Problem description

You are to predict whether a company will go bankrupt in the following year, based on financial attributes of the company.

Perhaps you are contemplating lending money to a company, and need to know whether the company is in near-term danger of not being able to repay.

Import modules

In [7]:

```
## Standard imports
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import sklearn

import os
import math

%matplotlib inline
```

API

In [8]:

```
## Load the bankruptcy_helper module

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# Reload all modules imported with %aimport
%load_ext autoreload
%autoreload 1

# Import bankruptcy_helper module
import bankruptcy_helper
%aimport bankruptcy_helper

helper = bankruptcy_helper.Helper()
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Get the data

The first step in our Recipe is Get the Data.

- Each example is a row of data corresponding to a single company
- There are 64 attributes, described in the section below
- The column `Bankrupt` is 1 if the company subsequently went bankrupt; 0 if it did not go bankrupt
- The column `Id` is a Company Identifier

In [9]:
```python
# Data directory
DATA_DIR = "./Data"

if not os.path.isdir(DATA_DIR):
    DATA_DIR = "../resource/asnlib/publicdata/bankruptcy/data"

data_file = "5th_yr.csv"
data = pd.read_csv( os.path.join(DATA_DIR, "train", data_file) )

target_attr = "Bankrupt"

n_samples, n_attrs = data.shape
print("Date shape: ", data.shape)
```

```
Date shape:  (4818, 66)
```

Have a look at the data

In [10]:
```python
data.head()
```

Out[10]:

	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	
0	0.025417	0.41769	0.0568	1.1605	-126.39	0.41355	0.025417	1.2395	1.16500	0.51773	0.02
1	-0.023834	0.2101	0.50839	4.2374	22.034	0.058412	-0.027621	3.6579	0.98183	0.76855	-0.02
2	0.030515	0.44606	0.19569	1.565	35.766	0.28196	0.039264	0.88456	1.05260	0.39457	0.03
3	0.052318	0.056366	0.54562	10.68	438.2	0.13649	0.058164	10.853	1.02790	0.61173	0.05
4	0.000992	0.49712	0.12316	1.3036	-71.398	0	0.001007	1.0116	1.29210	0.50288	0.0

In [23]:

```
print(data.head())
```

```
         X1        X2       X3       X4       X5        X6        X7        X8
X9      X10       X11       X12       X13       X14      X15       X16      X17
X18       X19      X20  \
0  0.025417  0.417690  0.05680   1.1605 -126.390  0.413550  0.025417   1.23950
1.16500  0.51773  0.025417  0.071822  0.042589  0.025417  4553.60  0.080156   2.
3941  0.025417  0.032332   64.985
1 -0.023834  0.210100  0.50839   4.2374   22.034  0.058412 -0.027621   3.65790
0.98183  0.76855 -0.027621 -0.175880  0.011274 -0.027621  3138.40  0.116300   4.
7595 -0.027621 -0.012744   62.936
2  0.030515  0.446060  0.19569   1.5650   35.766  0.281960  0.039264   0.88456
1.05260  0.39457  0.039264  0.113360  0.056198  0.039264  2400.00  0.152090   2.
2418  0.039264  0.032526   24.253
3  0.052318  0.056366  0.54562  10.6800  438.200  0.136490  0.058164  10.85300
1.02790  0.61173  0.058164  1.031900  0.186870  0.058164   260.91  1.399000  17.
7410  0.058164  0.137840   60.675
4  0.000992  0.497120  0.12316   1.3036  -71.398  0.000000  0.001007   1.01160
1.29210  0.50288  0.017035  0.002484  0.040636  0.001007  3455.90  0.105620   2.
0116  0.001007  0.000780  104.830

       X21       X22       X23       X24      X25       X26       X27       X28
X29        X30       X31      X32       X33       X34       X35      X36         X
37      X38       X39       X40  \
0  1.09350  0.113860  0.032332  0.413550  0.51773  0.080156  1.687400  0.096384
4.8078  0.513970  0.032332  191.430   1.9067  0.272590  0.113860  0.79532    4.2
43700  0.58152  0.144830  0.464990
1  1.15070 -0.015089 -0.010996  0.077482  0.76855  0.134330 -0.068354  1.519500
4.8829  0.092599 -0.012744   25.965  14.0570 -0.071818 -0.015089  2.19900    5.4
97200  0.82161 -0.006962  0.077370
2  0.82243  0.038380  0.025278  0.353230  0.39457  0.132470  0.334650  0.427330
4.5257  0.345420  0.032526  110.230   3.3111  0.086042  0.038380  1.23330    4.6
32700  0.49426  0.031794  0.096624
3  0.94921  0.059565  0.123980  0.149690  0.61173  1.295300  1.451000  1.370900
4.7352 -0.990460  0.137840   50.118   7.2828  1.056700  0.059565  0.54580  100.1
85857  0.61173  0.141160  8.432300
4  0.97245  0.017035  0.000768  0.002107  0.10133  0.105590  1.062900  0.261370
4.0033  0.381750  0.012043  112.330   3.2495  2.651300 -0.025970  1.29210   15.8
90000  0.51280 -0.020100  0.017617

       X41       X42     X43      X44       X45      X46      X47       X48
X49        X50       X51       X52      X53      X54      X55       X56       X57
X58        X59      X60  \
0  0.11263  0.144830  114.28   49.297  0.181600  0.76501   75.710  0.105790  0.1
34580   0.98325  0.353890  0.524470  0.87853  0.98679   3649.0  0.141650  0.0490
94  0.85835  0.123220   5.6167
1  0.18686 -0.006962  110.01   47.079 -0.063774  1.85760   61.792 -0.067146 -0.0
30980   3.16710  0.157040  0.071138  2.29710  2.45570  38823.0 -0.018502 -0.0310
11  1.01850  0.069047   5.7996
2  0.21903  0.031794  153.78  129.530  0.380420  1.33340   25.528  0.009805  0.0
08122   1.21520  0.346370  0.302010  0.86161  1.07930   6565.2  0.049940  0.0773
37  0.95006  0.252660  15.0490
3  0.02309  0.141160  109.59   48.911  0.745850  9.43550   62.370  0.038873  0.0
92122  10.68000  0.056366  0.137310  1.53700  1.53700  29652.0  0.027178  0.0855
24  0.97282  0.000000   6.0157
4  0.24180  0.013184  147.36   42.531  0.002675  0.38879  102.760 -0.034462 -0.0
26672   1.06370  0.405620  0.307740  1.06720  1.08820   1241.0 -0.020100  0.0019
74  0.99925  0.019736   3.4819

      X61      X62      X63     X64  Bankrupt    Id
0  7.4042  164.310   2.2214  1.3340         0  4510
1  7.7529   26.446  13.8020  6.4782         0  3537
2  2.8179  104.730   3.4852  2.6361         0  3920
3  7.4626   48.756   7.4863  1.0602         0  1806
4  8.5820  114.580   3.1854  2.7420         0  1529
```

Pretty *unhelpful* !

What are these mysteriously named features ?

Description of attributes

```
Attribute Information:

Id Company Identifier
- X1 net profit / total assets
- X2 total liabilities / total assets
- X3 working capital / total assets
- X4 current assets / short-term liabilities
- X5 [(cash + short-term securities + receivables - short-term liabilities) /
(operating expenses - depreciation)] * 365
- X6 retained earnings / total assets
- X7 EBIT / total assets
- X8 book value of equity / total liabilities
- X9 sales / total assets
- X10 equity / total assets
- X11 (gross profit + extraordinary items + financial expenses) / total assets
- X12 gross profit / short-term liabilities
- X13 (gross profit + depreciation) / sales
- X14 (gross profit + interest) / total assets
- X15 (total liabilities * 365) / (gross profit + depreciation)
- X16 (gross profit + depreciation) / total liabilities
- X17 total assets / total liabilities
- X18 gross profit / total assets
- X19 gross profit / sales
- X20 (inventory * 365) / sales
- X21 sales (n) / sales (n-1)
- X22 profit on operating activities / total assets
- X23 net profit / sales
- X24 gross profit (in 3 years) / total assets
- X25 (equity - share capital) / total assets
- X26 (net profit + depreciation) / total liabilities
- X27 profit on operating activities / financial expenses
- X28 working capital / fixed assets
- X29 logarithm of total assets
- X30 (total liabilities - cash) / sales
- X31 (gross profit + interest) / sales
- X32 (current liabilities * 365) / cost of products sold
- X33 operating expenses / short-term liabilities
- X34 operating expenses / total liabilities
- X35 profit on sales / total assets
- X36 total sales / total assets
- X37 (current assets - inventories) / long-term liabilities
- X38 constant capital / total assets
- X39 profit on sales / sales
- X40 (current assets - inventory - receivables) / short-term liabilities
- X41 total liabilities / ((profit on operating activities + depreciation) *
(12/365))
- X42 profit on operating activities / sales
- X43 rotation receivables + inventory turnover in days
- X44 (receivables * 365) / sales
- X45 net profit / inventory
- X46 (current assets - inventory) / short-term liabilities
- X47 (inventory * 365) / cost of products sold
- X48 EBITDA (profit on operating activities - depreciation) / total assets
- X49 EBITDA (profit on operating activities - depreciation) / sales
- X50 current assets / total liabilities
- X51 short-term liabilities / total assets
- X52 (short-term liabilities * 365) / cost of products sold)
```

```
- X53 equity / fixed assets
- X54 constant capital / fixed assets
- X55 working capital
- X56 (sales - cost of products sold) / sales
- X57 (current assets - inventory - short-term liabilities) / (sales - gross profit
- depreciation)
- X58 total costs /total sales
- X59 long-term liabilities / equity
- X60 sales / inventory
- X61 sales / receivables
- X62 (short-term liabilities *365) / sales
- X63 sales / short-term liabilities
- X64 sales / fixed assets
```

This may still be somewhat unhelpful for those of you not used to reading Financial Statements.

But that's partially the point of the exercise

- You can *still* perform Machine Learning *even if* you are not an expert in the problem domain
  - That's what makes this a good interview exercise: you can demonstrate your thought process even if you don't know the exact meaning of the terms
- Of course: becoming an expert in the domain *will improve* your ability to create better models
  - Feature engineering is easier if you understand the features, their inter-relationships, and the relationship to the target

Let's get a feel for the data

- What is the type of each attribute ?

In [11]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 4818 entries, 0 to 4817
Data columns (total 66 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   X1        4818 non-null   object
 1   X2        4818 non-null   object
 2   X3        4818 non-null   object
 3   X4        4818 non-null   object
 4   X5        4818 non-null   object
 5   X6        4818 non-null   object
 6   X7        4818 non-null   object
 7   X8        4818 non-null   object
 8   X9        4818 non-null   float64
 9   X10       4818 non-null   object
 10  X11       4818 non-null   object
 11  X12       4818 non-null   object
 12  X13       4818 non-null   float64
 13  X14       4818 non-null   object
 14  X15       4818 non-null   object
 15  X16       4818 non-null   object
 16  X17       4818 non-null   object
 17  X18       4818 non-null   object
 18  X19       4818 non-null   float64
 19  X20       4818 non-null   float64
 20  X21       4818 non-null   object
 21  X22       4818 non-null   object
 22  X23       4818 non-null   float64
 23  X24       4818 non-null   object
 24  X25       4818 non-null   object
 25  X26       4818 non-null   object
 26  X27       4818 non-null   object
 27  X28       4818 non-null   object
 28  X29       4818 non-null   object
 29  X30       4818 non-null   float64
 30  X31       4818 non-null   float64
 31  X32       4818 non-null   object
 32  X33       4818 non-null   object
 33  X34       4818 non-null   object
 34  X35       4818 non-null   object
 35  X36       4818 non-null   object
 36  X37       4818 non-null   object
 37  X38       4818 non-null   object
 38  X39       4818 non-null   float64
 39  X40       4818 non-null   object
 40  X41       4818 non-null   object
 41  X42       4818 non-null   float64
 42  X43       4818 non-null   float64
 43  X44       4818 non-null   float64
 44  X45       4818 non-null   object
 45  X46       4818 non-null   object
 46  X47       4818 non-null   object
 47  X48       4818 non-null   object
 48  X49       4818 non-null   float64
 49  X50       4818 non-null   object
 50  X51       4818 non-null   object
 51  X52       4818 non-null   object
 52  X53       4818 non-null   object
 53  X54       4818 non-null   object
 54  X55       4818 non-null   float64
 55  X56       4818 non-null   float64
 56  X57       4818 non-null   object
 57  X58       4818 non-null   float64
 58  X59       4818 non-null   object
 59  X60       4818 non-null   object
```

```
 60  X61        4818 non-null   object
 61  X62        4818 non-null   float64
 62  X63        4818 non-null   object
 63  X64        4818 non-null   object
 64  Bankrupt   4818 non-null   int64
 65  Id         4818 non-null   int64
dtypes: float64(16), int64(2), object(48)
memory usage: 2.4+ MB
```

You may be puzzled:

- Most attributes are `object` and *not* numeric (`float64`)
- But looking at the data via `data.head()` certainly gives the impression that all attributes are numeric

Welcome to the world of messy data ! The dataset has represented numbers as strings.

- These little unexpected challenges are common in the real-word
- Data is rarely perfect and clean

So you might want to first convert all attributes to numeric

Hint

- Look up the Pandas method `to_numeric`
  - We suggest you use the option `errors='coerce'`

Evaluating your project

In [12]:
```
holdout_data = pd.read_csv( os.path.join(DATA_DIR, "holdout", '5th_yr.csv') )

print("Data shape: ", holdout_data.shape)
```

```
Data shape:  (1092, 65)
```

We will evaluate your model on the holdout examples using metrics

- Accuracy
- Recall
- Precision

From our lecture: we may have to make a trade-off between Recall and Precision.

Our evaluation of your submission will be partially based on how you made (and described) the trade-off.

You may assume that it is 5 times worse to *fail to identify a company that will go bankrupt* than it is to fail to identify a company that won't go bankrupt.

Submission guidelines

Although your notebook may contain many models (e.g., due to your iterative development) we will only evaluate a single model. So choose one (explain why !) and do the following.

- You will implement the body of a subroutine `MyModel`
  - That takes as argument a Pandas DataFrame
    - Each row is an example on which to predict

    - The features of the example are elements of the row
  - Performs predictions on each example
  - Returns an array or predictions with a one-to-one correspondence with the examples in the test set

We will evaluate your model against the holdout data

- By reading the holdout examples `X_hold` (as above)
- Calling `y_hold_pred = MyModel(X_hold)` to get the predictions
- Comparing the predicted values `y_hold_pred` against the true labels `y_hold` which are known only to the instructors

See the following cell as an illustration:

X_hold = pd.read_csv(os.path.join(DATA_DIR, "holdout", '5th_yr.csv'))

Predict using MyModel

y_hold_pred = MyModel(X_hold)

Compute metrics

accuracy

accuracy_hold = accuracy_score(y_hold, y_hold_pred)

recall_

recall_hold = recall_score(y_hold, y_hold_pred, pos_label=1, average="binary")

precision

precision_hold = precision_score(y_hold, y_hold_pred, pos_label=1, average="binary")

print("\t{m:s} Accuracy: {a:3.1%}, Recall {r:3.1%}, Precision {p:3.1%}".format(m=name, a=accuracy_hold, r=recall_hold, p=precision_hold))

In [13]:

```
from sklearn.decomposition import PCA
from sklearn import svm
from sklearn.metrics import accuracy_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_predict, train_test_split
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn import tree
import math
```

In [22]:

```python
# Convert all data values to numeric, coercing errors to NaN
data = data.applymap(pd.to_numeric, errors='coerce')
holdout_data = holdout_data.applymap(pd.to_numeric, errors='coerce')

# Replace '?' with NaN to handle missing values
data.replace('?', np.nan, inplace=True)
data.fillna(data.mean(), inplace=True)

# Fill missing values with column-wise mean
holdout_data.replace('?', np.nan, inplace=True)
holdout_data.fillna(holdout_data.mean(), inplace=True)

# Convert DataFrame to NumPy array
df = data.values
out = holdout_data.values

# Display summary statistics of the dataset

data.describe()

# Extract features (all columns except last two) and normalize them
X=df[:, :-2]
X = StandardScaler().fit_transform(X) # Standardize features for ML models

# Extract target variable (second last column)
y=df[:, -2]
# Alternative feature and target extraction using Pandas indexing
X_data=data.iloc[:,:-2]
y_data=data.iloc[:, -2]
```

Out[22]:

	X1	X2	X3	X4	X5	X6	X7
count	4818.000000	4818.000000	4818.000000	4818.000000	4.818000e+03	4818.000000	4818.000000
mean	-0.055232	0.533272	0.188992	4.978602	1.956441e+01	-0.070060	-0.042516
std	6.705958	1.202660	1.282164	100.117705	2.382015e+04	7.776908	6.706577
min	-463.890000	0.000000	-72.067000	0.000000	-1.076400e+06	-463.890000	-463.890000
25%	0.004042	0.254765	0.044978	1.101525	-4.290075e+01	0.000000	0.005977
50%	0.046428	0.451610	0.218155	1.645450	4.932050e-01	0.000000	0.056653
75%	0.116725	0.662140	0.420033	2.943000	4.978025e+01	0.110387	0.135972
max	2.352300	72.416000	28.336000	6845.800000	1.250100e+06	203.150000	2.352300

In [21]:

```
pd.set_option('display.max_columns', None)
pd.set_option('display.expand_frame_repr', True)
pd.set_option('display.width', 200)
print(data.describe())
```

```
                X1            X2            X3            X4            X5
X6            X7            X8            X9           X10           X11           X12
X13           X14  \
count  4818.000000  4818.000000  4818.000000  4818.000000  4.818000e+03  4818.00
0000  4818.000000  4818.000000  4818.000000  4818.000000  4818.000000  4818.0000
00  4818.000000  4818.000000
mean     -0.055232     0.533272     0.188992     4.978602  1.956441e+01    -0.07
0060    -0.042516     5.739985     1.579277     0.503606    -0.024637     1.2183
57     0.452284    -0.042485
std       6.705958     1.202660     1.282164   100.117705  2.382015e+04     7.77
6908     6.706577   109.348749     1.342723     4.043196     6.706133    38.7865
94    34.196231     6.706578
min    -463.890000     0.000000   -72.067000     0.000000 -1.076400e+06  -463.89
0000  -463.890000    -3.735100     0.000191   -71.444000  -463.890000   -96.2390
00  -310.340000  -463.890000
25%       0.004042     0.254765     0.044978     1.101525 -4.290075e+01     0.00
0000     0.005977     0.482350     1.015600     0.319095     0.016183     0.0178
81     0.024954     0.005977
50%       0.046428     0.451610     0.218155     1.645450  4.932050e-01     0.00
0000     0.056653     1.154350     1.140500     0.522195     0.071071     0.1686
85     0.067723     0.056685
75%       0.116725     0.662140     0.420033     2.943000  4.978025e+01     0.11
0387     0.135972     2.814600     1.814050     0.721670     0.147890     0.5470
95     0.134847     0.136055
max       2.352300    72.416000    28.336000  6845.800000  1.250100e+06   203.15
0000     2.352300  6868.500000    37.807000   266.860000     2.352300  2470.3000
00  2340.200000     2.352300

                X15           X16           X17           X18           X19
X20           X21           X22           X23           X24           X25           X2
6             X27           X28  \
count  4.818000e+03  4818.000000  4818.000000  4818.000000  4818.000000  4818.00
0000  4818.000000  4818.000000  4818.000000  4818.000000  4818.000000  4818.0000
00    4818.000000  4818.000000
mean   2.768033e+03     1.261838     6.840618    -0.031071    -0.082028    57.00
2168     2.740360    -0.015683    -0.090357     0.137708     0.371927     1.1621
32     381.584380     6.822847
std    3.561790e+04    40.952597   109.355403     6.753507     5.754879   182.01
8911   110.389839     6.228986     5.725258     8.040588     4.109756    38.8129
95    9862.743624   108.277557
min    -4.667700e+05   -52.440000     0.000000  -463.890000  -310.800000   -29.34
0000  -135.150000  -431.590000  -310.890000  -463.890000   -71.444000   -52.4540
00 -158130.000000 -1089.700000
25%    2.492100e+02     0.076804     1.509850     0.005977     0.004368    18.53
7750     0.992880     0.000668     0.002728     0.026527     0.180500     0.0703
24       0.145345     0.091070
50%    8.972850e+02     0.236445     2.214850     0.056685     0.035307    38.62
3000     1.122300     0.062017     0.030074     0.154970     0.425990     0.2108
80       1.162200     0.543290
75%    2.311825e+03     0.638502     3.926975     0.136055     0.088287    66.85
0750     1.274500     0.136815     0.075800     0.350295     0.638653     0.5809
82       6.267475     1.732375
max    1.341700e+06  2837.400000  6869.500000    55.125000    77.244000  9928.50
0000  7661.500000    15.541000    77.244000   252.340000   266.860000  2689.1000
00  565940.000000  5534.000000

               X29           X30           X31           X32           X33
X34           X35           X36           X37           X38           X39            X
40           X41           X42  \
count  4818.000000  4818.000000  4818.000000  4.818000e+03  4818.000000  4818.00
0000  4818.000000  4818.000000   4818.000000  4818.000000  4818.000000  4818.000
000  4818.000000  4818.000000
mean      4.161201     0.691991    -0.068612  2.074823e+03     8.520843     5.07
4955    -0.017852     2.075176   100.185857     0.589316     0.019158     2.225
580     2.556513    -0.014902
```

```
std        0.829921    11.217691      5.748967  8.728196e+04   112.335444    109.97
3338     6.227517    14.101975    899.774590      4.035992     0.756891    63.131
677    91.177510     2.617310
min        0.006359   -23.060000  -310.800000  0.000000e+00     0.000000    -16.01
5000  -431.590000     0.000191     -4.325800    -71.444000   -47.047000    -0.335
160  -269.990000  -143.520000
25%        3.656250     0.085560      0.007006  5.013275e+01     2.782500      0.24
9400     0.008295     1.063300      2.846125      0.433462     0.005536     0.053
029     0.031838     0.000687
50%        4.175200     0.225955      0.042826  8.197500e+01     4.488750      1.71
4700     0.061804     1.552850     56.770000      0.620230     0.040252     0.183
200     0.094913     0.040963
75%        4.669700     0.406145      0.098331  1.321750e+02     7.298550      3.97
7275     0.138695     2.232275    100.185857      0.774872     0.091854     0.674
010     0.217020     0.090831
max        9.698300   656.450000     77.244000  4.277200e+06  7590.500000  7590.50
0000    15.541000   965.660000  40559.000000    266.860000     2.901100  4303.200
000  5043.300000    40.386000

                X43           X44            X45          X46            X47
X48           X49          X50          X51          X52          X53          X5
4           X55          X56  \
count   4818.000000   4818.000000    4818.000000  4818.000000    4818.000000  48
18.000000  4818.000000  4818.000000  4818.000000  4818.000000  4818.000000  481
8.000000  4.818000e+03  4818.000000
mean     155.612840     98.610765      81.795906     4.103433     153.579739
-0.103035    -0.072253     4.192413     0.426457     0.817538     8.089876
9.142221  1.074023e+04     0.056109
std      795.989622    725.594072    5273.616963   100.015679    3617.994648
7.829928     2.638443    99.598952     1.180774    14.189417   144.828893   160.
361029  8.284685e+04     0.755462
min    -3975.600000  -3946.200000   -3037.300000    -0.108060     -18.658000  -5
42.560000  -144.800000     0.000000     0.000000     0.000000 -1088.700000 -108
8.700000 -1.118500e+06   -46.788000
25%       76.214500     39.050250       0.026996     0.646130      19.849250
-0.030394    -0.022447     0.834400     0.186087     0.137313     0.764355
1.001350  9.771450e+01     0.011478
50%      106.670000     59.014000       0.291190     1.066600      42.389500
0.019769     0.012481     1.289150     0.328945     0.223895     1.296650     1.
440100  1.829500e+03     0.053663
75%      149.365000     86.087750       0.943597     2.000750      74.090750
0.095739     0.060499     2.372050     0.519905     0.360503     2.439775     2.
576300  7.786950e+03     0.124030
max    40515.000000  40515.000000  366030.000000  6845.800000  185610.000000
15.541000    16.866000  6845.800000    72.416000   666.110000  8309.600000  830
9.600000  4.212200e+06     1.000000

               X57          X58          X59           X60          X61
X62          X63           X64     Bankrupt           Id
count  4818.000000  4818.000000  4818.000000  4.818000e+03  4818.000000    4818.
000000  4818.000000   4818.000000  4818.000000  4818.000000
mean      0.022793     0.959585     0.273025  1.108795e+03    11.021303     177.
494445     9.287631    38.557533     0.063927  3499.858032
std       7.247517     0.932427     6.337285  6.942383e+04    43.766529    2279.
713700   113.049493   583.617986     0.244648  1392.049260
min    -468.670000    -0.085920  -184.980000 -1.244000e+01    -0.092493       0.
000000     0.000000    -3.726500     0.000000  1071.000000
25%       0.015582     0.876940     0.000000  5.456550e+00     4.236425      45.
065750     3.077000     2.137425     0.000000  2296.250000
50%       0.108860     0.950825     0.006365  9.449650e+00     6.181450      73.
879500     4.939500     4.224000     0.000000  3500.500000
75%       0.240200     0.990358     0.208242  1.968150e+01     9.342925     118.
597500     8.097875     9.834275     0.000000  4704.750000
max      87.981000    47.788000   308.150000  4.818700e+06  1308.500000  127450.
000000  7641.300000  28999.000000     1.000000  5909.000000
```

Handling Missing Data:

- Linear Regression Imputation: for data which shows linear trends

```
Train a linear regression model on the available data
Predict missing values using the trained model
Replace missing values with predicted values
```

- Interpolation (when missing is between values)

```
For each missing value:
Find the nearest known data points (x1, y1) and (x2, y2)
Compute slope: m = (y2 - y1) / (x2 - x1)
Estimate missing value: y = y1 + m * (x - x1)
Replace missing value with estimated y
```

- Extrapolation (time-series forecasting)

```
Select the two most recent known data points (x1, y1) and (x2, y2)
Compute slope: m = (y2 - y1) / (x2 - x1)
For each missing value:
Predict missing value: y = y2 + m * (x - x2)
Replace missing value with estimated y
```

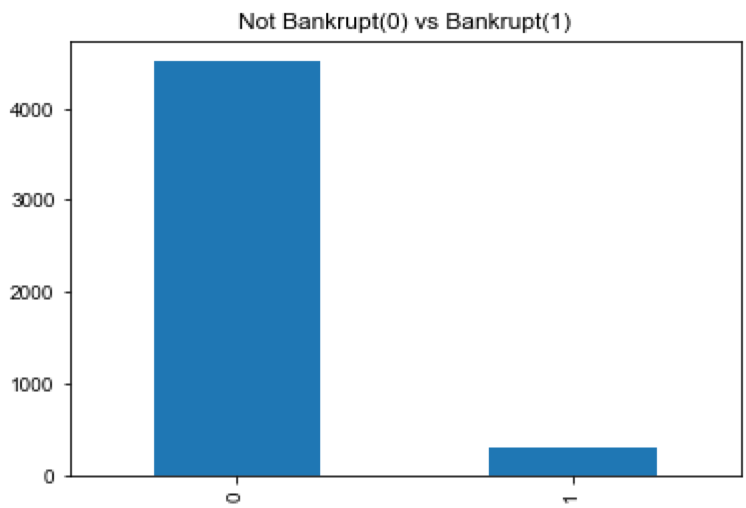
Can be used in ECM to find missing values

Visualize data, and find that the data is imbalanced, and the features have high correlation.

In [9]:
```
import seaborn as sns
y_data.value_counts().plot(kind='bar')
sns.set_style('whitegrid')
sns.set_palette('bwr')
plt.title('Not Bankrupt(0) vs Bankrupt(1)')
plt.show()
```

Out[9]: <AxesSubplot:>

Out[9]: Text(0.5, 1.0, 'Not Bankrupt(0) vs Bankrupt(1)')

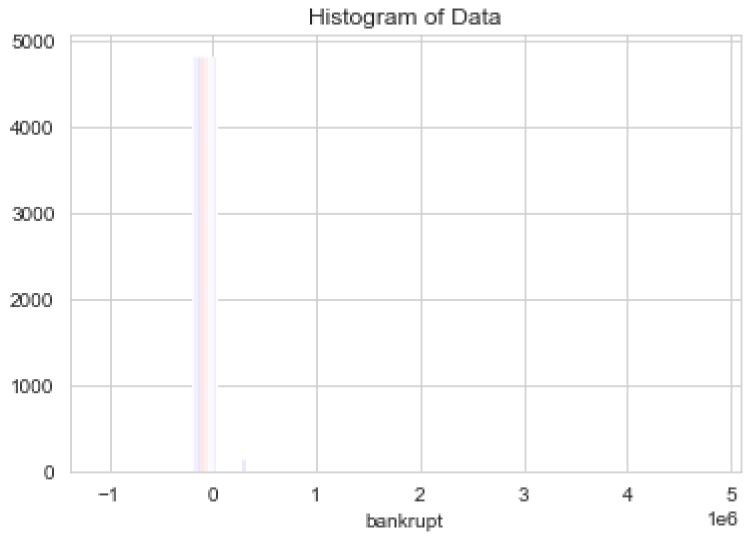


In [10]:
```
plt.hist(data, bins=20, alpha=0.5)
plt.xlabel('bankrupt')
plt.title('Histogram of Data')
plt.show()
```

Out[10]:
```
(array([[0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        ...,
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]]),
 array([-1118500.,  -821640.,  -524780.,  -227920.,    68940.,   365800.,
          662660.,   959520.,  1256380.,  1553240.,  1850100.,  2146960.,
         2443820.,  2740680.,  3037540.,  3334400.,  3631260.,  3928120.,
         4224980.,  4521840.,  4818700.]),
 <a list of 66 BarContainer objects>)
```

Out[10]:
```
Text(0.5, 0, 'bankrupt')
```

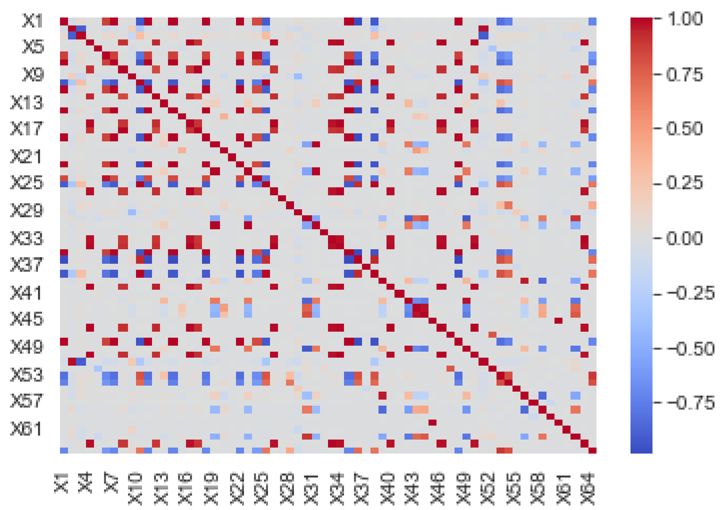
Out[10]:
```
Text(0.5, 1.0, 'Histogram of Data')
```



Heatmap

In [11]:
```
corr_matrix = X_data.corr()
sns.heatmap(corr_matrix, cmap='coolwarm')
plt.show()
```

Out[11]:
```
<AxesSubplot:>
```



Here I use randomoversampler to do random oversampling, and use PCA to lower the dimenision since we have dataset with features having high correlation with each other.

In [12]:
```
from imblearn.over_sampling import RandomOverSampler
# Random oversampling
# ros = RandomOverSampler(random_state=42)
# X_ros, y_ros = ros.fit_resample(X, y)
```

SMOTE

I also try SMOTE to balance the data to avoid overfitting. After applying SMOTE we can find the recall and precision seems to be balanced.

In [13]:
```
from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_ros, y_ros = smote.fit_resample(X, y)
# X_ros, y_ros = X, y
```

PCA

In [14]:
```
# Deciding the number of components for PCA
temp = []
n_components = np.arange(3,30)
for n in n_components:
    pca = PCA(n_components = n)
    pca.fit_transform(X_ros)
    total_variance = pca.explained_variance_ratio_.sum()
    temp.append(total_variance)
```

Out[14]:
```
array([[-0.34702704, -0.52941261, -0.12070452],
       [-0.21829935, -0.92479567, -0.04201231],
       [-0.33985283, -0.5397412 , -0.10783746],
       ...,
       [-0.33001267, -0.63287133, -0.10503995],
       [-0.64097052,  5.50196878,  0.01719124],
       [-0.4842435 ,  2.82453163, -0.05874565]])
```

Out[14]:
```
array([[-0.34702704, -0.5294126 , -0.12070446, -0.21890125],
       [-0.21829935, -0.92479567, -0.04201224, -0.08630365],
       [-0.33985283, -0.5397412 , -0.10783743, -0.16917515],
       ...,
       [-0.33001267, -0.63287133, -0.10503991, -0.14392445],
       [-0.64097052,  5.50196878,  0.01719121, -0.33927047],
       [-0.4842435 ,  2.82453162, -0.05874572,  0.33723751]])
```

Out[14]:
```
array([[-0.34702704, -0.5294126 , -0.12070438, -0.21890152, -0.09077936],
       [-0.21829935, -0.92479567, -0.04201216, -0.08630406, -0.11167949],
       [-0.33985283, -0.5397412 , -0.1078374 , -0.16917529, -0.09422986],
       ...,
```

In [15]:
```
sns.set_style('whitegrid')
sns.set_palette('bwr')
plt.figure(figsize=(10,10))
plt.plot(n_components, temp)
plt.xlabel('Number of components')
plt.ylabel('Total explained variance')
plt.title('Total explained variance per components')
plt.show()
```

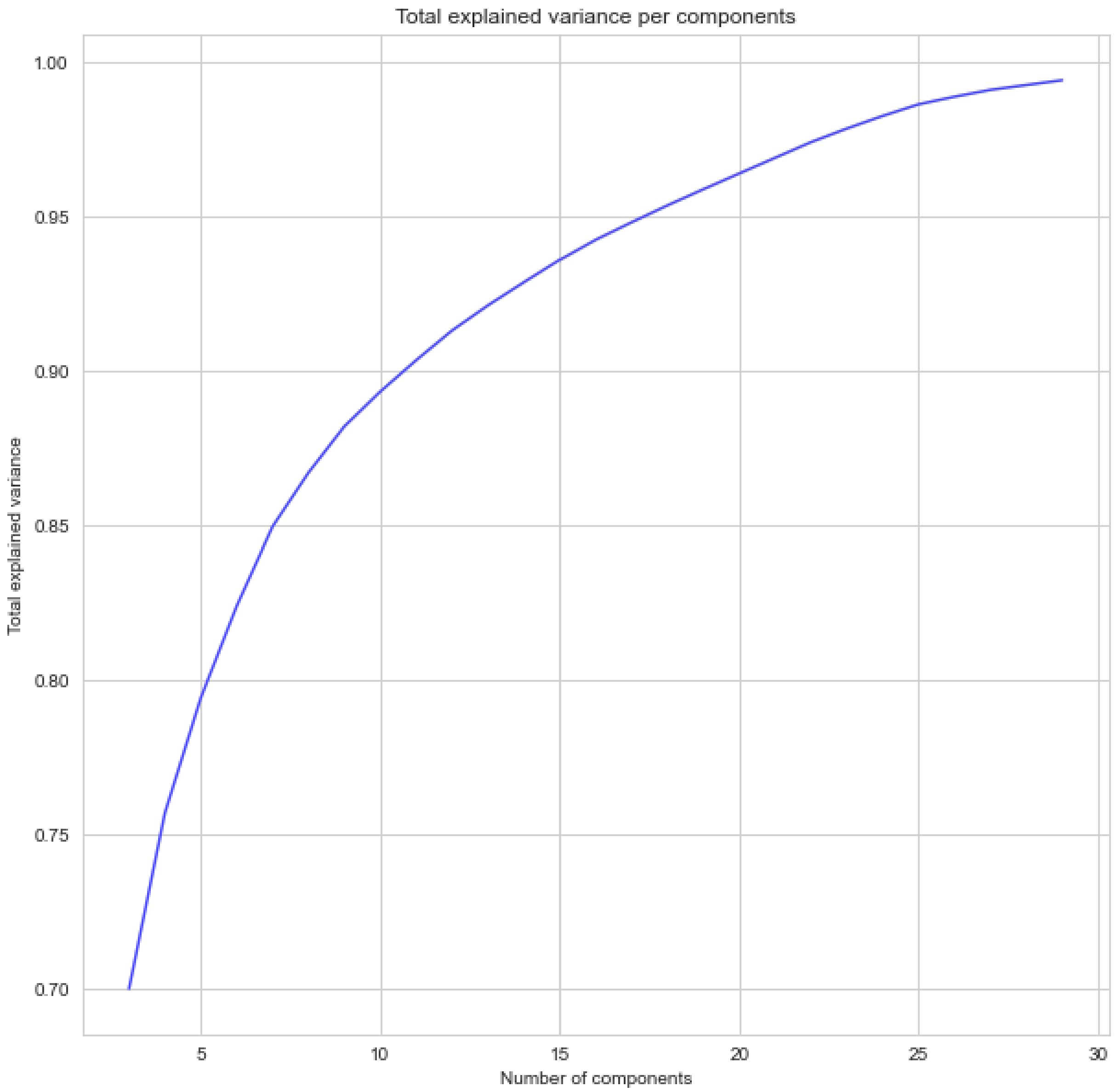
Out[15]: <Figure size 720x720 with 0 Axes>

Out[15]: [<matplotlib.lines.Line2D at 0x16cd9d30fd0>]

Out[15]: Text(0.5, 0, 'Number of components')

Out[15]: Text(0, 0.5, 'Total explained variance')

Out[15]: Text(0.5, 1.0, 'Total explained variance per components')



1. Create an empty list to store explained variance
2. Loop through different numbers of principal components
   - a. Initialize PCA with 'n' components
   - b. Fit PCA to the dataset and transform it
   - c. Calculate the total explained variance
   - d. Store the explained variance in the list
3. Plot the number of components against the total explained variance

In ECM, PCA can be used to identify key risk factors affecting economic capital.

Stress Testing: PCA can identify major risk drivers that contribute to capital adequacy.

Portfolio Risk Management: Helps group correlated financial metrics, improving model robustness.

We can see from the plot that the first 15 components have already explained 92% variance.

In [16]:
```
scaler = StandardScaler()
X_scaled_ros = scaler.fit_transform(X_ros)
pca = PCA(n_components=15)
X_pca_ros = pca.fit_transform(X_scaled_ros)
```

SVM

Since gridsearch takes quite a long time, I use randomsearch instead.

In [17]:
```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import uniform
from sklearn import svm
from sklearn.svm import SVC

# Define the hyperparameter search space for SVM
param_dist = {'C': [1, 10, 20], # Regularization Strength, control trade-off
                                # between achieving a low error on training
                                # data and maintaining model generalization
              'gamma': [0.1, 0.5], # Kernel coefficient for RBF Kernel,
                                   # control influence of individual training
                                   # points on decision boundaries
              'kernel': ['linear', 'rbf']} # Choice of Kernel Function, determine
                                           # how the SVM separates data in
                                           # feature space

# Initialize an SVM model with a default linear kernel
svm = svm.SVC(kernel='linear')
n_iter_search = 10
random_search = RandomizedSearchCV(svm, param_distributions=param_dist,
                                   n_iter=n_iter_search, cv=5)
# random search helps find the best combination of hyperparameters for a
# machine learning model.
# Fit Randomized Search to the resampled dataset
random_search.fit(X_ros, y_ros)
print("Best hyperparameters: {}".format(random_search.best_params_))
print("Best score: {:.2f}".format(random_search.best_score_))
```

Out[17]:
```
RandomizedSearchCV(cv=5, estimator=SVC(kernel='linear'),
                   param_distributions={'C': [1, 10, 20], 'gamma': [0.1, 0.5],
                                        'kernel': ['linear', 'rbf']})

Best hyperparameters: {'kernel': 'rbf', 'gamma': 0.5, 'C': 20}
Best score: 0.95
```

In [18]:

```
from sklearn.model_selection import cross_val_score
from sklearn import svm
from sklearn.svm import SVC
print("SVM")

# Initialize an SVM model with specified hyperparameters
svm_model=SVC(C=20, gamma=0.5)
# Split dataset into training (90%) and testing (10%) sets
X_train, X_test, y_train, y_test = train_test_split(X_pca_ros, y_ros,
                                                    test_size=0.1,
                                                    random_state=42)
svm_model.fit(X_train, y_train)
svm_pred = svm_model.predict(X_test)
```

```
SVM
```

Out[18]:

```
SVC(C=20, gamma=0.5)
```

In [19]:

```
name = "SVM model"

# Evaluate model performance
accuracy_test = accuracy_score(y_test, svm_pred)
recall_test = recall_score(y_test, svm_pred, pos_label=1,
                           average="binary")
precision_test = precision_score(y_test,   svm_pred, pos_label=1,
                                 average="binary", zero_division=1)

print("\t{m:s} Accuracy: {a:3.1%}, Recall {r:3.1%}, Precision {p:3.1%}".format
      (m=name,
       a=accuracy_test,
       r=recall_test,
       p=precision_test)
         )
```

```
        SVM model Accuracy: 83.6%, Recall 84.1%, Precision 83.0%
```

1. Initialize SVM model with hyperparameters: (depends on cases)
    - C = 20 (higher regularization, reducing misclassification penalties)
    - gamma = 0.5 (higher sensitivity to individual data points)
2. Split dataset into training (90%) and testing (10%) sets.
3. Train SVM model using the training set.
4. Make predictions on the test set.
5. Compute model evaluation metrics:
    - Accuracy: Overall classification correctness
    - Recall: Measures how well bankrupt companies are identified
    - Precision: Measures correctness of bankruptcy predictions
6. Print model performance results.
7. Compute and visualize confusion matrix to analyze prediction errors.

For ECM:

Can be used to identify firms at higher financial risk

Adjust parameters of SVM (c and gamma) to improve accuracy of default probability (Apply RandomizedSearchCV)

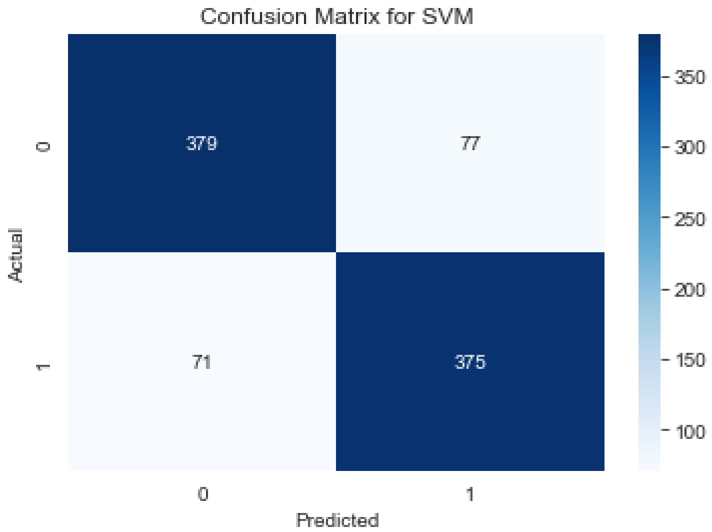
In [20]:
```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test, svm_pred)
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for SVM')
plt.show()
```

Out[20]: <AxesSubplot:>

Out[20]: Text(0.5, 16.0, 'Predicted')

Out[20]: Text(34.0, 0.5, 'Actual')

Out[20]: Text(0.5, 1.0, 'Confusion Matrix for SVM')



Logistic Regression

In [21]:

```python
from sklearn.linear_model import LogisticRegression

# Logistic Regression
print("Logistic Regression")

# Initialize Logistic Regression with balanced class weights to
# handle imbalance
log_reg = LogisticRegression(
    solver='lbfgs', # Solver for optimization
                    # (default, works well for small-medium datasets)
    max_iter=5000,  # Increase max iterations to ensure convergence
    class_weight='balanced',  # Adjusts weights to balance class distribution
    random_state=42
)

# Train the model on the training set
log_reg.fit(X_train, y_train)
# Predict on the test set
log_reg_pred = log_reg.predict(X_test)

# Evaluate model performance
accuracy_log_reg = accuracy_score(y_test, log_reg_pred)
recall_log_reg = recall_score(y_test, log_reg_pred, pos_label=1,
                              average="binary")
precision_log_reg = precision_score(y_test, log_reg_pred, pos_label=1,
                                    average="binary")

name = "Logistic Regression Model"
print("\t{m:s} Accuracy: {a:3.1%}, Recall {r:3.1%}, Precision {p:3.1%}".format(
    m=name, a=accuracy_log_reg, r=recall_log_reg, p=precision_log_reg
))

# Compute and visualize confusion matrix
cm = confusion_matrix(y_test, log_reg_pred)
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Logistic Regression')
plt.show()
```

```
Logistic Regression
```

Out[21]:

```
LogisticRegression(class_weight='balanced', max_iter=5000, random_state=42)
```

```
        Logistic Regression Model Accuracy: 75.4%, Recall 70.9%, Precision 77.5%
```

Out[21]:

```
<AxesSubplot:>
```

Out[21]:

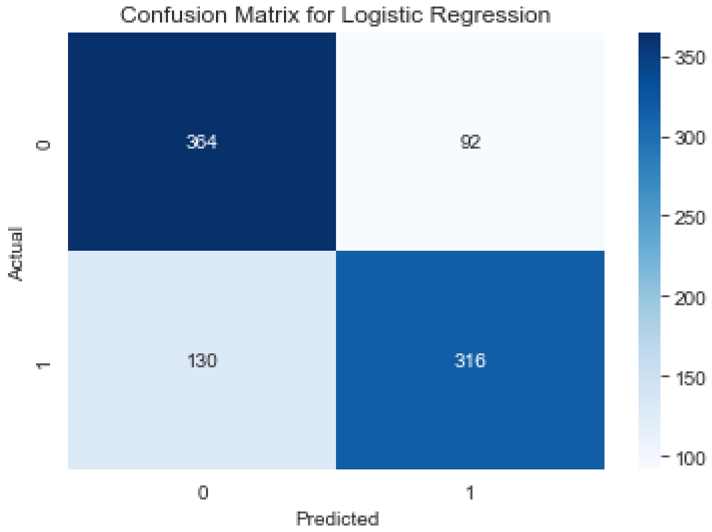
```
Text(0.5, 16.0, 'Predicted')
```

Out[21]:

```
Text(34.0, 0.5, 'Actual')
```

Out[21]:

```
Text(0.5, 1.0, 'Confusion Matrix for Logistic Regression')
```



1. Initialize Logistic Regression with:
   - 'lbfgs' solver for optimization
   - max_iter=5000 to prevent convergence issues
   - class_weight='balanced' to handle imbalanced bankruptcy data
2. Train the model using training data (X_train, y_train).
3. Predict bankruptcy probabilities for test data (X_test).
4. Compute model performance metrics:
   - Accuracy: Overall model correctness
   - Recall: Ensures bankrupt companies are correctly identified
   - Precision: Measures correctness of bankruptcy predictions
5. Print model evaluation results.
6. Compute confusion matrix to analyze classification errors.
7. Visualize confusion matrix using a heatmap.

For ECM:

Interpretable Model: Coefficients represent the impact of financial ratios on bankruptcy probability.

Helps in probability-based economic capital allocation.

Neural network model

In [22]:
```
from sklearn.metrics import classification_report
print("Neural network")
print("iteration : 50000")
logistic_activation = MLPClassifier(
    activation = 'logistic', # Uses sigmoid activation function
    hidden_layer_sizes = (90,90,3), # Three-layer neural network structure
    solver = "sgd", # Uses stochastic gradient descent optimizer
    random_state=1, max_iter=50000)

# Train the neural network model
logistic_activation.fit(X_train, y_train)
# Predict on the test set
logis_pred = logistic_activation.predict(X_test)
score_logis = accuracy_score(logis_pred, y_test)
print("sigmoid activation function test accuracy:", score_logis)
```

```
Neural network
iteration : 50000
```

Out[22]:
```
MLPClassifier(activation='logistic', hidden_layer_sizes=(90, 90, 3),
              max_iter=50000, random_state=1, solver='sgd')
```

```
sigmoid activation function test accuracy: 0.4911308203991131
```

In [23]:
```
name = "Neural network model"

# Evaluate model performance
accuracy_test = accuracy_score(y_test, logis_pred)
recall_test = recall_score(y_test, logis_pred, pos_label=1, average="binary")
precision_test = precision_score(y_test,   logis_pred, pos_label=1,
                                 average="binary", zero_division=1)

print("\t{m:s} Accuracy: {a:3.1%}, Recall {r:3.1%}, Precision {p:3.1%}".format
      (m=name,
       a=accuracy_test,
       r=recall_test,
       p=precision_test)
         )
```

```
        Neural network model Accuracy: 49.1%, Recall 99.3%, Precision 49.3%
```

1. Initialize a Neural Network (MLPClassifier) with:
    - Sigmoid activation function ('logistic')
    - Three hidden layers: 90, 90, 3 neurons
    - Stochastic Gradient Descent (SGD) optimizer
    - max_iter=50000 to ensure full convergence
2. Train the model using training data (X_train, y_train).
3. Predict bankruptcy outcomes for test data (X_test).
4. Compute performance metrics:
    - Accuracy: Overall correctness of predictions
    - Recall: Ability to correctly classify bankrupt companies
    - Precision: Accuracy of bankruptcy predictions
5. Print model evaluation results.

For ECM:

Handles nonlinear relationship in financial data

Sigmoid activation (logistic) is useful for probabilistic bankruptcy classification

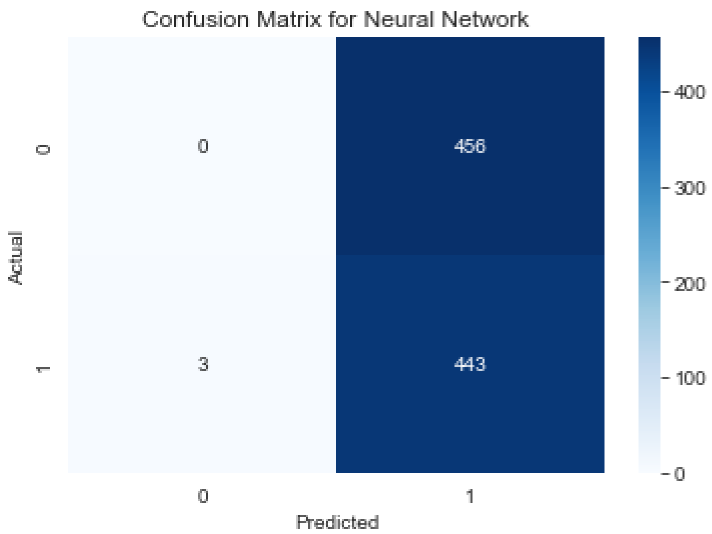
In [24]:
```
cm = confusion_matrix(y_test, logis_pred)
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Neural Network')
plt.show()
```

Out[24]: <AxesSubplot:>

Out[24]: Text(0.5, 16.0, 'Predicted')

Out[24]: Text(34.0, 0.5, 'Actual')

Out[24]: Text(0.5, 1.0, 'Confusion Matrix for Neural Network')



Decision Tree

In [25]:
```
import pandas as pd
import os

def MyModel(X):
    model = tree.DecisionTreeClassifier(
    max_depth=10, # Limit tree depth to avoid excessive complexity
    min_samples_split=10, # Require at least 10 samples to split a node
    min_samples_leaf=5, # Require at least 5 samples per leaf node
    random_state=42)
    model = model.fit(X_train, y_train)

    predictions = model.predict(X)
    return predictions
```

In [26]:
```
#random_out=holdout_data.iloc[:, :-1]
#X_test=random_out.sample(n=482,axis='rows')
```

In [27]:

```
name = "My best model---Decision tree model: "
y_test_pred = MyModel(X_test)

# Evaluate model performance
accuracy_test = accuracy_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred, pos_label=1, average="binary")
precision_test = precision_score(y_test, y_test_pred, pos_label=1,
                                 average="binary")

print("\t{m:s} Accuracy: {a:3.1%}, Recall {r:3.1%}, Precision {p:3.1%}".format
      (m=name,
       a=accuracy_test,
       r=recall_test,
       p=precision_test)
        )
```

```
        My best model---Decision tree model:  Accuracy: 84.1%, Recall 87.4%, Pre
cision 81.8%
```

1. Initialize a Decision Tree classifier with:
   - max_depth=10 (prevent overfitting)
   - min_samples_split=10 (ensure splits happen on sufficiently large nodes)
   - min_samples_leaf=5 (ensure leaf nodes are not too small)
2. Train the decision tree model on the training dataset (X_train, y_train).
3. Predict bankruptcy status for given input data (X).
4. Compute and return predictions.
5. Test the model using X_test and evaluate:
   - Accuracy (percentage of correct predictions)
   - Recall (how well bankrupt firms are identified)
   - Precision (correctness of bankruptcy predictions)
6. Print model performance results.

For ECM: Easy to visualize decision rules, help to find whichi financial metrics drive bankruptcy risk

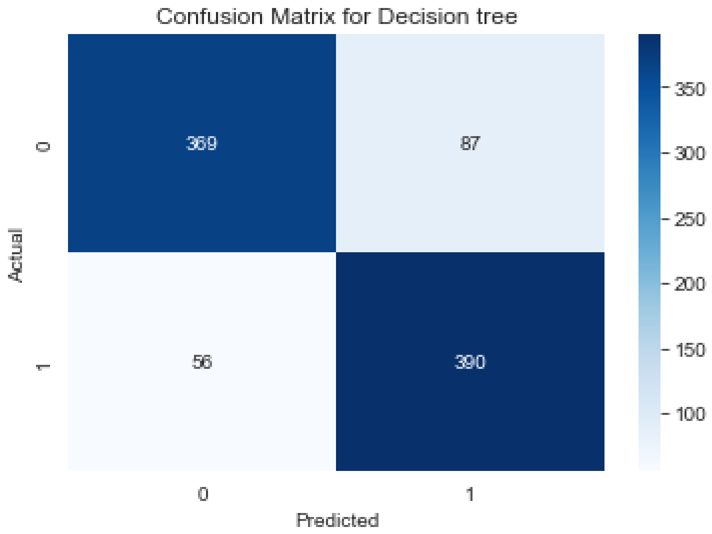
In [28]:
```python
cm = confusion_matrix(y_test, y_test_pred)
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Decision tree')
plt.show()
```

Out[28]: <AxesSubplot:>

Out[28]: Text(0.5, 16.0, 'Predicted')

Out[28]: Text(34.0, 0.5, 'Actual')

Out[28]: Text(0.5, 1.0, 'Confusion Matrix for Decision tree')



I choose decision tree as my best model as its recall and precision are more in line with my expectation. As we know that "You may assume that it is 5 times worse to fail to identify a company that will go bankrupt than it is to fail to identify a company that won't go bankrupt", it indicates that we should have the recall higher than precision when trying to make trade-off between Recall and Precision. I set the test size to be small in order to achive this goal which larger proportion of the data will be used for training the model. Thus the model has more training data to learn from and can better capture the patterns in the data that are associated with the positive class. However, SVM model and neural network model fail to achieve this goal. I try to increase the penalty parameter of the error term of SVM model "c", but even I increase to 20000 recall is still smaller than precision.

Decision Tree

Advantages:

- Easy to interpret and visualize.
- Captures non-linear relationships.
- Works well with small datasets and categorical variables.

Disadvantages:

- Prone to overfitting, especially with deep trees.
- Sensitive to noisy data.
- Not as effective for high-dimensional data without boosting or bagging.

Neural Network (MLP)

Advantages:

- Excellent for capturing complex, non-linear patterns.

- Works well with large datasets and high-dimensional data.
- Can model financial risk factors that interact in non-trivial ways.

Disadvantages:

- Hard to interpret (black-box model).
- Requires high computational power and tuning.
- Prone to overfitting if not properly regularized.

Logistic Regression

Advantages:

- Simple, interpretable, and easy to implement.
- Works well for linearly separable data.
- Computationally efficient, even for large datasets.

Disadvantages:

- Assumes linear relationships, which may not hold for complex financial data.
- Struggles with imbalanced data without resampling techniques.
- Less powerful compared to tree-based models or deep learning.

Support Vector Machine (SVM)

Advantages:

- Works well with small and medium-sized datasets.
- Effective for high-dimensional data.
- Can handle non-linear classification with the right kernel.

Disadvantages:

- Computationally expensive for large datasets.
- Hard to interpret compared to logistic regression or decision trees.
- Requires tuning of C and gamma to avoid overfitting or underfitting.

ECM:

Suggest for logistic regression (easy, interpretable) Neural network (For Complex Risk Patterns)

A single Decision Tree can easily memorize training data instead of learning general patterns (since variables for ECM interact in complex way)

SVM is quite expensive in large dataset