000000	115.000000
FP	18.000000	15.500000	13.500000	16.750000	14.750000
FN	18.000000	15.500000	13.500000	16.750000	14.750000
P	43.250000	43.250000	43.250000	43.250000	43.250000
N	129.750000	129.750000	129.750000	129.750000	129.750000
TPR	0.470373	0.470138	0.503194	0.483714	0.467397
TNR	0.823838	0.846150	0.868231	0.862582	0.847381
FPR	0.176162	0.153850	0.131769	0.137418	0.152619
FNR	0.529627	0.529862	0.496806	0.516286	0.532603
Precision	0.379752	0.350000	0.387741	0.394413	0.361048
F1_measure	0.319360	0.295971	0.346246	0.291893	0.262175
Accuracy	0.583815	0.641618	0.687861	0.612717	0.658960
Error_rate	0.416185	0.358382	0.312139	0.387283	0.341040
BACC	0.647106	0.658144	0.685713	0.673148	0.657389
TSS	0.294212	0.316288	0.371425	0.346296	0.314777
HSS	0.178148	0.178146	0.240213	0.192697	0.136996
Brier_score	0.155348	0.157572	0.139540	0.173229	0.147899
AUC	0.816756	0.811708	0.787456	0.755814	0.761661
Acc_by_package_fn	0.583815	0.641618	0.687861	0.612717	0.658960

	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
TP	24.750000	29.000000	26.000000	26.250000	27.000000
TN	111.250000	115.500000	112.500000	112.250000	113.000000
FP	18.500000	14.250000	17.250000	16.750000	16.000000

```
FN                 18.500000   14.250000   17.250000   16.750000   16.000000
P                  43.250000   43.250000   43.250000   43.000000   43.000000
N                 129.750000  129.750000  129.750000  129.000000  129.000000
TPR                 0.442149    0.485778    0.462508    0.483749    0.493374
TNR                 0.812366    0.853926    0.851611    0.838830    0.855411
FPR                 0.187634    0.146074    0.148389    0.161170    0.144589
FNR                 0.557851    0.514222    0.537492    0.516251    0.506626
Precision           0.227704    0.425887    0.356430    0.344046    0.369376
F1_measure          0.245383    0.332591    0.290421    0.310952    0.334899
Accuracy            0.572254    0.670520    0.601156    0.610465    0.627907
Error_rate          0.427746    0.329480    0.398844    0.389535    0.372093
BACC                0.627257    0.669852    0.657059    0.661290    0.674392
TSS                 0.254515    0.339704    0.314119    0.322579    0.348785
HSS                 0.109563    0.213132    0.175001    0.182451    0.227613
Brier_score         0.171573    0.141456    0.173214    0.154961    0.163264
AUC                 0.776182    0.818576    0.771867    0.826708    0.819196
Acc_by_package_fn   0.572254    0.670520    0.601156    0.610465    0.627907
```

0.3 3. Bidirectional-LSTM

Here I used Bidirectional-LSTM classifier to calculate values like Confusion matrix, Sensitivity, Specificity, False Positive Rate, False Negative Rate, precision, F1 score, Balanced Accuracy, True Skill Statistic, Heidke Skill Score and AUC. The results for each fold are stored for overall evaluation.

[27]:
```
# Standardize features
scaler = StandardScaler()
X = scaler.fit_transform(X)

# target variable to categorical
y = to_categorical(y)
```

[28]:
```
# Function for Bidirectional-LSTM model
def create_bidirectional_lstm(input_shape, num_classes):
    model = Sequential()
    model.add(Input(shape=input_shape))
    model.add(Bidirectional(LSTM(64)))
    model.add(Dense(32, activation='relu'))
    model.add(Dense(num_classes, activation='softmax'))
    model.compile(optimizer=Adam(learning_rate=0.001),␣
 ↪loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

[29]:
```
# Initialize an empty list
warnings.filterwarnings("ignore")
fold_value = []

# Reshape input data to be compatible with LSTM
```

```python
X = X.reshape(X.shape[0], X.shape[1], 1)
input_shape = (X.shape[1], 1)
num_classes = y.shape[1]

# Loop through each fold
for i, (train_index, test_index) in enumerate(kfold.split(X), start=1):
    # Splitting the data
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    # Create and train the Bidirectional-LSTM model
    model = create_bidirectional_lstm(input_shape, num_classes)
    model.fit(X_train, y_train, epochs=10, batch_size=32, verbose=0)


    y_pred_proba = model.predict(X_test)
    y_pred = np.argmax(y_pred_proba, axis=1)
    y_test_class = np.argmax(y_test, axis=1)

    # Confusion matrix
    cm = confusion_matrix(y_test_class, y_pred)
    tp = cm.diagonal()  # True Positives
    fn = cm.sum(axis=1) - tp  # False Negatives
    fp = cm.sum(axis=0) - tp  # False Positives
    tn = cm.sum() - (fp + fn + tp)  # True Negatives

    p = tp + fn
    n = tn + fp


    TPR = tp / (tp + fn)  # Sensitivity
    TNR = tn / (tn + fp)  # Specificity
    FPR = fp / (fp + tn)  # False Positive Rate
    FNR = fn / (fn + tp)  # False Negative Rate s

    Precision = np.divide(tp, (tp + fp), out=np.zeros_like(tp, dtype=float), where=(tp + fp) != 0)
    F1_measure = np.divide(2 * (Precision * TPR), (Precision + TPR), out=np.zeros_like(TPR, dtype=float), where=(Precision + TPR) != 0)

    Accuracy = accuracy_score(y_test_class, y_pred)
    Error_rate = 1 - Accuracy
    BACC = (TPR + TNR) / 2  # Balanced Accuracy
    TSS = TPR - FPR  # True Skill Statistic
    HSS = (2 * (tp * tn - fp * fn)) / ((tp + fn) * (fn + tn) + (tp + fp) * (fp + tn))  # Heidke Skill Score
```

```python
    #  Brier Score
    brier_score = np.mean([(y_pred_proba[:, i] - (y_test_class == i).
↪astype(int)) ** 2 for i in range(y_pred_proba.shape[1])])

    # AUC
    try:
        auc = roc_auc_score(y_test_class, y_pred_proba, multi_class='ovr')
    except ValueError:
        auc = np.nan  # NaN

    # averaged
    fold_value.append([
        tp.mean(), tn.mean(), fp.mean(), fn.mean(),p.mean(),n.mean(),
        TPR.mean(), TNR.mean(), FPR.mean(), FNR.mean(),
        Precision.mean(), F1_measure.mean(),
        Accuracy, Error_rate, BACC.mean(), TSS.mean(), HSS.mean(),
        brier_score, auc, Accuracy  # Acc_by_package_fn
    ])
```

```
6/6  0s 25ms/step
6/6  0s 23ms/step
WARNING:tensorflow:5 out of the last 13 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x31f9bbec0> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
6/6  0s 23ms/step
WARNING:tensorflow:5 out of the last 13 calls to <function
TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at
0x1218d9ee0> triggered tf.function retracing. Tracing is expensive and the
excessive number of tracings could be due to (1) creating @tf.function
repeatedly in a loop, (2) passing tensors with different shapes, (3) passing
Python objects instead of tensors. For (1), please define your @tf.function
outside of the loop. For (2), @tf.function has reduce_retracing=True option that
can avoid unnecessary retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
6/6  0s 24ms/step
6/6  0s 24ms/step
6/6  0s 24ms/step
6/6  0s 23ms/step
6/6  0s 23ms/step
6/6  0s 24ms/step
```

```
6/6  0s 24ms/step
```

0.3.1 Printing Output

[31]:
```python
# value to DataFrame
value_df = pd.DataFrame(fold_value, columns=[
    "TP", "TN", "FP", "FN","P","N", "TPR", "TNR", "FPR", "FNR", "Precision",␣
 ↪"F1_measure",
    "Accuracy", "Error_rate", "BACC", "TSS", "HSS", "Brier_score", "AUC",␣
 ↪"Acc_by_package_fn"
])

# Transpose
value_df_bilstm = value_df.T
value_df_bilstm.columns = [f"Fold {i+1}" for i in range(value_df_bilstm.
 ↪shape[1])]

# Display
value_df_bilstm
```

[31]:

	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5	\
TP	34.500000	38.000000	37.000000	37.750000	37.000000	
TN	121.000000	124.500000	123.500000	124.250000	123.500000	
FP	8.750000	5.250000	6.250000	5.500000	6.250000	
FN	8.750000	5.250000	6.250000	5.500000	6.250000	
P	43.250000	43.250000	43.250000	43.250000	43.250000	
N	129.750000	129.750000	129.750000	129.750000	129.750000	
TPR	0.481899	0.646744	0.626198	0.716652	0.532026	
TNR	0.889367	0.939240	0.927237	0.942802	0.868245	
FPR	0.110633	0.060760	0.072763	0.057198	0.131755	
FNR	0.518101	0.353256	0.373802	0.283348	0.467974	
Precision	0.569433	0.626078	0.789951	0.654372	0.754749	
F1_measure	0.506948	0.635727	0.620356	0.623122	0.580088	
Accuracy	0.797688	0.878613	0.855491	0.872832	0.855491	
Error_rate	0.202312	0.121387	0.144509	0.127168	0.144509	
BACC	0.685633	0.792992	0.776717	0.829727	0.700135	
TSS	0.371267	0.585984	0.553435	0.659454	0.400270	
HSS	0.418376	0.577227	0.555173	0.570266	0.472401	
Brier_score	0.063296	0.049729	0.050886	0.043895	0.045126	
AUC	0.957428	0.968859	0.964810	0.978026	0.967641	
Acc_by_package_fn	0.797688	0.878613	0.855491	0.872832	0.855491	

	Fold 6	Fold 7	Fold 8	Fold 9	Fold 10
TP	36.750000	37.750000	33.250000	34.000000	35.000000
TN	123.250000	124.250000	119.750000	120.000000	121.000000
FP	6.500000	5.500000	10.000000	9.000000	8.000000
FN	6.500000	5.500000	10.000000	9.000000	8.000000

```
P                   43.250000   43.250000   43.250000   43.000000   43.000000
N                  129.750000  129.750000  129.750000  129.000000  129.000000
TPR                  0.579455    0.630778    0.594991    0.559773    0.549643
TNR                  0.913426    0.940172    0.885191    0.899078    0.909446
FPR                  0.086574    0.059828    0.114809    0.100922    0.090554
FNR                  0.420545    0.369222    0.405009    0.440227    0.450357
Precision            0.641667    0.597222    0.465267    0.504663    0.590131
F1_measure           0.601899    0.611511    0.492458    0.530187    0.562408
Accuracy             0.849711    0.872832    0.768786    0.790698    0.813953
Error_rate           0.150289    0.127168    0.231214    0.209302    0.186047
BACC                 0.746440    0.785475    0.740091    0.729425    0.729545
TSS                  0.492881    0.570950    0.480182    0.458850    0.459089
HSS                  0.532715    0.548825    0.381295    0.441410    0.487524
Brier_score          0.053192    0.044602    0.071329    0.068684    0.058379
AUC                  0.962109    0.969837    0.930956    0.941302    0.963679
Acc_by_package_fn    0.849711    0.872832    0.768786    0.790698    0.813953
```

0.3.2 Average Output

In this section I calculae the average of each calculation criteria and show them in a table for easy comparison.

[33]:
```
values = [
    "TP", "TN", "FP", "FN","P","N", "TPR", "TNR", "FPR", "FNR",
    "Precision", "F1_measure", "Accuracy", "Error_rate",
    "BACC", "TSS", "HSS", "Brier_score", "AUC", "Acc_by_package_fn"
]

# names
value_df_rf.index = values
value_df_nb.index = values
value_df_bilstm.index = values

# Calculate the mean
avg_value_rf = value_df_rf.mean(axis=1)  # Average Random Forest
avg_value_nb = value_df_nb.mean(axis=1)  # Average  Naive Bayes
avg_value_bilstm = value_df_bilstm.mean(axis=1)  # Average Bidirectional LSTM

# averages to DataFrame
avg_values_combined = pd.DataFrame({
    "Random Forest": avg_value_rf,
    "Naive Bayes": avg_value_nb,
    "Bidirectional-LSTM": avg_value_bilstm
})

#index name
avg_values_combined.index.name = "Values"
```

```
# Display
avg_values_combined
```

[33]:

Values	Random Forest	Naive Bayes	Bidirectional-LSTM
TP	42.425000	27.075000	36.100000
TN	128.825000	113.475000	122.500000
FP	0.775000	16.125000	7.100000
FN	0.775000	16.125000	7.100000
P	43.200000	43.200000	43.200000
N	129.600000	129.600000	129.600000
TPR	0.946777	0.476237	0.591816
TNR	0.992933	0.846033	0.911420
FPR	0.007067	0.153967	0.088580
FNR	0.053223	0.523763	0.408184
Precision	0.968227	0.359640	0.619353
F1_measure	0.953192	0.302989	0.576470
Accuracy	0.982061	0.626727	0.835610
Error_rate	0.017939	0.373273	0.164390
BACC	0.969855	0.661135	0.751618
TSS	0.939711	0.322270	0.503236
HSS	0.946078	0.183396	0.498521
Brier_score	0.016538	0.157806	0.054912
AUC	0.998778	0.794592	0.960465
Acc_by_package_fn	0.982061	0.626727	0.835610

0.3.3 Conclusion:

The Random Forest model is the best performer among the three, with the highest accuracy (98.15%), precision (96.77%), True positive rate (TPR) (94.65%), and F1-measure (95.28%), as well as the lowest error rate (1.82%). It consistently delivers the most reliable results across all metrics. Bidirectional-LSTM performs moderately well, with an accuracy of 83.56%, but falls short compared to Random Forest. Naive Bayes, however, performs poorly, with a low accuracy of 62.67% and high error rate (37.33%), making it the least suitable option. Therefore, Random Forest is the best choice for this task, while Bidirectional-LSTM may be considered for sequential data, and Naive Bayes should be avoided.

[ ]:

```

```