Using ML to Aid in SBA Loan Approvals

By Joshua Ireland

Objective

The goal of this project will be to create a machine learning model that will aid a banker in deciding if a business owner should or should not be approved for a Small Business Administration (SBA) loan based on the likelihood the loan will go into default.

Background

Small Business Administration Backed Loans, or SBA loans for short, is a lending product designed to help small business owners fund, expand, or continue the operations of their businesses (1). This program allows the federal government to guarantee a loan for the bank to offer to a small business owner. This is an important, win-win-win, program. It allows the federal government to directly invest into new businesses and the growth they bring. This program builds the bank's confidence to lend to a group that would normally be too risky to lend to thus securing itself a whole new client base. Finally, this allows the average business owner, who doesn't have a rich family or investor friends, to secure the financing they need to make their dreams come to fruition.

The use of a model like this could be twofold. For the banker to have a model to use to quickly estimate if a business owner is likely to pay on their loan and not go into default. I think this model could also be helpful in the hands of the business owner. Preparing to secure an SBA loan can be intimidating. The business owner needs to assemble a business plan, current financials, projected financials and other items to persuade the banker that you would grow your business and be able to pay it back faithfully (2). This model could then be used to help test if the business owner has a compelling case before meeting with the banker to secure the SBA loan.

I have a secondary objective from this model creation. Coming from a background of franchise consulting and seeing how SBA loans can be critical to securing a franchise I am interested in seeing what could help my clients prepare. In the process I am also hoping to learn how investing in a franchise versus starting a new business could lead to a better or worse approval rating for SBA loans.

My hypothesis is that securing an SBA loan to fund the purchase of a franchise will have a positive correlation with SBA loan approval. The U.S. Small Business Administration website even explains some of the advantages of buying a franchise or existing business over the creation of a new one, as well as some of its shortcomings (3). Likewise, I think the SBA loan officer in the local bank would be aware of the differences and see the advantage in lending to a franchisee over someone starting from scratch.

My goal is to gather data from the small business association database that I found in an article for the Journal of Statistics Education entitled "'Should This Loan be Approved or Denied?': A Large Dataset with Class Assignment Guidelines" (4). After that I plan to review the data thoroughly, clean, and prepare it for use in training a machine learning model. After the model is created I will create an interface for bankers approving SBA loans, and those interested in applying for them, can use it to make more educated decisions of which loans to approve and which to deny.

1. Loans. (n.d.). Retrieved from U.S. Small Business Administration: https://www.sba.gov/funding-programs/loans
2. Fund Your Business. (n.d.). Retrieved from U.S. Small Business Administration: https://www.sba.gov/business-guide/plan-your-business/fund-your-business
3. Buy an existing business or franchise. (n.d.). Retrieved from U.S. Small Business Administration: https://www.sba.gov/business-guide/plan-your-business/buy-existing-business-or-franchise
4. Min Li, Amy Mickel & Stanley Taylor (2018) "Should This Loan be Approved or Denied?": A Large Dataset with Class Assignment Guidelines, Journal of Statistics Education, 26:1, 55-66, DOI: 10.1080/10691898.2018.1434342

Data Description

The data was taken from Min Li, Amy Mickel & Stanley Taylor (2018) "Should This Loan be Approved or Denied?": A Large Dataset with Class Assignment Guidelines, Journal of Statistics Education, 26:1, 55-66, DOI: 10.1080/10691898.2018.1434342.

This data was collected from the SBA from the years 1987 to 2014.

Below is a table describing the different variables that I took from the article above:

In [1]:
```
#Code taken from this source: https://www.delftstack.com/howto/python/python-display-im
import IPython.display as display
from PIL import Image
display.display(Image.open('DataSBADescriptions.png'))
```

Variable name	Data type	Description of variable
LoanNr_ChkDgt	Text	Identifier: Primary key
Name	Text	Borrower name
City	Text	Borrower city
State	Text	Borrower state
Zip	Text	Borrower zip code
Bank	Text	Bank name
BankState	Text	Bank state
NAICS	Text	North American industry classification system code
ApprovalDate	Date/Time	Date SBA commitment issued
ApprovalFY	Text	Fiscal year of commitment
Term	Number	Loan term in months
NoEmp	Number	Number of business employees
NewExist	Text	1 = Existing business, 2 = New business
CreateJob	Number	Number of jobs created
RetainedJob	Number	Number of jobs retained
FranchiseCode	Text	Franchise code, (00000 or 00001) = No franchise
UrbanRural	Text	1 = Urban, 2 = rural, 0 = undefined
RevLineCr	Text	Revolving line of credit: Y = Yes, N = No
LowDoc	Text	LowDoc Loan Program: Y = Yes, N = No
ChgOffDate	Date/Time	The date when a loan is declared to be in default
DisbursementDate	Date/Time	Disbursement date
DisbursementGross	Currency	Amount disbursed
BalanceGross	Currency	Gross amount outstanding
MIS_Status	Text	Loan status charged off = CHGOFF, Paid in full = PIF
ChgOffPrinGr	Currency	Charged-off amount
GrAppv	Currency	Gross amount of loan approved by bank
SBA_Appv	Currency	SBA's guaranteed amount of approved loan

Common Imports

In [3]:

```
#Importing Intel Sklearn patch
from sklearnex import patch_sklearn
patch_sklearn()
```

```
Intel(R) Extension for Scikit-learn* enabled (https://github.com/intel/scikit-learn-inte
lex)
```

In [4]:

```
#Common Imports
import matplotlib.pyplot as plt
import matplotlib as mpl
from matplotlib import cm
import numpy as np
import pandas as pd
import time
%matplotlib inline
import os
from datetime import datetime
from sklearn import svm
from sklearn.impute import SimpleImputer
from sklearn.impute import KNNImputer
```

```python
from sklearn.preprocessing import OneHotEncoder
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import mean_squared_error
from sklearn.metrics import plot_confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.callbacks import EarlyStopping
from functools import reduce
import pickle
```

Load Data

In [3]:

```python
#Dataset 'SBAnational.csv' was taken from the Min Li, Amy Mickel & Stanley Taylor (2018
#“Should This Loan be Approved or Denied?”: A Large Dataset with Class Assignment Guide
#Journal of Statistics Education, 26:1, 55-66, DOI: 10.1080/10691898.2018.1434342
#URL for the article: https://doi.org/10.1080/10691898.2018.1434342

#Reading from the file saving to SBAdf for Small Business Administration Dataset
SBAdf = pd.read_csv("SBAnational.csv", low_memory=False)

#Examining our data
SBAdf
```

Out[3]:

	LoanNr_ChkDgt	Name	City	State	Zip	Bank	BankState	NAI
0	1000014003	ABC HOBBYCRAFT	EVANSVILLE	IN	47711	FIFTH THIRD BANK	OH	4511
1	1000024006	LANDMARK BAR & GRILLE (THE)	NEW PARIS	IN	46526	1ST SOURCE BANK	IN	7224
2	1000034009	WHITLOCK DDS, TODD M.	BLOOMINGTON	IN	47401	GRANT COUNTY STATE BANK	IN	6212
3	1000044001	BIG BUCKS PAWN & JEWELRY, LLC	BROKEN ARROW	OK	74012	1ST NATL BK & TR CO OF BROKEN	OK	
4	1000054004	ANASTASIA CONFECTIONS, INC.	ORLANDO	FL	32801	FLORIDA BUS. DEVEL CORP	FL	
...	...	...	...	...	...	...	...	

	LoanNr_ChkDgt	Name	City	State	Zip	Bank	BankState	NAI
899159	9995573004	FABRIC FARMS	UPPER ARLINGTON	OH	43221	JPMORGAN CHASE BANK NATL ASSOC	IL	4511
899160	9995603000	FABRIC FARMS	COLUMBUS	OH	43221	JPMORGAN CHASE BANK NATL ASSOC	IL	4511
899161	9995613003	RADCO MANUFACTURING CO.,INC.	SANTA MARIA	CA	93455	RABOBANK, NATIONAL ASSOCIATION	CA	3323
899162	9995973006	MARUTAMA HAWAII, INC.	HONOLULU	HI	96830	BANK OF HAWAII	HI	
899163	9996003010	PACIFIC TRADEWINDS FAN & LIGHT	KAILUA	HI	96734	CENTRAL PACIFIC BANK	HI	

899164 rows × 27 columns

Data Preprocessing

In [4]:

```
#Droping variables that will have no use to our end user, justification in the markdown
SBAdf2 = SBAdf.drop(['LoanNr_ChkDgt', 'Name', 'City', 'State', 'Zip', 'Bank', 'BankStat
```

Eliminated the following variables at this stage of analysis:

LoanNr_ChkDgt - This was the primary key for the dataset and is not needed since we have our own index and will not need to find specific loans in the future

Name - This is the name of the business who is borrowing the SBA backed loan. The name of the business won't have an impact on the final approval and can be dropped.

City - This is being dropped since the time it would take to encode it and the amount it would contribute to a federally backed loan decision. The work does not justify to payoff.

Zip - Same as City

State - Same as City

BankState - Same as City

Bank - This is being dropped because the final user interface will be for use by any banker or business owner, so knowing the approval rate of individual banks will not benefit our end user

ApprovalFY - This could impact the default rate depending on loans being approved during times of feast and famine. However, recessions don't always take entire years. So we will encode approvaldata and drop approvalfy

ChgOffDate - This column only contains the date for the loans that went into default. We only need to know which loans went into default for our model, we do not need the exact date it occured.

DisbursementDate - Knowing whether the loan was dispersed during a recession could be a good indicator of whether a loan will default. However, since this variable is based on when the loan was dispersed after approval, it won't be of use to our pre-approval end user

BalanceGross - Our end user won't be able to know the outstanding balance. It will not be relvant for approvals.

In [5]:

```
#Reviewing out data after the dropped variables
SBAdf2
```

Out[5]:

	NAICS	ApprovalDate	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	Urba
0	451120	28-Feb-97	84	4	2.0	0	0	1	
1	722410	28-Feb-97	60	2	2.0	0	0	1	
2	621210	28-Feb-97	180	7	1.0	0	0	1	
3	0	28-Feb-97	60	2	1.0	0	0	1	
4	0	28-Feb-97	240	14	1.0	7	7	1	
...	...	...	...	...	...	...	...	...	
899159	451120	27-Feb-97	60	6	1.0	0	0	1	
899160	451130	27-Feb-97	60	6	1.0	0	0	1	
899161	332321	27-Feb-97	108	26	1.0	0	0	1	
899162	0	27-Feb-97	60	6	1.0	0	0	1	
899163	0	27-Feb-97	48	1	2.0	0	0	1	

899164 rows × 15 columns

In [6]:

```
#Examing the datatypes of our dataframe to decide next steps
SBAdf2.dtypes
```

Out[6]:

```
NAICS                 int64
ApprovalDate         object
Term                  int64
NoEmp                 int64
NewExist            float64
CreateJob             int64
RetainedJob           int64
FranchiseCode         int64
UrbanRural            int64
RevLineCr            object
LowDoc               object
DisbursementGross    object
MIS_Status           object
GrAppv               object
```

```
SBA_Appv            object
dtype: object
```

In [7]:

```
#Transforming our currency amounts with type(object) to floats
SBAdf2['DisbursementGross'] = SBAdf2['DisbursementGross'].replace("\$|,","", regex=True
SBAdf2['DisbursementGross'] = pd.to_numeric(SBAdf2['DisbursementGross'])

SBAdf2['GrAppv'] = SBAdf2['GrAppv'].replace("\$|,","", regex=True)
SBAdf2['GrAppv'] = pd.to_numeric(SBAdf2['GrAppv'])

SBAdf2['SBA_Appv'] = SBAdf2['SBA_Appv'].replace("\$|,","", regex=True)
SBAdf2['SBA_Appv'] = pd.to_numeric(SBAdf2['DisbursementGross'])
```

Knowing whether the economy was in a boom or recession year when the loan was approved could be valuable information for predicting our target variable. However, just as a collection of years we cannot extrapulate much from it. So I will transfrom it into a categorical variable that represents whether the year was a boom or recession. Economic conditions was pulled from this source: https://www.nber.org/research/data/us-business-cycle-expansions-and-contractions

In [8]:

```
#Processing the variable 'ApprovalDate'
#Our ApprovalDate column has three letter month appreviations.
#To be able to interact with it better we will change the appreviation to the respectiv
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Jan','1')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Feb','2')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Mar','3')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Apr','4')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('May','5')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Jun','6')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Jul','7')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Aug','8')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Sep','9')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Oct','10')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Nov','11')
SBAdf2['ApprovalDate'] = SBAdf2['ApprovalDate'].str.replace('Dec','12')

#Converting our variable to the datetime type
SBAdf2['ApprovalDate'] = pd.to_datetime(SBAdf2['ApprovalDate'])
SBAdf2
```

Out[8]:

	NAICS	ApprovalDate	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	Urba
0	451120	1997-02-28	84	4	2.0	0	0	1	
1	722410	1997-02-28	60	2	2.0	0	0	1	
2	621210	1997-02-28	180	7	1.0	0	0	1	
3	0	1997-02-28	60	2	1.0	0	0	1	
4	0	1997-02-28	240	14	1.0	7	7	1	
...	...	...	...	...	...	...	...	...	
899159	451120	1997-02-27	60	6	1.0	0	0	1	
899160	451130	1997-02-27	60	6	1.0	0	0	1	
899161	332321	1997-02-27	108	26	1.0	0	0	1	

	NAICS	ApprovalDate	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	Urba
899162	0	1997-02-27	60	6	1.0	0	0	1	
899163	0	1997-02-27	48	1	2.0	0	0	1	

899164 rows × 15 columns

In [9]:

```
#Loading a dataset to view recession periods in US history
#Data sources from: https://www.nber.org/research/data/us-business-cycle-expansions-and
RDdf = pd.read_csv("Recessiondates.csv", low_memory=False)
RDdf['peak']= pd.to_datetime(RDdf['trough'])
RDdf
```

Out[9]:

	peak	trough
0	1854-12-01	1854-12-01
1	1858-12-01	1858-12-01
2	1861-06-01	1861-06-01
3	1867-12-01	1867-12-01
4	1870-12-01	1870-12-01
5	1879-03-01	1879-03-01
6	1885-05-01	1885-05-01
7	1888-04-01	1888-04-01
8	1891-05-01	1891-05-01
9	1894-06-01	1894-06-01
10	1897-06-01	1897-06-01
11	1900-12-01	1900-12-01
12	1904-08-01	1904-08-01
13	1908-06-01	1908-06-01
14	1912-01-01	1912-01-01
15	1914-12-01	1914-12-01
16	1919-03-01	1919-03-01
17	1921-07-01	1921-07-01
18	1924-07-01	1924-07-01
19	1927-11-01	1927-11-01
20	1933-03-01	1933-03-01
21	1938-06-01	1938-06-01
22	1945-10-01	1945-10-01
23	1949-10-01	1949-10-01

	peak	trough
24	1954-05-01	1954-05-01
25	1958-04-01	1958-04-01
26	1961-02-01	1961-02-01
27	1970-11-01	1970-11-01
28	1975-03-01	1975-03-01
29	1980-07-01	1980-07-01
30	1982-11-01	1982-11-01
31	1991-03-01	1991-03-01
32	2001-11-01	2001-11-01
33	2009-06-01	2009-06-01
34	2020-04-01	2020-04-01

In [10]:

```
#Using the dates from 'ApprovalDate' to create a new variable 'RecessionYN' this variab
#a 1 for "Recession Year" or 0 for "Not a Recession Year"
#Peak and end dates are taken from the dataset above
peak = 19600401
peak = pd.to_datetime(str(peak), format='%Y%m%d')
end = 19610201
end = pd.to_datetime(str(end), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak) & (SBAdf2['ApprovalDate'] <= end), 'Recessi

peak2 = 19691201
peak2 = pd.to_datetime(str(peak2), format='%Y%m%d')
end2 = 19701101
end2 = pd.to_datetime(str(end2), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak2) & (SBAdf2['ApprovalDate'] <= end2), 'Reces

peak3 = 19731101
peak3 = pd.to_datetime(str(peak3), format='%Y%m%d')
end3 = 19750301
end3 = pd.to_datetime(str(end3), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak3) & (SBAdf2['ApprovalDate'] <= end3), 'Reces

peak4 = 19800101
peak4 = pd.to_datetime(str(peak4), format='%Y%m%d')
end4 = 19800701
end4 = pd.to_datetime(str(end4), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak4) & (SBAdf2['ApprovalDate'] <= end4), 'Reces

peak5 = 19810701
peak5 = pd.to_datetime(str(peak5), format='%Y%m%d')
end5 = 19821101
end5 = pd.to_datetime(str(end5), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak5) & (SBAdf2['ApprovalDate'] <= end5), 'Reces

peak6 = 19900701
peak6 = pd.to_datetime(str(peak6), format='%Y%m%d')
end6 = 19910301
end6 = pd.to_datetime(str(end6), format='%Y%m%d')
```

```
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak6) & (SBAdf2['ApprovalDate'] <= end6), 'Reces

peak7 = 20010301
peak7 = pd.to_datetime(str(peak7), format='%Y%m%d')
end7 = 20011101
end7 = pd.to_datetime(str(end7), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak7) & (SBAdf2['ApprovalDate'] <= end7), 'Reces

peak7 = 20071201
peak7 = pd.to_datetime(str(peak7), format='%Y%m%d')
end7 = 20090601
end7 = pd.to_datetime(str(end7), format='%Y%m%d')
SBAdf2.loc[(SBAdf2['ApprovalDate'] >= peak7) & (SBAdf2['ApprovalDate'] <= end7), 'Reces

#Replacing our missing values with 0
#Code taken from: https://stackoverflow.com/questions/52835971/fill-nan-with-zero-pytho
SBAdf2['RecessionYN'] = pd.to_numeric(SBAdf2['RecessionYN'], errors='coerce').fillna(0)
SBAdf2['RecessionYN'] = SBAdf2['RecessionYN'].astype(int)
```

The Franchise Code is an important categorical variable for our analysis. However, we do not need to know what the specific franchise is for

In [11]:

```
#Reviewing our data processing to this point.  From examining the 'ApprovalDate' we kno
#171 was a loan approved during a recession.  If our processing worked the rest will sh
SBAdf2.iloc[[160,161,162,164,165,166,167,168,169,170,171,172,173,174,175,176,177,178,17
```

Out[11]:

	NAICS	ApprovalDate	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	UrbanRu
160	0	1997-12-06	84	10	1.0	0	0	1	
161	441120	1997-02-28	300	1	1.0	0	0	1	
162	541330	1997-12-06	84	5	1.0	0	0	1	
164	0	1997-12-06	84	3	1.0	0	0	1	
165	0	1997-12-06	12	2	1.0	0	0	1	
166	621210	1997-02-28	108	2	1.0	0	0	1	
167	422810	1997-02-28	102	10	1.0	0	0	1	
168	0	1997-06-13	84	9	1.0	0	0	1	
169	0	1997-02-28	180	1	1.0	0	0	1	
170	722410	1997-06-16	180	2	0.0	0	0	1	
171	0	1980-06-18	10	22	2.0	0	0	0	
172	541430	1997-06-17	12	1	1.0	0	0	1	
173	421810	1997-06-18	12	1	1.0	0	0	1	
174	0	1997-02-28	240	6	1.0	6	4	1	
175	0	1997-06-19	13	1	1.0	0	0	1	
176	0	1997-06-20	12	3	1.0	0	0	1	
177	512110	1997-06-20	12	3	1.0	0	0	1	

	NAICS	ApprovalDate	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	UrbanRu
178	812320	1997-06-23	60	40	1.0	0	0	1	
179	332117	2006-07-02	90	72	2.0	1	3	1	
180	454110	1997-02-28	120	1	2.0	0	0	1	

In [12]:

```
#Dropping the ApprovalDate now that our new variable has been successfully created
SBAdf2 = SBAdf2.drop(['ApprovalDate'], axis=1)
```

The Franchise Code is an important categorical variable for our analysis. However, we do not need to know what the specific franchise each loan represents since there are many options. Instead, we will encode our 'FranchiseCode' variable so it just shows 1 for is franchise, and 0 for is not franchise.

We see from our documentation above that if the franchise code is 0 or 1 it is not a franchise, but any series of numbers larger than that was a code for a specific franchise. However, examining our CSV file we see that there are 12 that were coded '3' in error. So we need to process 0,1,3 as "Not a Franchise" and anything else as "Is a Franchise"

In [13]:

```
#Encoding our 'FranchiseCode' variable to become a hot code, 0 for "Not Franchise" and
SBAdf2.loc[SBAdf2['FranchiseCode'] < 4, 'FranchiseCode']=0
SBAdf2.loc[SBAdf2['FranchiseCode'] >= 4, 'FranchiseCode']=1
#Reviewing to see if our changes worked.  From examining the CSV I know observation 12
#So those two should show '1' and the rest '0'
SBAdf2.iloc[[8,9,10,11,12,13,14,15,16,17,18]]
```

Out[13]:

	NAICS	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	UrbanRural	RevLineCr
8	721310	297	2	2.0	0	0	0	0	N
9	0	84	3	2.0	0	0	0	0	N
10	811111	84	1	2.0	0	0	0	0	N
11	235950	60	24	1.0	0	0	0	0	N
12	445299	162	2	2.0	0	0	1	1	N
13	0	120	2	2.0	0	0	0	0	N
14	0	240	1	1.0	30	0	0	0	N
15	421330	12	5	2.0	0	0	0	0	N
16	0	60	5	1.0	0	0	0	0	N
17	0	60	16	1.0	0	0	0	0	N
18	0	84	12	2.0	0	0	1	0	N

NAICS is a categorical variable reflecting which industry the company applying for the loan is in. OneHotEncoding this in its current state is not feasabile since there are too many represented. However, the first 2 digits of this 6 digit code represents its larger industry, with the other 4 representing subcategories. So we can first reduce this column to its first 2 digits and then OneHotEncode from there.

In [14]:
```
#Processing the variable 'NAICS'
SBAdf2['NAICS_Short'] = SBAdf2.NAICS.astype(str).str[:2].astype(int)
SBAdf3 = SBAdf2.drop(['NAICS'], axis = 1)
SBAdf3
```

Out[14]:

	Term	NoEmp	NewExist	CreateJob	RetainedJob	FranchiseCode	UrbanRural	RevLineCr	Low
0	84	4	2.0	0	0	0	0	N	
1	60	2	2.0	0	0	0	0	N	
2	180	7	1.0	0	0	0	0	N	
3	60	2	1.0	0	0	0	0	N	
4	240	14	1.0	7	7	0	0	N	
...	...	...	...	...	...	...	...	...	
899159	60	6	1.0	0	0	0	0	0	
899160	60	6	1.0	0	0	0	0	Y	
899161	108	26	1.0	0	0	0	0	N	
899162	60	6	1.0	0	0	0	0	N	
899163	48	1	2.0	0	0	0	0	N	

899164 rows × 15 columns

In [15]:
```
#Looking for NaN or any other values that do not fit in some of our other variables
UniqueT = pd.unique(SBAdf3.Term)
UniqueNO = pd.unique(SBAdf3.NoEmp)
UniqueCJ = pd.unique(SBAdf3.CreateJob)
UniqueRJ = pd.unique(SBAdf3.RetainedJob)
UniqueDG = pd.unique(SBAdf3.DisbursementGross)
UniqueGA = pd.unique(SBAdf3.GrAppv)
UniqueSBA = pd.unique(SBAdf3.SBA_Appv)

print(f"Unique values in Term {UniqueT}")
print(f"Unique values in NoEmp {UniqueNO}")
print(f"Unique values in CreateJob {UniqueCJ}")
print(f"Unique values in RetainedJob {UniqueRJ}")
print(f"Unique values in DisbursementGross {UniqueDG}")
print(f"Unique values in GrAppv {UniqueGA}")
print(f"Unique values in SBA_Appv {UniqueSBA}")
```

```
Unique values in Term [ 84  60 180 240 120  45 297 162  12 300  87 114 144 126  83 102
80 137
  42  96 167   7  36  37  26 264  72  24   5  54  66 161  71   4  93 288
 108  10  13  90  19  16   3  27 149  41 246  18  57 104  82 298  14  61
 127  58  44  32  85  48  31 112  38  73  47  11 134  15  79  53  39   6
 255  55 133  95  35  59  62  68 123  46  70 138  40  52  25  65  91   1
  74  49 103  77  86  63  56  22   0  97  23  17  69  21  43  89 276  92
```

```
 183   2 132  34 131   9  78  99 129 216   8  29 289  30 119 228 168 208
  81 147 125  94  51 211  64 111 266  75 306  28 232 117 118 309 303  98
 191 116  76 113 292  88 166 244 176 258 203 231 142  33 157 165  50 210
 294 301 106  20 318 229 204 269 241 178 115 174 192  67 100 141 282 122
 156 153 268 249 238 233 105 263 124 279 140 186 107 190 308 128 243 302
 299 280 223 311 222 202 257 130 101 121 278 272 319 283 221 250 290 199
 252 187 310 304 136 261 196 181 175 195 177 139 110 242 270 277 184 150
 207 358 213 273 357 248 275 164 239 206 215 170 254 217 172 158 218 189
 256 179 262 193 146 155 135 185 307 200 109 265 349 209 236 251 145 152
 169 245 230 285 201 293 171 227 188 225 247 163 143 235 160 148 253 159
 295 198 271 234 237 336 220 291 287 154 219 197 312 284 281 151 267 274
 182 212 305 205 226 194 259 214 173 224 385 313 340 461 296 343 348 286
 260 334 320 315 360 421 342 351 372 314 354 435 316 480 329 387 325 321
 317 352 359 389 333 330 417 404 370 324 398 419 425 414 365 345 481 350
 335 369 341 355 362 322 339 375 323 347 327 505 326 418 328 363 413 361
 356 396 438 382 364 367 374 442 353 527 569 338 366 386 368 428 346 388
 430 443 381 409 445 384 391 511 412 449 403 434 402 423 440 429]
Unique values in NoEmp [   4    2    7   14   19   45    1    3   24    5   16   12    6
90
   18    9   20   10    8   50   17   32   31   60   22   40   72   55
   30   25   46   15  214   28   23   11   57   13  112   26   80   42
   65   21   97  100  200  126   48   33   58   38   37   35    0   75
   36   70   66   27 2000   56   34   93   85  150   29   41  290   67
   44   47  119   39  155   54   49   82   95  300   51  120  265  133
   86  160   68   61  220   43   71   98  350   78  233  263   62 7941
   63  210  125  107  450  165  130 9992   77   64  424  257   52  600
  190  142   99   59   73  135   74  109  250   69  500  140  116  260
   96  339   87  110  161   88  105   53  400 2725  605  103   91   81
   94  147  144  175  136  182  156  118  375  345  121  145  180  101
   84   89   76  550  216  108  138  171  117  249  279  208  170   79
 9999  115  104  900   83  153  106  191  270  102  295 3000  230  195
  127  313  205  141  128  199  124  162  305  185  152  129  275  576
  132  196  640  178  750  720  414  315  143  299  217  179  320 1145
 4000  167  114  203  134 8000  111 5511 5921   92  240  280  148  340
  285 1300 1800  680  258  231  173 1980  510 5812  401 7212  192 5084
  480  186  131  146  168 1050  189  343 7216  312 8041  394  307 7389
  174  828  485  266 2100 1000 5947  368 1600 5000 1500  298 6000  221
 4953  360  355  137  325  123  122  344  163  386  530  237  330  188
  387 1003  761  421  113  308  158  310  187  238  235  222 7111 2112
  254 1200  225  228 2400  317  425 8018  430  197  157  176  198  256
 3200  183  277  362  202  289  139  520  154 2151  177  316  224  149
  151  215  395  426  262 6501  169  282 4100  383  172  688  287 1250
  390  318 4847 7231 5149 3900  967 3500  314  602  476  206 1461  274
  408  441  246  336  570  227 1940  735  523  204 1233 3170 1711  255
  351 1451  207  365  454 1550  823  544 1150  294 9000  226 7000  463
  211  370 2900 4005 1900  241  288  194  213  278  332  484  281  322
  650  181  521  218  800  420 3400 1400  700  242 2200 2500  625  495
  358  243  466  354  223  184  479  269  435  342  385  193  232  827
 2401  273  713  253  261  296 2501 1629 1700  429 2010  455  660  164
 1235  252  376  380  464  245 1100  608  209  247 5555  329  604  456
  166 1280 3089  985 1020  505 1502  234 5200  284  609  259  475  324
 5680 1981  323  251  740  575  396 1030  229 2610  515  328  442  433
 2232  341  306 3732  346  447  850  427  407  782  293  236  356 4685
 7241  363 1005  369  458  267 7999 2020  445 2121 1125 1010 4658  712
  212  271  377 1718 1515  560  404  302  276  248 1015  268 3737  319
 2120  304  512  585  292  808  244 9090 3030  606  840  460  301 2300
 3600  159  525  353 7991 5211 4012 1112 1440  413  410  488 4501 4800
 3100 3334  538 1603 1706 2520  283 1520 2202  357  201 1012  499  423
  635 1073  465 2510 1644 1101  403 4300  382  498  448 3009  685 1340
 2700  367  535  760 1524  309 7007  384  327 1960  540 5013  780  348
  717 8500 7538  405 2005 1382  858 9945 1542 1920 3713]
Unique values in CreateJob [   0    7   30    5    4    1   20   10    3    8   16   15
6   11
    2   40   55   25   12   21   50    9   13   47   18   17   14   29
   23   35   43   75   22   45   27   65   19   58   48   72   38   28
```

```
   24  150  200   82   68   41   80   70   33   97   32   26   34   36
   31  100   56   60   90   77   99   39   44   51  120   85   69   95
   42  160   37   57  600   49 1000   53   54   46   59  163  450  456
 3000  452  451  198   79  454   62  136   64   52  126  180   74  303
   63  386   78   98  455   76  152  221  110   84  153  127 2020  225
  453  125  458  457  174  104   89  320  154  300  102  149 8800  800
  130  235 5199  250  137  500  121  105   96  360  255  140  122  175
 1200   66  112 3500  118  220  115   73   93  151  195   67  138  400
   61  124   91 1711  131  184   83  409 1618 1150   88 1530  157  145
  166  135  210  226  183 3100  252  116   71  129  223   81  569  139
  144 1011  179  214  146  171  141  350   92  101  119  280  123  205
 1229  128  103  189  114  108  158  167   87  186   86  134 1100  750
  206  375  109  433 2140  177  264  168  240 5621  170  169  165  222
  106  148  363 1118  310  164 5085  143  480  256  365  155  190  397
 1027  270   94 2515  162  182 1016  860]
Unique values in RetainedJob [   0    7   23    4    6    1    9   20    2    5   19
8    3   10
   24   12   15   11   25   44   17   14   65   28   38   16   42   26
   18   13   50   93   40   37   60   21   30   31   34   35  150   22
   73   41   45  100  180   58   75  165   36  130   29   27  125   99
   46   32  257   43   47   80   70   54   62   33   39  400   55   95
   48  120   71   63   81   52   94   78  160  109   86   77  155   85
   90   64 3225   61   69   66  210  107   97   51   83  112   53   72
   76   87   68  118  138   67   57   56  117  171  229  115  275  153
  300  105  140  135   59   79  200  295  205  206  128  186  137  250
   89   49  131   92  404  110  320  139   82  108   88  104  114  134
  230  102  103   96   98   84  101  220  233   74  267   91 9500  355
  123  175  550  500  450  170  195  116  305  147  610  187  235  157
  124  127  106  254 4441  277  225  207  111  312  317  173  350  216
  143  430  197  176  145  126  133  256 2200  362  202  148  316 8800
  215  146  185  154  212  141  163  184 5000 3200  132  194  113  161
  172  330  366  190 1300  390 4000  476 3900  967  268  136  602  121
  240  122  162  523  204  159 1711  119  251  152  417  291  544  129
  142  231  189  203  360  213  278  280  484  260  177  281  675  226
  263  700  247  600  245  750  151  270  375  191  182  223 7250  214
  169  342  221  217  232  815  287  285  188 1000 1700  428  660  156
 1500  318  265  167  236  370  310  609  475  322  208  515  259  328
  497  356  255  158  192  166  219  363  274  144  262  315  178  420
  286  585  325  201  710  196  384  237  940  302  371  394 1600 3860
  244  393  410  472  720  168  252  290  297  548  485  183  800  149
  387  298  480  266  164  403  369  498  448  685  535  292  327  911
 3100  540  304 1111  243  199  900  198]
Unique values in DisbursementGross [  60000.   40000.  287000. ...  377446.  123770. 108
6300.]
Unique values in GrAppv [  60000.   40000.  287000. ...    12480.   62425. 1086300.]
Unique values in SBA_Appv [  60000.   40000.  287000. ...  377446.  123770. 1086300.]
```

Now I need to OneHotEncode our categorical data and Label Encode our target variable. I am using the OneHotEncoder instead of the Ordinal Encoder since our categorical variables are not in order.

In [16]:

```python
#OneHotEncode our feature data
encoder = OneHotEncoder()

encoder_df = pd.DataFrame(encoder.fit_transform(SBAdf3[['NAICS_Short']]).toarray())

# label_encoder for our response data.
label_encoder = LabelEncoder()
# Encode labels in column 'MIS_Status'.
SBAdf3['MIS_Status']= label_encoder.fit_transform(SBAdf3['MIS_Status'])

#Defining our Feature_data and Response_data and Encoded_data seperately so each can be
Feature_data = encoder_df
Response_data = pd.DataFrame(SBAdf3['MIS_Status'])
```

```python
Encoded_data = Feature_data.join(Response_data)
Other_data = SBAdf3.filter(['Term', 'NoEmp', 'NewExist', 'CreateJob', 'RetainedJob', 'F
                    'RevLineCr', 'LowDoc', 'DisbursementGross', 'GrAppv', 'SBA_Appv', 'R

#Review our now encoded data
SBAdf4 = Other_data.join(Encoded_data)
```

Three variables, "UrbanRural", "RevLineCr", and "LowDoc" all have a mess of answers.

UrbanRural - This lists 1 for urban, 2 for rural and 0 for undefined.

RevLineCr, LowDoc - This is supposed to be Y and N but as seen below, both contain a lot of random values

In [17]:

```python
#Reviewing our 4 categorical variables to see what missing values we are working with
UniqueUR = pd.unique(SBAdf4.UrbanRural)
UniqueRLC = pd.unique(SBAdf4.RevLineCr)
UniqueLD = pd.unique(SBAdf4.LowDoc)
UniqueNE = pd.unique(SBAdf4.NewExist)

print(f"Unique values in UrbanRural {UniqueUR}")
print(f"Unique values in RevLineCr {UniqueRLC}")
print(f"Unique values in LowDoc {UniqueLD}")
print(f"Unique values in NewExist {UniqueNE}")
```

```
Unique values in UrbanRural [0 1 2]
Unique values in RevLineCr ['N' '0' 'Y' 'T' nan '`' ',' '1' 'C' '3' '2' 'R' '7' 'A' '5'
'.' '4' '-'
 'Q']
Unique values in LowDoc ['Y' 'N' 'C' '1' nan 'S' 'R' 'A' '0']
Unique values in NewExist [ 2.  1.  0. nan]
```

In [18]:

```python
#Processing our 4 categorical variables to be ready for a KNN imputer
#Urban Rural 1 and 2 are acceptable, so we will replace 0 with NaN for our imputer
SBAdf4['UrbanRural'] = SBAdf4['UrbanRural'].replace(0, np.NaN)

#For NewExisit, like Urban Rural, 1 and 2 are acceptable so we will replace 0 with NaN
SBAdf4['NewExist'] = SBAdf4['NewExist'].replace(0, np.NaN)

#Only acceptable values of RevLineCr are Y and N, the rest will be replaced with Nan
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('0', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('T', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('`', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace(',', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('1', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('C', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('3', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('2', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('R', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('7', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('A', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('5', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('.', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('4', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('-', np.NaN)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('Q', np.NaN)

#Only acceptable values for LowDoc are Y and N, the rest will be replaced with NaN
```

```python
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('C', np.NaN)
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('1', np.NaN)
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('S', np.NaN)
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('R', np.NaN)
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('A', np.NaN)
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('0', np.NaN)

#Encode our Y and N to 0 and  1 for LowDoc and RevLineCr
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('Y', 1)
SBAdf4['LowDoc'] = SBAdf4['LowDoc'].replace('N', 0)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('Y', 1)
SBAdf4['RevLineCr'] = SBAdf4['RevLineCr'].replace('N', 0)
```

In [19]:

```python
#Ensure we simplified all of our values into acceptable values or NaN
UniqueUR = pd.unique(SBAdf4.UrbanRural)
UniqueRLC = pd.unique(SBAdf4.RevLineCr)
UniqueLD = pd.unique(SBAdf4.LowDoc)
UniqueNE = pd.unique(SBAdf4.NewExist)

print(f"Unique values in UrbanRural {UniqueUR}")
print(f"Unique values in RevLineCr {UniqueRLC}")
print(f"Unique values in LowDoc {UniqueLD}")
print(f"Unique values in NewExist {UniqueNE}")
```

```
Unique values in UrbanRural [nan  1.  2.]
Unique values in RevLineCr [ 0. nan  1.]
Unique values in LowDoc [ 1.  0. nan]
Unique values in NewExist [ 2.  1. nan]
```

In [20]:

```python
#Gather our data for KNN Impute
BeforeImpute = SBAdf4.filter(['UrbanRural', 'NewExist', 'RevLineCr', 'LowDoc'], axis =
BeforeImpute
```

Out[20]:

	UrbanRural	NewExist	RevLineCr	LowDoc
0	NaN	2.0	0.0	1.0
1	NaN	2.0	0.0	1.0
2	NaN	1.0	0.0	0.0
3	NaN	1.0	0.0	1.0
4	NaN	1.0	0.0	0.0
...	...	...	...	...
899159	NaN	1.0	NaN	0.0
899160	NaN	1.0	1.0	0.0
899161	NaN	1.0	0.0	0.0
899162	NaN	1.0	0.0	1.0
899163	NaN	2.0	0.0	0.0

899164 rows × 4 columns

In [21]:
```
#Run KNN Imputer
imputer = KNNImputer(n_neighbors=3)
AfterImpute = imputer.fit_transform(BeforeImpute)
AfterImpute
```

Out[21]:
```
array([[1., 2., 0., 1.],
       [1., 2., 0., 1.],
       [1., 1., 0., 0.],
       ...,
       [1., 1., 0., 0.],
       [2., 1., 0., 1.],
       [1., 2., 0., 0.]])
```

In [22]:
```
#Create our After Impute dataframe as AIdf and prepare to combine
AIdf = pd.DataFrame(AfterImpute)
AIdf
```

Out[22]:

	UrbanRural	NewExist	RevLineCr	LowDoc
0	1.000000	2.0	0.0	1.0
1	1.000000	2.0	0.0	1.0
2	1.000000	1.0	0.0	0.0
3	2.000000	1.0	0.0	1.0
4	1.000000	1.0	0.0	0.0
...	...	...	...	...
899159	1.000000	1.0	0.0	0.0
899160	1.333333	1.0	1.0	0.0
899161	1.000000	1.0	0.0	0.0
899162	2.000000	1.0	0.0	1.0
899163	1.000000	2.0	0.0	0.0

899164 rows × 4 columns

In [23]:
```
#Second verfication that we only have acceptable values
UniqueUR = pd.unique(AIdf.UrbanRural)
UniqueRLC = pd.unique(AIdf.RevLineCr)
UniqueLD = pd.unique(AIdf.LowDoc)
UniqueNE = pd.unique(AIdf.NewExist)

print(f"Unique values in UrbanRural {UniqueUR}")
print(f"Unique values in RevLineCr {UniqueRLC}")
print(f"Unique values in LowDoc {UniqueLD}")
print(f"Unique values in NewExist {UniqueNE}")
```

```
Unique values in UrbanRural [1.         2.         1.33333333 1.66666667]
Unique values in RevLineCr [0. 1.]
Unique values in LowDoc [1.         0.         0.33333333 0.66666667]
Unique values in NewExist [2.         1.         1.66666667 1.33333333]
```

In [24]:
```
#Bring our imputed data back into our master dataframe
```

```
SBAdf4 = SBAdf4.drop(['UrbanRural', 'NewExist', 'RevLineCr', 'LowDoc'], axis = 1)
SBAdf5 = SBAdf4.join(AIdf)
SBAdf5
```

Out[24]:

	Term	NoEmp	CreateJob	RetainedJob	FranchiseCode	DisbursementGross	GrAppv	SBA_App
0	84	4	0	0	0	60000.0	60000.0	60000.
1	60	2	0	0	0	40000.0	40000.0	40000.
2	180	7	0	0	0	287000.0	287000.0	287000.
3	60	2	0	0	0	35000.0	35000.0	35000.
4	240	14	7	7	0	229000.0	229000.0	229000.
...	...	...	...	...	...	...	...	.
899159	60	6	0	0	0	70000.0	70000.0	70000.
899160	60	6	0	0	0	85000.0	85000.0	85000.
899161	108	26	0	0	0	300000.0	300000.0	300000.
899162	60	6	0	0	0	75000.0	75000.0	75000.
899163	48	1	0	0	0	30000.0	30000.0	30000.

899164 rows × 39 columns

In [65]:

```
#Scaling our continous variables
continuous = SBAdf5.filter(['Term', 'NoEmp', 'DisbursementGross', 'GrAppv', 'SBA_Appv']
scaler = StandardScaler()
continous_scaled = scaler.fit_transform(continuous)
continous_scaled = pd.DataFrame(continous_scaled)
continous_scaled.columns = (['Term', 'NoEmp', 'DisbursementGross', 'GrAppv', 'SBA_Appv'
nocont = SBAdf5.drop(['Term', 'NoEmp', 'DisbursementGross', 'GrAppv', 'SBA_Appv'], axis
SBAdf5 = nocont.join(continous_scaled)
```

Selecting The Most Relevant Variables

We will be using MIS_Status as our target variable. This variable ends in two possible outcomes: PIF for Paid in Full and CHGOFF for Charged-Off for loans that have defaulted. The rest of the variables selected will be the features. Below, we are going to check the correlation between each feature and our target to decide which are worth keeping and which will be eliminated.

In [26]:

```
#Verify our data types to ensure everything is ready for our correlation test
SBAdf5.dtypes
```

Out[26]:

```
CreateJob              int64
RetainedJob            int64
FranchiseCode          int64
RecessionYN            int32
0                    float64
1                    float64
2                    float64
```

```
3                     float64
4                     float64
5                     float64
6                     float64
7                     float64
8                     float64
9                     float64
10                    float64
11                    float64
12                    float64
13                    float64
14                    float64
15                    float64
16                    float64
17                    float64
18                    float64
19                    float64
20                    float64
21                    float64
22                    float64
23                    float64
24                    float64
MIS_Status              int32
UrbanRural            float64
NewExist              float64
RevLineCr             float64
LowDoc                float64
Term                  float64
NoEmp                 float64
DisbursementGross     float64
GrAppv                float64
SBA_Appv              float64
dtype: object
```

In [7]:

```
#Examine our correlations to see if any other variables can be dropped
SBAdf5.corr().style.background_gradient(cmap="Blues")
```

Out[7]:

	Unnamed: 0	Term	NoEmp	FranchiseCode	DisbursementGross	GrAppv	SBA_
Unnamed: 0	1.000000	0.110847	0.011346	0.054213	0.070392	0.083208	0.0
Term	0.110847	1.000000	0.046140	0.038391	0.466391	0.502610	0.4
NoEmp	0.011346	0.046140	1.000000	0.007385	0.088651	0.090430	0.0
FranchiseCode	0.054213	0.038391	0.007385	1.000000	0.079022	0.088967	0.0
DisbursementGross	0.070392	0.466391	0.088651	0.079022	1.000000	0.971242	1.0
GrAppv	0.083208	0.502610	0.090430	0.088967	0.971242	1.000000	0.9
SBA_Appv	0.070392	0.466391	0.088651	0.079022	1.000000	0.971242	1.0
RecessionYN	-0.082273	-0.025499	-0.002880	-0.003111	0.005789	0.003329	0.0
RevLineCr	-0.160000	-0.335331	-0.032915	-0.098916	-0.179223	-0.250212	-0.1
MIS_Status	0.180063	0.307745	0.025802	0.014188	0.104869	0.115089	0.1
UrbanRural	0.053243	-0.079810	-0.017598	-0.014146	-0.064504	-0.067768	-0.0
NewExist	-0.045932	-0.072494	-0.040048	0.142210	-0.073748	-0.065721	-0.0
LowDoc	0.189514	-0.108677	-0.022464	0.028401	-0.173250	-0.163415	-0.1

In [7]:
```
#Dropping 'CreateJob' and 'RetainedJob' for having little to no correlation to MIS_Stat
#dropping as well our NAICS variables since most are not strongly coordinated and the l
SBAdf5 = SBAdf5.filter(['Term', 'NoEmp', 'FranchiseCode', 'DisbursementGross', 'GrAppv'
                        'RecessionYN', 'RevLineCr','MIS_Status','UrbanRural', 'NewExist'
```

In [29]:
```
#Saving our now processed data to a new CSV file for future reference
SBAdf5.to_csv(r'C:\Users\joshu\DTSC 691 Project\SBAdf5.csv', header = True)
```

In [5]:
```
#7/1/22 experienced a computer error, restarted from importing saved CSV file
SBAdf5 = pd.read_csv('SBAdf5.csv', low_memory=False)
```

In [6]:
```
#Reviewing out fully cleaned dataframe before moving on
SBAdf5
```

Out[6]:

	Unnamed: 0	Term	NoEmp	FranchiseCode	DisbursementGross	GrAppv	SBA_Appv	Rec
0	0	-0.339513	-0.100007	0	-0.490730	-0.468423	-0.490730	
1	1	-0.643861	-0.126995	0	-0.560262	-0.539029	-0.560262	
2	2	0.877876	-0.059526	0	0.298449	0.332952	0.298449	
3	3	-0.643861	-0.126995	0	-0.577644	-0.556680	-0.577644	
4	4	1.638745	0.034931	0	0.096808	0.128195	0.096808	
...	...	...	...	...	...	...	...	
899159	899159	-0.643861	-0.073020	0	-0.455965	-0.433120	-0.455965	
899160	899160	-0.643861	-0.073020	0	-0.403816	-0.380166	-0.403816	
899161	899161	-0.035166	0.196856	0	0.343644	0.378846	0.343644	
899162	899162	-0.643861	-0.073020	0	-0.438582	-0.415469	-0.438582	
899163	899163	-0.796034	-0.140489	0	-0.595027	-0.574331	-0.595027	

899164 rows × 13 columns

In [8]:
```
SBAdf5 = SBAdf5.drop(["Unnamed: 0"], axis=1)
```

In [9]:
```
#Reviewing our feature data to ensure we have only a 1 for 'PIF' or a 0 for 'Chgoff'
UniqueFeature = pd.unique(SBAdf5.MIS_Status)
print(f"Unique values in MIS_Status {UniqueFeature}")
```

```
Unique values in MIS_Status [1 0 2]
```

In [10]:
```
print(SBAdf5['MIS_Status'].value_counts())
```

```
1    739609
0    157558
2      1997
Name: MIS_Status, dtype: int64
```

In [11]:
```
#With plenty of instances left, we will drop the rows containing a 2
SBAdf5 = SBAdf5[SBAdf5.MIS_Status != 2]
```

In [12]:
```
print(SBAdf5['MIS_Status'].value_counts())
```

```
1    739609
0    157558
Name: MIS_Status, dtype: int64
```

Train Test Split

In [14]:
```
#Train test split our data for our 6 models we will test below
#X is our features data
X = SBAdf5.drop(["MIS_Status"], axis=1)
#Y is our response data
Y = SBAdf5["MIS_Status"]

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1, random_state=4
```

Random Forest Classifier

In [61]:
```
# -----
# Coarse-Grained RandomForestClassifier GridSearch
# -----

#Took code and language from my assignment 3 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"max_depth": [3,4,5,8], "n_estimators": [50,100,250],
    "min_samples_split": [4,5,8,12]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
rfc_gs_coarse = GridSearchCV(RandomForestClassifier(random_state=42), param_grid,
                              verbose=1, cv=3, n_jobs=-1)

#Fit our grid search with our training data
rfc_gs_coarse.fit(X_train, Y_train)

#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (rfc_gs_coarse.best_params_))
```

```
Fitting 3 folds for each of 48 candidates, totalling 144 fits
The best parameters are:  {'max_depth': 8, 'min_samples_split': 4, 'n_estimators': 250}
```

In [65]:
```
# -----
# Refined-Grained RandomForestClassifier GridSearch
# -----
```

```
#Took code and language from my assignment 3 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"max_depth": [30,32,34,36], "n_estimators": [250,275,300,325,350],
    "min_samples_split": [2,3,4]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
rfc_gs_refined = GridSearchCV(RandomForestClassifier(random_state=42), param_grid,
                              verbose=1, cv=3, n_jobs = -1)

#Fit our grid search with our training data
rfc_gs_refined.fit(X_train, Y_train)

#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (rfc_gs_refined.best_params_))
```

```
Fitting 3 folds for each of 60 candidates, totalling 180 fits
The best parameters are:  {'max_depth': 30, 'min_samples_split': 4, 'n_estimators': 300}
```

In [67]:

```
# -----
# Final-Grained RandomForestClassifier GridSearch
# -----

#Took code and language from my assignment 3 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"max_depth": [10,15,20,25,30], "n_estimators": [294,296,298,299,300],
    "min_samples_split": [4]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
rfc_gs_final = GridSearchCV(RandomForestClassifier(random_state=42), param_grid,
                              verbose=1, cv=3, n_jobs = -1)

#Fit our grid search with our training data
rfc_gs_final.fit(X_train, Y_train)

#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (rfc_gs_final.best_params_))
```

```
Fitting 3 folds for each of 25 candidates, totalling 75 fits
The best parameters are:  {'max_depth': 25, 'min_samples_split': 4, 'n_estimators': 299}
```

In [68]:

```
# -----
# Final2-Grained RandomForestClassifier GridSearch
# -----

#Took code and language from my assignment 3 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"max_depth": [23,24,25,26,27], "n_estimators": [299],
    "min_samples_split": [4]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
rfc_gs_final2 = GridSearchCV(RandomForestClassifier(random_state=42), param_grid,
                              verbose=1, cv=3, n_jobs = -1)
```

```
#Fit our grid search with our training data
rfc_gs_final2.fit(X_train, Y_train)

#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (rfc_gs_final.best_params_))
```

```
Fitting 3 folds for each of 5 candidates, totalling 15 fits
The best parameters are:  {'max_depth': 25, 'min_samples_split': 4, 'n_estimators': 299}
```

In [15]:

```
%%time
#Train our RandomForestClassifier with the best paramaters found from our grid search a
#Creating a new RandomForestClassifier

rfc = RandomForestClassifier(n_estimators=299, max_depth=25, random_state=42, min_sampl
rfc.fit(X_train, Y_train)
#running our model prediction using our X_train data
rfcpred = rfc.predict(X_train)
y_predrfc = rfc.predict(X_test)
```

```
Wall time: 2min 58s
```

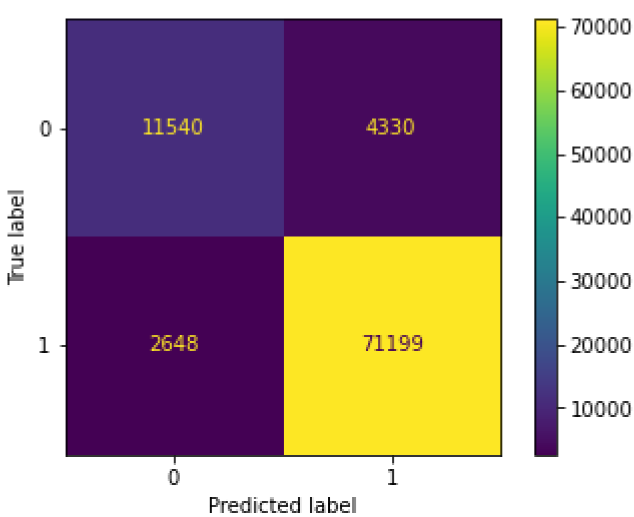
In [16]:

```
#Evaluating the accuracy, precision, and recall of our model
acc_scorerfc = accuracy_score(Y_test, y_predrfc)
prec_scorerfc = precision_score(Y_test, y_predrfc, average='micro')
recall_scorerfc = recall_score(Y_test, y_predrfc, average='micro')

print('Random Forest Classifier Accuracy=%s' % (acc_scorerfc))
print('Random Forest Classifier Precision=%s' % (prec_scorerfc))
print('Random Forest Classifier Recall=%s' % (recall_scorerfc))
```

```
Random Forest Classifier Accuracy=0.9222220983760046
Random Forest Classifier Precision=0.9222220983760046
Random Forest Classifier Recall=0.9222220983760046
```

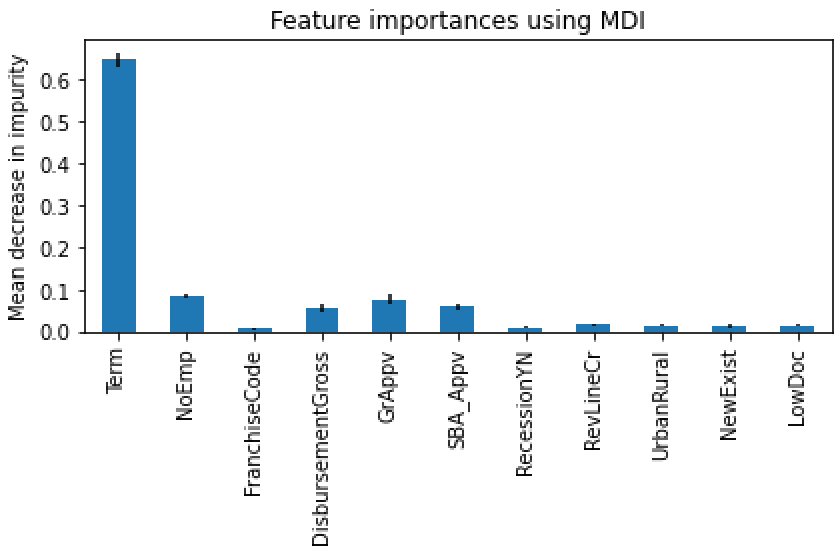
In [17]:

```
#Confusion Matrix for our Random Forest Classifier

plot_confusion_matrix(rfc, X_test, Y_test)
plt.show()
```



In [18]:

```
#Examing feature importance of our Random Forest Classifier
#Some code taken from: https://scikit-learn.org/stable/auto_examples/ensemble/plot_fore

#Defining our feature names
feature_names = (['Term', 'NoEmp', 'FranchiseCode', 'DisbursementGross', 'GrAppv', 'SBA
                       'RecessionYN', 'RevLineCr','UrbanRural', 'NewExist', 'LowDoc'])

#Gathering our importances
importances = rfc.feature_importances_
std = np.std([tree.feature_importances_ for tree in rfc.estimators_], axis=0)

forest_importances = pd.Series(importances, index=feature_names)

#Plotting our feature importance
fig, ax = plt.subplots()
forest_importances.plot.bar(yerr=std, ax=ax)
ax.set_title("Feature importances using MDI")
ax.set_ylabel("Mean decrease in impurity")
fig.tight_layout()
```



Decision Tree Classifier

In [74]:

```
# -----
# Coarse-Grained DecisionTreeClassifier GridSearch
# -----

#Took code and language from my assignment 2 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"splitter": ["best", "random"], "max_depth": [1,2,3,4,5,8,16,32],
    "min_samples_split": [2,3,4,5,8,12,16,20]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
dtc_gs_coarse = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid,
                              verbose=1, cv=3, n_jobs = -1)

#Fit our grid search with our training data
dtc_gs_coarse.fit(X_train, Y_train)
```

```
#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (dtc_gs_coarse.best_params_))
```

```
Fitting 3 folds for each of 128 candidates, totalling 384 fits
The best parameters are:  {'max_depth': 16, 'min_samples_split': 20, 'splitter': 'best'}
```

In [76]:

```
# -----
# Refined-Grained DecisionTreeClassifier GridSearch
# -----

#Took code and language from my assignment 1 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"splitter": ["best"], "max_depth": [13,14,15,16,17,18],
    "min_samples_split": [69,70,71,72,73,74,75]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
dtc_gs_refined = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid,
                              verbose=1, cv=3)

#Fit our grid search with our training data
dtc_gs_refined.fit(X_train, Y_train)

#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (dtc_gs_refined.best_params_))
```

```
Fitting 3 folds for each of 42 candidates, totalling 126 fits
The best parameters are:  {'max_depth': 13, 'min_samples_split': 70, 'splitter': 'best'}
```

In [78]:

```
# -----
# Final-Grained DecisionTreeClassifier GridSearch
# -----

#Took code and language from my assignment 1 submission for DTSC680
#create our paramater grid dictionary to be passed to the grid search
param_grid = [
    {"splitter": ["best"], "max_depth": [9,10,11,12,13,14,15],
    "min_samples_split": [70]},
]

#Initiate grid search CV, passing it our parameter grid dictionary
dtc_gs_final = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid,
                              verbose=1, cv=3)

#Fit our grid search with our training data
dtc_gs_final.fit(X_train, Y_train)

#Find the best parameters from our given dictionary options we passed the grid search
print("The best parameters are: ", (dtc_gs_final.best_params_))
```

```
Fitting 3 folds for each of 7 candidates, totalling 21 fits
The best parameters are:  {'max_depth': 13, 'min_samples_split': 70, 'splitter': 'best'}
```

In [19]:

```
%%time
#Train our DecisionTreeClassifier with the best paramaters found from our grid search a
```

```
#Creating a new DecisionTreeClassifier
dtc = DecisionTreeClassifier(max_depth=13, min_samples_split=70, splitter='best', rando
dtc.fit(X_train, Y_train)
#running our model prediction using our X_train data
dtcpred = dtc.predict(X_train)
y_preddtc = dtc.predict(X_test)
```

```
Wall time: 4.76 s
```

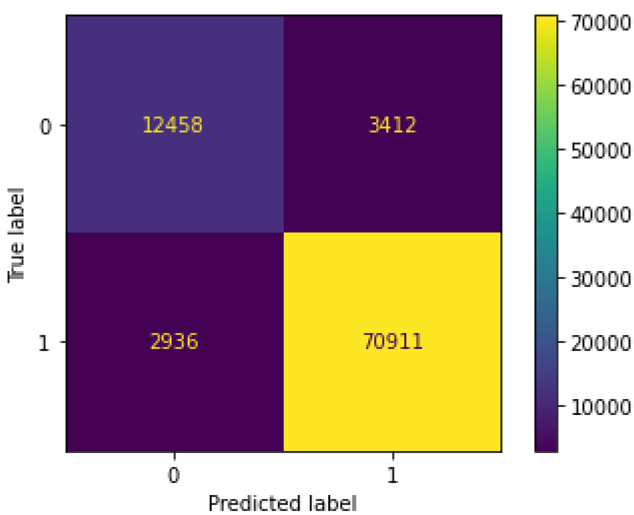
In [20]:

```
#Evaluating the accuracy, precision, and recall of our model
acc_scoredtc = accuracy_score(Y_test, y_preddtc)
prec_scoredtc = precision_score(Y_test, y_preddtc, average='micro')
recall_scoredtc = recall_score(Y_test, y_preddtc, average='micro')

print('Decision Tree Classifier Accuracy=%s' % (acc_scoredtc))
print('Decision Tree Classifier Precision=%s' % (prec_scoredtc))
print('Decision Tree Classifier Recall=%s' % (recall_scoredtc))
```

```
Decision Tree Classifier Accuracy=0.9292441789181537
Decision Tree Classifier Precision=0.9292441789181537
Decision Tree Classifier Recall=0.9292441789181537
```

In [21]:

```
#Confusion Matrix for our Decision Tree Classifier
plot_confusion_matrix(dtc, X_test, Y_test)
plt.show()
```



Logistic Regression

In [22]:

```
%%time
#Train our Logistic Regression model and evaluate how long it takes to train

lr = LogisticRegression()
lr.fit(X_train, Y_train)
y_predlr = lr.predict(X_test)
```

```
Wall time: 1.31 s
C:\Users\joshu\anaconda3\lib\site-packages\daal4py\sklearn\linear_model\logistic_path.p
y:548: ConvergenceWarning: lbfgs failed to converge (status=2):
ABNORMAL_TERMINATION_IN_LNSRCH.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

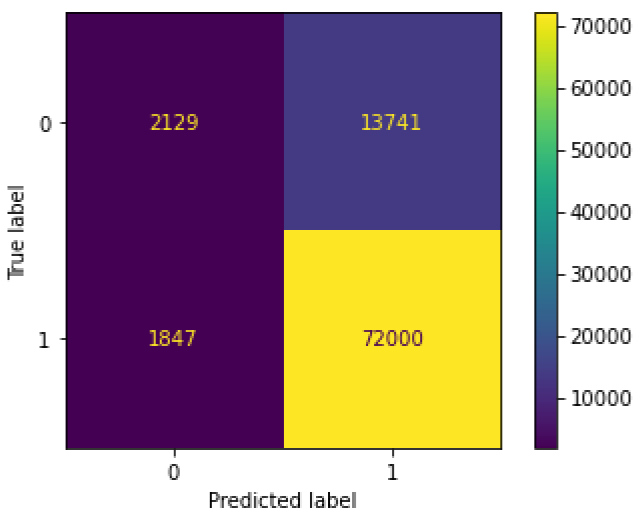
In [23]:

```
#Evaluating the accuracy, precision, and recall of our model
acc_scorelr = accuracy_score(Y_test, y_predlr)
prec_scorelr = precision_score(Y_test, y_predlr, average='micro')
recall_scorelr = recall_score(Y_test, y_predlr, average='micro')

print('Logistic Regression Accuracy=%s' % (acc_scorelr))
print('Logistic Regression Precision=%s' % (prec_scorelr))
print('Logistic Regression Recall=%s' % (recall_scorelr))
```

```
Logistic Regression Accuracy=0.8262536642999655
Logistic Regression Precision=0.8262536642999655
Logistic Regression Recall=0.8262536642999655
```

In [24]:

```
#Confusion Matrix for our Logistic Regression
plot_confusion_matrix(lr, X_test, Y_test)
plt.show()
```



KNN Classifier

In [ ]:

```
#Perform gridsearch to identify the best paramater for our KNN Classifier
knn = KNeighborsClassifier()
k_range = list(range(1, 100))
param_grid = dict(n_neighbors=k_range)

# defining parameter range
grid = GridSearchCV(knn, param_grid, cv=3, scoring='accuracy', return_train_score=False
                    verbose=1)

# fitting the model for grid search
grid_search=grid.fit(X_train, Y_train)
print("The best parameters are: ", (grid.best_params_))
```

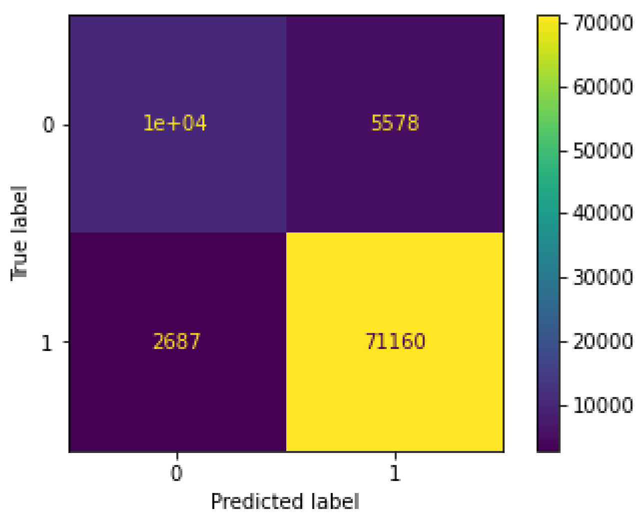
In [25]:
```
%%time
#Train our KNN Classifier and evaluate how long it takes to train

knn = KNeighborsClassifier(n_neighbors=7)
knn.fit(X_train, Y_train)
y_predknn = knn.predict(X_test)
```

```
Wall time: 22.5 s
```

In [26]:
```
#Evaluating the accuracy, precision, and recall of our model
acc_scoreknn = accuracy_score(Y_test, y_predknn)
prec_scoreknn = precision_score(Y_test, y_predknn, average='micro')
recall_scoreknn = recall_score(Y_test, y_predknn, average='micro')

print('KNN Classifier Accuracy=%s' % (acc_scoreknn))
print('KNN Classifier Precision=%s' % (prec_scoreknn))
print('KNN Classifier Recall=%s' % (recall_scoreknn))
```

```
KNN Classifier Accuracy=0.9078769909827569
KNN Classifier Precision=0.9078769909827569
KNN Classifier Recall=0.9078769909827569
```

In [27]:
```
#Confusion Matrix for our KNN Classifier
plot_confusion_matrix(knn, X_test, Y_test)
plt.show()
```



SVM Classifier

In [28]:
```
#Train test split our data for our SVM model we will test below
#X is our features data
X = SBAdf5.drop(["MIS_Status"], axis=1)
#Y is our response data
Y = SBAdf5["MIS_Status"]

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.025, random_state
```

In [29]:
```
%%time
#Train our SVM classifier and evaluate how long it takes to train
```

```
svm = SVC(kernel= 'linear', C=0.01)
svm.fit(X_train, Y_train)
y_predsvm = svm.predict(X_test)
```

```
Wall time: 4h 29min 59s
```

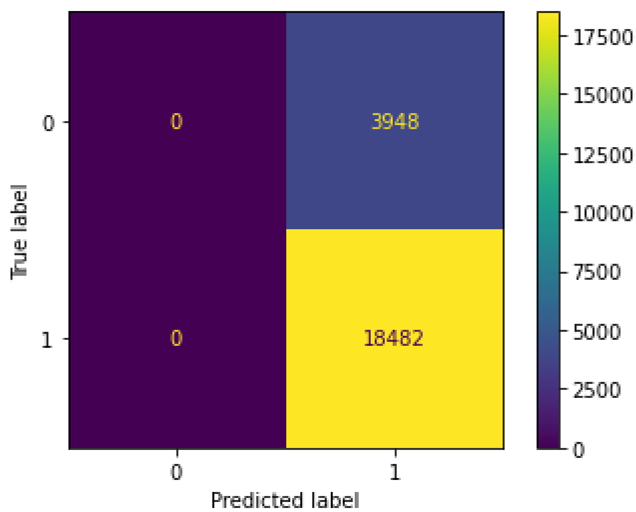
In [30]:

```
#Evaluating the accuracy, precision, and recall of our model
acc_scoreSVM = accuracy_score(Y_test, y_predsvm)
prec_scoreSVM = precision_score(Y_test, y_predsvm, average='micro')
recall_scoreSVM = recall_score(Y_test, y_predsvm, average='micro')

print('SVM Classifier Accuracy=%s' % (acc_scoreSVM))
print('SVM Classifier Precision=%s' % (prec_scoreSVM))
print('SVM Classifier Recall=%s' % (recall_scoreSVM))
```

```
SVM Classifier Accuracy=0.8239857333927775
SVM Classifier Precision=0.8239857333927775
SVM Classifier Recall=0.8239857333927775
```

In [31]:

```
#Confusion Matrix for our SVM Classifier
plot_confusion_matrix(svm, X_test, Y_test)
plt.show()
```



ANN Classifier

In [32]:

```
#Train test split our data back to 10% for our ANN Classifier
#X is our features data
X = SBAdf5.drop(["MIS_Status"], axis=1)
#Y is our response data
Y = SBAdf5["MIS_Status"]

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.1, random_state=4
```

In [33]:

```
#Creating our early stopping object
early_stop = EarlyStopping()
```

In [38]:

```
%%time
#Train our ANN classifier and evaluate how long it takes to train

tf.keras.backend.set_floatx('float64')

ann = keras.models.Sequential([
            keras.layers.Dense(100, activation="relu"),
            keras.layers.Dense(50),
            keras.layers.Dense(10),
            keras.layers.Dense(1)
])

#Compiling our model
ann.compile(loss="mean_squared_error",
              optimizer="sgd",
              metrics=["accuracy"])


#Fitting our model
callback = tf.keras.callbacks.EarlyStopping(monitor="loss", patience=0)
ann_fit = ann.fit(X_train, Y_train, epochs=5000, callbacks=[callback],
                  validation_data=(X_test, Y_test))
```

```
Epoch 1/5000
25233/25233 [==============================] - 28s 1ms/step - loss: 0.1111 - accuracy:
0.8409 - val_loss: 0.1046 - val_accuracy: 0.8483
Epoch 2/5000
25233/25233 [==============================] - 28s 1ms/step - loss: 0.1031 - accuracy:
0.8539 - val_loss: 0.1013 - val_accuracy: 0.8554
Epoch 3/5000
25233/25233 [==============================] - 32s 1ms/step - loss: 0.1007 - accuracy:
0.8591 - val_loss: 0.0996 - val_accuracy: 0.8611
Epoch 4/5000
25233/25233 [==============================] - 29s 1ms/step - loss: 0.0995 - accuracy:
0.8611 - val_loss: 0.1001 - val_accuracy: 0.861895 - accuracy: 0.
Epoch 5/5000
25233/25233 [==============================] - 31s 1ms/step - loss: 0.0988 - accuracy:
0.8622 - val_loss: 0.1002 - val_accuracy: 0.8594
Epoch 6/5000
25233/25233 [==============================] - 38s 2ms/step - loss: 0.0984 - accuracy:
0.8626 - val_loss: 0.0977 - val_accuracy: 0.8630
Epoch 7/5000
25233/25233 [==============================] - 38s 2ms/step - loss: 0.0978 - accuracy:
0.8629 - val_loss: 0.0976 - val_accuracy: 0.8623
Epoch 8/5000
25233/25233 [==============================] - 38s 2ms/step - loss: 0.0975 - accuracy:
0.8633 - val_loss: 0.0970 - val_accuracy: 0.8627
Epoch 9/5000
25233/25233 [==============================] - 38s 2ms/step - loss: 0.0972 - accuracy:
0.8636 - val_loss: 0.0967 - val_accuracy: 0.8634
Epoch 10/5000
25233/25233 [==============================] - 43s 2ms/step - loss: 0.0969 - accuracy:
0.8637 - val_loss: 0.0966 - val_accuracy: 0.8629
Epoch 11/5000
25233/25233 [==============================] - 34s 1ms/step - loss: 0.0966 - accuracy:
0.8637 - val_loss: 0.0967 - val_accuracy: 0.8654
Epoch 12/5000
25233/25233 [==============================] - 41s 2ms/step - loss: 0.0964 - accuracy:
0.8643 - val_loss: 0.0956 - val_accuracy: 0.8648
Epoch 13/5000
25233/25233 [==============================] - 47s 2ms/step - loss: 0.0968 - accuracy:
0.8641 - val_loss: 0.0954 - val_accuracy: 0.8646
Wall time: 7min 47s
```

In [39]:
```
#Making
y_pred = ann.predict(X_train)
```

In [40]:
```
#calculating our MSE for our ANN model prediction above
ANN_mse = mean_squared_error(Y_train, y_pred)

#Printing our MSE results
print("Artifical Neural Network MSE is: ", round((ANN_mse),4))
```

```
Artifical Neural Network MSE is:  0.0951
```

Summary of Our 6 Models Performance

In [41]:
```
#Random Forest Classifier Performance
print('Random Forest Classifier Accuracy=%s' % (acc_scorerfc))
print('Random Forest Classifier Precision=%s' % (prec_scorerfc))
print('Random Forest Classifier Recall=%s' % (recall_scorerfc))
print('The Training Time for This Classifier Was: 2min 58sec')

#Decision Tree Classifier Performance
print('Decision Tree Classifier Accuracy=%s' % (acc_scoredtc))
print('Decision Tree Classifier Precision=%s' % (prec_scoredtc))
print('Decision Tree Classifier Recall=%s' % (recall_scoredtc))
print('The Training Time for This Classifier Was: 4.76sec')

#Linear Regression Performance
print('Logistic Regression Accuracy=%s' % (acc_scorelr))
print('Logistic Regression Precision=%s' % (prec_scorelr))
print('Logistic Regression Recall=%s' % (recall_scorelr))
print('The Training Time for This Classifier Was: 1.31sec')

#KNN Classifier Performance
print('KNN Classifier Accuracy=%s' % (acc_scoreknn))
print('KNN Classifier Precision=%s' % (prec_scoreknn))
print('KNN Classifier Recall=%s' % (recall_scoreknn))
print('The Training Time for This Classifier Was: 22.5sec')

#SVM Classifier Performance
print('SVM Classifier Accuracy=%s' % (acc_scoreSVM))
print('SVM Classifier Precision=%s' % (prec_scoreSVM))
print('SVM Classifier Recall=%s' % (recall_scoreSVM))
print('The Training Time for This Classifier Was:4h 29min 59s')

#Ann Classifier Perfomance
print("Artifical Neural Network MSE is: ", round((ANN_mse),4))
print('The Training Time for This Classifier Was:7min 47s')
```

```
Random Forest Classifier Accuracy=0.9222220983760046
Random Forest Classifier Precision=0.9222220983760046
Random Forest Classifier Recall=0.9222220983760046
The Training Time for This Classifier Was: 2min 58sec
Decision Tree Classifier Accuracy=0.9292441789181537
Decision Tree Classifier Precision=0.9292441789181537
Decision Tree Classifier Recall=0.9292441789181537
The Training Time for This Classifier Was: 4.76sec
Logistic Regression Accuracy=0.8262536642999655
```

```
Logistic Regression Precision=0.8262536642999655
Logistic Regression Recall=0.8262536642999655
The Training Time for This Classifier Was: 1.31sec
KNN Classifier Accuracy=0.9078769909827569
KNN Classifier Precision=0.9078769909827569
KNN Classifier Recall=0.9078769909827569
The Training Time for This Classifier Was: 22.5sec
SVM Classifier Accuracy=0.8239857333927775
SVM Classifier Precision=0.8239857333927775
SVM Classifier Recall=0.8239857333927775
The Training Time for This Classifier Was:4h 29min 59s
Artifical Neural Network MSE is:  0.0951
The Training Time for This Classifier Was:7min 47s
```

Based on the above scores, I have chosen the Decision Tree Classifier as our model of choice.

Using Pickle on Selected Model

In [61]:

```
#Selecting our model
model = dtc

#saving to our pickle file
pickle.dump(model, open('model.pkl', 'wb'))
```