

MachineLearningAlgorithms

December 22, 2023

1 Machine Learning Algorithms

A simple script that analyses a data frame using fundamental statistical learning algorithms and a feed forward neural network. Written by: Da'Vel Reed Johnson

```
[1]: #Load Required Modules
import os
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

#Ignore Unnecessary Warnings
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

#Set path
main_path = os.path.dirname(os.path.abspath("__file__"))
```

```
[2]: #Opening the data file
filename = main_path + '/data/Employee.csv'

df = pd.read_csv(filename)
```

1.1 Gathering information about the dataframe

```
[3]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4653 entries, 0 to 4652
Data columns (total 9 columns):
 #   Column              Non-Null Count  Dtype
---  -
 0   Education            4653 non-null   object
 1   JoiningYear          4653 non-null   int64
 2   City                 4653 non-null   object
```

```

3   PaymentTier          4653 non-null   int64
4   Age                  4653 non-null   int64
5   Gender                4653 non-null   object
6   EverBenched           4653 non-null   object
7   ExperienceInCurrentDomain 4653 non-null   int64
8   LeaveOrNot            4653 non-null   int64
dtypes: int64(5), object(4)
memory usage: 327.3+ KB

```

```
[4]: df.describe()
```

```

[4]:      JoiningYear  PaymentTier      Age  ExperienceInCurrentDomain \
count  4653.000000  4653.000000  4653.000000  4653.000000
mean   2015.062970    2.698259   29.393295    2.905652
std     1.863377    0.561435    4.826087    1.558240
min    2012.000000    1.000000   22.000000    0.000000
25%    2013.000000    3.000000   26.000000    2.000000
50%    2015.000000    3.000000   28.000000    3.000000
75%    2017.000000    3.000000   32.000000    4.000000
max    2018.000000    3.000000   41.000000    7.000000

      LeaveOrNot
count  4653.000000
mean     0.343864
std     0.475047
min     0.000000
25%     0.000000
50%     0.000000
75%     1.000000
max     1.000000

```

```
[5]: df.head()
```

```

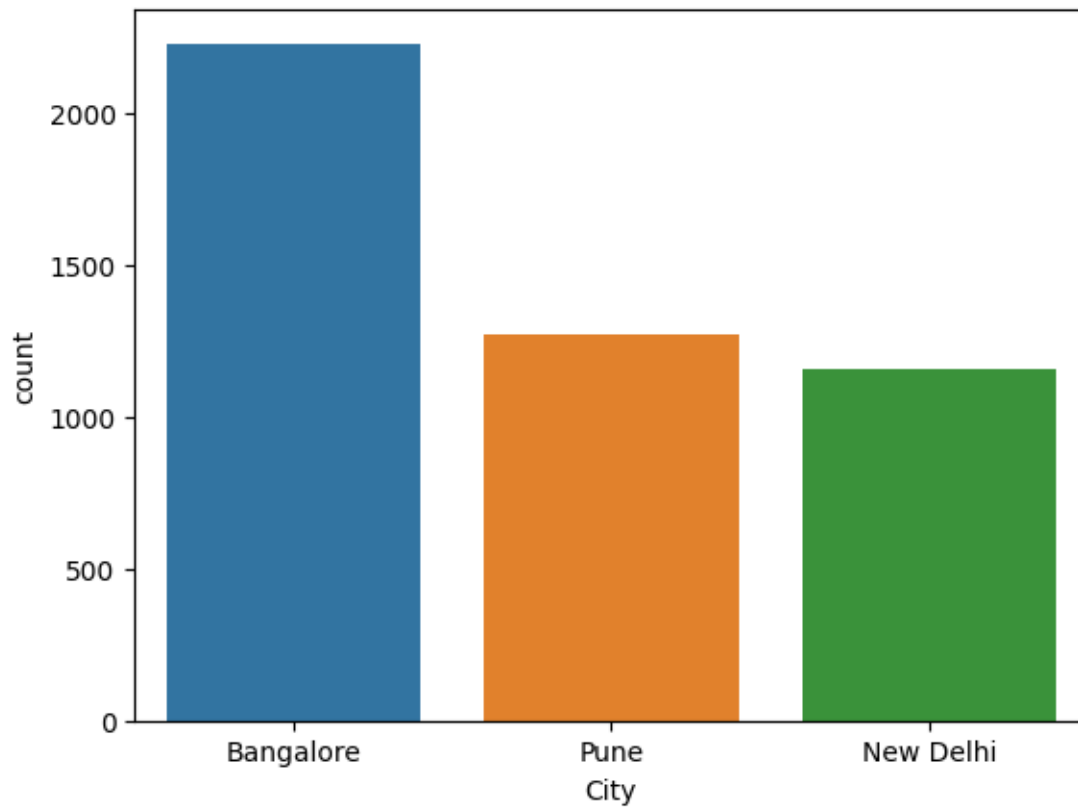
[5]:      Education  JoiningYear      City  PaymentTier  Age  Gender  EverBenched \
0  Bachelors      2017  Bangalore          3   34   Male          No
1  Bachelors      2013    Pune          1   28  Female          No
2  Bachelors      2014  New Delhi          3   38  Female          No
3   Masters      2016  Bangalore          3   27   Male          No
4   Masters      2017    Pune          3   24   Male          Yes

      ExperienceInCurrentDomain  LeaveOrNot
0                             0           0
1                             3           1
2                             2           0
3                             5           1
4                             2           1

```

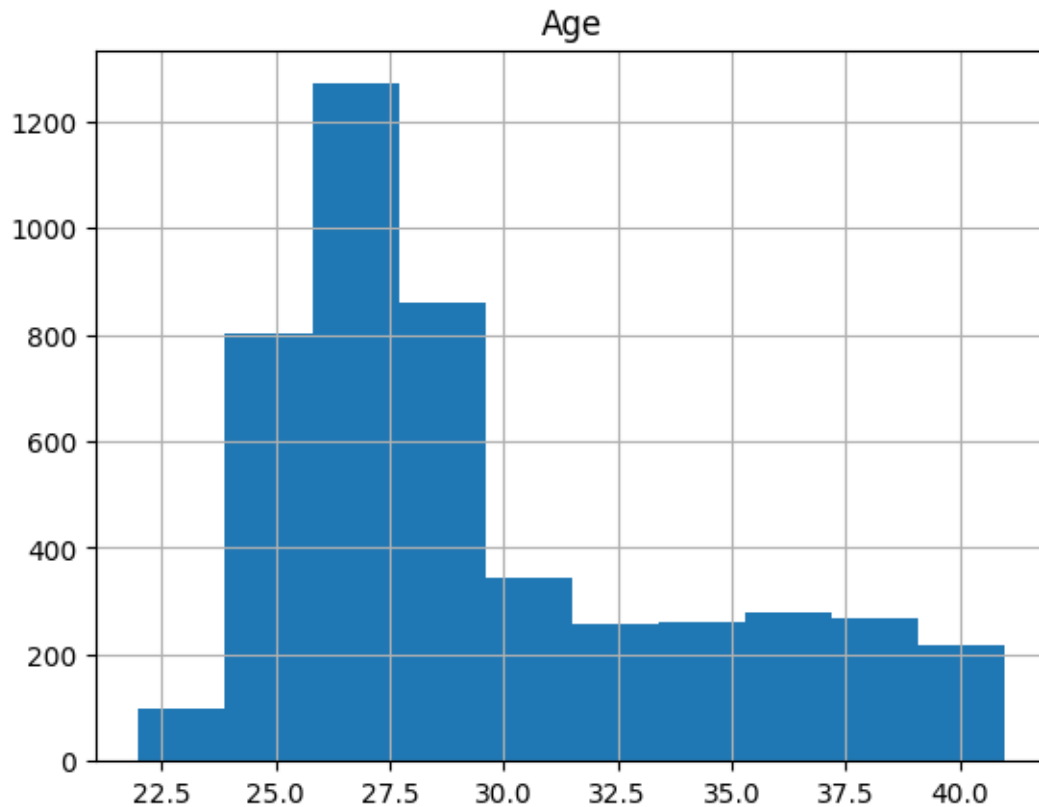
1.2 Base plots and graphics

```
[6]: p = sns.countplot(data=df, x = 'City')
```



```
[7]: df.hist(column='Age')
```

```
[7]: array([[<AxesSubplot:title={'center':'Age'}>]], dtype=object)
```



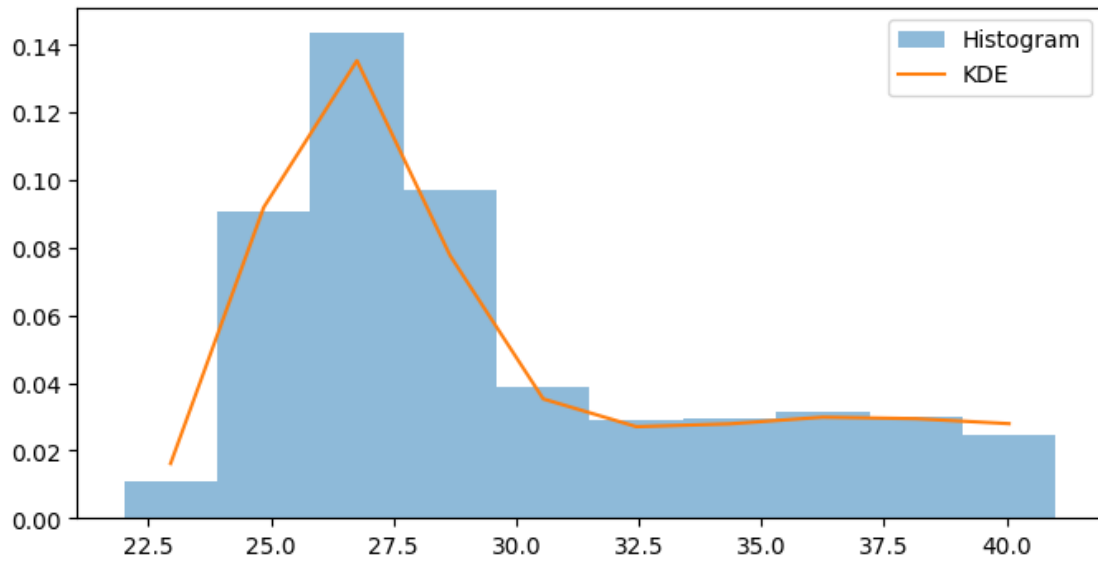
```
[8]: from scipy.stats import gaussian_kde

# Create histogram data
hist_counts, bin_edges = np.histogram(df['Age'], bins=10, density=True)
bin_centers = 0.5 * (bin_edges[1:] + bin_edges[:-1])

# Adjust the bandwidth
bandwidth_factor = 0.1 # Adjust this factor as needed (less than 1 for smaller
↳ bandwidth)

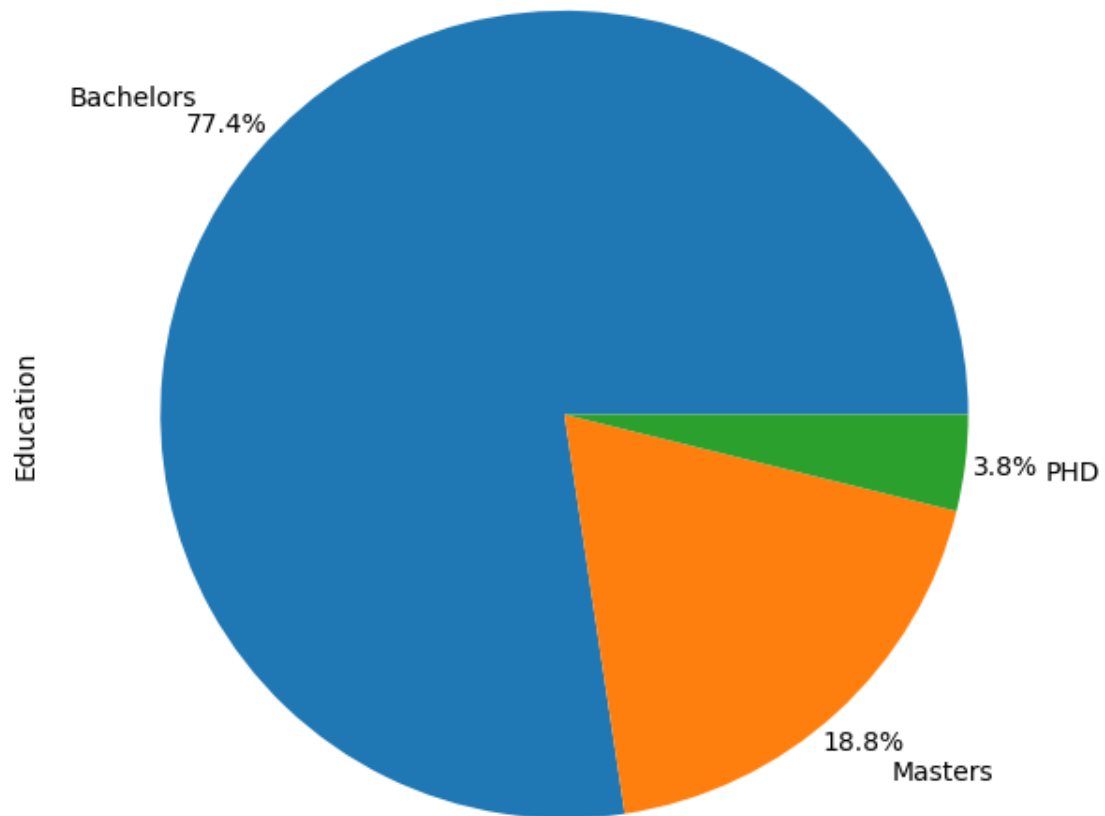
# Estimate KDE
kde = gaussian_kde(df['Age'], bw_method=bandwidth_factor)
kde_values = kde(bin_centers)

# Plot histogram and KDE
plt.figure(figsize=(8, 4))
plt.hist(df['Age'], bins=10, density=True, alpha=0.5, label='Histogram')
plt.plot(bin_centers, kde_values, label='KDE')
plt.legend()
plt.show()
```



```
[9]: df['Education'].value_counts().plot(kind='pie',figsize=(8, 7), autopct='%1.1f%%',pctdistance=1.1,labeldistance=1.2)
```

```
[9]: <AxesSubplot:ylabel='Education'>
```



```
[10]: #Check to see if there are any empty entries  
df.isnull().sum()
```

```
[10]: Education          0  
      JoiningYear      0  
      City            0  
      PaymentTier     0  
      Age             0  
      Gender          0  
      EverBenched     0  
      ExperienceInCurrentDomain  0  
      LeaveOrNot      0  
      dtype: int64
```

2 Preparing Data for model training

```
[11]: #Assigning features and classes to training variables
X =
    ↪df[['Education','JoiningYear','PaymentTier','Gender','EverBenched','Age','ExperienceInCurrentDomain']]
y = df['LeaveOrNot']
```

```
[12]: #Creating dummy variables from categorical variables for numerical analysis
df_dummies = pd.get_dummies(X)
```

```
[13]: df_dummies.head()
```

```
[13]:
```

	JoiningYear	PaymentTier	Age	ExperienceInCurrentDomain	\
0	2017	3	34		0
1	2013	1	28		3
2	2014	3	38		2
3	2016	3	27		5
4	2017	3	24		2

	Education_Bachelors	Education_Masters	Education_PHD	Gender_Female	\
0	1	0	0	0	
1	1	0	0	1	
2	1	0	0	1	
3	0	1	0	0	
4	0	1	0	0	

	Gender_Male	EverBenched_No	EverBenched_Yes
0	1	1	0
1	0	1	0
2	0	1	0
3	1	1	0
4	1	0	1

```
[14]: # Removing Gender and EverBenched for duplication
print(f"Number of columns before deleting: {df_dummies.shape[1]}")

del_cols = ['Gender_Male','EverBenched_No']
df_dummies.drop(labels = del_cols,axis = 1,inplace = True)
print(f"Number of columns after deleting: {df_dummies.shape[1]}")
```

Number of columns before deleting: 11

Number of columns after deleting: 9

```
[15]: #Normalizing data for processing
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
df_normalized = pd.DataFrame(scaler.fit_transform(df_dummies),
    ↪columns=df_dummies.columns)
```

```
[16]: df_normalized.head()
```

```
[16]:   JoiningYear  PaymentTier      Age  ExperienceInCurrentDomain  \
0      1.039638      0.537503  0.954645                -1.864901
1     -1.107233     -3.025177 -0.288732                 0.060554
2     -0.570515      0.537503  1.783563                -0.581264
3      0.502921      0.537503 -0.495961                 1.344191
4      1.039638      0.537503 -1.117650                -0.581264

      Education_Bachelors  Education_Masters  Education_PHD  Gender_Female  \
0              0.540501          -0.480575        -0.200022        -0.821551
1              0.540501          -0.480575        -0.200022         1.217210
2              0.540501          -0.480575        -0.200022         1.217210
3             -1.850136           2.080840        -0.200022        -0.821551
4             -1.850136           2.080840        -0.200022        -0.821551

      EverBenched_Yes
0          -0.338365
1          -0.338365
2          -0.338365
3          -0.338365
4           2.955387
```

```
[17]: #Converting the dependent variable to binary
dy = pd.get_dummies(y)
```

```
[18]: dy.head()
```

```
[18]:   0  1
0  1  0
1  0  1
2  1  0
3  0  1
4  0  1
```

```
[19]: dy.columns
```

```
[19]: Int64Index([0, 1], dtype='int64')
```

```
[20]: dummyy = dy[1]
```

```
[21]: dummyy.head()
```

```
[21]: 0    0
1    1
2    0
3    1
4    1
```


Name: 1, dtype: uint8

2.1 Training and Running Fundamental Models

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

```
[23]: X_train, X_test, y_train, y_test = train_test_split(df_normalized, dummyy,
↳ test_size=0.3, stratify=y)
```

2.1.1 Gaussian Naive Bayes

```
[24]: from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import classification_report
```

```
[25]: # Fit train set for Gaussian Naive Bayes
GNB = GaussianNB()
GNB.fit(X_train, y_train)
```

```
[25]: GaussianNB()
```

```
[26]: # Predict for test set
GNBy_pred = GNB.predict(X_test)
print(classification_report(y_test, GNBy_pred))
```

	precision	recall	f1-score	support
0	0.72	0.76	0.74	916
1	0.49	0.44	0.47	480
accuracy			0.65	1396
macro avg	0.61	0.60	0.60	1396
weighted avg	0.64	0.65	0.65	1396

2.1.2 Random Forest Classifier

```
[27]: from sklearn.ensemble import RandomForestClassifier
```

```
[28]: rfc = RandomForestClassifier(n_estimators=600, max_features='sqrt',
↳ oob_score=True, random_state=None, n_jobs=-1)
rfc.fit(X_train, y_train)
```

```
[28]: RandomForestClassifier(max_features='sqrt', n_estimators=600, n_jobs=-1,
oob_score=True)
```

```
[29]: predictions = rfc.predict(X_test)
```

```
[30]: from sklearn.metrics import classification_report, confusion_matrix
print(classification_report(y_test, predictions))
```

	precision	recall	f1-score	support
0	0.81	0.89	0.85	916
1	0.75	0.61	0.67	480
accuracy			0.80	1396
macro avg	0.78	0.75	0.76	1396
weighted avg	0.79	0.80	0.79	1396

```
[31]: print(confusion_matrix(y_test, predictions))
```

```
[[818  98]
 [187 293]]
```

Plotting feature importance

```
[32]: rfc.feature_importances_
```

```
[32]: array([0.36679592, 0.15083207, 0.20706674, 0.09950798, 0.02861508,
          0.0380703 , 0.00916082, 0.07798808, 0.02196301])
```

```
[33]: def plot_feature_importance(importance, names, model_type):

    #Create arrays from feature importance and feature names
    feature_importance = np.array(importance)
    feature_names = np.array(names)

    #Create a DataFrame using a Dictionary
    data={'feature_names':feature_names, 'feature_importance':feature_importance}
    fi_df = pd.DataFrame(data)

    #Sort the DataFrame in order decreasing feature importance
    fi_df.sort_values(by=['feature_importance'], ascending=False, inplace=True)

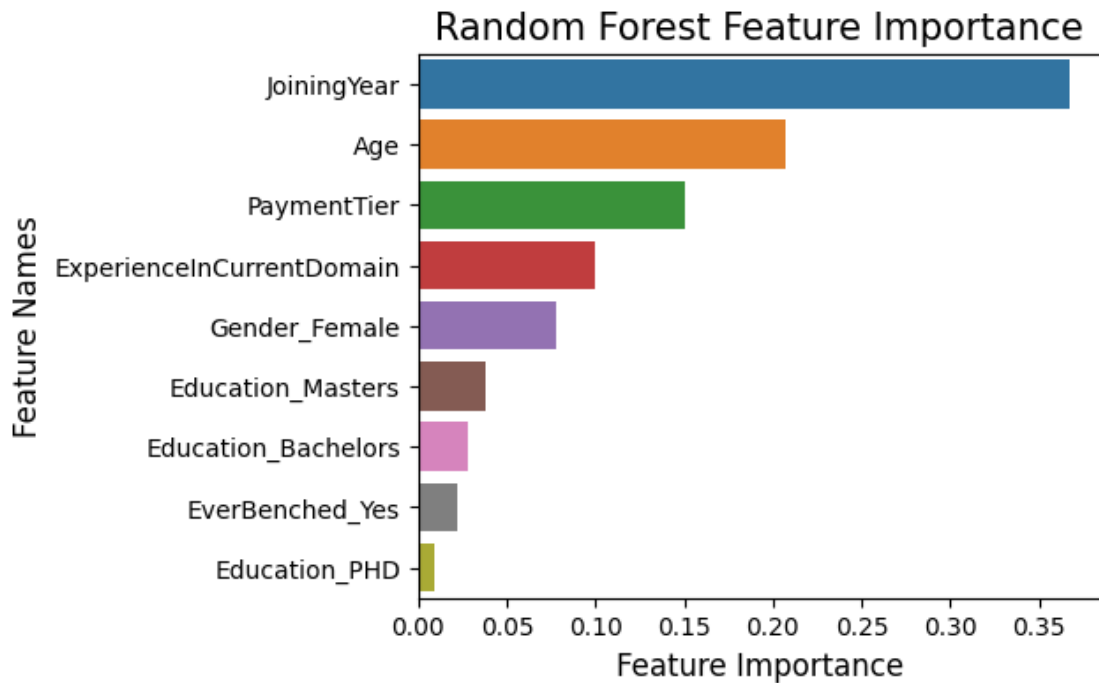
    #Define size of bar plot
    plt.figure(figsize=(100,80))

    #Plot Seaborn bar chart
    sns.barplot(x=fi_df['feature_importance'], y=fi_df['feature_names'])

    #Add chart labels
    plt.title(model_type + ' FEATURE IMPORTANCE')
    plt.xlabel('FEATURE IMPORTANCE')
    plt.ylabel('FEATURE NAMES')
```

```
[34]: plot_feature_importance(rfc.feature_importances_, X_train.columns, 'RANDOM_
    ↪FOREST')
```

```
plt.title('Random Forest Feature Importance', fontsize=15)
plt.xlabel('Feature Importance', fontsize=12)
plt.ylabel('Feature Names', fontsize=12)
fig = plt.gcf()
fig.set_size_inches(5, 4)
plt.show()
```



2.1.3 Gradient Boosting Classifier

```
[35]: from sklearn.metrics import mean_squared_error as MSE
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import GradientBoostingRegressor

gb = GradientBoostingClassifier(n_estimators=300,
                               max_depth=2,
                               learning_rate=0.8)

gb.fit(X_train, y_train)
gby_pred = gb.predict(X_test)
```

Finding optimal hyperparameters

```
[36]: hyperparameter_space = {'n_estimators':[300, 350, 400, 450, 500],
                              'learning_rate':[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
```

```

        'max_depth':[1, 2]}

from sklearn.model_selection import RandomizedSearchCV

rs = RandomizedSearchCV(GradientBoostingClassifier(),
                        param_distributions=hyperparameter_space,
                        n_iter=10, scoring="neg_root_mean_squared_error",
                        random_state=None, n_jobs=-1, cv=5)

rs.fit(X_train, y_train)
print("Optimal hyperparameter combination:", rs.best_params_)

```

Optimal hyperparameter combination: {'n_estimators': 350, 'max_depth': 2, 'learning_rate': 0.3}

```
[37]: print(classification_report(y_test,gby_pred))
```

	precision	recall	f1-score	support
0	0.81	0.91	0.86	916
1	0.78	0.58	0.67	480
accuracy			0.80	1396
macro avg	0.79	0.75	0.76	1396
weighted avg	0.80	0.80	0.79	1396

```
[38]: print(confusion_matrix(y_test,gby_pred))
```

```
[[837  79]
 [201 279]]
```

2.1.4 Extreme Gradient Boosting Classifier

```
[39]: from xgboost import XGBClassifier
      from sklearn.metrics import accuracy_score
```

```
[40]: xgb = XGBClassifier()
      #XGB
      xgb.fit(X_train,y_train)
      y_pred_xgb = xgb.predict(X_test)
```

```
[41]: print(classification_report(y_test,y_pred_xgb))
      print(confusion_matrix(y_test,y_pred_xgb))
```

	precision	recall	f1-score	support
0	0.82	0.91	0.86	916

	1	0.78	0.61	0.69	480
accuracy				0.81	1396
macro avg	0.80	0.76	0.77		1396
weighted avg	0.81	0.81	0.80		1396

```
[[836  80]
 [188 292]]
```

```
[42]: param_dist = {
        'n_estimators': [50, 100, 200],
        'max_depth': [3, 4, 5, 6, 7, 8],
        'gamma': [0, 0.1, 0.2, 0.3, 0.4],
        'reg_lambda': [1, 1.5, 2, 2.5, 3]
    }

    xgb = XGBClassifier()
    rs = RandomizedSearchCV(xgb, param_dist, n_iter=25, scoring='accuracy', cv=3,
        verbose=1, random_state=42)
    rs.fit(X_train, y_train)

    rs.fit(X_train, y_train)
    print("Optimal hyperparameter combination:", rs.best_params_)
```

Fitting 3 folds for each of 25 candidates, totalling 75 fits

Fitting 3 folds for each of 25 candidates, totalling 75 fits

Optimal hyperparameter combination: {'reg_lambda': 2, 'n_estimators': 50, 'max_depth': 6, 'gamma': 0.4}

```
[78]: xgb = XGBClassifier(
        n_estimators=50,
        reg_lambda=2,
        gamma=0.4,
        max_depth=6
    )
```

```
[79]: xgb.fit(X_train, y_train)
    y_pred_xgb = xgb.predict(X_test)
```

```
[80]: print(classification_report(y_test, y_pred_xgb))
    print(confusion_matrix(y_test, y_pred_xgb))
    # Calculate the accuracy
    accuracy = accuracy_score(y_test, y_pred_xgb)

    # Print the accuracy
    print("Accuracy:", accuracy)
```

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	0.84	0.94	0.88	775
1	0.84	0.63	0.72	389
accuracy			0.84	1164
macro avg	0.84	0.79	0.80	1164
weighted avg	0.84	0.84	0.83	1164

```
[[727 48]
 [142 247]]
Accuracy: 0.8367697594501718
```

2.1.5 Feed Forward Neural Network

```
[46]: import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
```

```
[47]: class BasicNeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(BasicNeuralNetwork, self).__init__()
        # Define the first hidden layer
        self.hidden1 = nn.Linear(input_size, hidden_size1)

        # Define the second hidden layer
        self.hidden2 = nn.Linear(hidden_size1, hidden_size2)

        # Define the output layer
        self.output = nn.Linear(hidden_size2, output_size)

    def forward(self, x):
        # Apply a non-linear activation function / ReLU after each hidden layer
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))

        # The output layer
        x = self.output(x)
        return x #Use F.softmax(x, dim=1) to apply a softmax for multi-class
        ↪problems
```

Designing a Neural Network through rules of thumb

```
[48]: num_features = df_normalized.shape[1]
num_output = 1
hidden_size = int((num_features * num_output)**0.5)
print(f'Number of features: {num_features}')
```

```
print(f'Output Size: {num_output}')
print(f'Number of neurons: {hidden_size}')
```

Number of features: 9

Output Size: 1

Number of neurons: 3

```
[49]: # Example usage
input_size = num_features # Size of input (number of input features)
hidden_size1 = hidden_size #Size of first hidden layer
hidden_size2 = hidden_size #Size of second hidden layer
output_size = num_output # Size of output (number of classes for
    ↪classification)

model = BasicNeuralNetwork(input_size, hidden_size1, hidden_size2, output_size)
```

```
[50]: X_train, X_test, y_train, y_test = train_test_split(df_normalized, dummyy,
    ↪test_size=0.3, stratify=y)
```

```
[51]: # Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train.values, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.float32)
X_test_tensor = torch.tensor(X_test.values, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.float32)

# Create datasets
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

# Data Loaders
batch_size = 64 # Define your batch size
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
    ↪shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
    ↪shuffle=False)

# Initialize the model, loss function, and optimizer
model = BasicNeuralNetwork(input_size, hidden_size1, hidden_size2, output_size)
criterion = nn.BCEWithLogitsLoss()
#For binary classification, use nn.BCEWithLogitsLoss and for multi-class, use
    ↪nn.CrossEntropyLoss.

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Lists to store metrics
epoch_losses = []
epoch_accuracies = []
```

```

val_epoch_losses = []
val_epoch_accuracies = []

# Training Loop
num_iterations = 100 # Define the number of epochs
for epoch in range(num_iterations):
    model.train()
    total_loss = 0
    for inputs, targets in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()

        # Forward pass
        outputs = model(inputs)
        outputs = outputs.squeeze()

        # Compute the loss
        loss = criterion(outputs, targets)

        # Backward pass and optimize
        loss.backward()
        optimizer.step()
        total_loss += loss.item()

    avg_loss = total_loss / len(train_loader)
    epoch_losses.append(avg_loss)

# Testing Loop Adjustment for Binary Classification
model.eval()
val_total_loss = 0
correct = 0
total = 0
with torch.no_grad():
    correct = 0
    total = 0
    for inputs, targets in test_loader:
        outputs = model(inputs)
        outputs = outputs.squeeze()
        val_loss = criterion(outputs, targets)
        val_total_loss += val_loss.item()

        # Threshold the outputs to get binary predictions
        predicted = outputs > 0 # or use a different threshold like 0.5
        total += targets.size(0)
        correct += (predicted == targets).sum().item()
accuracy = 100 * correct / total
epoch_accuracies.append(accuracy)

```



```

avg_val_loss = val_total_loss / len(test_loader)
val_epoch_losses.append(avg_val_loss)
val_accuracy = 100 * correct / total
val_epoch_accuracies.append(val_accuracy)

print(f'Epoch [{epoch+1}/{num_iterations}], Loss: {avg_loss:.4f}, Val Loss: {avg_val_loss:.4f}, Val Accuracy: {val_accuracy:.2f}%')

print(f'Accuracy of the model on the test set: {accuracy:.2f}%')

```

```

Epoch [1/100], Loss: 0.6486, Val Loss: 0.6482, Val Accuracy: 65.62%
Epoch [2/100], Loss: 0.6438, Val Loss: 0.6451, Val Accuracy: 65.62%
Epoch [3/100], Loss: 0.6411, Val Loss: 0.6425, Val Accuracy: 65.62%
Epoch [4/100], Loss: 0.6380, Val Loss: 0.6387, Val Accuracy: 65.62%
Epoch [5/100], Loss: 0.6322, Val Loss: 0.6321, Val Accuracy: 65.62%
Epoch [6/100], Loss: 0.6241, Val Loss: 0.6239, Val Accuracy: 65.62%
Epoch [7/100], Loss: 0.6140, Val Loss: 0.6131, Val Accuracy: 66.48%
Epoch [8/100], Loss: 0.6019, Val Loss: 0.6013, Val Accuracy: 71.42%
Epoch [9/100], Loss: 0.5891, Val Loss: 0.5866, Val Accuracy: 71.56%
Epoch [10/100], Loss: 0.5749, Val Loss: 0.5729, Val Accuracy: 72.49%
Epoch [11/100], Loss: 0.5620, Val Loss: 0.5627, Val Accuracy: 73.07%
Epoch [12/100], Loss: 0.5534, Val Loss: 0.5563, Val Accuracy: 73.42%
Epoch [13/100], Loss: 0.5481, Val Loss: 0.5525, Val Accuracy: 72.71%
Epoch [14/100], Loss: 0.5438, Val Loss: 0.5488, Val Accuracy: 72.56%
Epoch [15/100], Loss: 0.5401, Val Loss: 0.5450, Val Accuracy: 72.49%
Epoch [16/100], Loss: 0.5365, Val Loss: 0.5417, Val Accuracy: 72.35%
Epoch [17/100], Loss: 0.5333, Val Loss: 0.5386, Val Accuracy: 72.49%
Epoch [18/100], Loss: 0.5302, Val Loss: 0.5361, Val Accuracy: 72.35%
Epoch [19/100], Loss: 0.5279, Val Loss: 0.5349, Val Accuracy: 72.13%
Epoch [20/100], Loss: 0.5267, Val Loss: 0.5328, Val Accuracy: 72.13%
Epoch [21/100], Loss: 0.5244, Val Loss: 0.5313, Val Accuracy: 72.06%
Epoch [22/100], Loss: 0.5227, Val Loss: 0.5299, Val Accuracy: 72.56%
Epoch [23/100], Loss: 0.5219, Val Loss: 0.5290, Val Accuracy: 72.28%
Epoch [24/100], Loss: 0.5202, Val Loss: 0.5280, Val Accuracy: 72.56%
Epoch [25/100], Loss: 0.5191, Val Loss: 0.5275, Val Accuracy: 72.49%
Epoch [26/100], Loss: 0.5179, Val Loss: 0.5266, Val Accuracy: 72.21%
Epoch [27/100], Loss: 0.5169, Val Loss: 0.5257, Val Accuracy: 72.21%
Epoch [28/100], Loss: 0.5157, Val Loss: 0.5248, Val Accuracy: 72.42%
Epoch [29/100], Loss: 0.5150, Val Loss: 0.5244, Val Accuracy: 72.21%
Epoch [30/100], Loss: 0.5143, Val Loss: 0.5236, Val Accuracy: 72.28%
Epoch [31/100], Loss: 0.5138, Val Loss: 0.5235, Val Accuracy: 72.21%
Epoch [32/100], Loss: 0.5133, Val Loss: 0.5231, Val Accuracy: 72.06%
Epoch [33/100], Loss: 0.5126, Val Loss: 0.5226, Val Accuracy: 71.99%
Epoch [34/100], Loss: 0.5120, Val Loss: 0.5223, Val Accuracy: 72.28%
Epoch [35/100], Loss: 0.5116, Val Loss: 0.5222, Val Accuracy: 72.13%
Epoch [36/100], Loss: 0.5116, Val Loss: 0.5217, Val Accuracy: 71.99%
Epoch [37/100], Loss: 0.5105, Val Loss: 0.5214, Val Accuracy: 72.13%
Epoch [38/100], Loss: 0.5108, Val Loss: 0.5214, Val Accuracy: 71.99%

```

Epoch [39/100], Loss: 0.5097, Val Loss: 0.5203, Val Accuracy: 72.13%
 Epoch [40/100], Loss: 0.5092, Val Loss: 0.5202, Val Accuracy: 72.28%
 Epoch [41/100], Loss: 0.5089, Val Loss: 0.5200, Val Accuracy: 72.42%
 Epoch [42/100], Loss: 0.5084, Val Loss: 0.5196, Val Accuracy: 72.42%
 Epoch [43/100], Loss: 0.5079, Val Loss: 0.5195, Val Accuracy: 72.42%
 Epoch [44/100], Loss: 0.5074, Val Loss: 0.5192, Val Accuracy: 72.64%
 Epoch [45/100], Loss: 0.5070, Val Loss: 0.5188, Val Accuracy: 72.42%
 Epoch [46/100], Loss: 0.5069, Val Loss: 0.5189, Val Accuracy: 72.64%
 Epoch [47/100], Loss: 0.5066, Val Loss: 0.5186, Val Accuracy: 72.78%
 Epoch [48/100], Loss: 0.5062, Val Loss: 0.5186, Val Accuracy: 72.92%
 Epoch [49/100], Loss: 0.5061, Val Loss: 0.5180, Val Accuracy: 72.64%
 Epoch [50/100], Loss: 0.5058, Val Loss: 0.5177, Val Accuracy: 72.99%
 Epoch [51/100], Loss: 0.5054, Val Loss: 0.5180, Val Accuracy: 73.21%
 Epoch [52/100], Loss: 0.5054, Val Loss: 0.5178, Val Accuracy: 72.64%
 Epoch [53/100], Loss: 0.5049, Val Loss: 0.5176, Val Accuracy: 74.07%
 Epoch [54/100], Loss: 0.5047, Val Loss: 0.5172, Val Accuracy: 73.64%
 Epoch [55/100], Loss: 0.5043, Val Loss: 0.5175, Val Accuracy: 74.57%
 Epoch [56/100], Loss: 0.5043, Val Loss: 0.5171, Val Accuracy: 73.78%
 Epoch [57/100], Loss: 0.5042, Val Loss: 0.5170, Val Accuracy: 73.78%
 Epoch [58/100], Loss: 0.5039, Val Loss: 0.5172, Val Accuracy: 74.64%
 Epoch [59/100], Loss: 0.5035, Val Loss: 0.5167, Val Accuracy: 75.14%
 Epoch [60/100], Loss: 0.5035, Val Loss: 0.5167, Val Accuracy: 74.64%
 Epoch [61/100], Loss: 0.5029, Val Loss: 0.5165, Val Accuracy: 74.79%
 Epoch [62/100], Loss: 0.5032, Val Loss: 0.5166, Val Accuracy: 74.64%
 Epoch [63/100], Loss: 0.5032, Val Loss: 0.5160, Val Accuracy: 75.29%
 Epoch [64/100], Loss: 0.5028, Val Loss: 0.5163, Val Accuracy: 75.07%
 Epoch [65/100], Loss: 0.5028, Val Loss: 0.5160, Val Accuracy: 75.00%
 Epoch [66/100], Loss: 0.5026, Val Loss: 0.5159, Val Accuracy: 75.07%
 Epoch [67/100], Loss: 0.5022, Val Loss: 0.5159, Val Accuracy: 74.93%
 Epoch [68/100], Loss: 0.5023, Val Loss: 0.5156, Val Accuracy: 75.21%
 Epoch [69/100], Loss: 0.5021, Val Loss: 0.5160, Val Accuracy: 74.79%
 Epoch [70/100], Loss: 0.5018, Val Loss: 0.5156, Val Accuracy: 74.86%
 Epoch [71/100], Loss: 0.5014, Val Loss: 0.5155, Val Accuracy: 75.07%
 Epoch [72/100], Loss: 0.5011, Val Loss: 0.5154, Val Accuracy: 74.93%
 Epoch [73/100], Loss: 0.5012, Val Loss: 0.5152, Val Accuracy: 74.86%
 Epoch [74/100], Loss: 0.5012, Val Loss: 0.5147, Val Accuracy: 75.14%
 Epoch [75/100], Loss: 0.5004, Val Loss: 0.5149, Val Accuracy: 74.93%
 Epoch [76/100], Loss: 0.5006, Val Loss: 0.5143, Val Accuracy: 74.71%
 Epoch [77/100], Loss: 0.5002, Val Loss: 0.5143, Val Accuracy: 75.14%
 Epoch [78/100], Loss: 0.5003, Val Loss: 0.5142, Val Accuracy: 74.93%
 Epoch [79/100], Loss: 0.5000, Val Loss: 0.5139, Val Accuracy: 74.93%
 Epoch [80/100], Loss: 0.4998, Val Loss: 0.5142, Val Accuracy: 75.07%
 Epoch [81/100], Loss: 0.4996, Val Loss: 0.5137, Val Accuracy: 74.93%
 Epoch [82/100], Loss: 0.4993, Val Loss: 0.5137, Val Accuracy: 75.21%
 Epoch [83/100], Loss: 0.4989, Val Loss: 0.5137, Val Accuracy: 75.14%
 Epoch [84/100], Loss: 0.4992, Val Loss: 0.5131, Val Accuracy: 75.36%
 Epoch [85/100], Loss: 0.4990, Val Loss: 0.5131, Val Accuracy: 75.14%
 Epoch [86/100], Loss: 0.4986, Val Loss: 0.5129, Val Accuracy: 74.93%

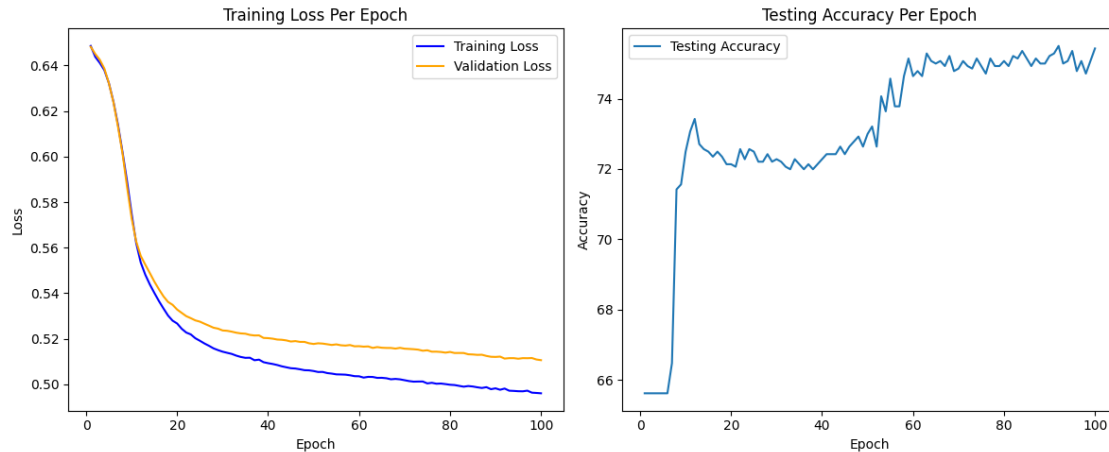
```
Epoch [87/100], Loss: 0.4983, Val Loss: 0.5129, Val Accuracy: 75.14%
Epoch [88/100], Loss: 0.4987, Val Loss: 0.5124, Val Accuracy: 75.00%
Epoch [89/100], Loss: 0.4978, Val Loss: 0.5121, Val Accuracy: 75.00%
Epoch [90/100], Loss: 0.4982, Val Loss: 0.5120, Val Accuracy: 75.21%
Epoch [91/100], Loss: 0.4976, Val Loss: 0.5121, Val Accuracy: 75.29%
Epoch [92/100], Loss: 0.4981, Val Loss: 0.5113, Val Accuracy: 75.50%
Epoch [93/100], Loss: 0.4971, Val Loss: 0.5115, Val Accuracy: 75.00%
Epoch [94/100], Loss: 0.4971, Val Loss: 0.5115, Val Accuracy: 75.07%
Epoch [95/100], Loss: 0.4969, Val Loss: 0.5112, Val Accuracy: 75.36%
Epoch [96/100], Loss: 0.4969, Val Loss: 0.5115, Val Accuracy: 74.79%
Epoch [97/100], Loss: 0.4972, Val Loss: 0.5114, Val Accuracy: 75.07%
Epoch [98/100], Loss: 0.4963, Val Loss: 0.5115, Val Accuracy: 74.71%
Epoch [99/100], Loss: 0.4962, Val Loss: 0.5109, Val Accuracy: 75.07%
Epoch [100/100], Loss: 0.4960, Val Loss: 0.5106, Val Accuracy: 75.43%
Accuracy of the model on the test set: 75.43%
```

```
[52]: # Plotting Loss and Accuracy
plt.figure(figsize=(12, 5))

# Plotting training loss
plt.subplot(1, 2, 1)
plt.plot(range(1, num_iterations + 1), epoch_losses, label='Training Loss',
         color="blue")
plt.plot(range(1, num_iterations + 1), val_epoch_losses, label="Validation
         Loss", color="orange")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Per Epoch')
plt.legend()

# Plotting accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, num_iterations + 1), val_epoch_accuracies, label='Testing
         Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Testing Accuracy Per Epoch')
plt.legend()

plt.tight_layout()
plt.show()
```



Designing a Neural Network through trial and error

```
[53]: class TE_NeuralNetwork(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, hidden_size3,
        ↪ output_size):
        super(TE_NeuralNetwork, self).__init__()

        self.hidden1 = nn.Linear(input_size, hidden_size1)
        self.hidden2 = nn.Linear(hidden_size1, hidden_size2)
        self.hidden3 = nn.Linear(hidden_size2, hidden_size3)

        # Define the output layer
        self.output = nn.Linear(hidden_size3, output_size)

    def forward(self, x):
        # Apply a non-linear activation function / ReLU after each hidden layer
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        x = F.relu(self.hidden3(x))

        # The output layer
        x = self.output(x)
        return x
```

```
[54]: num_features = df_normalized.shape[1]
hidden_size = 45
hidden_step = 9
num_output = 1
print(f'Number of features: {num_features}')
print(f'Output Size: {num_output}')
print(f'Number of neurons: {hidden_size}')
print(f'Number of neurons layer 3: {hidden_step}')
```

Number of features: 9
Output Size: 1
Number of neurons: 45
Number of neurons layer 3: 9

```
[55]: # Example usage
input_size = num_features # Size of input (number of input features)
hidden_size1 = hidden_size #Size of first hidden layer
hidden_size2 = hidden_size #Size of second hidden layer
hidden_size3 = hidden_step #Size of third hidden layer
output_size = num_output # Size of output (number of classes for
    ↪classification)

model = TE_NeuralNetwork(input_size, hidden_size1, hidden_size2, hidden_size3,
    ↪output_size)
```

```
[56]: # Data Loaders
batch_size = 64 # Define your batch size
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size,
    ↪shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size,
    ↪shuffle=False)

# Initialize the model, loss function, and optimizer
model = TE_NeuralNetwork(input_size, hidden_size1, hidden_size2, hidden_size3,
    ↪output_size)
criterion = nn.BCEWithLogitsLoss()
#For binary classification, use nn.BCEWithLogitsLoss and for multi-class, use
    ↪nn.CrossEntropyLoss.

optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training Loop
num_iterations = 128 # Define the number of epochs
def train_and_evaluate_model(model, train_loader, test_loader, optimizer,
    ↪criterion, num_iterations):
    epoch_losses = []
    epoch_accuracies = []
    val_epoch_losses = []
    val_epoch_accuracies = []

    for epoch in range(num_iterations):
        # Training Loop
        model.train()
        total_loss = 0
        for inputs, targets in train_loader:
```

```

        optimizer.zero_grad()
        outputs = model(inputs)
        outputs = outputs.squeeze()
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(train_loader)
    epoch_losses.append(avg_loss)

    # Testing Loop
    model.eval()
    val_total_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, targets in test_loader:
            outputs = model(inputs)
            outputs = outputs.squeeze()
            val_loss = criterion(outputs, targets)
            val_total_loss += val_loss.item()
            predicted = outputs > 0 # Adjust the threshold as needed
            total += targets.size(0)
            correct += (predicted == targets).sum().item()
    accuracy = 100 * correct / total
    epoch_accuracies.append(accuracy)
    avg_val_loss = val_total_loss / len(test_loader)
    val_epoch_losses.append(avg_val_loss)
    val_accuracy = 100 * correct / total
    val_epoch_accuracies.append(val_accuracy)
    return epoch_losses, epoch_accuracies, val_epoch_losses,
    ↪ val_epoch_accuracies

epoch_losses, epoch_accuracies, val_epoch_losses, val_epoch_accuracies =
    ↪ train_and_evaluate_model(model, train_loader, test_loader, optimizer,
    ↪ criterion, num_iterations)

# Print the final accuracy
print(f'Accuracy of the model on the test set: {epoch_accuracies[-1]:.2f}%')

```

Accuracy of the model on the test set: 81.23%

```

[57]: # Plotting Loss and Accuracy
plt.figure(figsize=(12, 5))

# Plotting training loss
plt.subplot(1, 2, 1)

```

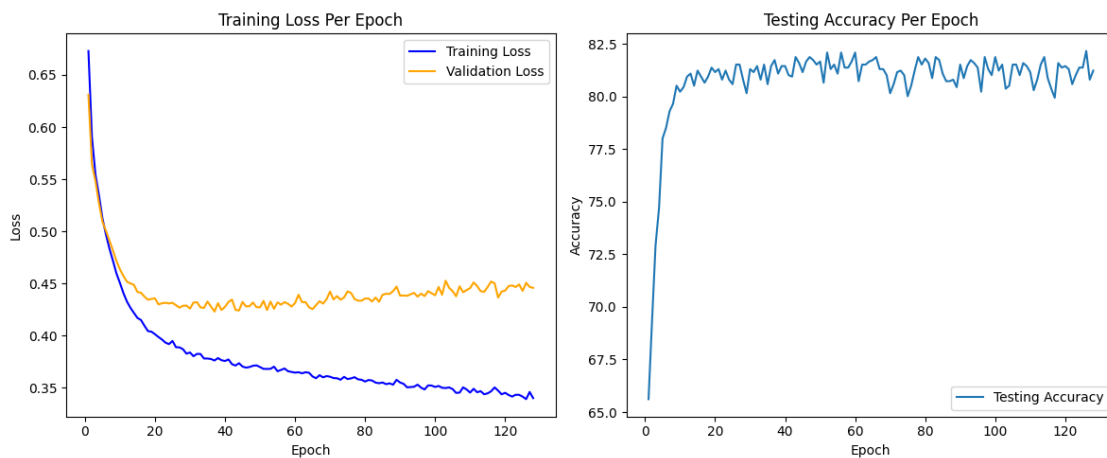
```

plt.plot(range(1, num_iterations + 1), epoch_losses, label='Training Loss',
         color="blue")
plt.plot(range(1, num_iterations + 1), val_epoch_losses, label="Validation
         Loss", color="orange")
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Per Epoch')
plt.legend()

# Plotting accuracy
plt.subplot(1, 2, 2)
plt.plot(range(1, num_iterations + 1), val_epoch_accuracies, label='Testing
         Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Testing Accuracy Per Epoch')
plt.legend()

plt.tight_layout()
plt.show()

```



We appear to reach maximum accuracy at around 40 to 60 Epochs. This is also where the Validation loss begins to level off or increase indicating model overfit.

```

[58]: num_iterations = 50
epoch_losses, epoch_accuracies, val_epoch_losses, val_epoch_accuracies =
    train_and_evaluate_model(model, train_loader, test_loader, optimizer,
    criterion, num_iterations)

# Print the final accuracy
print(f'Accuracy of the model on the test set: {epoch_accuracies[-1]:.2f}%')

```

Accuracy of the model on the test set: 81.52%

2.1.6 Training and Testing Additional Machine Learning Algorithms

Adapted from Pooja Joshi

```
[59]: from sklearn.model_selection import   
      ↪ train_test_split, cross_val_score, GridSearchCV   
      from sklearn.linear_model import LogisticRegression   
      from sklearn.tree import DecisionTreeClassifier   
      from sklearn.svm import SVC   
      from sklearn.neighbors import KNeighborsClassifier   
      from sklearn.ensemble import   
      ↪ RandomForestClassifier, AdaBoostClassifier, BaggingClassifier, ExtraTreesClassifier   
      from sklearn.naive_bayes import GaussianNB   
      from sklearn.metrics import   
      ↪ accuracy_score, confusion_matrix, classification_report, roc_curve, auc   
      from datetime import datetime   
      from sklearn.feature_selection import RFE   
  
[69]: X_train, X_test, y_train, y_test = train_test_split(df_normalized,   
      ↪ dummyy, test_size = 0.25, random_state = 42)   
  
[70]: #Instantiate the classifiers   
      clf_logreg = LogisticRegression()   
      clf_tree = DecisionTreeClassifier()   
      clf_knn = KNeighborsClassifier()   
      clf_svc = SVC()   
      clf_forest = RandomForestClassifier()   
      clf_ada = AdaBoostClassifier()   
      clf_bagging = BaggingClassifier()   
      clf_extratrees = ExtraTreesClassifier()   
      clf_gnb = GaussianNB()   
  
[71]: classifiers = ['LogisticRegression', 'DecisionTree', 'KNN', 'SVC',   
      ↪ 'RandomForest', 'Adaboost', 'Bagging', 'Extratrees', 'Naive']   
  
[72]: models = {clf_logreg: 'LogisticRegression',   
      clf_tree: 'DecisionTree',   
      clf_knn: 'KNN',   
      clf_svc: 'SVC',   
      clf_forest: 'RandomForest',   
      clf_ada: 'Adaboost',   
      clf_bagging: 'Bagging',   
      clf_extratrees: 'Extratrees' ,   
      clf_gnb: 'Naive'}
```



```

[73]: # train function fits the model and returns accuracy score
def train(algo,name,X_train,y_train,X_test,y_test):
    algo.fit(X_train,y_train)
    y_pred = algo.predict(X_test)
    score = accuracy_score(y_test,y_pred)

    print(f"-----{name}-----")
    print(f"Accuracy Score for {name}: {score*100:.4f}%")
    return y_test,y_pred,score

# acc_res function calculates confusion matrix
def acc_res(y_test,y_pred):
    null_accuracy = y_test.value_counts()[0]/len(y_test)
    print(f"Null Accuracy: {null_accuracy*100:.4f}%")
    print("Confusion Matrix")
    matrix = confusion_matrix(y_test,y_pred)
    print(matrix)

    print("+++++")
    TN = matrix[0,0]
    FP = matrix[0,1]
    FN = matrix[1,0]
    TP = matrix[1,1]
    accuracy_score=(TN+TP) / float(TP+TN+FP+FN)
    recall_score = (TP)/ float(TP+FN)
    specificity = TN / float(TN+FP)
    FPR = FP / float(FP+TN)
    precision_score = TP / float(TP+FP)
    print(f"Accuracy Score: {accuracy_score*100:.4f}%")
    print(f"Recall Score: {recall_score*100:.4f}%")
    print(f"Specificity Score: {specificity*100:.4f}%")
    print(f"False Positive Rate: {FPR*100:.4f}%")
    print(f"Precision Score: {precision_score*100:.4f}%")

    print("+++++")
    print("Classification Report")
    print(classification_report(y_test,y_pred))

def main(models):
    accuracy_scores = []
    for algo,name in models.items():
        y_test_train,y_pred,acc_score =
    train(algo,name,X_train,y_train,X_test,y_test)
        acc_res(y_test_train,y_pred)
        accuracy_scores.append(acc_score)
    return accuracy_scores

```

```
accuracy_scores = main(models)
```

-----LogisticRegression-----

Accuracy Score for LogisticRegression: 70.6186%

Null Accuracy: 66.5808%

Confusion Matrix

```
[[694  81]
```

```
 [261 128]]
```

+++++

Accuracy Score: 70.6186%

Recall Score: 32.9049%

Specificity Score: 89.5484%

False Positive Rate: 10.4516%

Precision Score: 61.2440%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.73	0.90	0.80	775
1	0.61	0.33	0.43	389
accuracy			0.71	1164
macro avg	0.67	0.61	0.62	1164
weighted avg	0.69	0.71	0.68	1164

-----DecisionTree-----

Accuracy Score for DecisionTree: 81.2715%

Null Accuracy: 66.5808%

Confusion Matrix

```
[[691  84]
```

```
 [134 255]]
```

+++++

Accuracy Score: 81.2715%

Recall Score: 65.5527%

Specificity Score: 89.1613%

False Positive Rate: 10.8387%

Precision Score: 75.2212%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.84	0.89	0.86	775
1	0.75	0.66	0.70	389
accuracy			0.81	1164
macro avg	0.79	0.77	0.78	1164

weighted avg 0.81 0.81 0.81 1164

-----KNN-----

Accuracy Score for KNN: 80.0687%

Null Accuracy: 66.5808%

Confusion Matrix

[[693 82]

[150 239]]

+++++

Accuracy Score: 80.0687%

Recall Score: 61.4396%

Specificity Score: 89.4194%

False Positive Rate: 10.5806%

Precision Score: 74.4548%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.82	0.89	0.86	775
1	0.74	0.61	0.67	389
accuracy			0.80	1164
macro avg	0.78	0.75	0.76	1164
weighted avg	0.80	0.80	0.80	1164

-----SVC-----

Accuracy Score for SVC: 82.0447%

Null Accuracy: 66.5808%

Confusion Matrix

[[728 47]

[162 227]]

+++++

Accuracy Score: 82.0447%

Recall Score: 58.3548%

Specificity Score: 93.9355%

False Positive Rate: 6.0645%

Precision Score: 82.8467%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.82	0.94	0.87	775
1	0.83	0.58	0.68	389
accuracy			0.82	1164
macro avg	0.82	0.76	0.78	1164

weighted avg 0.82 0.82 0.81 1164

-----RandomForest-----

Accuracy Score for RandomForest: 82.3883%

Null Accuracy: 66.5808%

Confusion Matrix

[[701 74]

[131 258]]

+++++

Accuracy Score: 82.3883%

Recall Score: 66.3239%

Specificity Score: 90.4516%

False Positive Rate: 9.5484%

Precision Score: 77.7108%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.84	0.90	0.87	775
1	0.78	0.66	0.72	389
accuracy			0.82	1164
macro avg	0.81	0.78	0.79	1164
weighted avg	0.82	0.82	0.82	1164

-----Adaboost-----

Accuracy Score for Adaboost: 81.6151%

Null Accuracy: 66.5808%

Confusion Matrix

[[731 44]

[170 219]]

+++++

Accuracy Score: 81.6151%

Recall Score: 56.2982%

Specificity Score: 94.3226%

False Positive Rate: 5.6774%

Precision Score: 83.2700%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.81	0.94	0.87	775
1	0.83	0.56	0.67	389
accuracy			0.82	1164
macro avg	0.82	0.75	0.77	1164

weighted avg 0.82 0.82 0.81 1164

-----Bagging-----

Accuracy Score for Bagging: 80.6701%

Null Accuracy: 66.5808%

Confusion Matrix

[[681 94]

[131 258]]

+++++

Accuracy Score: 80.6701%

Recall Score: 66.3239%

Specificity Score: 87.8710%

False Positive Rate: 12.1290%

Precision Score: 73.2955%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.84	0.88	0.86	775
1	0.73	0.66	0.70	389
accuracy			0.81	1164
macro avg	0.79	0.77	0.78	1164
weighted avg	0.80	0.81	0.80	1164

-----Extratrees-----

Accuracy Score for Extratrees: 82.0447%

Null Accuracy: 66.5808%

Confusion Matrix

[[697 78]

[131 258]]

+++++

Accuracy Score: 82.0447%

Recall Score: 66.3239%

Specificity Score: 89.9355%

False Positive Rate: 10.0645%

Precision Score: 76.7857%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.84	0.90	0.87	775
1	0.77	0.66	0.71	389
accuracy			0.82	1164
macro avg	0.80	0.78	0.79	1164

weighted avg	0.82	0.82	0.82	1164
--------------	------	------	------	------

-----Naive-----

Accuracy Score for Naive: 64.9485%

Null Accuracy: 66.5808%

Confusion Matrix

[[580 195]

[213 176]]

+++++

Accuracy Score: 64.9485%

Recall Score: 45.2442%

Specificity Score: 74.8387%

False Positive Rate: 25.1613%

Precision Score: 47.4394%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.73	0.75	0.74	775
1	0.47	0.45	0.46	389
accuracy			0.65	1164
macro avg	0.60	0.60	0.60	1164
weighted avg	0.65	0.65	0.65	1164

```
[74]: pd.DataFrame(accuracy_scores, columns = ['Accuracy Scores'], index = classifiers).
      ↪ sort_values(by = 'Accuracy Scores',
      ↪
      ↪ ascending = False)
```

```
[74]: Accuracy Scores
RandomForest    0.823883
SVC              0.820447
Extratrees      0.820447
Adaboost        0.816151
DecisionTree    0.812715
Bagging         0.806701
KNN             0.800687
LogisticRegression 0.706186
Naive           0.649485
```

A Random Forest appears to perform the best of the common models but still under perform XGBoost.

Running the models without normalization or dropping features

```

[75]: X_train,X_test,y_train,y_test = train_test_split(pd.get_dummies(X),
↳dummyy,test_size =0.25,random_state = 42)

[76]: warnings.simplefilter(action='ignore', category=RuntimeWarning)
import sklearn.exceptions
warnings.simplefilter(action='ignore', category=sklearn.exceptions.
↳UndefinedMetricWarning)

# train function fits the model and returns accuracy score
def train(algo,name,X_train,y_train,X_test,y_test):
    algo.fit(X_train,y_train)
    y_pred = algo.predict(X_test)
    score = accuracy_score(y_test,y_pred)
    ↳
    ↳print(f"-----{name}-----")
    print(f"Accuracy Score for {name}: {score*100:.4f}%")
    return y_test,y_pred,score

# acc_res function calculates confusion matrix
def acc_res(y_test,y_pred):
    null_accuracy = y_test.value_counts()[0]/len(y_test)
    print(f"Null Accuracy: {null_accuracy*100:.4f}%")
    print("Confusion Matrix")
    matrix = confusion_matrix(y_test,y_pred)
    print(matrix)
    ↳
    ↳print("+++++")
    TN = matrix[0,0]
    FP = matrix[0,1]
    FN = matrix[1,0]
    TP = matrix[1,1]
    accuracy_score=(TN+TP) / float(TP+TN+FP+FN)
    recall_score = (TP)/ float(TP+FN)
    specificity = TN / float(TN+FP)
    FPR = FP / float(FP+TN)
    precision_score = TP / float(TP+FP)
    print(f"Accuracy Score: {accuracy_score*100:.4f}%")
    print(f"Recall Score: {recall_score*100:.4f}%")
    print(f"Specificity Score: {specificity*100:.4f}%")
    print(f"False Positive Rate: {FPR*100:.4f}%")
    print(f"Precision Score: {precision_score*100:.4f}%")
    ↳
    ↳print("+++++")
    print("Classification Report")
    print(classification_report(y_test,y_pred))

def main(models):

```

```

    accuracy_scores = []
    for algo,name in models.items():
        y_test_train,y_pred,acc_score =
→train(algo,name,X_train,y_train,X_test,y_test)
        acc_res(y_test_train,y_pred)
        accuracy_scores.append(acc_score)
    return accuracy_scores

accuracy_scores = main(models)

```

-----LogisticRegression-----

Accuracy Score for LogisticRegression: 69.7595%

Null Accuracy: 66.5808%

Confusion Matrix

[[697 78]

[274 115]]

+++++

Accuracy Score: 69.7595%

Recall Score: 29.5630%

Specificity Score: 89.9355%

False Positive Rate: 10.0645%

Precision Score: 59.5855%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.72	0.90	0.80	775
1	0.60	0.30	0.40	389
accuracy			0.70	1164
macro avg	0.66	0.60	0.60	1164
weighted avg	0.68	0.70	0.66	1164

-----DecisionTree-----

Accuracy Score for DecisionTree: 80.4983%

Null Accuracy: 66.5808%

Confusion Matrix

[[684 91]

[136 253]]

+++++

Accuracy Score: 80.4983%

Recall Score: 65.0386%

Specificity Score: 88.2581%

False Positive Rate: 11.7419%

Precision Score: 73.5465%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.83	0.88	0.86	775
1	0.74	0.65	0.69	389
accuracy			0.80	1164
macro avg	0.78	0.77	0.77	1164
weighted avg	0.80	0.80	0.80	1164

-----KNN-----

Accuracy Score for KNN: 77.5773%

Null Accuracy: 66.5808%

Confusion Matrix

[[687 88]

[173 216]]

+++++

Accuracy Score: 77.5773%

Recall Score: 55.5270%

Specificity Score: 88.6452%

False Positive Rate: 11.3548%

Precision Score: 71.0526%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.80	0.89	0.84	775
1	0.71	0.56	0.62	389
accuracy			0.78	1164
macro avg	0.75	0.72	0.73	1164
weighted avg	0.77	0.78	0.77	1164

-----SVC-----

Accuracy Score for SVC: 66.5808%

Null Accuracy: 66.5808%

Confusion Matrix

[[775 0]

[389 0]]

+++++

Accuracy Score: 66.5808%

Recall Score: 0.0000%

Specificity Score: 100.0000%

False Positive Rate: 0.0000%

Precision Score: nan%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.67	1.00	0.80	775
1	0.00	0.00	0.00	389
accuracy			0.67	1164
macro avg	0.33	0.50	0.40	1164
weighted avg	0.44	0.67	0.53	1164

-----RandomForest-----

Accuracy Score for RandomForest: 81.5292%

Null Accuracy: 66.5808%

Confusion Matrix

[[688 87]

[128 261]]

+++++

Accuracy Score: 81.5292%

Recall Score: 67.0951%

Specificity Score: 88.7742%

False Positive Rate: 11.2258%

Precision Score: 75.0000%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.84	0.89	0.86	775
1	0.75	0.67	0.71	389
accuracy			0.82	1164
macro avg	0.80	0.78	0.79	1164
weighted avg	0.81	0.82	0.81	1164

-----Adaboost-----

Accuracy Score for Adaboost: 81.6151%

Null Accuracy: 66.5808%

Confusion Matrix

[[731 44]

[170 219]]

+++++

Accuracy Score: 81.6151%

Recall Score: 56.2982%

Specificity Score: 94.3226%

False Positive Rate: 5.6774%

Precision Score: 83.2700%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.81	0.94	0.87	775
1	0.83	0.56	0.67	389
accuracy			0.82	1164
macro avg	0.82	0.75	0.77	1164
weighted avg	0.82	0.82	0.81	1164

-----Bagging-----

Accuracy Score for Bagging: 81.0137%

Null Accuracy: 66.5808%

Confusion Matrix

[[683 92]

[129 260]]

+++++

Accuracy Score: 81.0137%

Recall Score: 66.8380%

Specificity Score: 88.1290%

False Positive Rate: 11.8710%

Precision Score: 73.8636%

+++++

Classification Report

	precision	recall	f1-score	support
0	0.84	0.88	0.86	775
1	0.74	0.67	0.70	389
accuracy			0.81	1164
macro avg	0.79	0.77	0.78	1164
weighted avg	0.81	0.81	0.81	1164

-----Extratrees-----

Accuracy Score for Extratrees: 82.0447%

Null Accuracy: 66.5808%

Confusion Matrix

[[698 77]

[132 257]]

+++++

Accuracy Score: 82.0447%

Recall Score: 66.0668%

Specificity Score: 90.0645%

False Positive Rate: 9.9355%

Precision Score: 76.9461%

+++++

```

Classification Report
              precision    recall  f1-score   support

     0       0.84         0.90         0.87         775
     1       0.77         0.66         0.71         389

 accuracy
macro avg       0.81         0.78         0.79         1164
weighted avg    0.82         0.82         0.82         1164

```

-----Naive-----

Accuracy Score for Naive: 70.2749%

Null Accuracy: 66.5808%

Confusion Matrix

[[605 170]

[176 213]]

+++++

Accuracy Score: 70.2749%

Recall Score: 54.7558%

Specificity Score: 78.0645%

False Positive Rate: 21.9355%

Precision Score: 55.6136%

+++++

Classification Report

```

              precision    recall  f1-score   support

     0       0.77         0.78         0.78         775
     1       0.56         0.55         0.55         389

 accuracy
macro avg       0.67         0.66         0.66         1164
weighted avg    0.70         0.70         0.70         1164

```

```

[77]: pd.DataFrame(accuracy_scores,columns = ['Accuracy Scores'],index = classifiers).
      ↪sort_values(by = 'Accuracy Scores',
      ↪
      ↪          ascending = False)

```

```

[77]: Accuracy Scores
Extratrees      0.820447
Adaboost        0.816151
RandomForest    0.815292
Bagging         0.810137
DecisionTree    0.804983
KNN             0.775773

```

Naive	0.702749
LogisticRegression	0.697595
SVC	0.665808

Improvements can be noted in the Naive Bayes and Bagging models, but there's a big drop in the Support Vector Classifier performance. Overall the models seem have greater accuracy with normalization.