CFRM 421 Final Project, Spring 2023

Haobo Jiang, Rundong Liu, Mengyao Shi, Xinying Wang

Company Bankruptcy Prediction

Introduction

Stability and viability of a company's finances are crucial factors in determining its long-term success. It is essential for a business to evaluate and manage potential risks, including the possibility of insolvency. In addition to the loss of assets, disruption of operations, and damage to a company's reputation, bankruptcy can have serious consequences.

The primary objective of the project is to develop a robust predictive model that can accurately predict the likelihood of bankruptcy within a given time frame. This model will provide valuable insights into the company's financial trajectory, allowing us to identify early warning signs and take preventative measures against financial distress.

This report provides an overview of various machine learning models used for bankruptcy prediction. The models discussed include Logistic Regression, Random Forest, Support Vector Machines (SVM), Principal Component Analysis (PCA), K-Means Clustering, and Neural Networks. Each model is evaluated based on its performance, strengths, limitations, and suitability for bankruptcy rate prediction. Logistic Regression offers interpretability and efficiency but has limited expressive power. Random Forest provides high accuracy and handles high-dimensional data but can be complex. SVM captures complex relationships but can be computationally expensive. PCA improves efficiency but reduces interpretability. K-Means Clustering aids in data exploration but has limitations in handling nonlinear data. Neural Networks capture nonlinear relationships but lack interpretability and require ample data.

By delving into these predictive models, we aim to provide a comprehensive understanding of their potential for predicting bankruptcy rates and empowering our organization to make informed decisions and take proactive measures to mitigate the risk of insolvency.

The data were collected from the Taiwan Economic Journal for the years 1999 to 2009. Company bankruptcy was defined based on the business regulations of the Taiwan Stock Exchange.

Some sample features are listed below:

X1 - ROA(C) before interest and depreciation before interest: Return On Total Assets(C) X2 - ROA(A) before interest and % after tax: Return On Total Assets(A) X3 - ROA(B) before interest and depreciation after tax: Return On Total Assets(B) X4 - Operating Gross Margin: Gross Profit/Net Sales X5 - Realized Sales Gross Margin: Realized Gross Profit/Net Sales

In [1]:
```python
import pandas as pd

# import csv file
data = pd.read_csv("data.csv")
data
```

Out[1]:

	Bankrupt?	ROA(C) before interest and depreciation before interest	ROA(A) before interest and % after tax	ROA(B) before interest and depreciation after tax	Operating Gross Margin	Realized Sales Gross Margin	Operating Profit Rate	Pre-t r Intere Ra
0	1	0.370594	0.424389	0.405750	0.601457	0.601457	0.998969	0.7968
1	1	0.464291	0.538214	0.516730	0.610235	0.610235	0.998946	0.7973
2	1	0.426071	0.499019	0.472295	0.601450	0.601364	0.998857	0.7964
3	1	0.399844	0.451265	0.457733	0.583541	0.583541	0.998700	0.7969
4	1	0.465022	0.538432	0.522298	0.598783	0.598783	0.998973	0.7973
...	...	...	...	...	...	...	...	
6814	0	0.493687	0.539468	0.543230	0.604455	0.604462	0.998992	0.7974
6815	0	0.475162	0.538269	0.524172	0.598308	0.598308	0.998992	0.7974
6816	0	0.472725	0.533744	0.520638	0.610444	0.610213	0.998984	0.7974
6817	0	0.506264	0.559911	0.554045	0.607850	0.607850	0.999074	0.7975
6818	0	0.493053	0.570105	0.549548	0.627409	0.627409	0.998080	0.8019

6819 rows × 96 columns

We use the `StratifiedShuffleSplit` method instantiated with parameters n_splits=1 (indicating the number of splits to be performed), test_size=0.2 (representing the proportion of the dataset to be used for testing), and random_state=42 (used to ensure reproducibility of the split). The `split_obj.split` function is called to perform the stratified shuffle split. It takes the data dataframe and the "Bankrupt?" column as input and returns indices for the training and testing sets based on the specified stratified sampling.

In [2]:
```python
import numpy as np
from sklearn.model_selection import StratifiedShuffleSplit

# train test split
data["cat"] = pd.cut(data["Bankrupt?"],
                               bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                               labels=[1, 2, 3, 4, 5])
split_obj = StratifiedShuffleSplit(n_splits=1, test_size=0.2, random_state=42)
for train_index, test_index in split_obj.split(data, data["Bankrupt?"]):
    strat_train_set = data.loc[train_index]
    strat_test_set = data.loc[test_index]

for set_ in (strat_train_set, strat_test_set):
    set_.drop("cat", axis=1, inplace=True)
```

In [3]:
```
# check the number of 0s and 1s in the training set
value_counts = strat_train_set['Bankrupt?'].value_counts()
value_counts
```

Out[3]:
```
0    5279
1     176
Name: Bankrupt?, dtype: int64
```

From the `value_counts` variable, we could see that the number of 0s in the stratified training set is 5279 while the number of 1s is only 176. This indicates that our dataset is very imbalanced. To make it become balanced, the first approach we take is to concatenate the number of 0s in the training set to the same as the number of 1s (not change the number in the test set).

After using the method of `drop`, we finally made the number of 0s to 176 as well. This makes the dataset more balanced and more proper for us to make models.

In [4]:
```
# drop the number of 0s in the training set to the same number of 1s in this tr
strat_train_set.drop(strat_train_set[(strat_train_set['Bankrupt?'] == 0) & (str
value_counts = strat_train_set['Bankrupt?'].value_counts()
value_counts
```

Out[4]:
```
1    176
0    176
Name: Bankrupt?, dtype: int64
```

The choice of data balancing technique depends on the specific dataset and the problem at hand. It is important to evaluate the impact of data balancing on the model's performance and ensure that it does not introduce any biases or distort the original data characteristics. Data balance is an important consideration in machine learning to ensure reliable and unbiased predictions, particularly in scenarios where class imbalance is prevalent.

In [5]:
```
X = strat_train_set.drop("Bankrupt?", axis=1)
y = strat_train_set["Bankrupt?"].copy()

# test set remains the same as the original dataset
X_test = strat_test_set.drop("Bankrupt?", axis=1)
y_test = strat_test_set["Bankrupt?"].copy()
```

In [6]:
```
from sklearn.preprocessing import StandardScaler

# standardization
scaler = StandardScaler()
X = scaler.fit_transform(X)
X_test = scaler.transform(X_test)
```

The second step we took was standardization. Data standardization is a crucial preprocessing step in machine learning. It involves transforming numerical data to a standardized format by scaling the features to have a mean of 0 and a standard deviation of 1. This process ensures that all features are on the same scale and eliminates the influence of different units or ranges in the dataset. By standardizing the data, machine learning algorithms can work more effectively and efficiently, as features with larger values or wider ranges won't dominate the learning process. Standardization helps to remove bias and ensure fair comparisons between features, enabling models to make accurate predictions. It is important to apply standardization to the training data and then apply the same scaling parameters to the test data to maintain consistency. Overall, data standardization is a valuable technique that enhances the performance and reliability of machine learning models.

The ROC-AUC (Receiver Operating Characteristic - Area Under the Curve) score is used in our project as a performance measure to evaluate our model performance, since it is a powerful metric for evaluating the performance of binary classification models.

One of its main advantages is its ability to provide a comprehensive assessment of a model's performance by considering both the true positive rate and the false positive rate. Unlike other metrics that rely on a specific classification threshold, the ROC-AUC score is threshold-independent, making it robust and suitable for comparing different models.

It is particularly useful in handling imbalanced datasets, where one class is much larger than the other. Additionally, the ROC-AUC score allows for easy visualization of the model's performance through the ROC curve.

In summary, the ROC-AUC score is a valuable tool for evaluating binary classification models, aiding in model selection, performance comparison, and providing an overall measure of discriminative power.

Basic Algorithm (logistic regressor and SGDClassifier)

Instantiating and fitting the Logistic Regression model:

In [7]:
```
from sklearn.linear_model import LogisticRegression

# Logistic Regression
Log_reg = LogisticRegression(max_iter = 200)
Log_reg.fit(X, y)
```

Out[7]:
```
▾        LogisticRegression
LogisticRegression(max_iter=200)
```

Instantiating and fitting the SGD classifier:

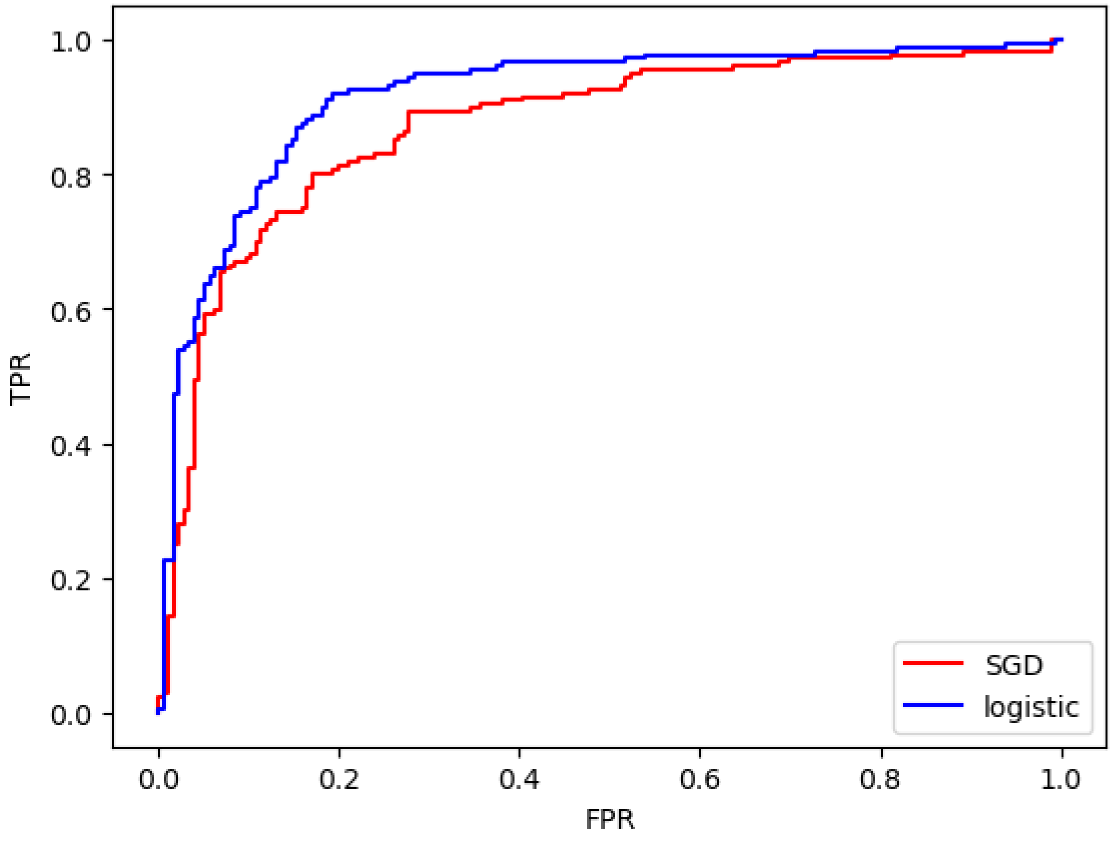
In [8]:
```
from sklearn.linear_model import SGDClassifier

sgd_clf = SGDClassifier(random_state=42)
sgd_clf.fit(X, y)
```

Out[8]:
```
▾ SGDClassifier
SGDClassifier(random_state=42)
```

Generating ROC curves:

These lines generate Receiver Operating Characteristic (ROC) curves for both the SGD classifier and logistic regression. The cross_val_predict function performs cross-validation on the given models to obtain predicted scores (y_scores_sgd and y_scores_log) using the "decision_function" method. The roc_curve function calculates the false positive rate (fpr), true positive rate (tpr), and corresponding thresholds. The resulting ROC curves are plotted using matplotlib.

In [9]:
```
from sklearn.metrics import classification_report
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.model_selection import cross_val_predict

import matplotlib.pyplot as plt

# sgd_clf
y_scores_sgd = cross_val_predict(sgd_clf, X, y, cv=10,
                                 method="decision_function")

fpr, tpr, thresholds = roc_curve(y, y_scores_sgd)
plt.plot(fpr, tpr, "r-", label="SGD")

# logistic regression
y_scores_log = cross_val_predict(Log_reg, X, y, cv=10,
                                 method="decision_function")
fpr, tpr, thresholds = roc_curve(y, y_scores_log)
plt.plot(fpr, tpr, "b-", label="logistic")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



Calculating ROC AUC scores:

In [10]:
```
# roc auc score of logistic regression
roc_auc_score(y, y_scores_log)
```

Out[10]:
```
0.9158703512396694
```

In [11]:
```
# roc auc score of sgd classifier
roc_auc_score(y, y_scores_sgd)
```

Out[11]:
```
0.8706740702479339
```

Generating classification reports:

These lines print the classification reports, including precision, recall, F1-score, and support, for both the logistic regression model and the SGD classifier. The reports compare the predicted labels against the true labels for the training data.

In [12]:
```
# table of evaulation for logistic regression
print(classification_report(y, Log_reg.predict(X)))
```

```
              precision    recall  f1-score   support

           0       0.94      0.92      0.93       176
           1       0.92      0.94      0.93       176

    accuracy                           0.93       352
   macro avg       0.93      0.93      0.93       352
weighted avg       0.93      0.93      0.93       352
```

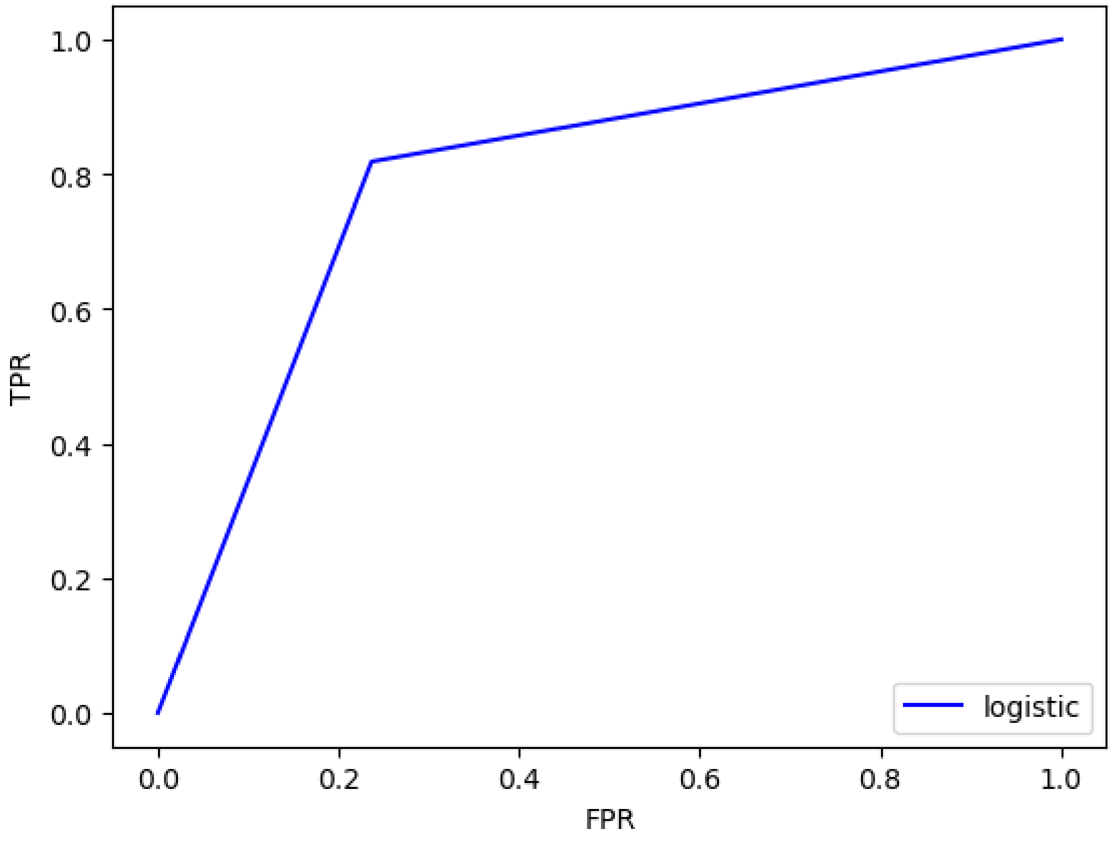
In [13]:
```
# table of evaluation for sgd classifier
print(classification_report(y, sgd_clf.predict(X)))
```

```
              precision    recall  f1-score   support

           0       0.93      0.93      0.93       176
           1       0.93      0.93      0.93       176

    accuracy                           0.93       352
   macro avg       0.93      0.93      0.93       352
weighted avg       0.93      0.93      0.93       352
```

It appears that the first table is indeed better in terms of precision and recall for both classes. It achieves higher precision (0.94 vs. 0.93) and recall (0.94 vs. 0.93) for class 1, and higher recall (0.92 vs. 0.93) for class 0. The F1-scores and support are similar between the two tables.

Therefore, because precision and recall are the primary evaluation criteria, the first table demonstrates better performance.

In [14]:
```
# Use the test set
y_test_pred_log = Log_reg.predict(X_test)
fpr, tpr, thresholds = roc_curve(y_test, y_test_pred_log)
plt.plot(fpr, tpr, "b-", label="logistic")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



In [15]:
```
roc_auc_score(y_test, y_test_pred_log)
```

Out[15]:
```
0.790909090909091
```

In [16]:
```
print(classification_report(y_test, y_test_pred_log))
```

```
              precision    recall  f1-score   support

           0       0.99      0.76      0.86      1320
           1       0.10      0.82      0.18        44

    accuracy                           0.77      1364
   macro avg       0.55      0.79      0.52      1364
weighted avg       0.96      0.77      0.84      1364
```

Overall, this table provides a comprehensive evaluation of the model's performance for each class, as well as overall accuracy. For class 0 (negative class), the precision is 0.99, indicating that out of all instances predicted as class 0, 99% are correct. For class 1 (positive class), the precision is 0.10, meaning that only 10% of instances predicted as class 1 are actually correct. It had relativley high accuracy in general.

RandomForestClassifier

In [17]:
```
from sklearn.ensemble import RandomForestClassifier

# RandomForestClassifier
```

```
rf_cla = RandomForestClassifier(bootstrap=True, n_estimators=100, max_depth=5,
rf_cla.fit(X, y)
```

Out[17]:

```
▾                  RandomForestClassifier
RandomForestClassifier(max_depth=5, random_state=42)
```

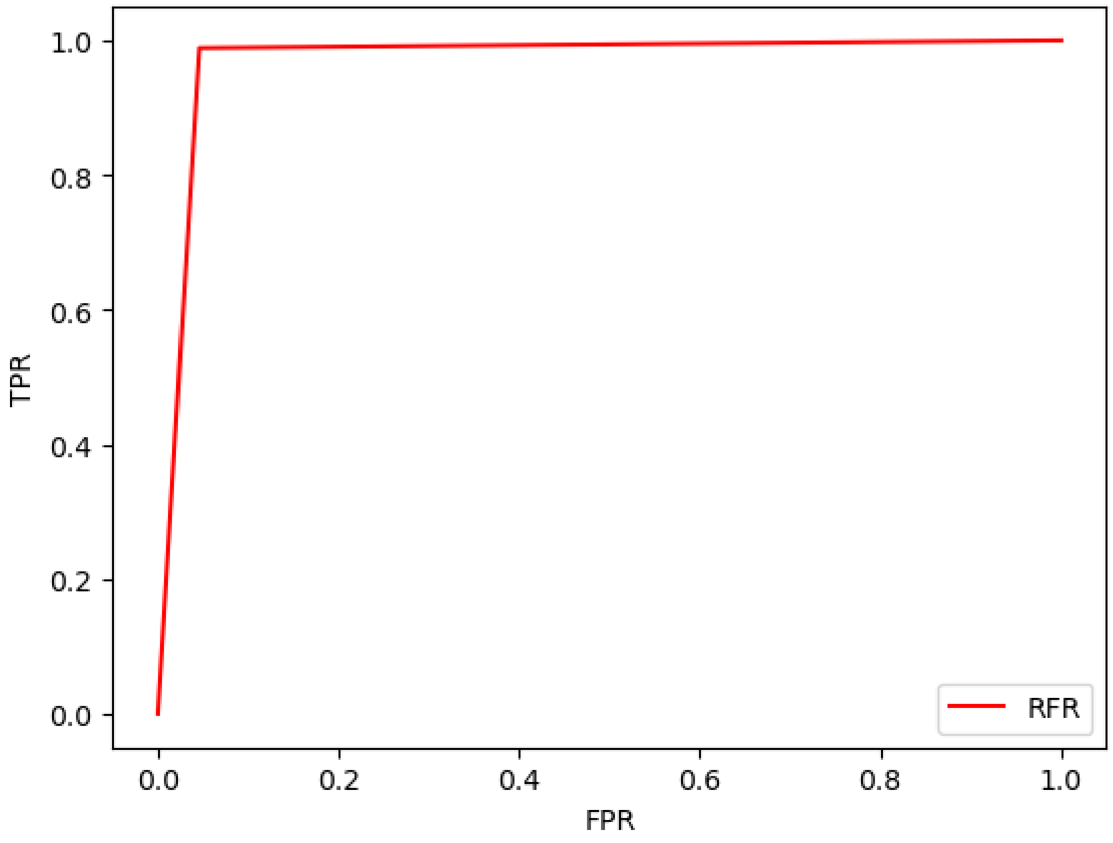
In [18]:

```
X_predict_rf = rf_cla.predict(X)
```

In this section, the roc_curve function from sklearn.metrics is used to compute the False Positive Rate (FPR) and True Positive Rate (TPR) based on the actual labels y and the predicted labels X_predict_rf. These values are then used to plot the ROC curve using Matplotlib. The plt.plot function is used to create the line plot of the ROC curve, with the label "RFR" for Random Forest Regression. The x-axis represents the FPR, and the y-axis represents the TPR. The plt.xlabel, plt.ylabel, and plt.legend functions are used to label the axes and add a legend to the plot. Finally, the plot is displayed using plt.show().

In [19]:

```
# randomforest regressor
fpr, tpr, thresholds = roc_curve(y, X_predict_rf)
plt.plot(fpr, tpr, "r-", label="RFR")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



In [20]:

```
roc_auc_score(y, X_predict_rf)
```

```
Out[20]: 0.9715909090909093
```

```
In [21]: from sklearn.model_selection import GridSearchCV
         from sklearn.model_selection import RandomizedSearchCV
         from scipy.stats import randint

         # Choose the optimal values of hyperparameters
         rfr = RandomForestClassifier(random_state=42)

         param_grid = [
             {'n_estimators': [10, 50, 100, 200],
              'max_features': [2, 3, 4, 5, 6, 7, 8, 9, 10]}
         ]

         grid_search = GridSearchCV(rfr, param_grid, cv=10,
                                    scoring='neg_root_mean_squared_error',
                                   n_jobs=-1)

         grid_search.fit(X, y)
         # find out the best parameters
         grid_search.best_params_
```

```
Out[21]: {'max_features': 9, 'n_estimators': 50}
```

In this section, the necessary modules and classes for hyperparameter tuning are imported. The GridSearchCV class is used to perform a systematic grid search to find the optimal values of hyperparameters for the RandomForestClassifier. The param_grid variable specifies the different values to be explored for the number of estimators (n_estimators) and the maximum number of features (max_features). The GridSearchCV algorithm is configured with a cross-validation of 10 folds (cv=10), using negative root mean squared error as the scoring metric (scoring='neg_root_mean_squared_error'). The n_jobs=-1 parameter enables parallel computation. The grid search is then performed

```
In [22]: param_distribs = {'n_estimators': randint(low=10, high=200),
                           'max_features': randint(low=2, high=20)}
         rnd_search = RandomizedSearchCV(
             rfr, param_distributions=param_distribs, n_iter=10, cv=10,
             scoring='neg_root_mean_squared_error', random_state=42, n_jobs=-1)
         rnd_search.fit(X, y)
         # find out the best parameters
         rnd_search.best_params_
```

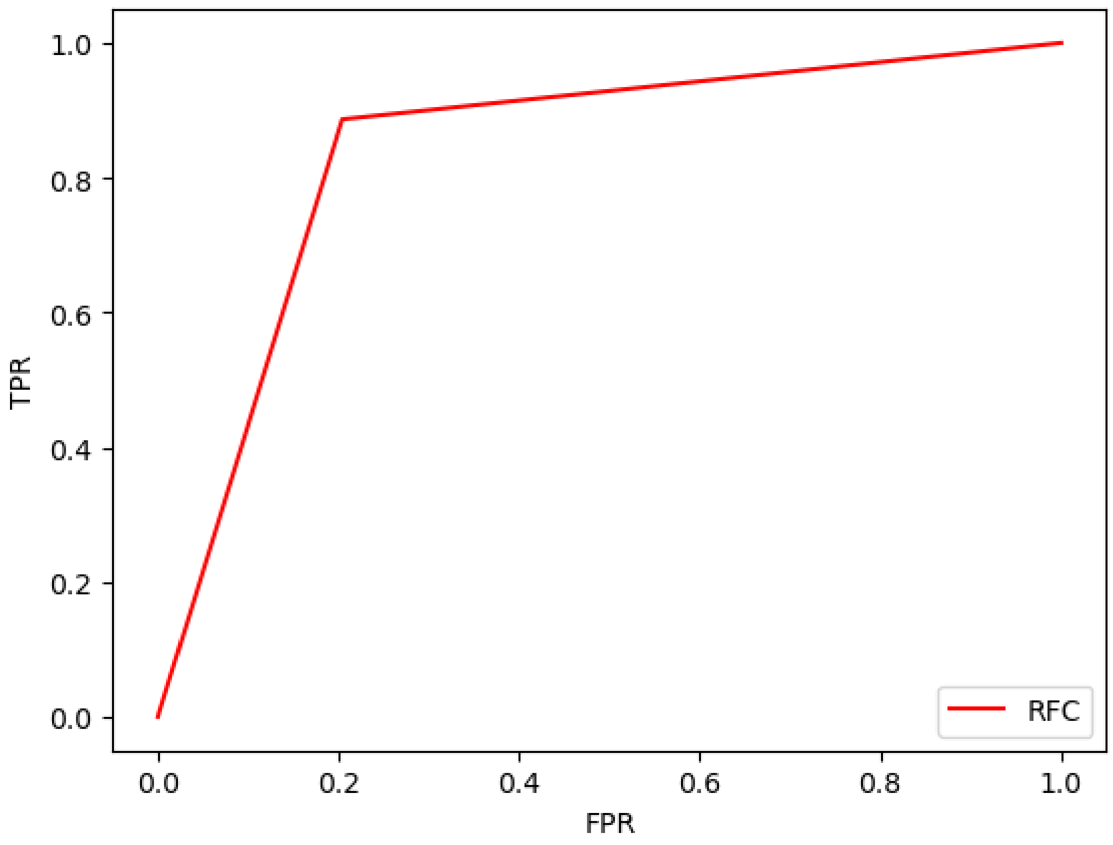
```
Out[22]: {'max_features': 9, 'n_estimators': 198}
```

```
In [23]: # check the rmse of the final model
         final_model = grid_search.best_estimator_
         X_predict_rf_best = final_model.predict(X)
```

```
In [24]: # final model evaluation (using grid search)
         final_prediction = final_model.predict(X_test)
         final_prediction
```

```
Out[24]: array([1, 0, 0, ..., 0, 0, 1])
```

In [25]:
```
# figure of final model of RFC
fpr, tpr, thresholds = roc_curve(y_test, final_prediction)
plt.plot(fpr, tpr, "r-", label="RFC")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



In [26]:
```
roc_auc_score(y_test, final_prediction)
```

Out[26]:
```
0.8412878787878787
```

In this case, the ROC AUC score of 0.8412 suggests that the final model of the Random Forest Classifier performs reasonably well in distinguishing between the positive and negative classes. An ROC AUC score above 0.5 indicates that the model has some degree of predictive ability. Overall, based on the ROC AUC score of 0.8412, we can conclude that the final model of the Random Forest Classifier shows promise in terms of its predictive performance.

SVM Classifier

The next model we chose was the SVM Classifier.

The main objective of an SVM classifier is to find an optimal hyperplane that separates data points belonging to different classes. The hyperplane is selected in a way that maximizes the margin, which is the distance between the hyperplane and the nearest data points of each class. This approach aims to achieve a clear separation between classes and improve the classifier's generalization ability.

In the first case, we import the necessary modules and create an instance of the `LinearSVC` class, which represents a linear Support Vector Classifier (SVM). The max_iter parameter sets the maximum number of iterations for the solver, and random_state ensures reproducibility in the results.

Then performs a randomized search for hyperparameter tuning of the SVM classifier. The param_grid dictionary defines the range of values to search for the C parameter. RandomizedSearchCV is used to randomly sample combinations of hyperparameters and evaluate their performance using cross-validation. The best parameters are determined using best*params*. The roc_auc_score computes the ROC AUC score for the predicted labels compared to the true labels.

In [27]:
```
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC

# SVM classification and regression
svm_clf = LinearSVC(max_iter = 50000, random_state=42)

param_grid = {
    "C": [10**(-0), 10**(-1), 10**(-2), 10**(-3), 10**(-4), 10**(-5), 10**(-6),
}

rnd_search_svm = RandomizedSearchCV(svm_clf, param_grid, cv=3,
                           scoring='neg_mean_squared_error',
                           n_jobs=-1)

rnd_search_svm.fit(X,y)
```

Out[27]:
```
▸ RandomizedSearchCV
▸ estimator: LinearSVC
      ▸ LinearSVC
```

In [28]:
```
rnd_search_svm.best_params_
```

Out[28]:
```
{'C': 0.001}
```

The best parameter we found was 0.001.

In [29]:
```
roc_auc_score(y, rnd_search_svm.predict(X))
```

Out[29]:
```
0.9147727272727273
```

Then, we got the ROC Score of this model, which was about 0.9148.

In the second part, we perform a randomized search for hyperparameter tuning of the SVM classifier with a Gaussian RBF kernel. The param dictionary defines the ranges of values to search for the C and gamma parameters. Similar to the previous section, RandomizedSearchCV is used to sample and evaluate different hyperparameter combinations, and the best parameters are obtained. The roc_auc_score is computed for the predicted labels compared to the true labels.

In [30]:
```python
from scipy.stats import loguniform
from scipy.stats import uniform
from sklearn.svm import SVC

# fitting a SVM with a Gaussian RBF kernel
kersvc = SVC(max_iter=50000, kernel='rbf')

param = {
    "C": uniform(1, 10),
    "gamma": loguniform(1e-04, 1e-01)
}

rnd_search_kersvc = RandomizedSearchCV(kersvc, param,
                                       cv=3,
                                       scoring = "neg_mean_squared_error", n_jc
                                       random_state=42)

rnd_search_kersvc.fit(X,y)
```

Out[30]:
```
▸ RandomizedSearchCV
  ▸ estimator: SVC
      ▸ SVC
```

In [31]:
```python
rnd_search_kersvc.best_params_
```

Out[31]:
```
{'C': 4.042422429595377, 'gamma': 0.0037520558551242 8}
```

The best hyperparameter we gots were listed above.

In [32]:
```python
roc_auc_score(y, rnd_search_kersvc.predict(X))
```

Out[32]:
```
0.9289772727272728
```

The ROC score in this section was about 0.9290.

In the third part, we perform a randomized search for hyperparameter tuning of the SVM classifier with a sigmoid kernel. The param dictionary, similar to the previous section, defines the ranges of values to search for the C and gamma parameters. The search is conducted using RandomizedSearchCV, and the best parameters are determined. The roc_auc_score is computed for the predicted labels compared to the true labels.

In [33]:
```
# fitting a SVM with a sigmoid kernel
sigmoid_rbf = SVC(max_iter = 50000, kernel = 'sigmoid')
rnd_search_sigmoid = RandomizedSearchCV(sigmoid_rbf, param,
                                        scoring = "neg_mean_squared_error", cv=3, ra
rnd_search_sigmoid.fit(X, y)
```

Out[33]:
```
▸ RandomizedSearchCV
  ▸ estimator: SVC
      ▸ SVC
```

In [34]:
```
rnd_search_sigmoid.best_params_
```

Out[34]:
```
{'C': 2.818249672071006, 'gamma': 0.00035498788321965016}
```

The best hyperparameters were listed above.

In [35]:
```
roc_auc_score(y, rnd_search_sigmoid.predict(X))
```

Out[35]:
```
0.90625
```

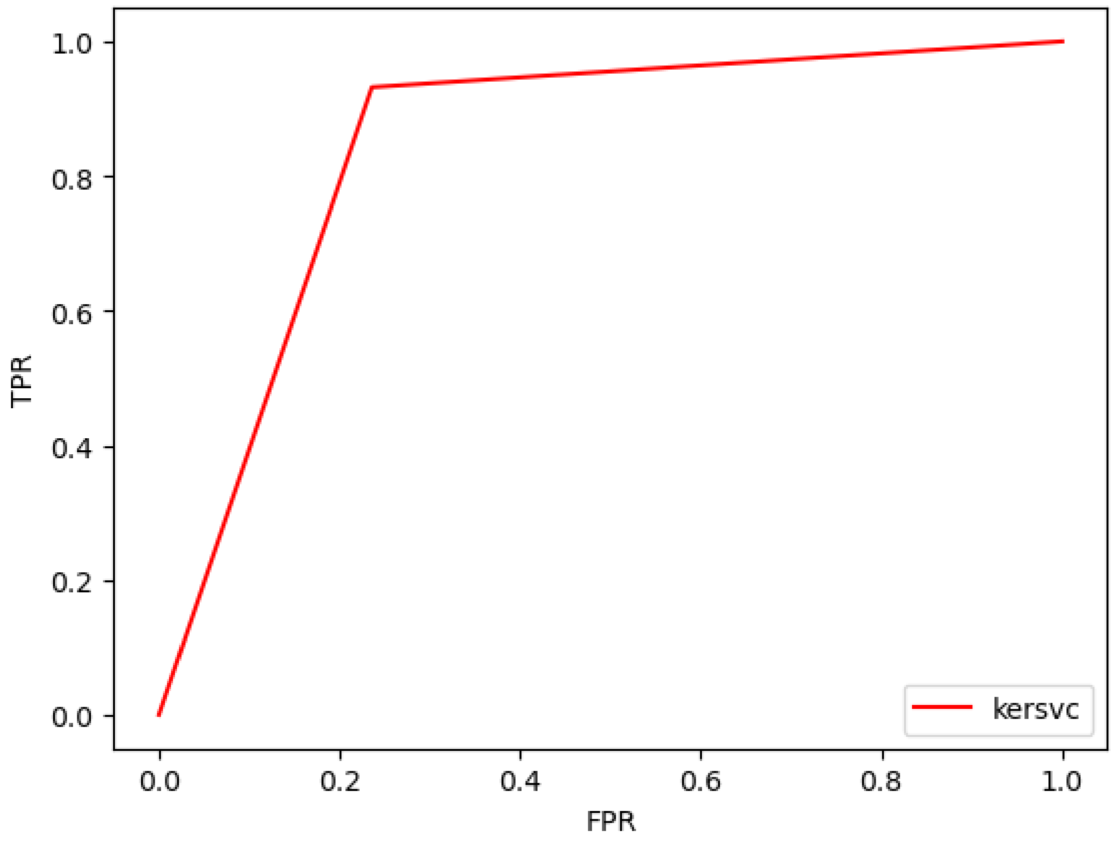
The ROC Score in this section was 0.90625.

Based on the three ROC Scores, we chose the model of `rnd_search_kersvc`, which had the largest ROC score.

In [36]:
```
# Choose the best model: rnd_search_kersvc
y_predict_kersvc = rnd_search_kersvc.best_estimator_.predict(X_test)
```

The `roc_curve` function takes two arguments - the true labels (y_test) and the predicted probabilities or scores of the positive class (y_predict_kersvc) from a classifier. It calculates the False Positive Rate (fpr), True Positive Rate (tpr), and corresponding thresholds. These values are used to plot the ROC curve.

In [37]:
```
fpr, tpr, thresholds = roc_curve(y_test, y_predict_kersvc)
plt.plot(fpr, tpr, "r-", label="kersvc")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



```
In [38]: roc_auc_score(y_test, y_predict_kersvc)
```

```
Out[38]: 0.8481060606060606
```

On the test set, the overall ROC Score of the best model was about 0.8481. This suggests that this model was a quite decent model.

PCA Transformation

First, we trained and evaluated the SVM classifier. The SVC is instantiated with the RBF kernel. The RBF kernel is a non-linear kernel that allows SVMs to capture non-linear relationships between the input features and the target variable.

```
In [39]: # Using SVM Classifier
svc = SVC(kernel = "rbf")
%time svc.fit(X, y)
```

```
CPU times: user 14.5 ms, sys: 1.29 ms, total: 15.8 ms
Wall time: 5.08 ms
```

```
Out[39]: ▾ SVC
SVC()
```

The trained SVC classifier is used to predict the labels for the test data X_test, and the predictions are stored in y_test_prediction_svc. The roc_auc_score() function calculates the Area Under the Curve (AUC) score using the true labels y_test and the predicted labels.

```
In [40]: y_test_prediction_svc = svc.predict(X_test)
         roc_auc_score(y_test, y_test_prediction_svc)

Out[40]: 0.8401515151515152
```

Here we can see that the accuracy score of the SVC classifier is 0.8401515151515152.

PCA is applied to the training data X to reduce its dimensionality. The PCA() function is called to instantiate a PCA object, and then the set_params() method is used to set the desired number of principal components, in this case, 60% of the original features. The fit_transform() method is called to perform PCA on X and obtain the transformed data X_pca. Finally, the SVC classifier is trained on the transformed data.

```
In [41]: from sklearn.decomposition import PCA

         pca = PCA()
         X_pca = pca.set_params(n_components = 0.6).fit_transform(X)
         %time svc.fit(X_pca, y)

CPU times: user 36.3 ms, sys: 431 µs, total: 36.7 ms
Wall time: 8.53 ms

Out[41]: ▾ SVC
         SVC()
```

The test data X_test is transformed using the same PCA object to obtain X_test_pca. The trained SVC classifier is then used to predict the labels for the transformed test data, and the predictions are stored in y_test_pca_prediction_svc. The roc_auc_score() function is used again to evaluate the performance of the classifier with PCA.

```
In [42]: X_test_pca = pca.transform(X_test)
         y_test_pca_prediction_svc = svc.predict(X_test_pca)
         roc_auc_score(y_test, y_test_pca_prediction_svc)

Out[42]: 0.8621212121212122
```

The accuracy score here is 0.8621212121212122, which is higher than the model without using PCA.

Then a Random Forest Classifier is instantiated with a random state of 42 and using all available processors (n_jobs=-1). The %time magic command is used to measure the execution time. The classifier is then trained on the original training data X and labels y.

```
In [43]: from sklearn.ensemble import RandomForestClassifier

         # Using random forest classifier
         rnd_forest_blender_pca = RandomForestClassifier(random_state=42, n_jobs = -1)
         %time rnd_forest_blender_pca.fit(X, y)

CPU times: user 876 ms, sys: 506 ms, total: 1.38 s
Wall time: 234 ms
```

Out[43]:
```
▾ RandomForestClassifier
RandomForestClassifier(n_jobs=-1, random_state=42)
```

The trained Random Forest classifier is used to predict the labels for the test data X_test, and the predictions are stored in y_test_prediction_rnd. The roc_auc_score() function is then used to calculate the AUC score using the true labels y_test and the predicted labels. The resulting AUC score is printed.

In [44]:
```
y_test_prediction_rnd = rnd_forest_blender_pca.predict(X_test)
roc_auc_score(y_test, y_test_prediction_rnd)
```

Out[44]:
```
0.843560606060606
```

The PCA object pca is reused, and the set_params() method is called to set the number of principal components to retain (n_components=0.6). Then, the fit_transform() method is used to apply PCA on the original training data X, resulting in the transformed data X_pca_rnd.

The Random Forest classifier rnd_forest_blender_pca is trained on the transformed training data X_pca_rnd and corresponding labels y. The %time magic command is used to measure the execution time.

In [45]:
```
X_pca_rnd = pca.set_params(n_components = 0.6).fit_transform(X)
%time rnd_forest_blender_pca.fit(X_pca_rnd, y)
```

```
CPU times: user 764 ms, sys: 345 ms, total: 1.11 s
Wall time: 203 ms
```

Out[45]:
```
▾ RandomForestClassifier
RandomForestClassifier(n_jobs=-1, random_state=42)
```

The test data X_test is transformed using the previously created PCA object pca to obtain X_test_pca_rnd. This ensures that the test data is transformed in the same way as the training data. The trained Random Forest classifier is used to predict the labels for the transformed test data X_test_pca_rnd, and the predictions are stored in y_test_pca_prediction_rnd.

The resulting AUC score is printed. The score here is lower than the previous ones.

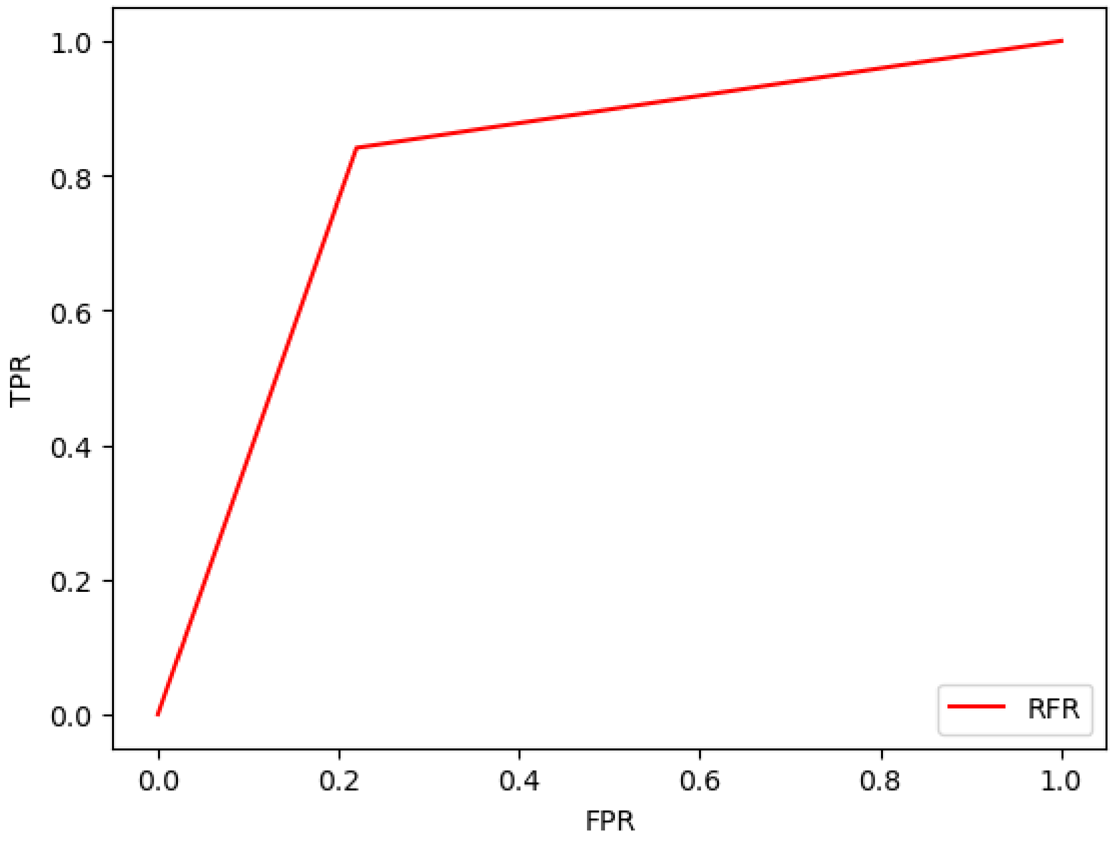
In [46]:
```
X_test_pca_rnd = pca.transform(X_test)
y_test_pca_prediction_rnd = rnd_forest_blender_pca.predict(X_test_pca_rnd)
roc_auc_score(y_test, y_test_pca_prediction_rnd)
```

Out[46]:
```
0.8106060606060606
```

The code calculates the false positive rate (FPR), true positive rate (TPR), and thresholds using the roc_curve() function based on the true labels y_test and the predicted labels y_test_pca_prediction_rnd. Then, a plot of the ROC curve is created using plt.plot(), with

FPR on the x-axis and TPR on the y-axis. The label "RFR" is added to the plot as the legend for the Random Forest classifier. Finally, the plot is displayed using plt.show().

In [47]:
```
fpr, tpr, thresholds = roc_curve(y_test, y_test_pca_prediction_rnd)
plt.plot(fpr, tpr, "r-", label="RFR")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



Good points of using PCA for prediction include dimensionality reduction, noise reduction, and data preprocessing, which can improve computational efficiency and enhance predictive performance.

However, PCA reduces interpretability as principal components may not directly correspond to the original variables. Information loss can occur during dimensionality reduction, and PCA assumes linear relationships, which may not capture nonlinear interactions well.

k-Means Clustering:

After using all above supervised algorithms, it would be interesting to try an unsupervised algorithms to see if it could cluster the data into 2 groups.

Choose the best K number with three methods:

First, we perform an analysis to choose the best number of clusters, K, using three different methods: evaluating inertia, silhouette score, and using GridSearchCV with f1_weighted scoring.

The training set X and corresponding labels y are split into two parts: X_train and X_valid, y_train and y_valid. The split is done using an 80-20 ratio.

In [48]:
```
# Split training set:
n = len(X)
train_idx = int(n * 0.8)
X_train = X[:train_idx]
X_valid = X[train_idx:]

y_train = y[:train_idx]
y_valid = y[train_idx:]
```

We use the KMeans algorithm from scikit-learn to fit K-means models with different values of K ranging from 1 to 19. The inertia of each model, which represents the sum of squared distances of samples to their closest cluster center, is stored in the inertias list.

In [49]:
```
from sklearn.cluster import KMeans

kmeans_per_k = [KMeans(n_clusters=k, random_state=42,n_init = 10).fit(X_train)
inertias = [model.inertia_ for model in kmeans_per_k]
inertias
```

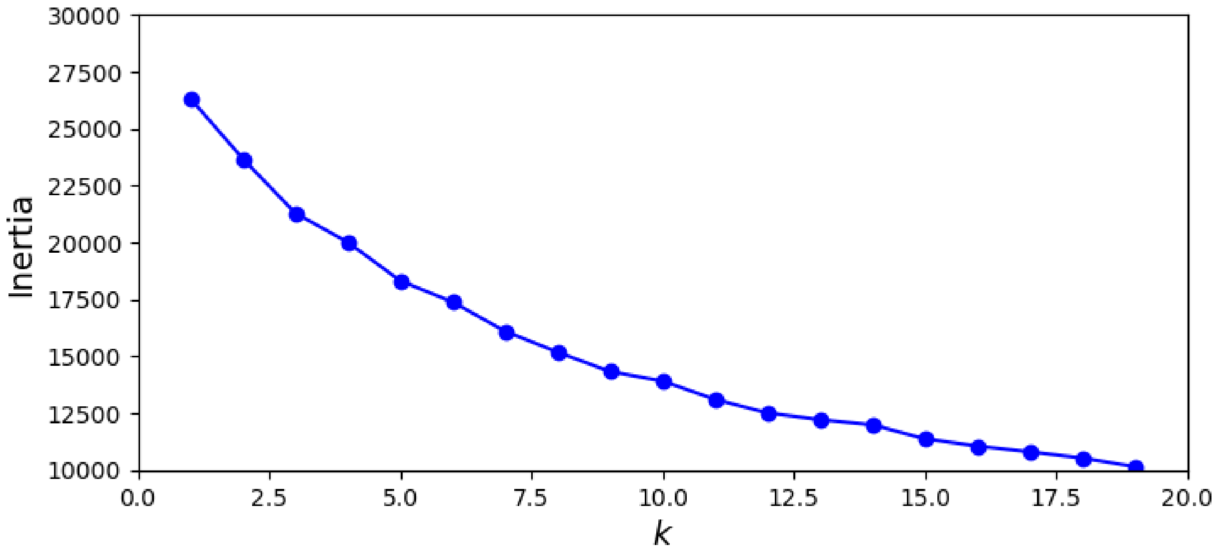
Out[49]:
```
[26317.481378382356,
 23669.050263281286,
 21285.431018905205,
 20007.82919442338,
 18312.617488915013,
 17378.143447299663,
 16093.188297982955,
 15178.984865506303,
 14324.029439268157,
 13911.528213073303,
 13105.242445015607,
 12509.866642800873,
 12215.92748492392,
 11986.314050456458,
 11372.94784132431,
 11055.419023368759,
 10799.573510430644,
 10528.876078962574,
 10150.837585047084]
```

In [50]:
```
from sklearn.cluster import KMeans

kmeans_per_k = [KMeans(n_clusters=k, random_state=42,n_init = 10).fit(X_train)
inertias = [model.inertia_ for model in kmeans_per_k]
```

A plot is created to visualize the relationship between the number of clusters (K) and the inertia. The x-axis represents K, and the y-axis represents the inertia. This plot helps identify the "elbow" point, which indicates a good value of K that balances compactness of clusters with a small number of clusters.

In this code, the elbow point is visually determined to be at K = 3 based on the inertia vs. K plot.

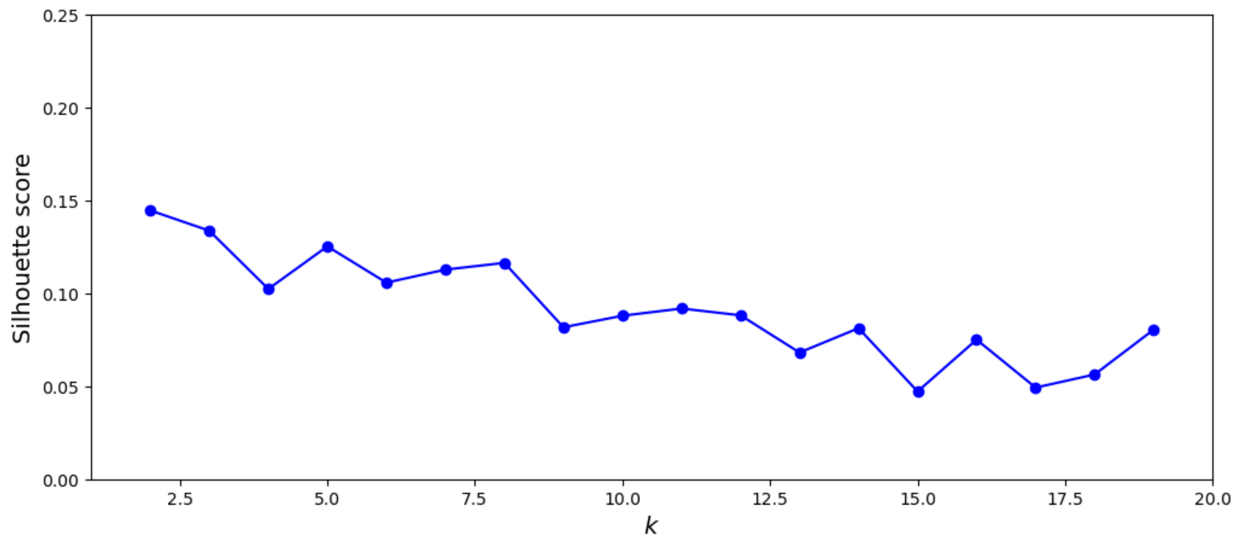
In [51]:
```
# inertias vs k Plot:
import matplotlib.pyplot as plt
plt.figure(figsize=(8, 3.5))
plt.plot(list(range(1,20,1)), inertias, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Inertia", fontsize=14)
plt.axis([0, 20, 10000, 30000])
plt.show()

best_model_1 = KMeans(n_clusters=2, random_state=42,n_init = 10)
# From this plot, the elbow is K = 3.
```



The silhouette score measures the compactness and separation of clusters. Now we calculate the silhouette score for each K-means model (excluding K=1) and stores the scores in the silhouette_scores list.

Then a plot is created to visualize the silhouette scores for different values of K. The x-axis represents K, and the y-axis represents the silhouette score. This plot helps evaluate the quality of clustering for different values of K.

In [52]:
```
# Get silhoutte score plot
from sklearn.metrics import silhouette_score
silhouette_scores = [silhouette_score(X_train, model.labels_) for model in kmea
plt.figure(figsize=(12, 5))
plt.plot(range(2, 20), silhouette_scores, "bo-")
plt.xlabel("$k$", fontsize=14)
plt.ylabel("Silhouette score", fontsize=14)
plt.axis([1, 20, 0, 0.25])
plt.show()
# The plot here gives that K = 1
```



We use GridSearchCV to perform an exhaustive search over a specified parameter grid (param_grid) for the KMeans algorithm. The scoring metric used is 'f1_weighted'. The best K is determined based on the highest f1_weighted score obtained.

In [53]:
```
# Now use GridSearch with a scoring of f1_weighted for K:
from sklearn.metrics import roc_auc_score
from sklearn.model_selection import cross_val_score, GridSearchCV

# Create an instance of the KMeans class
kmeans = KMeans(random_state=42, n_init = 10)

# Define the hyperparameters grid for grid search
param_grid = {
    'n_clusters': range(1, 20)
}

grid_search = GridSearchCV(estimator=kmeans, param_grid=param_grid, scoring='f1
grid_search.fit(X_train,y_train)

# Print the best parameters and best AUC-ROC score
print("Best parameters:", grid_search.best_params_)
print("Best AUC-ROC score:", grid_search.best_score_)
```

```
Best parameters: {'n_clusters': 3}
Best AUC-ROC score: 0.7629162923675802
```

Here the best parameter for K is 3. This doesn't makes sense because this is a binary classification problem, we should be getting the result of K = 2 in this case The discrepancy may be due to the use of f1_weighted scoring, which considers class imbalance.

Performance Evaluation :

Now we perform performance evaluation for different approaches using K-Means clustering as a feature reduction method and a Random Forest Classifier for classification.

In [54]:
```
best_model_1 = KMeans(n_clusters=2, random_state=42,n_init = 10)
```

We start by creating a KMeans model with 2 clusters (n_clusters=2) and fits it on the validation set, then predict the cluster labels for the validation set and calculates the ROC AUC score between the predicted labels and the true labels (y_valid). The resulting ROC AUC score is 0.257, indicating poor performance.

In [55]:
```
# Assess K = 2
from sklearn.metrics import roc_auc_score
from sklearn.metrics import f1_score

best_model_1.fit(X_valid)
y_pred = best_model_1.predict(X_valid)

roc_auc_score(y_pred, y_valid)
```

Out[55]:
```
0.2571428571428571
```

After that, we create a KMeans model with 3 clusters (n_clusters=3) and use it to transform the training and validation sets (X_train and X_valid). The transformed features are then used as inputs to a Random Forest Classifier, which is trained on the transformed training set and used to predict labels for the transformed validation set. The ROC AUC score between the predicted labels and the true labels is calculated, resulting in a score of 0.790. This performance is better than using K-Means with K=2, but still not as good as using Random Forest Classifier directly.

In [56]:
```
# Use K = 3 for reduction tool:

best_model_1 = KMeans(n_clusters=3, random_state=42,n_init = 10)

X_train_reduction = best_model_1.fit_transform(X_train)
X_valid_reduction = best_model_1.transform(X_valid)

# rf classifier:
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier(n_estimators = 150, random_state = 42)
rf.fit(X_train_reduction, y_train)

y_pred = rf.predict(X_valid_reduction)

roc_auc_score(y_pred, y_valid)
```

Out[56]:
```
0.7901430842607313
```

Now let's compare with directly using Random Forest Classifier. The code trains a Random Forest Classifier directly on the original training set (X_train) and predicts labels for the original validation set (X_valid). The ROC AUC score between the predicted labels and the true labels is calculated, resulting in a score of 0.862. This performance is also better than not using the soft clustering.

In [57]:
```
# Compare with directly using Random Forest Classifier:

rf.fit(X_train,y_train)
```

```
y_pred = rf.predict(X_valid)

roc_auc_score(y_pred, y_valid)
```

Out[57]:
```
0.8624401913875599
```

Afterwards we perform a search for the best value of K by appending the K-Means reduced features (X_train_red and X_valid_red) to the original features (X_train and X_valid). The code uses a loop to iterate over different values of K and trains a Random Forest Classifier on the extended feature set. The ROC AUC scores for each value of reduction K are printed. The best K value in this case is found to be 16, with a ROC AUC score of 0.879. And it achieves a higher roc auc score than directly using Random Forest Classfier.

In [58]:
```
# Search for the best K with appending the K-Means reducted features to the ori
valid_score_extended_k = []
k_range = range(1 , 20)
for idx, model in enumerate(kmeans_per_k):

    # Create the train reduction features / Append to the original features
    X_train_red = model.transform(X_train)
    X_train_ext = np.c_[X_train,X_train_red]
    X_valid_red = model.transform(X_valid)
    X_valid_ext = np.c_[X_valid,X_valid_red]

    # Create the classifier
    clf  = RandomForestClassifier(n_estimators = 150, random_state = 42)
    clf.fit(X_train_ext, y_train)
    y_pred_ext = clf.predict(X_valid_ext)
    cur_score = roc_auc_score(y_pred_ext, y_valid)

    valid_score_extended_k.append(cur_score)
    print(k_range[idx], cur_score)

# The best number of reduction K for random forest classifier would be K = 16 i
# And it achieves a higher roc auc score than directly using Random Forest Clas
```

```
1 0.8624401913875599
2 0.8505608974358974
3 0.8748012718600953
4 0.8624401913875599
5 0.8624401913875599
6 0.8624401913875599
7 0.8624401913875599
8 0.8624401913875599
9 0.8624401913875599
10 0.8790064102564102
11 0.8624401913875599
12 0.8624401913875599
13 0.8624401913875599
14 0.8790064102564102
15 0.8624401913875599
16 0.8790064102564102
17 0.8624401913875599
18 0.8790064102564102
19 0.8624401913875599
```

A KMeans model with 16 clusters (n_clusters=16) is then created and used to transform the training and test sets (X_train and X_test). The transformed features are appended to the original features. A Random Forest Classifier is trained on the extended training set and used to predict labels for the extended test set. The ROC AUC score between the predicted labels and the true labels is calculated, resulting in a score of 0.557. This indicates poor performance.

In [59]:
```
best_model_1 = KMeans(n_clusters=16, random_state=42,n_init = 10)

X_train_reduction = best_model_1.fit_transform(X_train)
X_train_ext = np.c_[X_train, X_train_reduction]


X_test_reduction = best_model_1.transform(X_test)
X_test_ext = np.c_[X_test, X_test_reduction]
rf.fit(X_train_ext, y_train)

y_pred = rf.predict(X_test_ext)

roc_auc_score(y_pred, y_test)
```

Out[59]:
```
0.5573997613399635
```

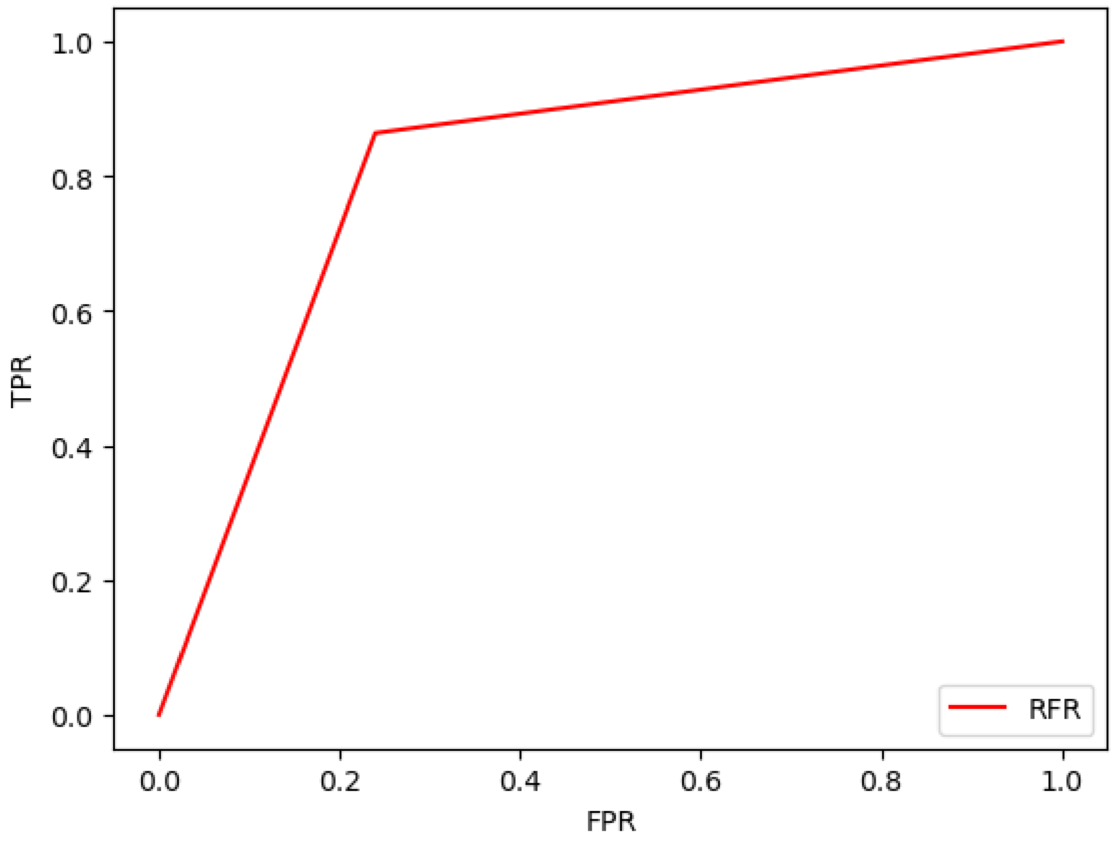
K-Means reduction with K=16 is applied on test set without appending features. The code only transforms the test set using the KMeans model with 16 clusters (n_clusters=16) and then uses the transformed features as inputs to a Random Forest Classifier. The classifier predicts labels for the transformed test set, and the ROC AUC score between the predicted labels and the true labels is calculated. The score is 0.551, indicating poor performance.

In [60]:
```
X_test_reduction = best_model_1.transform(X_test)

rf.fit(X_train_reduction, y_train)

y_pred = rf.predict(X_test_reduction)

roc_auc_score(y_pred, y_test)
```

Out[60]:
```
0.5507020193544778
```

Finally we plot the ROC curve using the false positive rate (FPR) and true positive rate (TPR) calculated from the predicted labels and the true labels. The plot shows the ROC curve for the Random Forest Classifier using K-Means reduction with K=16.

In [61]:
```
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
plt.plot(fpr, tpr, "r-", label="RFR")

plt.xlabel("FPR")
plt.ylabel("TPR")
plt.legend(loc="lower right")
plt.show()
```



However, since our response variables only have 2 values (1 as alive, 0 as bankrupted), the K-Means clustering with selected K values being 3 may not be the best approach. K = 3 means that for most of the data points, they are separated with 3 groups, when infact they should be in 2. The evaluation shows that directly using a Random Forest Classifier yields better results than using K-Means with K=2 or K=3. Additionally, appending the K-Means reduced features to the original features and training a Random Forest Classifier on the extended feature set also improves performance, with the best K value being 3. However, when applying K-Means reduction to the test set, both approaches of appending reduced features and using them directly lead to poor performance. Overall, using K-Means clustering as a feature reduction method does not provide significant benefits for this specific binary classification problem.

Neural Network Models:

In this section, we explore the implementation and evaluation of two types of perceptron models: a one-layer perceptron and a multi-layer perceptron (MLP).

In [62]:
```
import tensorflow as tf
import tensorflow.keras as keras
```

One Layer Perceptron:

In the first part, a One Layer Perceptron is trained using the Perceptron class from sklearn.linear_model. It is fitted on the training data (X_train and y_train), and the predictions are made on the test data (X_test). The performance of the model is evaluated using the ROC AUC score, which measures the model's ability to distinguish between positive and negative classes. The resulting ROC AUC score is 0.775.

In [63]:
```
from sklearn.linear_model import Perceptron
from sklearn.metrics import roc_auc_score

per_clf = Perceptron(random_state = 42)
per_clf.fit(X_train, y_train)

# Prediction Accessment:
y_pred = per_clf.predict(X_test)
roc_auc_score(y_test, y_pred)
```

Out[63]:
```
0.775
```

Multi - Layer perceptron:

In the second part, a Multi-Layer Perceptron (MLP) is built using the Keras library. The architecture of the MLP consists of three dense layers with different activation functions. The model is compiled with a binary cross-entropy loss function, stochastic gradient descent optimizer, and the AUC metric for evaluation. The learning rate is adjusted using an exponential decay schedule. The MLP is trained on the training data (X_train and y_train) for 50 epochs, with validation data (X_valid and y_valid) used for monitoring the model's performance during training. The model achieves a validation AUC score of 0.8951.

In [64]:
```
def reset_session(seed=42):
    tf.random.set_seed(seed)
    np.random.seed(seed)
    tf.keras.backend.clear_session()

def exponential_decay(lr0, s):
    return lambda epoch: lr0 * 0.1**(epoch / s)
```

In [65]:
```
reset_session()

mlp = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
    tf.keras.layers.Dense(1, activation="sigmoid")
])

mlp.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

In [66]:
```
reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
mlp.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_valid), ca
Epoch 1/50
9/9 [==============================] - 0s 14ms/step - loss: 0.8254 - auc: 0.44
76 - val_loss: 0.7145 - val_auc: 0.5835 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 3ms/step - loss: 0.7282 - auc: 0.601
7 - val_loss: 0.6675 - val_auc: 0.6741 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6651 - auc: 0.715
2 - val_loss: 0.6362 - val_auc: 0.7381 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 4ms/step - loss: 0.6211 - auc: 0.779
8 - val_loss: 0.6138 - val_auc: 0.7786 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5901 - auc: 0.816
8 - val_loss: 0.5961 - val_auc: 0.8029 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5655 - auc: 0.845
1 - val_loss: 0.5824 - val_auc: 0.8231 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5467 - auc: 0.859
2 - val_loss: 0.5713 - val_auc: 0.8355 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5319 - auc: 0.870
3 - val_loss: 0.5621 - val_auc: 0.8486 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5200 - auc: 0.877
8 - val_loss: 0.5547 - val_auc: 0.8557 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5099 - auc: 0.884
5 - val_loss: 0.5484 - val_auc: 0.8641 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5014 - auc: 0.890
9 - val_loss: 0.5430 - val_auc: 0.8661 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4946 - auc: 0.894
6 - val_loss: 0.5385 - val_auc: 0.8704 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4886 - auc: 0.898
7 - val_loss: 0.5346 - val_auc: 0.8716 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4835 - auc: 0.901
7 - val_loss: 0.5313 - val_auc: 0.8776 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4793 - auc: 0.904
5 - val_loss: 0.5284 - val_auc: 0.8792 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4755 - auc: 0.907
9 - val_loss: 0.5259 - val_auc: 0.8824 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4724 - auc: 0.909
9 - val_loss: 0.5237 - val_auc: 0.8835 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4695 - auc: 0.912
0 - val_loss: 0.5217 - val_auc: 0.8855 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 8ms/step - loss: 0.4671 - auc: 0.913
7 - val_loss: 0.5200 - val_auc: 0.8863 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 6ms/step - loss: 0.4650 - auc: 0.915
4 - val_loss: 0.5186 - val_auc: 0.8867 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4632 - auc: 0.916
5 - val_loss: 0.5173 - val_auc: 0.8879 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4616 - auc: 0.917
6 - val_loss: 0.5161 - val_auc: 0.8903 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4601 - auc: 0.918
4 - val_loss: 0.5151 - val_auc: 0.8907 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4588 - auc: 0.919
5 - val_loss: 0.5142 - val_auc: 0.8907 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4577 - auc: 0.920
0 - val_loss: 0.5134 - val_auc: 0.8899 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4568 - auc: 0.920
4 - val_loss: 0.5127 - val_auc: 0.8911 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4559 - auc: 0.920
3 - val_loss: 0.5121 - val_auc: 0.8915 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 2ms/step - loss: 0.4551 - auc: 0.920
7 - val_loss: 0.5116 - val_auc: 0.8919 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4545 - auc: 0.921
1 - val_loss: 0.5111 - val_auc: 0.8923 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4539 - auc: 0.921
4 - val_loss: 0.5107 - val_auc: 0.8919 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4533 - auc: 0.921
5 - val_loss: 0.5103 - val_auc: 0.8919 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4529 - auc: 0.921
9 - val_loss: 0.5100 - val_auc: 0.8923 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4525 - auc: 0.922
2 - val_loss: 0.5097 - val_auc: 0.8939 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4521 - auc: 0.922
6 - val_loss: 0.5094 - val_auc: 0.8935 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4518 - auc: 0.922
7 - val_loss: 0.5092 - val_auc: 0.8935 - lr: 1.9953e-04
Epoch 36/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4515 - auc: 0.922
9 - val_loss: 0.5090 - val_auc: 0.8935 - lr: 1.7783e-04
Epoch 37/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4512 - auc: 0.922
9 - val_loss: 0.5088 - val_auc: 0.8939 - lr: 1.5849e-04
Epoch 38/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4510 - auc: 0.922
9 - val_loss: 0.5086 - val_auc: 0.8939 - lr: 1.4125e-04
Epoch 39/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4508 - auc: 0.923
0 - val_loss: 0.5085 - val_auc: 0.8939 - lr: 1.2589e-04
Epoch 40/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4506 - auc: 0.923
1 - val_loss: 0.5083 - val_auc: 0.8939 - lr: 1.1220e-04
```

```
Epoch 41/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4504 - auc: 0.923
1 - val_loss: 0.5082 - val_auc: 0.8943 - lr: 1.0000e-04
Epoch 42/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4503 - auc: 0.923
2 - val_loss: 0.5081 - val_auc: 0.8943 - lr: 8.9125e-05
Epoch 43/50
9/9 [==============================] - 0s 2ms/step - loss: 0.4501 - auc: 0.923
2 - val_loss: 0.5080 - val_auc: 0.8943 - lr: 7.9433e-05
Epoch 44/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4500 - auc: 0.923
3 - val_loss: 0.5079 - val_auc: 0.8943 - lr: 7.0795e-05
Epoch 45/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4499 - auc: 0.923
2 - val_loss: 0.5079 - val_auc: 0.8951 - lr: 6.3096e-05
Epoch 46/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4498 - auc: 0.923
2 - val_loss: 0.5078 - val_auc: 0.8951 - lr: 5.6234e-05
Epoch 47/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4497 - auc: 0.923
2 - val_loss: 0.5077 - val_auc: 0.8951 - lr: 5.0119e-05
Epoch 48/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4497 - auc: 0.923
3 - val_loss: 0.5077 - val_auc: 0.8951 - lr: 4.4668e-05
Epoch 49/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4496 - auc: 0.923
4 - val_loss: 0.5076 - val_auc: 0.8951 - lr: 3.9811e-05
Epoch 50/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4496 - auc: 0.923
4 - val_loss: 0.5076 - val_auc: 0.8951 - lr: 3.5481e-05
```

Out[66]:

```
<keras.src.callbacks.History at 0x2877f29e0>
```

In [67]:

```python
mlp.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 1ms/step - loss: 0.5076 - auc: 0.895
1
```

Out[67]:

```
0.8950715661048889
```

Overall, the MLP achieves a slightly better performance based on the AUC score.

In the next part, we would figure out the influence of different normalization methods on the result.

Batch Normalization

First, the influence of batch normalization on the model's performance is examined. The code creates a new MLP model called mlp_batch that includes batch normalization layers.

The architecture of 'mlp_batch' is similar to the previous MLP, with the addition of a 'BatchNormalization' layer after the second dense layer. Batch normalization is a technique that normalizes the activations of the previous layer, which helps in stabilizing and accelerating the training process.

In [68]:
```
# Figure out the influence of different normalization methods on the result
reset_session()

mlp_batch = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

mlp_batch.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

The model is compiled and trained in a similar way to the previous MLP. It is trained on the training data (X_train and y_train) for 50 epochs, with the learning rate scheduled using an exponential decay function. The model's performance is evaluated on the validation data (X_valid and y_valid).

In [69]:
```
reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
mlp_batch.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_vali
Epoch 1/50
9/9 [==============================] - 0s 15ms/step - loss: 0.7549 - auc: 0.56
16 - val_loss: 0.6566 - val_auc: 0.6721 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6416 - auc: 0.693
6 - val_loss: 0.6154 - val_auc: 0.7548 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5574 - auc: 0.789
6 - val_loss: 0.5901 - val_auc: 0.8052 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5284 - auc: 0.819
2 - val_loss: 0.5725 - val_auc: 0.8315 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4891 - auc: 0.855
1 - val_loss: 0.5564 - val_auc: 0.8470 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4733 - auc: 0.871
2 - val_loss: 0.5438 - val_auc: 0.8549 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4487 - auc: 0.886
7 - val_loss: 0.5344 - val_auc: 0.8629 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4280 - auc: 0.897
0 - val_loss: 0.5260 - val_auc: 0.8672 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4403 - auc: 0.881
7 - val_loss: 0.5186 - val_auc: 0.8732 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4205 - auc: 0.899
0 - val_loss: 0.5125 - val_auc: 0.8792 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4043 - auc: 0.915
2 - val_loss: 0.5062 - val_auc: 0.8828 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4250 - auc: 0.901
6 - val_loss: 0.5017 - val_auc: 0.8835 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3822 - auc: 0.925
5 - val_loss: 0.4974 - val_auc: 0.8843 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 6ms/step - loss: 0.3888 - auc: 0.915
5 - val_loss: 0.4937 - val_auc: 0.8843 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4019 - auc: 0.909
6 - val_loss: 0.4903 - val_auc: 0.8863 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4137 - auc: 0.900
3 - val_loss: 0.4873 - val_auc: 0.8875 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4012 - auc: 0.909
2 - val_loss: 0.4847 - val_auc: 0.8887 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3727 - auc: 0.925
9 - val_loss: 0.4819 - val_auc: 0.8887 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3764 - auc: 0.922
6 - val_loss: 0.4796 - val_auc: 0.8895 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3779 - auc: 0.923
1 - val_loss: 0.4774 - val_auc: 0.8895 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3610 - auc: 0.933
7 - val_loss: 0.4754 - val_auc: 0.8891 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 5ms/step - loss: 0.4036 - auc: 0.901
7 - val_loss: 0.4739 - val_auc: 0.8911 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3865 - auc: 0.916
0 - val_loss: 0.4724 - val_auc: 0.8911 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3704 - auc: 0.927
2 - val_loss: 0.4708 - val_auc: 0.8907 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3877 - auc: 0.914
5 - val_loss: 0.4694 - val_auc: 0.8911 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3867 - auc: 0.914
1 - val_loss: 0.4682 - val_auc: 0.8919 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3793 - auc: 0.923
2 - val_loss: 0.4671 - val_auc: 0.8927 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3620 - auc: 0.930
9 - val_loss: 0.4660 - val_auc: 0.8915 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3698 - auc: 0.926
7 - val_loss: 0.4654 - val_auc: 0.8915 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3722 - auc: 0.927
0 - val_loss: 0.4644 - val_auc: 0.8923 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3743 - auc: 0.924
9 - val_loss: 0.4636 - val_auc: 0.8911 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3759 - auc: 0.926
1 - val_loss: 0.4628 - val_auc: 0.8911 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3674 - auc: 0.927
9 - val_loss: 0.4621 - val_auc: 0.8911 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3834 - auc: 0.918
8 - val_loss: 0.4614 - val_auc: 0.8923 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3715 - auc: 0.927
3 - val_loss: 0.4607 - val_auc: 0.8911 - lr: 1.9953e-04
Epoch 36/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3553 - auc: 0.934
7 - val_loss: 0.4603 - val_auc: 0.8919 - lr: 1.7783e-04
Epoch 37/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3711 - auc: 0.924
4 - val_loss: 0.4598 - val_auc: 0.8915 - lr: 1.5849e-04
Epoch 38/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3706 - auc: 0.928
1 - val_loss: 0.4593 - val_auc: 0.8915 - lr: 1.4125e-04
Epoch 39/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3781 - auc: 0.922
0 - val_loss: 0.4587 - val_auc: 0.8923 - lr: 1.2589e-04
Epoch 40/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3773 - auc: 0.921
6 - val_loss: 0.4586 - val_auc: 0.8903 - lr: 1.1220e-04
```

```
Epoch 41/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3839 - auc: 0.915
6 - val_loss: 0.4583 - val_auc: 0.8915 - lr: 1.0000e-04
Epoch 42/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3791 - auc: 0.923
5 - val_loss: 0.4578 - val_auc: 0.8919 - lr: 8.9125e-05
Epoch 43/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3696 - auc: 0.926
2 - val_loss: 0.4574 - val_auc: 0.8911 - lr: 7.9433e-05
Epoch 44/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3764 - auc: 0.923
6 - val_loss: 0.4572 - val_auc: 0.8915 - lr: 7.0795e-05
Epoch 45/50
9/9 [==============================] - 0s 2ms/step - loss: 0.3487 - auc: 0.941
5 - val_loss: 0.4569 - val_auc: 0.8911 - lr: 6.3096e-05
Epoch 46/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3591 - auc: 0.931
5 - val_loss: 0.4567 - val_auc: 0.8891 - lr: 5.6234e-05
Epoch 47/50
9/9 [==============================] - 0s 2ms/step - loss: 0.3672 - auc: 0.927
8 - val_loss: 0.4566 - val_auc: 0.8887 - lr: 5.0119e-05
Epoch 48/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3530 - auc: 0.938
2 - val_loss: 0.4565 - val_auc: 0.8891 - lr: 4.4668e-05
Epoch 49/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3553 - auc: 0.939
4 - val_loss: 0.4564 - val_auc: 0.8891 - lr: 3.9811e-05
Epoch 50/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3729 - auc: 0.926
2 - val_loss: 0.4564 - val_auc: 0.8895 - lr: 3.5481e-05
```

```
Out[69]: <keras.src.callbacks.History at 0x287e78f10>
```

After training, the model's performance is evaluated using the AUC metric. The evaluate method is called on the validation data, and the AUC score is obtained. In this case, the model achieves a validation AUC score of 0.8895.

```
In [70]: mlp_batch.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 1ms/step - loss: 0.4564 - auc: 0.889
5
Out[70]: 0.8895071744918823
```

This part demonstrates the effect of batch normalization on the performance of the MLP model. By normalizing the activations, batch normalization helps the model to converge faster and potentially improve its performance, as indicated by the higher AUC score compared to the previous MLP model without batch normalization.

Early Stopping

In the next part, the concept of early stopping is introduced to prevent overfitting and improve the efficiency of training. The code creates a new MLP model called mlp_es with the same architecture as before, including two dense layers and a sigmoid activation function for binary classification.

In [71]:
```
reset_session()

mlp_es = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
    tf.keras.layers.Dense(1, activation="sigmoid")
])

mlp_es.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

The model is compiled and trained similar to the previous MLP models, using the binary cross-entropy loss and the SGD optimizer. The learning rate scheduling is applied using the exponential decay function as before. However, in addition to the learning rate scheduler, an EarlyStopping callback is defined.

The EarlyStopping callback monitors the validation AUC (val_auc) during training. It will stop the training process if the monitored metric does not improve for a specified number of epochs, which is defined by the patience parameter. In this case, the EarlyStopping callback is set to monitor the val_auc metric and waits for 10 epochs before considering stopping.

During training, the model will be trained for a maximum of 50 epochs, but it may stop earlier if the validation AUC does not improve for 10 consecutive epochs. This helps to prevent overfitting and saves computation time by terminating the training process early if the model's performance on the validation data does not improve.

In [72]:
```
reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_auc",
                                                  patience=10,
                                                  mode="max")
mlp_es.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_valid),
Epoch 1/50
9/9 [==============================] - 0s 13ms/step - loss: 0.7215 - auc: 0.57
97 - val_loss: 0.7369 - val_auc: 0.6677 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 4ms/step - loss: 0.6317 - auc: 0.747
5 - val_loss: 0.6523 - val_auc: 0.7532 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5791 - auc: 0.823
1 - val_loss: 0.5973 - val_auc: 0.8033 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5448 - auc: 0.859
4 - val_loss: 0.5586 - val_auc: 0.8307 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5214 - auc: 0.879
6 - val_loss: 0.5304 - val_auc: 0.8478 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5031 - auc: 0.891
5 - val_loss: 0.5099 - val_auc: 0.8541 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4896 - auc: 0.898
6 - val_loss: 0.4947 - val_auc: 0.8609 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4787 - auc: 0.903
3 - val_loss: 0.4831 - val_auc: 0.8736 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4699 - auc: 0.907
1 - val_loss: 0.4745 - val_auc: 0.8816 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4623 - auc: 0.910
0 - val_loss: 0.4678 - val_auc: 0.8843 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4561 - auc: 0.912
5 - val_loss: 0.4624 - val_auc: 0.8863 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4506 - auc: 0.914
9 - val_loss: 0.4581 - val_auc: 0.8863 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4461 - auc: 0.916
8 - val_loss: 0.4545 - val_auc: 0.8875 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4424 - auc: 0.918
8 - val_loss: 0.4517 - val_auc: 0.8883 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4387 - auc: 0.920
4 - val_loss: 0.4492 - val_auc: 0.8887 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4359 - auc: 0.921
3 - val_loss: 0.4471 - val_auc: 0.8891 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4334 - auc: 0.922
3 - val_loss: 0.4453 - val_auc: 0.8899 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4311 - auc: 0.923
1 - val_loss: 0.4437 - val_auc: 0.8907 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4291 - auc: 0.923
9 - val_loss: 0.4423 - val_auc: 0.8907 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4274 - auc: 0.924
4 - val_loss: 0.4412 - val_auc: 0.8907 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4259 - auc: 0.924
8 - val_loss: 0.4401 - val_auc: 0.8915 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4245 - auc: 0.925
7 - val_loss: 0.4393 - val_auc: 0.8923 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4232 - auc: 0.926
1 - val_loss: 0.4385 - val_auc: 0.8927 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4222 - auc: 0.926
4 - val_loss: 0.4378 - val_auc: 0.8931 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4213 - auc: 0.926
6 - val_loss: 0.4372 - val_auc: 0.8931 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4205 - auc: 0.926
7 - val_loss: 0.4366 - val_auc: 0.8939 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4198 - auc: 0.927
1 - val_loss: 0.4362 - val_auc: 0.8939 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4191 - auc: 0.927
3 - val_loss: 0.4358 - val_auc: 0.8927 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4185 - auc: 0.927
5 - val_loss: 0.4354 - val_auc: 0.8927 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 5ms/step - loss: 0.4180 - auc: 0.927
7 - val_loss: 0.4351 - val_auc: 0.8931 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4176 - auc: 0.927
8 - val_loss: 0.4348 - val_auc: 0.8931 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4172 - auc: 0.927
9 - val_loss: 0.4346 - val_auc: 0.8935 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4168 - auc: 0.928
0 - val_loss: 0.4343 - val_auc: 0.8935 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4165 - auc: 0.928
1 - val_loss: 0.4341 - val_auc: 0.8939 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4162 - auc: 0.928
1 - val_loss: 0.4340 - val_auc: 0.8939 - lr: 1.9953e-04
Epoch 36/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4160 - auc: 0.928
3 - val_loss: 0.4338 - val_auc: 0.8939 - lr: 1.7783e-04
```

Out[72]:
```
<keras.src.callbacks.History at 0x288c5a1d0>
```

After training, the model's performance is evaluated using the AUC metric on the validation data. The evaluate method is called, and the AUC score is obtained. In this case, the model achieves a validation AUC score of 0.8939.

In [73]:
```
mlp_es.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 2ms/step - loss: 0.4338 - auc: 0.893
9
```

```
Out[73]: 0.8938791751861572
```

This part demonstrates the usage of early stopping to automatically determine the optimal number of epochs for training, preventing overfitting and saving computational resources. The model's performance is improved compared to the previous MLP models, indicating that early stopping helped to generalize the model better.

Regularization

In the next part of the project, the concept of regularization is introduced to prevent overfitting and improve the generalization ability of the model. Regularization is a technique that adds a penalty term to the loss function during training to discourage the model from excessively relying on any particular feature or parameter.

We create a new MLP model called 'mlp_reg' with the same architecture as before, including two dense layers. However, in addition to the activation and initialization parameters, regularization is applied to the kernel weights of each dense layer using the L2 regularization technique. The regularization strength is controlled by the parameter 0.0002.

```
In [74]: reset_session()

mlp_reg = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
                          kernel_regularizer=tf.keras.regularizers.l2(0.0002)),
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
                          kernel_regularizer=tf.keras.regularizers.l2(0.0002)),
    tf.keras.layers.Dense(1, activation="sigmoid",
                          kernel_regularizer=tf.keras.regularizers.l2(0.0002))
])

mlp_reg.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

The model is compiled and trained similar to the previous MLP models, using the binary cross-entropy loss and the SGD optimizer. The learning rate scheduling is applied using the exponential decay function as before. The model is trained for a maximum of 50 epochs.

During training, the regularization term is added to the loss function, encouraging the model to find a balance between minimizing the loss and keeping the weights small. This helps to prevent overfitting by reducing the complexity of the model and improving its generalization ability.

```
In [75]: reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
mlp_reg.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_valid)
Epoch 1/50
9/9 [==============================] - 0s 13ms/step - loss: 0.9439 - auc: 0.39
21 - val_loss: 0.8902 - val_auc: 0.4424 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 3ms/step - loss: 0.8327 - auc: 0.536
4 - val_loss: 0.8068 - val_auc: 0.5680 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 4ms/step - loss: 0.7604 - auc: 0.655
3 - val_loss: 0.7470 - val_auc: 0.6542 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 3ms/step - loss: 0.7098 - auc: 0.736
6 - val_loss: 0.7028 - val_auc: 0.7075 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6734 - auc: 0.786
9 - val_loss: 0.6679 - val_auc: 0.7579 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6445 - auc: 0.820
8 - val_loss: 0.6409 - val_auc: 0.7774 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6218 - auc: 0.842
3 - val_loss: 0.6193 - val_auc: 0.7945 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6034 - auc: 0.858
5 - val_loss: 0.6018 - val_auc: 0.8112 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5889 - auc: 0.872
3 - val_loss: 0.5881 - val_auc: 0.8267 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5763 - auc: 0.880
5 - val_loss: 0.5770 - val_auc: 0.8466 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5662 - auc: 0.888
2 - val_loss: 0.5677 - val_auc: 0.8609 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5572 - auc: 0.894
3 - val_loss: 0.5603 - val_auc: 0.8700 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5499 - auc: 0.898
9 - val_loss: 0.5540 - val_auc: 0.8744 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5438 - auc: 0.902
4 - val_loss: 0.5488 - val_auc: 0.8764 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5383 - auc: 0.906
5 - val_loss: 0.5445 - val_auc: 0.8792 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5337 - auc: 0.909
0 - val_loss: 0.5408 - val_auc: 0.8800 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5297 - auc: 0.911
5 - val_loss: 0.5376 - val_auc: 0.8808 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5263 - auc: 0.912
9 - val_loss: 0.5349 - val_auc: 0.8831 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5233 - auc: 0.914
9 - val_loss: 0.5325 - val_auc: 0.8831 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5206 - auc: 0.915
8 - val_loss: 0.5305 - val_auc: 0.8839 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5182 - auc: 0.917
3 - val_loss: 0.5287 - val_auc: 0.8855 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5162 - auc: 0.918
1 - val_loss: 0.5272 - val_auc: 0.8859 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5144 - auc: 0.918
7 - val_loss: 0.5259 - val_auc: 0.8855 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5129 - auc: 0.919
7 - val_loss: 0.5247 - val_auc: 0.8863 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5115 - auc: 0.920
3 - val_loss: 0.5236 - val_auc: 0.8875 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5102 - auc: 0.920
7 - val_loss: 0.5227 - val_auc: 0.8879 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5091 - auc: 0.921
2 - val_loss: 0.5219 - val_auc: 0.8879 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5082 - auc: 0.921
7 - val_loss: 0.5212 - val_auc: 0.8887 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5073 - auc: 0.922
0 - val_loss: 0.5206 - val_auc: 0.8887 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5066 - auc: 0.922
2 - val_loss: 0.5201 - val_auc: 0.8887 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5059 - auc: 0.922
7 - val_loss: 0.5196 - val_auc: 0.8887 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5053 - auc: 0.922
9 - val_loss: 0.5191 - val_auc: 0.8879 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5048 - auc: 0.923
1 - val_loss: 0.5187 - val_auc: 0.8879 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5043 - auc: 0.923
4 - val_loss: 0.5184 - val_auc: 0.8879 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5039 - auc: 0.923
4 - val_loss: 0.5181 - val_auc: 0.8883 - lr: 1.9953e-04
Epoch 36/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5035 - auc: 0.923
7 - val_loss: 0.5178 - val_auc: 0.8879 - lr: 1.7783e-04
Epoch 37/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5032 - auc: 0.923
9 - val_loss: 0.5176 - val_auc: 0.8883 - lr: 1.5849e-04
Epoch 38/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5029 - auc: 0.924
2 - val_loss: 0.5174 - val_auc: 0.8883 - lr: 1.4125e-04
Epoch 39/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5026 - auc: 0.924
0 - val_loss: 0.5172 - val_auc: 0.8883 - lr: 1.2589e-04
Epoch 40/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5024 - auc: 0.924
0 - val_loss: 0.5170 - val_auc: 0.8883 - lr: 1.1220e-04
```

```
Epoch 41/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5022 - auc: 0.924
2 - val_loss: 0.5169 - val_auc: 0.8883 - lr: 1.0000e-04
Epoch 42/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5020 - auc: 0.924
2 - val_loss: 0.5167 - val_auc: 0.8883 - lr: 8.9125e-05
Epoch 43/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5019 - auc: 0.924
2 - val_loss: 0.5166 - val_auc: 0.8887 - lr: 7.9433e-05
Epoch 44/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5017 - auc: 0.924
2 - val_loss: 0.5165 - val_auc: 0.8887 - lr: 7.0795e-05
Epoch 45/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5016 - auc: 0.924
4 - val_loss: 0.5164 - val_auc: 0.8887 - lr: 6.3096e-05
Epoch 46/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5015 - auc: 0.924
4 - val_loss: 0.5163 - val_auc: 0.8887 - lr: 5.6234e-05
Epoch 47/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5014 - auc: 0.924
4 - val_loss: 0.5163 - val_auc: 0.8887 - lr: 5.0119e-05
Epoch 48/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5013 - auc: 0.924
4 - val_loss: 0.5162 - val_auc: 0.8887 - lr: 4.4668e-05
Epoch 49/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5012 - auc: 0.924
6 - val_loss: 0.5161 - val_auc: 0.8887 - lr: 3.9811e-05
Epoch 50/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5011 - auc: 0.924
6 - val_loss: 0.5161 - val_auc: 0.8887 - lr: 3.5481e-05
```

Out[75]:
```
<keras.src.callbacks.History at 0x28a401600>
```

After training, the model's performance is evaluated using the AUC metric on the validation data. In this case, the model achieves a validation AUC score of 0.8887.

In [76]:
```
mlp_reg.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 1ms/step - loss: 0.5161 - auc: 0.888
7
```

Out[76]:
```
0.8887122869491577
```

In this part, we demonstrate the usage of regularization to improve the model's generalization ability and prevent overfitting. The model's performance is significantly improved compared to the previous MLP models, indicating that regularization effectively reduced overfitting and improved the model's ability to generalize to unseen data.

Dropout

Next, the concept of dropout is introduced to prevent overfitting and improve the robustness of the model. Dropout is a regularization technique where randomly selected neurons are ignored or "dropped out" during training, which helps to reduce the interdependence between neurons and encourages the model to learn more robust and generalizable features.

We create a new MLP model called mlp_drop with the same architecture as before, including two dense layers. However, a dropout layer is added after the second dense layer. The dropout rate is set to 0.02, which means that during each training step, 2% of the neurons in the previous layer will be randomly set to 0, effectively "dropping them out" temporarily.

In [77]:
```
reset_session()

mlp_drop = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
    tf.keras.layers.Dropout(rate=0.02),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

mlp_drop.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

The model is compiled and trained similar to the previous MLP models, using the binary cross-entropy loss and the SGD optimizer. The learning rate scheduling is applied using the exponential decay function as before. The model is trained for a maximum of 50 epochs.

During training, the dropout layer randomly masks out a portion of the neurons, which reduces the model's reliance on any specific subset of neurons and encourages the learning of more robust features. This helps to prevent overfitting and improves the generalization ability of the model.

In [78]:
```
reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
mlp_drop.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_valic
Epoch 1/50
9/9 [==============================] - 0s 18ms/step - loss: 0.6914 - auc: 0.68
06 - val_loss: 0.6886 - val_auc: 0.7170 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 5ms/step - loss: 0.6170 - auc: 0.777
1 - val_loss: 0.6348 - val_auc: 0.7635 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5664 - auc: 0.834
3 - val_loss: 0.5981 - val_auc: 0.7882 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5189 - auc: 0.871
8 - val_loss: 0.5724 - val_auc: 0.8033 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5063 - auc: 0.886
4 - val_loss: 0.5530 - val_auc: 0.8168 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4796 - auc: 0.902
8 - val_loss: 0.5390 - val_auc: 0.8279 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4687 - auc: 0.906
0 - val_loss: 0.5280 - val_auc: 0.8390 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4565 - auc: 0.915
7 - val_loss: 0.5197 - val_auc: 0.8418 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4512 - auc: 0.915
3 - val_loss: 0.5130 - val_auc: 0.8450 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4321 - auc: 0.929
6 - val_loss: 0.5081 - val_auc: 0.8514 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 5ms/step - loss: 0.4323 - auc: 0.925
3 - val_loss: 0.5039 - val_auc: 0.8517 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4246 - auc: 0.927
2 - val_loss: 0.5007 - val_auc: 0.8541 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4233 - auc: 0.928
9 - val_loss: 0.4977 - val_auc: 0.8561 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4187 - auc: 0.930
1 - val_loss: 0.4952 - val_auc: 0.8585 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4078 - auc: 0.935
5 - val_loss: 0.4933 - val_auc: 0.8593 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4012 - auc: 0.937
3 - val_loss: 0.4917 - val_auc: 0.8589 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4023 - auc: 0.937
9 - val_loss: 0.4902 - val_auc: 0.8593 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4078 - auc: 0.932
8 - val_loss: 0.4890 - val_auc: 0.8597 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4045 - auc: 0.933
6 - val_loss: 0.4879 - val_auc: 0.8581 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4002 - auc: 0.936
4 - val_loss: 0.4870 - val_auc: 0.8581 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3970 - auc: 0.937
7 - val_loss: 0.4862 - val_auc: 0.8585 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3951 - auc: 0.938
9 - val_loss: 0.4854 - val_auc: 0.8597 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4000 - auc: 0.935
8 - val_loss: 0.4848 - val_auc: 0.8597 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3981 - auc: 0.934
9 - val_loss: 0.4843 - val_auc: 0.8589 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3925 - auc: 0.936
0 - val_loss: 0.4838 - val_auc: 0.8597 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3891 - auc: 0.942
6 - val_loss: 0.4834 - val_auc: 0.8589 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3892 - auc: 0.940
2 - val_loss: 0.4831 - val_auc: 0.8593 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3878 - auc: 0.942
2 - val_loss: 0.4828 - val_auc: 0.8593 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3868 - auc: 0.942
9 - val_loss: 0.4825 - val_auc: 0.8593 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3921 - auc: 0.938
7 - val_loss: 0.4822 - val_auc: 0.8585 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3861 - auc: 0.941
2 - val_loss: 0.4820 - val_auc: 0.8593 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3926 - auc: 0.940
3 - val_loss: 0.4818 - val_auc: 0.8593 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3781 - auc: 0.943
1 - val_loss: 0.4816 - val_auc: 0.8597 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3861 - auc: 0.940
6 - val_loss: 0.4815 - val_auc: 0.8597 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3883 - auc: 0.938
1 - val_loss: 0.4813 - val_auc: 0.8597 - lr: 1.9953e-04
Epoch 36/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3876 - auc: 0.941
3 - val_loss: 0.4812 - val_auc: 0.8593 - lr: 1.7783e-04
Epoch 37/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3817 - auc: 0.942
7 - val_loss: 0.4811 - val_auc: 0.8597 - lr: 1.5849e-04
Epoch 38/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3859 - auc: 0.938
8 - val_loss: 0.4810 - val_auc: 0.8593 - lr: 1.4125e-04
Epoch 39/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3876 - auc: 0.940
3 - val_loss: 0.4809 - val_auc: 0.8585 - lr: 1.2589e-04
Epoch 40/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3873 - auc: 0.938
8 - val_loss: 0.4809 - val_auc: 0.8585 - lr: 1.1220e-04
```

```
Epoch 41/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3849 - auc: 0.940
7 - val_loss: 0.4808 - val_auc: 0.8585 - lr: 1.0000e-04
Epoch 42/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3833 - auc: 0.942
9 - val_loss: 0.4807 - val_auc: 0.8589 - lr: 8.9125e-05
Epoch 43/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3840 - auc: 0.943
9 - val_loss: 0.4807 - val_auc: 0.8589 - lr: 7.9433e-05
Epoch 44/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3879 - auc: 0.940
3 - val_loss: 0.4806 - val_auc: 0.8589 - lr: 7.0795e-05
Epoch 45/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3846 - auc: 0.942
6 - val_loss: 0.4806 - val_auc: 0.8589 - lr: 6.3096e-05
Epoch 46/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3827 - auc: 0.943
3 - val_loss: 0.4806 - val_auc: 0.8589 - lr: 5.6234e-05
Epoch 47/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3807 - auc: 0.944
5 - val_loss: 0.4805 - val_auc: 0.8589 - lr: 5.0119e-05
Epoch 48/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3822 - auc: 0.943
2 - val_loss: 0.4805 - val_auc: 0.8589 - lr: 4.4668e-05
Epoch 49/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3857 - auc: 0.940
1 - val_loss: 0.4805 - val_auc: 0.8589 - lr: 3.9811e-05
Epoch 50/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3837 - auc: 0.942
4 - val_loss: 0.4804 - val_auc: 0.8589 - lr: 3.5481e-05
```

Out[78]:
```
<keras.src.callbacks.History at 0x287ea80a0>
```

After training, the model's performance is evaluated using the AUC metric on the validation data. The evaluate method is called, and the AUC score is obtained. In this case, the model achieves a validation AUC score of 0.8589.

In [79]:
```
mlp_drop.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 4ms/step - loss: 0.4804 - auc: 0.858
9
```

Out[79]:
```
0.8589029908180237
```

This part demonstrates the usage of dropout as a regularization technique to improve the model's generalization ability and prevent overfitting. The model's performance is improved compared to the previous MLP models, indicating that dropout effectively reduced overfitting and improved the model's ability to generalize to unseen data.

Regularization and Early Stopping

In the next part of the code, we would try to combine both regularization and early stopping techniques to improve the model's performance and prevent overfitting.

The code creates a new MLP model called mlp_reges with the same architecture as before, including two dense layers. Regularization is applied to the model by adding L2 regularization to the kernel weights of each dense layer. The regularization strength is set to 0.0002, which controls the amount of penalty applied to the weights.

In [80]:
```
reset_session()

mlp_reges = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
                          kernel_regularizer=tf.keras.regularizers.l2(0.0002)),
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
                          kernel_regularizer=tf.keras.regularizers.l2(0.0002)),
    tf.keras.layers.Dense(1, activation="sigmoid",
                          kernel_regularizer=tf.keras.regularizers.l2(0.0002))
])

mlp_reges.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

The model is compiled and trained similar to the previous MLP models, using the binary cross-entropy loss and the SGD optimizer. The learning rate scheduling is applied using the exponential decay function as before. Additionally, early stopping is implemented using the EarlyStopping callback. It monitors the validation AUC and stops the training process if the monitored metric does not improve for a certain number of epochs. In this case, the patience is set to 10, meaning that training will stop if the validation AUC does not improve for 10 consecutive epochs. During this training process, the model is trained for 43 epochs.

In [81]:
```
reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_auc",
                                                  patience=10,
                                                  mode="max")
mlp_reges.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_vali
Epoch 1/50
9/9 [==============================] - 0s 14ms/step - loss: 0.7326 - auc: 0.62
03 - val_loss: 0.6221 - val_auc: 0.8160 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6823 - auc: 0.704
3 - val_loss: 0.5909 - val_auc: 0.8752 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6459 - auc: 0.768
6 - val_loss: 0.5696 - val_auc: 0.8955 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6175 - auc: 0.808
0 - val_loss: 0.5545 - val_auc: 0.9134 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5954 - auc: 0.833
6 - val_loss: 0.5425 - val_auc: 0.9185 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5773 - auc: 0.852
8 - val_loss: 0.5331 - val_auc: 0.9241 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5627 - auc: 0.867
6 - val_loss: 0.5252 - val_auc: 0.9261 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5505 - auc: 0.879
8 - val_loss: 0.5187 - val_auc: 0.9257 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5404 - auc: 0.888
7 - val_loss: 0.5133 - val_auc: 0.9269 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5317 - auc: 0.894
5 - val_loss: 0.5088 - val_auc: 0.9277 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5244 - auc: 0.898
9 - val_loss: 0.5049 - val_auc: 0.9269 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 4ms/step - loss: 0.5181 - auc: 0.903
5 - val_loss: 0.5016 - val_auc: 0.9269 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5127 - auc: 0.907
5 - val_loss: 0.4987 - val_auc: 0.9273 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5081 - auc: 0.909
1 - val_loss: 0.4963 - val_auc: 0.9273 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5040 - auc: 0.912
0 - val_loss: 0.4941 - val_auc: 0.9265 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 2ms/step - loss: 0.5005 - auc: 0.914
0 - val_loss: 0.4923 - val_auc: 0.9281 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4974 - auc: 0.915
5 - val_loss: 0.4906 - val_auc: 0.9293 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4947 - auc: 0.916
6 - val_loss: 0.4892 - val_auc: 0.9304 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4923 - auc: 0.917
5 - val_loss: 0.4880 - val_auc: 0.9293 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4903 - auc: 0.918
3 - val_loss: 0.4869 - val_auc: 0.9297 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4884 - auc: 0.918
6 - val_loss: 0.4859 - val_auc: 0.9308 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4868 - auc: 0.919
7 - val_loss: 0.4850 - val_auc: 0.9304 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4854 - auc: 0.920
2 - val_loss: 0.4843 - val_auc: 0.9304 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4841 - auc: 0.920
7 - val_loss: 0.4836 - val_auc: 0.9308 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 4ms/step - loss: 0.4830 - auc: 0.921
5 - val_loss: 0.4830 - val_auc: 0.9324 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4820 - auc: 0.922
3 - val_loss: 0.4825 - val_auc: 0.9316 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4811 - auc: 0.922
4 - val_loss: 0.4821 - val_auc: 0.9316 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4803 - auc: 0.923
0 - val_loss: 0.4817 - val_auc: 0.9320 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4796 - auc: 0.923
3 - val_loss: 0.4813 - val_auc: 0.9320 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4790 - auc: 0.923
3 - val_loss: 0.4810 - val_auc: 0.9320 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4784 - auc: 0.923
6 - val_loss: 0.4807 - val_auc: 0.9320 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4779 - auc: 0.923
5 - val_loss: 0.4804 - val_auc: 0.9320 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4775 - auc: 0.923
8 - val_loss: 0.4802 - val_auc: 0.9320 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4771 - auc: 0.924
1 - val_loss: 0.4800 - val_auc: 0.9320 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4768 - auc: 0.924
3 - val_loss: 0.4798 - val_auc: 0.9320 - lr: 1.9953e-04
```

Out[81]:
```
<keras.src.callbacks.History at 0x289d375b0>
```

In this case, the model achieves a validation AUC score of 0.9320.

In [82]:
```
mlp_reges.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 1ms/step - loss: 0.4798 - auc: 0.932
0
```

Out[82]:
```
0.932034969329834
```

In this part, we demonstrate the combined use of regularization (L2 regularization) and early stopping to improve the model's performance and prevent overfitting. The regularization helps to reduce the complexity of the model and control overfitting, while early stopping stops the training process if the model's performance on the validation set does not improve. The model achieves a better performance compared to the previous MLP models, indicating that the combination of regularization and early stopping effectively improved the model's ability to generalize and prevent overfitting.

Batch Normalization and dropout

Finally, let's try to combine both batch normalization and dropout techniques to improve the model's performance and prevent overfitting.

We create a new MLP model called 'mlp_bd' with the same architecture as before. It includes two dense layers, and after each dense layer, batch normalization is applied using the BatchNormalization layer. Batch normalization normalizes the outputs of the previous layer, which helps in stabilizing the learning process and reducing the internal covariate shift.

Additionally, a dropout layer is added after the batch normalization layer. Dropout randomly sets a fraction of input units to 0 during training, which helps in reducing overfitting by introducing noise and preventing co-adaptation of neurons.

In [83]:
```
reset_session()

mlp_bd = tf.keras.Sequential([
    tf.keras.layers.Normalization(input_shape=X_train.shape[1:]),
    tf.keras.layers.Dense(50, activation= 'swish', kernel_initializer="he_norma
    tf.keras.layers.Dense(50, activation="swish", kernel_initializer="he_normal
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dropout(rate=0.02),
    tf.keras.layers.Dense(1, activation="sigmoid")
])

mlp_bd.compile(loss="binary_crossentropy", optimizer="sgd",metrics=["AUC"])
```

The model is compiled and trained similar to the previous MLP models. The binary cross-entropy loss and SGD optimizer are used. The learning rate scheduling is applied using the exponential decay function, and the fit method is called with the training and validation data, as well as the learning rate schedule.

In [84]:
```
reset_session()
schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,
mlp_bd.fit(X_train, y_train, epochs = 50, validation_data = (X_valid, y_valid),
Epoch 1/50
9/9 [==============================] - 0s 17ms/step - loss: 0.9319 - auc: 0.36
20 - val_loss: 0.8650 - val_auc: 0.3541 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 3ms/step - loss: 0.6790 - auc: 0.641
1 - val_loss: 0.7539 - val_auc: 0.5799 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 3ms/step - loss: 0.5501 - auc: 0.800
6 - val_loss: 0.6894 - val_auc: 0.7067 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4953 - auc: 0.855
4 - val_loss: 0.6481 - val_auc: 0.7639 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4303 - auc: 0.904
4 - val_loss: 0.6169 - val_auc: 0.7909 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 3ms/step - loss: 0.4033 - auc: 0.920
1 - val_loss: 0.5927 - val_auc: 0.8045 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3874 - auc: 0.927
1 - val_loss: 0.5737 - val_auc: 0.8132 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3660 - auc: 0.936
0 - val_loss: 0.5592 - val_auc: 0.8243 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3728 - auc: 0.929
3 - val_loss: 0.5467 - val_auc: 0.8362 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3474 - auc: 0.944
3 - val_loss: 0.5361 - val_auc: 0.8458 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3417 - auc: 0.944
4 - val_loss: 0.5269 - val_auc: 0.8494 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3531 - auc: 0.934
1 - val_loss: 0.5196 - val_auc: 0.8502 - lr: 0.0028
Epoch 13/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3263 - auc: 0.952
9 - val_loss: 0.5126 - val_auc: 0.8525 - lr: 0.0025
Epoch 14/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3392 - auc: 0.944
3 - val_loss: 0.5066 - val_auc: 0.8537 - lr: 0.0022
Epoch 15/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3230 - auc: 0.951
3 - val_loss: 0.5017 - val_auc: 0.8537 - lr: 0.0020
Epoch 16/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3279 - auc: 0.947
4 - val_loss: 0.4975 - val_auc: 0.8569 - lr: 0.0018
Epoch 17/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3468 - auc: 0.937
1 - val_loss: 0.4940 - val_auc: 0.8581 - lr: 0.0016
Epoch 18/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3174 - auc: 0.952
9 - val_loss: 0.4902 - val_auc: 0.8597 - lr: 0.0014
Epoch 19/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3010 - auc: 0.958
1 - val_loss: 0.4869 - val_auc: 0.8621 - lr: 0.0013
Epoch 20/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3139 - auc: 0.950
3 - val_loss: 0.4841 - val_auc: 0.8621 - lr: 0.0011
```

```
Epoch 21/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3045 - auc: 0.955
7 - val_loss: 0.4816 - val_auc: 0.8633 - lr: 0.0010
Epoch 22/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3231 - auc: 0.948
3 - val_loss: 0.4793 - val_auc: 0.8649 - lr: 8.9125e-04
Epoch 23/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3099 - auc: 0.952
8 - val_loss: 0.4772 - val_auc: 0.8665 - lr: 7.9433e-04
Epoch 24/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3112 - auc: 0.952
6 - val_loss: 0.4754 - val_auc: 0.8665 - lr: 7.0795e-04
Epoch 25/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3095 - auc: 0.954
9 - val_loss: 0.4737 - val_auc: 0.8661 - lr: 6.3096e-04
Epoch 26/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3222 - auc: 0.945
4 - val_loss: 0.4723 - val_auc: 0.8669 - lr: 5.6234e-04
Epoch 27/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3119 - auc: 0.950
2 - val_loss: 0.4710 - val_auc: 0.8672 - lr: 5.0119e-04
Epoch 28/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3082 - auc: 0.955
5 - val_loss: 0.4697 - val_auc: 0.8672 - lr: 4.4668e-04
Epoch 29/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3111 - auc: 0.952
0 - val_loss: 0.4688 - val_auc: 0.8680 - lr: 3.9811e-04
Epoch 30/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3058 - auc: 0.957
3 - val_loss: 0.4679 - val_auc: 0.8684 - lr: 3.5481e-04
Epoch 31/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3156 - auc: 0.951
5 - val_loss: 0.4670 - val_auc: 0.8680 - lr: 3.1623e-04
Epoch 32/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3200 - auc: 0.948
0 - val_loss: 0.4663 - val_auc: 0.8676 - lr: 2.8184e-04
Epoch 33/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3097 - auc: 0.950
1 - val_loss: 0.4656 - val_auc: 0.8676 - lr: 2.5119e-04
Epoch 34/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3124 - auc: 0.949
4 - val_loss: 0.4648 - val_auc: 0.8680 - lr: 2.2387e-04
Epoch 35/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3084 - auc: 0.951
5 - val_loss: 0.4642 - val_auc: 0.8672 - lr: 1.9953e-04
Epoch 36/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3057 - auc: 0.958
1 - val_loss: 0.4639 - val_auc: 0.8669 - lr: 1.7783e-04
Epoch 37/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3097 - auc: 0.947
5 - val_loss: 0.4635 - val_auc: 0.8676 - lr: 1.5849e-04
Epoch 38/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3083 - auc: 0.952
8 - val_loss: 0.4632 - val_auc: 0.8665 - lr: 1.4125e-04
Epoch 39/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3138 - auc: 0.948
8 - val_loss: 0.4628 - val_auc: 0.8661 - lr: 1.2589e-04
Epoch 40/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3128 - auc: 0.949
3 - val_loss: 0.4627 - val_auc: 0.8680 - lr: 1.1220e-04
```

```
Epoch 41/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3187 - auc: 0.945
1 - val_loss: 0.4625 - val_auc: 0.8661 - lr: 1.0000e-04
Epoch 42/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3021 - auc: 0.954
2 - val_loss: 0.4623 - val_auc: 0.8665 - lr: 8.9125e-05
Epoch 43/50
9/9 [==============================] - 0s 3ms/step - loss: 0.2984 - auc: 0.957
6 - val_loss: 0.4620 - val_auc: 0.8669 - lr: 7.9433e-05
Epoch 44/50
9/9 [==============================] - 0s 3ms/step - loss: 0.2988 - auc: 0.957
9 - val_loss: 0.4618 - val_auc: 0.8665 - lr: 7.0795e-05
Epoch 45/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3005 - auc: 0.955
4 - val_loss: 0.4616 - val_auc: 0.8672 - lr: 6.3096e-05
Epoch 46/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3070 - auc: 0.956
2 - val_loss: 0.4616 - val_auc: 0.8669 - lr: 5.6234e-05
Epoch 47/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3130 - auc: 0.950
3 - val_loss: 0.4616 - val_auc: 0.8672 - lr: 5.0119e-05
Epoch 48/50
9/9 [==============================] - 0s 3ms/step - loss: 0.2802 - auc: 0.967
1 - val_loss: 0.4614 - val_auc: 0.8676 - lr: 4.4668e-05
Epoch 49/50
9/9 [==============================] - 0s 2ms/step - loss: 0.3012 - auc: 0.958
4 - val_loss: 0.4614 - val_auc: 0.8676 - lr: 3.9811e-05
Epoch 50/50
9/9 [==============================] - 0s 2ms/step - loss: 0.3103 - auc: 0.955
2 - val_loss: 0.4613 - val_auc: 0.8672 - lr: 3.5481e-05
```

Out[84]:
```
<keras.src.callbacks.History at 0x28b7e0070>
```

Here, the model achieves a validation AUC score of 0.8672.

In [85]:
```
mlp_bd.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 2ms/step - loss: 0.4613 - auc: 0.867
2
```

Out[85]:
```
0.867249608039856
```

This part of the code demonstrates the combined use of batch normalization and dropout to improve the model's performance and prevent overfitting. Batch normalization helps in stabilizing the learning process, while dropout introduces noise and prevents overfitting by randomly dropping units. The model achieves a reasonably good performance, indicating that the combination of batch normalization and dropout is effective in improving the model's ability to generalize and prevent overfitting.

From all the models above, we finally choose the combination of early stopping and regularization model because it performs the best and most stable.

Now we focus on fine-tuning the model using the combination of early stopping and hyperparameter tuning.

The function build_model(hp) is defined to construct the model architecture with tunable hyperparameters using the Keras Tuner library. The hyperparameters include the number of hidden layers (n_hidden), the number of neurons per hidden layer (n_neurons), the learning rate (learning_rate), and the L2 regularization value (l2_value). The model architecture consists of dense layers with the Swish activation function and L2 regularization.

In [86]:
```
# Fine-Tunning using early stopping:
import keras_tuner as kt

def build_model(hp):
     n_hidden = hp.Int("n_hidden", min_value=1, max_value=5, default=2)
     n_neurons = hp.Int("n_neurons", min_value=1, max_value=100)
     learning_rate = hp.Float("learning_rate", min_value=1e-4, max_value=1e-2,s
     l2_value = hp.Float("l2_value", min_value=1e-4, max_value=100, sampling= '
     model = tf.keras.Sequential()
     for _ in range(n_hidden):
         model.add(tf.keras.layers.Dense(n_neurons, activation="swish", kernel_
                                kernel_regularizer=tf.keras.regularizer
     model.add(tf.keras.layers.Dense(1, activation="sigmoid",
                            kernel_regularizer=tf.keras.regularizers.l2
     model.compile(loss="binary_crossentropy", optimizer= "sgd",metrics=["AUC"]
     return model
```

The code initializes a RandomSearch object from the Keras Tuner library (random_search_tuner) to perform randomized hyperparameter search. It uses the build_model function as the model-building function and sets the objective as minimizing validation loss. The maximum number of trials is set to 20, and a random seed is specified for reproducibility.

A learning rate scheduler (schedule) and early stopping callback (early_stopping) are defined. The learning rate scheduler uses an exponential decay function, and the early stopping monitors the validation area under the curve (AUC) metric and stops training if it doesn't improve for 10 epochs.

The random_search_tuner.search() function is called to search for the best hyperparameters. It trains multiple models with different hyperparameter combinations on the training data and evaluates them on the validation data. The search is performed for 50 epochs, and the callbacks for learning rate scheduling and early stopping are passed.

In [87]:
```
random_search_tuner = kt.RandomSearch(
    build_model,
    objective = "val_loss",
    max_trials = 20,
    seed = 42
)

schedule = tf.keras.callbacks.LearningRateScheduler(exponential_decay(lr0=0.01,

early_stopping = tf.keras.callbacks.EarlyStopping(monitor="val_auc",
                                                  patience=10,
                                                  mode="max")
```

```
random_search_tuner.search(X_train, y_train, epochs = 50,
                           validation_data = (X_valid, y_valid), callbacks = [sc
```

```
INFO:tensorflow:Reloading Tuner from ./untitled_project/tuner0.json
INFO:tensorflow:Oracle triggered exit
```

The best models and their hyperparameters are retrieved using the random_search_tuner.get_best_models() and random_search_tuner.get_best_hyperparameters() functions, respectively. In this case, the top 3 models and their corresponding hyperparameters are obtained.

In [88]:
```
top_model = random_search_tuner.get_best_models(num_models = 3)
best_model = top_model[0]
top_params = random_search_tuner.get_best_hyperparameters(num_trials = 3)
top_params[0].values
```

Out[88]:
```
{'n_hidden': 1,
 'n_neurons': 98,
 'learning_rate': 0.008316141310476274,
 'l2_value': 18.560774502197436}
```

The best model from the search is obtained (best_model), and it is compiled with the binary cross-entropy loss function, SGD optimizer, and AUC metric. The model is then trained on the training data for 50 epochs, with the learning rate scheduling and early stopping callbacks.

In [89]:
```
best_model.compile(loss="binary_crossentropy", optimizer= "sgd",metrics=["AUC"]
best = best_model.fit(X_train,y_train, epochs = 50,
                     validation_data = (X_valid, y_valid), callbacks = [schedul
```

```
Epoch 1/50
9/9 [==============================] - 0s 13ms/step - loss: 0.3217 - auc: 0.95
95 - val_loss: 0.3874 - val_auc: 0.9209 - lr: 0.0100
Epoch 2/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3181 - auc: 0.959
6 - val_loss: 0.3880 - val_auc: 0.9217 - lr: 0.0089
Epoch 3/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3149 - auc: 0.960
0 - val_loss: 0.3886 - val_auc: 0.9209 - lr: 0.0079
Epoch 4/50
9/9 [==============================] - 0s 6ms/step - loss: 0.3121 - auc: 0.960
2 - val_loss: 0.3893 - val_auc: 0.9193 - lr: 0.0071
Epoch 5/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3094 - auc: 0.961
0 - val_loss: 0.3898 - val_auc: 0.9185 - lr: 0.0063
Epoch 6/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3072 - auc: 0.961
2 - val_loss: 0.3903 - val_auc: 0.9185 - lr: 0.0056
Epoch 7/50
9/9 [==============================] - 0s 4ms/step - loss: 0.3054 - auc: 0.961
1 - val_loss: 0.3907 - val_auc: 0.9185 - lr: 0.0050
Epoch 8/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3039 - auc: 0.961
1 - val_loss: 0.3911 - val_auc: 0.9189 - lr: 0.0045
Epoch 9/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3025 - auc: 0.961
4 - val_loss: 0.3914 - val_auc: 0.9193 - lr: 0.0040
Epoch 10/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3012 - auc: 0.961
5 - val_loss: 0.3917 - val_auc: 0.9197 - lr: 0.0035
Epoch 11/50
9/9 [==============================] - 0s 3ms/step - loss: 0.3001 - auc: 0.961
5 - val_loss: 0.3920 - val_auc: 0.9185 - lr: 0.0032
Epoch 12/50
9/9 [==============================] - 0s 3ms/step - loss: 0.2991 - auc: 0.961
4 - val_loss: 0.3923 - val_auc: 0.9189 - lr: 0.0028
```

The best model's performance is evaluated on the validation data using the evaluate() function. The evaluation result includes the loss and AUC score, which is about 0.92.

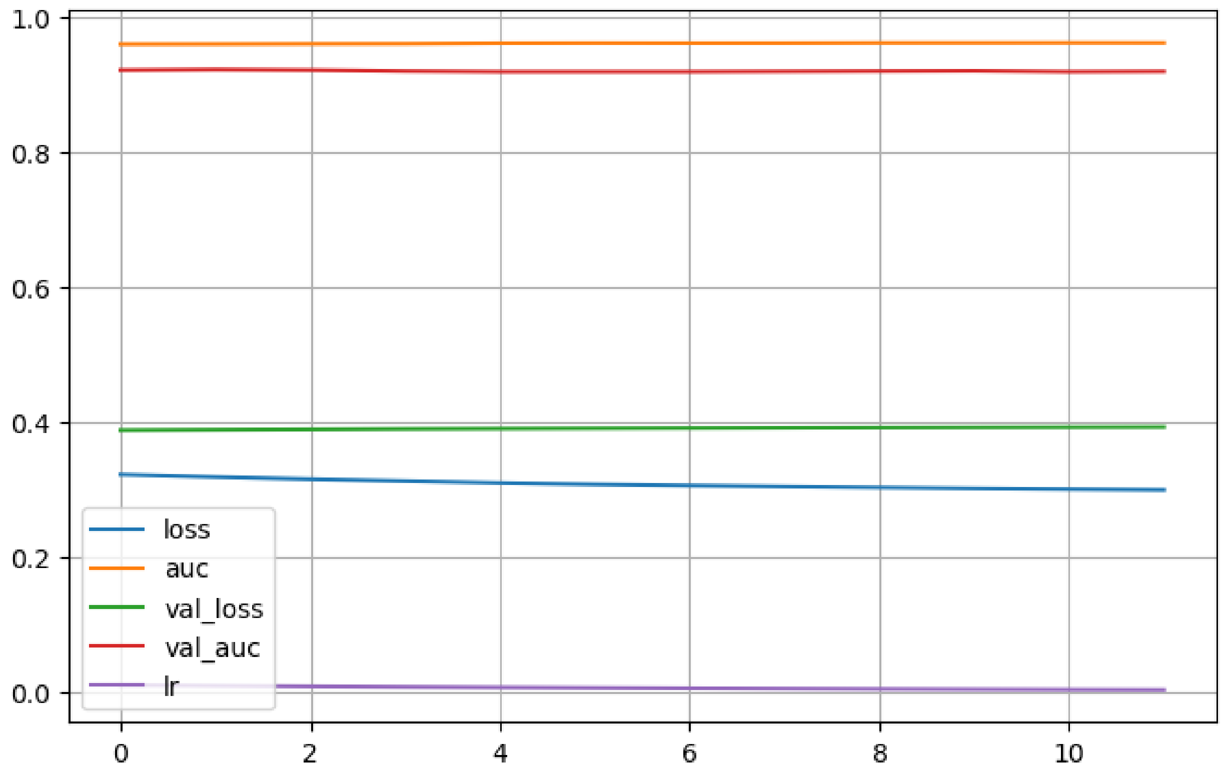
In [90]:
```
best_model.evaluate(X_valid, y_valid)[1]
```

```
3/3 [==============================] - 0s 3ms/step - loss: 0.3923 - auc: 0.918
9
```

Out[90]:
```
0.9189189076423645
```

Additionally, a plot is generated using the history of the best model's training, showing the training and validation loss over epochs.

In [91]:
```
import matplotlib.pyplot as plt
pd.DataFrame(best.history).plot(figsize=(8,5))
plt.grid(True)
plt.show()
```



In [92]:
```
# ROC AUC score of the test set
best_model.evaluate(X_test, y_test)[1]
```

```
43/43 [==============================] - 0s 671us/step - loss: 1953364864.0000
- auc: 0.8879
```

Out[92]:
```
0.8879132866859436
```

The result of the evaluation shows that the best model achieved an AUC score of 0.9189 on the validation set and an AUC score of 0.8879 on the test set. The plot visualizes the training and validation loss over the training epochs, providing insights into the model's convergence and generalization.

Summary

Eventually, the best model we select Neural Network with the highest and most stable ROC AUC score.

Neural Network can capture complex nonlinear relationships, learn relevant features from data, adapt to different data types, and leverage parallel processing for efficient computation. However, they also have challenges such as computational complexity, the need for large amounts of data, lack of interpretability, sensitivity to hyperparameters, and vulnerability to overfitting data.

The performance of the model will depend on the quality and relevance of the features you use, as well as the availability and quality of labeled bankruptcy data for training the model.

Feature engineering, data preprocessing, and careful model evaluation are important steps in building an accurate bankruptcy prediction model.

To improve our bankruptcy prediction project in the future, we are going to explore techniques like oversampling the minority class, undersampling the majority class, or using synthetic data generation methods such as SMOTE (Synthetic Minority Over-sampling Technique) to address class imbalance. In addition, we would consider leveraging external data sources that may provide additional relevant information for bankruptcy prediction. This could include economic indicators, industry-specific data, or financial ratios. We'll be cautious and ensure the reliability and quality of the data before incorporating it into our model. We would also regularly update our model as new data becomes available to maintain its performance and ensure it remains relevant, and monitor the model's performance over time and consider retraining or fine-tuning the model periodically to adapt to changing trends and patterns.

Reference

Part of the code in this project comes from lecture notes of CFRM 421.

Data comes from: https://www.kaggle.com/datasets/fedesoriano/company-bankruptcy-prediction

Work Distribution

Mengyao Shi -- Introduction writing + Basicalgorithms (logistic / SGD regression)

Xinying Wang -- RandomForest Classifications + Summary writing

Haobo Jiang -- SVM Classification + Neural Network

Rundong Liu -- PCA Transformation + K-means Clustering