Financial Distress Prediction Report

Rui Shen

shen.ru@husky.neu.edu

CSYE 7245, INFO 7390, Spring 2018, Northeastern University

Abstract

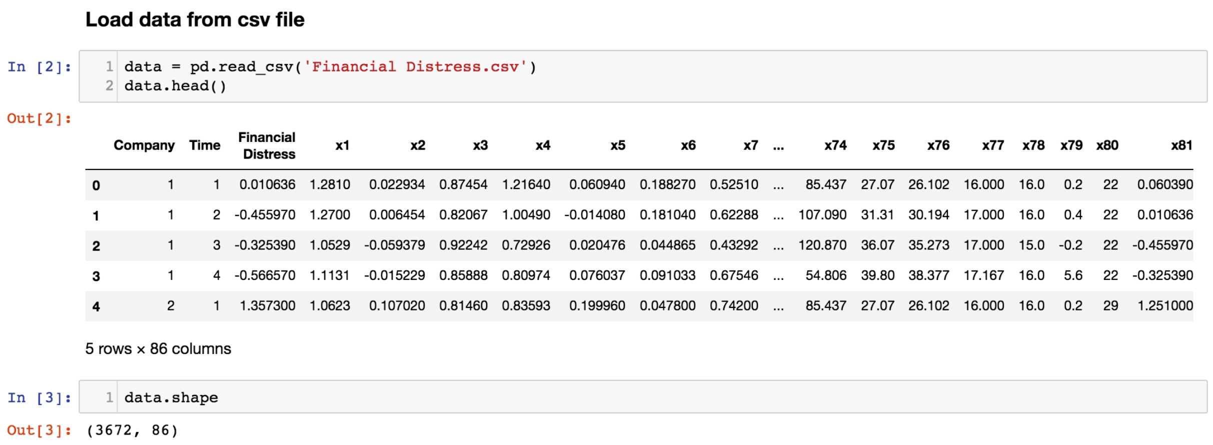
While investing a company - no matter capital venture, loan, or acquisition, the financial health of that company is one of the most important considerations to the most. Therefore, it's crutial for financial companies to have an accurate evaluation of the financial distress index. It is considerably difficult to perdict the future trend of a company for several years, but much easier to identify the current situation. Gathering public data and calcluate a financial distress index before further dicsion is an effective way to prevent loss.

Introduction

kaggle.com has a lot of datasets regarding "predict financial distress", one of them contains as much as 83 independent variables. This project will use the dataset to predict whether a company is financially healthy or not when given a particular combination of 83 varaibles. In the project, I am going to use four machine learning algorithms: multiple linear regression, support vector regression, random forest regression and neural network. I wil use GridSearchCV to auto identify the best R-squared value for tune each the model, and compare the accuracy rate based on R-square value.

Understanding dataset

- Understanding how dependent and independent variables distributed is important. One of the difficulties with this dataset is it's not properly labelled, independent variables are named as x1-x83, which causes troubles to identify inappropriate values, also make it hard to remove outliers.



Load data from csv file

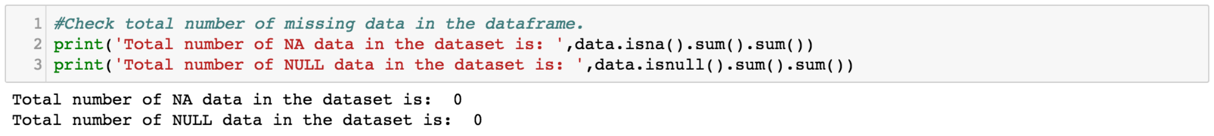
```
In [2]: data = pd.read_csv('Financial Distress.csv')
        data.head()
Out[2]:
```

	Company	Time	Financial Distress	x1	x2	x3	x4	x5	x6	x7	...	x74	x75	x76	x77	x78	x79	x80	x81
0	1	1	0.010636	1.2810	0.022934	0.87454	1.21640	0.060940	0.188270	0.52510	...	85.437	27.07	26.102	16.000	16.0	0.2	22	0.060390
1	1	2	-0.455970	1.2700	0.006454	0.82067	1.00490	-0.014080	0.181040	0.62288	...	107.090	31.31	30.194	17.000	16.0	0.4	22	0.010636
2	1	3	-0.325390	1.0529	-0.059379	0.92242	0.72926	0.020476	0.044865	0.43292	...	120.870	36.07	35.273	17.000	15.0	-0.2	22	-0.455970
3	1	4	-0.566570	1.1131	-0.015229	0.85888	0.80974	0.076037	0.091033	0.67546	...	54.806	39.80	38.377	17.167	16.0	5.6	22	-0.325390
4	2	1	1.357300	1.0623	0.107020	0.81460	0.83593	0.199960	0.047800	0.74200	...	85.437	27.07	26.102	16.000	16.0	0.2	29	1.251000

5 rows × 86 columns

```
In [3]: data.shape
Out[3]: (3672, 86)
```

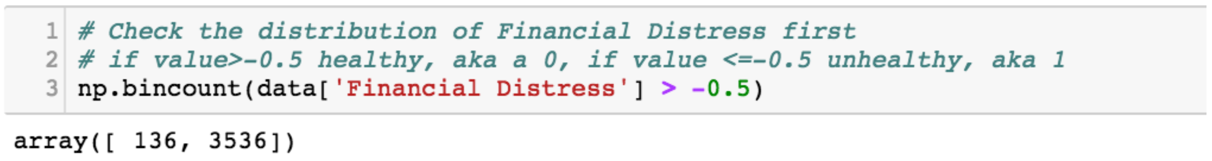
- It's also easy to see that no missing value exist in the dataset.



```
#Check total number of missing data in the dataframe.
print('Total number of NA data in the dataset is: ',data.isna().sum().sum())
print('Total number of NULL data in the dataset is: ',data.isnull().sum().sum())
```

```
Total number of NA data in the dataset is:  0
Total number of NULL data in the dataset is:  0
```

An obvious trap of this dataset is the highly imbalance of the output.

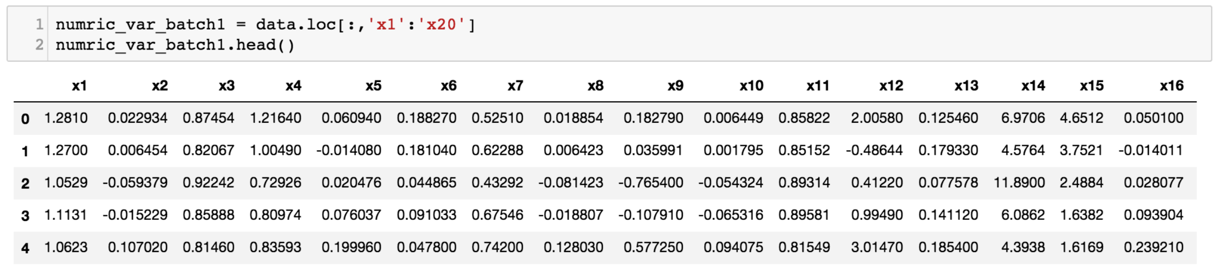


```
# Check the distribution of Financial Distress first
# if value>-0.5 healthy, aka a 0, if value <=-0.5 unhealthy, aka 1
np.bincount(data['Financial Distress'] > -0.5)
```

```
array([ 136, 3536])
```

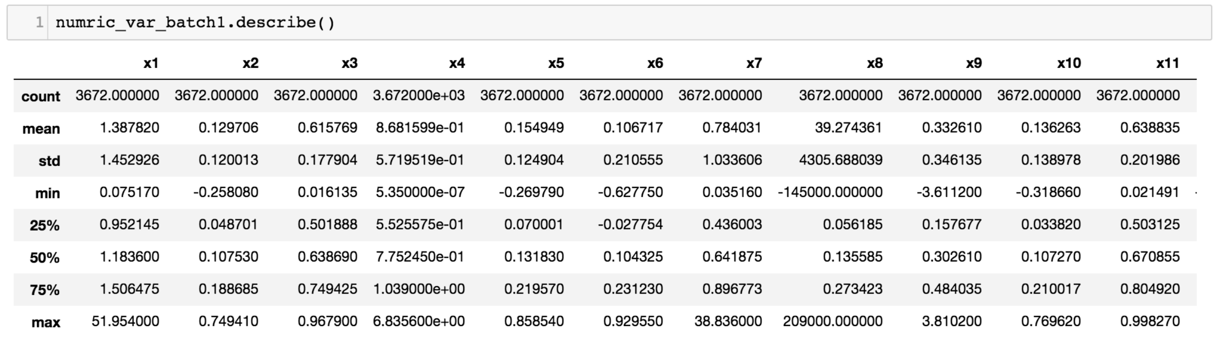
- Now we will take a look at each column by plotting boxplot, distance plot and trend plot. It is difficult to show and compare all 83 columns once, so a nice way would be to divide them into four batches. The code below only shows one batch.

1.



```
numric_var_batch1 = data.loc[:,'x1':'x20']
numric_var_batch1.head()
```

	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x14	x15	x16
0	1.2810	0.022934	0.87454	1.21640	0.060940	0.188270	0.52510	0.018854	0.182790	0.006449	0.85822	2.00580	0.125460	6.9706	4.6512	0.050100
1	1.2700	0.006454	0.82067	1.00490	-0.014080	0.181040	0.62288	0.006423	0.035991	0.001795	0.85152	-0.48644	0.179330	4.5764	3.7521	-0.014011
2	1.0529	-0.059379	0.92242	0.72926	0.020476	0.044865	0.43292	-0.081423	-0.765400	-0.054324	0.89314	0.41220	0.077578	11.8900	2.4884	0.028077
3	1.1131	-0.015229	0.85888	0.80974	0.076037	0.091033	0.67546	-0.018807	-0.107910	-0.065316	0.89581	0.99490	0.141120	6.0862	1.6382	0.093904
4	1.0623	0.107020	0.81460	0.83593	0.199960	0.047800	0.74200	0.128030	0.577250	0.094075	0.81549	3.01470	0.185400	4.3938	1.6169	0.239210



```
numric_var_batch1.describe()
```

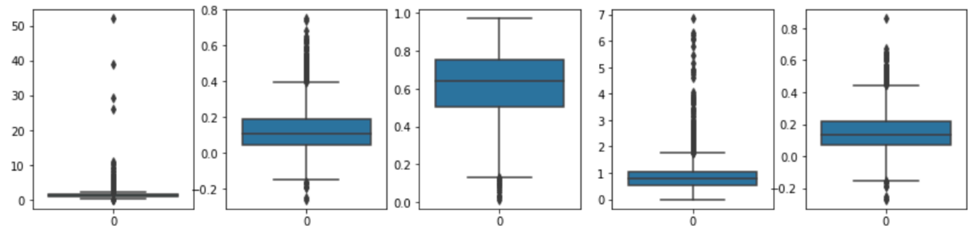
	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11
count	3672.000000	3672.000000	3672.000000	3.672000e+03	3672.000000	3672.000000	3672.000000	3672.000000	3672.000000	3672.000000	3672.000000
mean	1.387820	0.129706	0.615769	8.681599e-01	0.154949	0.106717	0.784031	39.274361	0.332610	0.136263	0.638835
std	1.452926	0.120013	0.177904	5.719519e-01	0.124904	0.210555	1.033606	4305.688039	0.346135	0.138978	0.201986
min	0.075170	-0.258080	0.016135	5.350000e-07	-0.269790	-0.627750	0.035160	-145000.000000	-3.611200	-0.318660	0.021491
25%	0.952145	0.048701	0.501888	5.525575e-01	0.070001	-0.027754	0.436003	0.056185	0.157677	0.033820	0.503125
50%	1.183600	0.107530	0.638690	7.752450e-01	0.131830	0.104325	0.641875	0.135585	0.302610	0.107270	0.670855
75%	1.506475	0.188685	0.749425	1.039000e+00	0.219570	0.231230	0.896773	0.273423	0.484035	0.210017	0.804920
max	51.954000	0.749410	0.967900	6.835600e+00	0.858540	0.929550	38.836000	209000.000000	3.810200	0.769620	0.998270

2

```
In [9]:
batch1_list = []
for n in range(1,21):
    batch1_list.append('x' + str(n))
```

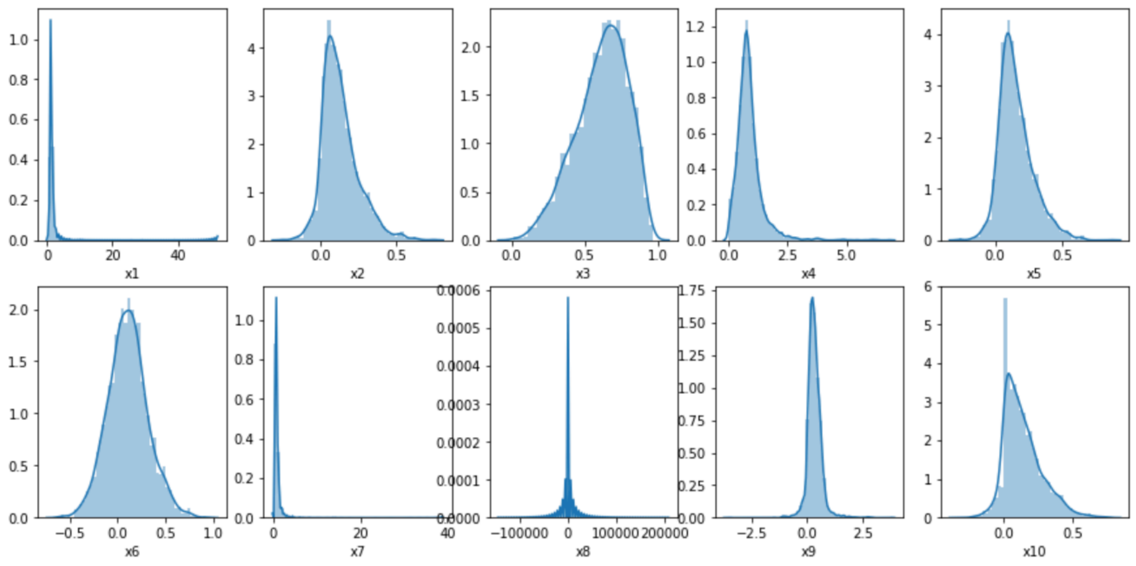
```
In [10]:
count = 1
for x in batch1_list:
    plt.subplot(4,5,count)
    sns.boxplot(data=data[x])
    count += 1
plt.show()
```

```
/Users/Rui/anaconda3/lib/python3.6/site-packages/seaborn/categorical.py:462: FutureWarning: remove_na is deprecated a
nd is a private function. Do not use.
  box_data = remove_na(group_data)
```



3

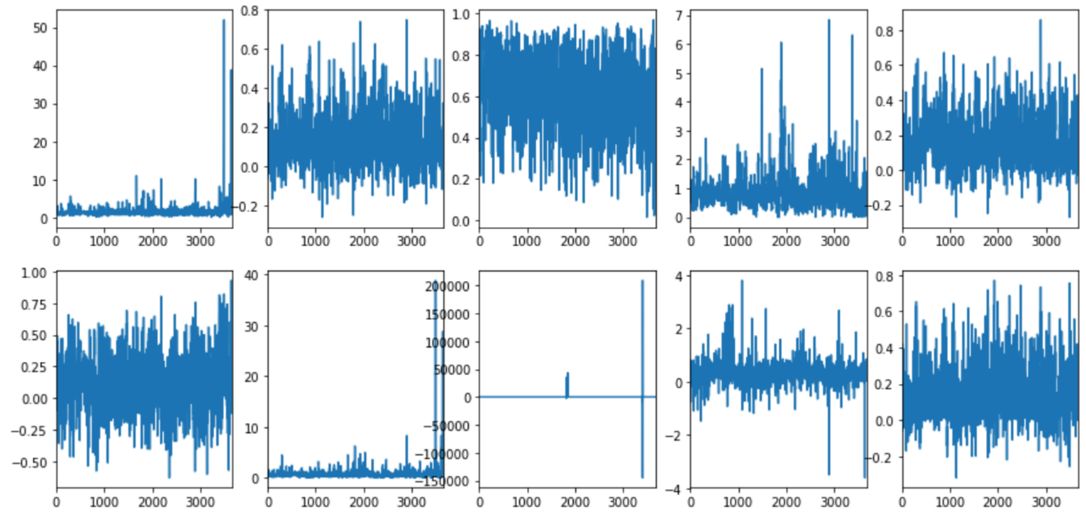
```
count = 1
for x in batch1_list:
    plt.subplot(4,5,count)
    sns.distplot(data[x])
    count += 1
plt.show()
```



```
count = 1
for x in batch1_list:
    plt.subplot(4,5,count)
    sns.tsplot(data=data[x])
    count += 1
plt.show()
```

```
/Users/Rui/anaconda3/lib/python3.6/site-packages/seaborn/timeseries.py:183: UserWarning: The tsplot function is
cated and will be removed or replaced (in a substantially altered version) in a future release.
  warnings.warn(msg, UserWarning)
```

4.



- I also tried to detect outliers, however due to unhighly imbalance and lack of proper tagging of the variables, it is hard to remove any of them.

```
def outliers_iqr(ys):
    quartile_1, quartile_3 = np.percentile(ys, [15, 85])
    iqr = quartile_3 - quartile_1
    lower_bound = quartile_1 - (iqr * 1.5)
    upper_bound = quartile_3 + (iqr * 1.5)
    return np.where((ys > upper_bound) | (ys < lower_bound))
```

```
outlier_list = []
for column_index in list(range(1,80)) + list(range(81,83)):
    outlier_list = outlier_list+ outliers_iqr(data['x' + str(column_index)])[0].tolist()
test_d = dict([i, outlier_list.count(i)] for i in outlier_list)
print('Number of rows has outliers is: ' + str(len(test_d)))
```

```
Number of rows has outliers is: 2296
```

```
a = 0
b = 0
for x in list(test_d.keys()):
    if data.iloc[x,2]>-0.5:
        a = a + 1
    else: b = b + 1
print('Number of healthy records with outliers is: ' + str(a))
print('Number of healthy records with outliers is: ' + str(b))
```

```
Number of healthy records with outliers is: 2201
Number of healthy records with outliers is: 95
```

- Another necessary step for understanding dataset is to look at thier correlation, since in previous plots we can see that many vairables seems to have similar curves and range. I drew a correlation matrix using the corr() function and also plot a heatmap using sns. I'll drop these columns later using variance inflation factor(vif).

```python
corr = data.loc[:,'Financial Distress':'x83'].corr()
corr
```

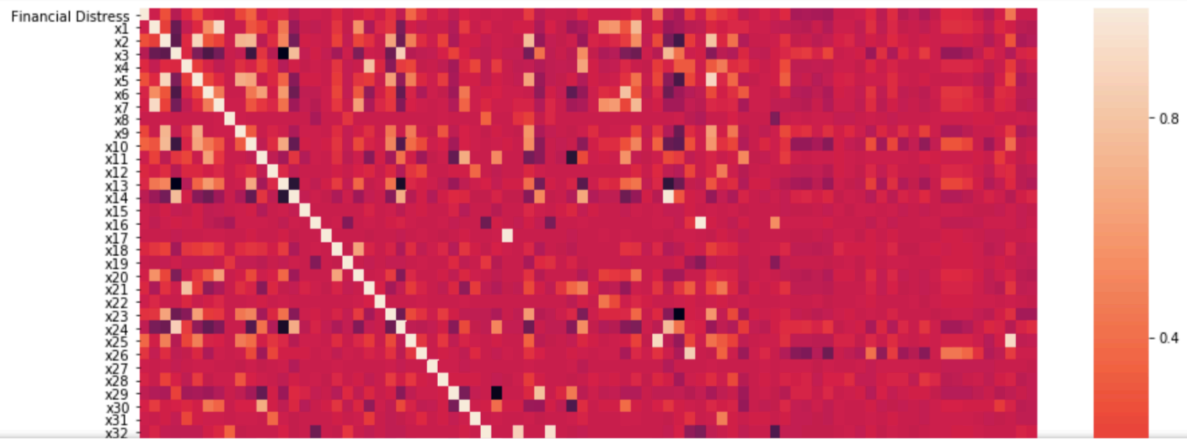
	Financial Distress	x1	x2	x3	x4	x5	x6	x7	x8	x9	...	x74	x75
Financial Distress	1.000000	0.030928	0.272978	-0.089264	0.116837	0.274435	0.077260	0.042412	-0.006382	0.219091	...	-0.018746	0.110030
x1	0.030928	1.000000	0.208851	-0.440321	-0.035504	0.109543	0.535425	0.914489	0.007329	-0.000506	...	0.007974	0.135594
x2	0.272978	0.208851	1.000000	-0.504509	0.114046	0.892480	0.298078	0.214098	0.014020	0.729858	...	0.014677	0.010825
x3	-0.089264	-0.440321	-0.504509	1.000000	0.094481	-0.356252	-0.593620	-0.398730	-0.006290	-0.030976	...	0.014103	-0.227217

```python
# For each column in corr matrix, a value over 0.7 will be considered as  highly correlated
# And a value over 0.9 is very highly correlated
corr_dict = {}
for col in range(1,84):
    corr_dict.update(
        {'x'+str(col):['x'+str(x+1)
                       for x in [i for i, x in enumerate((abs(corr['x'+str(col)]>=0.7)))if x if i!=col]]}
    )
```

```python
corr_dict = {i:j for i,j in corr_dict.items() if j!=[]}
print('The number of all columns that has high correlationship with others is: ' + str(len(corr_dict)))
corr_dict
```

```
The number of all columns that has high correlationship with others is: 43
```

```python
sns.heatmap(corr,
            xticklabels=corr.columns.values,
            yticklabels=corr.columns.values)
plt.show()
```



- Now since there is one categorical column x80 in the dataset, we need to perform one hot coding first.

```python
#Remove first two columns as they are irrelavent, also rename first column
data_preprocess = data.drop(data.columns[[0,1]], axis=1, inplace=False)
data_preprocess.rename(columns={'Financial Distress':'Financial_Distress'}, inplace=True)
data_preprocess.head(3)
```

	Financial_Distress	x1	x2	x3	x4	x5	x6	x7	x8	x9	...	x74	x75	x76	x77
0	0.010636	1.2810	0.022934	0.87454	1.21640	0.060940	0.188270	0.52510	0.018854	0.182790	...	85.437	27.07	26.102	16.0
1	-0.455970	1.2700	0.006454	0.82067	1.00490	-0.014080	0.181040	0.62288	0.006423	0.035991	...	107.090	31.31	30.194	17.0
2	-0.325390	1.0529	-0.059379	0.92242	0.72926	0.020476	0.044865	0.43292	-0.081423	-0.765400	...	120.870	36.07	35.273	17.0

3 rows × 84 columns

```python
#Check how many categories for x80
x80_cats = data_preprocess['x80'].value_counts()
len(x80_cats)
```

```
37
```

```python
#Use one hot coding to encode the categorical variable
data_preprocess = pd.get_dummies(data_preprocess, columns=["x80"])
data_preprocess.head()
```

x4	x5	x6	x7	x8	x9	...	x80_28	x80_29	x80_30	x80_31	x80_32	x80_33	x80_34	x80_35	x80_36	x80_37
.21640	0.060940	0.188270	0.52510	0.018854	0.182790	...	0	0	0	0	0	0	0	0	0	0
.00490	-0.014080	0.181040	0.62288	0.006423	0.035991	...	0	0	0	0	0	0	0	0	0	0
.72926	0.020476	0.044865	0.43292	-0.081423	-0.765400	...	0	0	0	0	0	0	0	0	0	0
.80974	0.076037	0.091033	0.67546	-0.018807	-0.107910	...	0	0	0	0	0	0	0	0	0	0
.83593	0.199960	0.047800	0.74200	0.128030	0.577250	...	0	1	0	0	0	0	0	0	0	0

- Then normalize the dataset by following code.

```python
#Normalization
x = data_preprocess.values #returns a numpy array
min_max_scaler = preprocessing.MinMaxScaler()
x_scaled = min_max_scaler.fit_transform(x)
data_preprocess = pd.DataFrame(x_scaled, columns=data_preprocess.columns)
data_preprocess.head(3)
```

	Financial_Distress	x1	x2	x3	x4	x5	x6	x7	x8	x9	...	x80_28	x80_29	x80_30	x8
0	0.063068	0.023243	0.278925	0.901909	0.177951	0.293115	0.523997	0.012627	0.409605	0.511223	...	0.0	0.0	0.0	
1	0.059663	0.023031	0.262568	0.845308	0.147010	0.226627	0.519354	0.015147	0.409605	0.491442	...	0.0	0.0	0.0	
2	0.060616	0.018846	0.197224	0.952215	0.106686	0.257253	0.431911	0.010251	0.409604	0.383459	...	0.0	0.0	0.0	

- Reducing dimension was done by two algorithms, one by removing columns with high correlation value using vif, one by checking p-value of the linear analysis result. Once both processes were done, there were only 25 independent variables left.

```python
#VIF
#gather features
features = "+".join(data_preprocess.drop(['Financial_Distress'], axis=1))
# get y and X dataframes based on this regression:
y, X = dmatrices('Financial_Distress ~' + features, data_preprocess, return_type='dataframe')

# For each X, calculate VIF and save in dataframe
vif = pd.DataFrame()
vif["VIF Factor"] = [variance_inflation_factor(X.values, i) for i in range(X.shape[1])]
vif["features"] = X.columns
```

```
#70 columns left, run a multiple linear analysis
X = data_preprocess.drop(['Financial_Distress'], axis=1)
y = data_preprocess.Financial_Distress

X2 = sm.add_constant(X)
est = sm.OLS(y, X2)
est2 = est.fit()
print(est2.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:     Financial_Distress   R-squared:                       0.395
Model:                            OLS   Adj. R-squared:                  0.383
Method:                 Least Squares   F-statistic:                     34.01
Date:                Sat, 03 Feb 2018   Prob (F-statistic):               0.00
Time:                        21:42:01   Log-Likelihood:                 10197.
No. Observations:                3672   AIC:                        -2.025e+04
Df Residuals:                    3602   BIC:                        -1.982e+04
Df Model:                          69
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         -5.7682      7.591     -0.760      0.447     -20.651       9.114
x1             0.0393      0.034      1.163      0.245      -0.027       0.106
```

Run machine learning algorithms

Split dataset and define cost methods

```
random.seed(0)
y = data_preprocess['Financial_Distress'] #Define dependent variable as y
df = data_preprocess.drop(data_preprocess.columns[[0]], axis=1, inplace=False) #Define independent variables x1-x83
```

```
X_train, X_test, y_train, y_test = train_test_split(df, y, test_size=0.3)
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(2570, 69) (2570,)
(1102, 69) (1102,)
```

```
def MSE(y_true,y_pred):
    mse = mean_squared_error(y_true, y_pred)
    return mse

def R2(y_true,y_pred):
    #Best possible score is 1, if it's negative, then the model perform worse than a horizontal line.
    r2 = r2_score(y_true, y_pred)
    return r2
def two_score(y_true,y_pred):
    MSE(y_true,y_pred) #set score here and not below if using MSE in GridCV
    score = R2(y_true,y_pred)
    return score

def two_scorer():
    return make_scorer(two_score, greater_is_better=True)
```

Multiple linear regression

Multiple linear regression is straight-forward with no parameters to adjust. I am running this model because in machine learning, sometimes a complex question can be solved by a seemly easy model. However, even though the MSE score is low in this case, the negative R-square score indicates a poor model

```
mlinear_model = linear_model.LinearRegression()
lm = mlinear_model.fit(X_train, y_train)
mlinear_model.predict(X_test)
```

```
array([ 0.07379511,  0.06480222,  0.1422211 , ...,  0.07048367,
        0.06392812,  0.07432553])
```

```
# The coefficients
print('Coefficients: \n', mlinear_model.coef_)
print('Mean squared error: %.2f' % MSE(y_test, mlinear_model.predict(X_test))) #not bad.
print('Variance score: %.2f' % R2(y_test, mlinear_model.predict(X_test))) #...this is horrible
```

```
Coefficients: 
 [  4.13568773e-02   7.70896915e-02  -4.51245590e-02   3.33198246e-02
   1.38424775e-02   8.59817358e+00   1.52963061e-01  -8.16134246e-03
   3.76491538e-02  -1.15240082e-02   1.86389623e-02   1.23182681e+02
   1.74891557e-01   1.44443834e-02   2.39493641e+00  -1.30625404e-02
  -1.36100193e-02  -2.07323993e-02  -1.64373616e+00  -4.09804119e-03
   2.74027664e-02   5.49181503e-03   2.65005330e-02  -9.64699555e-03
  -1.99338513e-02  -1.06058248e+01   1.67095155e-02  -1.71410608e-01
   2.43293088e+01  -2.87166492e-02  -4.71252613e-02  -2.81451686e+00
   9.08527885e-03  -2.56614416e-04   1.24087866e-01  -2.17718529e-02
   7.23848407e-03   2.39162527e-02   2.99368795e-02  -1.12972876e-01
  -1.69193627e-02   8.87417963e-01   3.75654856e-03  -1.46151491e+02
   1.16273258e-03   8.74124918e-03  -4.37222568e-03  -8.03338888e-03
   1.11044840e-02   1.22934869e-02   9.13332923e+00  -1.32233631e-02
  -7.35148799e-03   4.99774125e-03  -2.07522219e-03  -3.24958597e-03
   3.19854781e-03   1.00859992e-02  -5.40172488e-03   1.68823911e-03
  -5.26369253e-03  -3.31635318e-03  -4.37536061e-03   6.47845553e-06
   9.16840153e-01   2.70120015e-03  -2.53629854e-03   8.33762448e-03
   5.45033012e-04]
Mean squared error: 0.02
Variance score: -167.00
```

Support vector regression

For SVR, I tried two different kernels: rbf and linear, and it turns out the rbf always performs better. However, since all changes were done in the same cell, no records were left for these efforts. The last set of parameters are listed here, the best paramter chose by GridSearchCV can reach a MSE with nearly 0, and a R2 to 0.54.

SVR

```
svr_model = svm.SVR(gamma=0.001, C=100, epsilon = 0.001)
svr_model.fit(X_train, y_train)
```

```
SVR(C=100, cache_size=200, coef0=0.0, degree=3, epsilon=0.001, gamma=0.001,
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

```
print('Mean squared error: %.2f' % MSE(y_test, svr_model.predict(X_test))) #good!
print('Variance score: %.2f' % R2(y_test, svr_model.predict(X_test))) #fine
```

```
Mean squared error: 0.00
Variance score: 0.51
```

```
def svr_param_selection(X, y, nfolds):
    tuned_parameters = {'kernel': ['rbf'], 'gamma': [0.001, 0.01, 0.1, 1],
                     'C': [1, 5, 10, 100, 1000], 'epsilon' : [0.001, 0.01, 0.1]}
    grid_search = GridSearchCV(svm.SVR(), tuned_parameters, cv=nfolds, scoring=two_scorer())
    grid_search.fit(X, y)
    grid_search.best_params_
    return grid_search.best_params_
```

```
svr_param_selection(X_train, y_train, 5)
```

```
{'C': 1, 'epsilon': 0.001, 'gamma': 0.1, 'kernel': 'rbf'}
```

```
svr_model_2 = svm.SVR(kernel='rbf', C=1, gamma=0.1, epsilon=0.001)
svr_model_2.fit(X_train, y_train)
```

```
SVR(C=1, cache_size=200, coef0=0.0, degree=3, epsilon=0.001, gamma=0.1,
  kernel='rbf', max_iter=-1, shrinking=True, tol=0.001, verbose=False)
```

```
print('Mean squared error: %.2f' % MSE(y_test, svr_model_2.predict(X_test)))
print('Variance score: %.2f' % R2(y_test, svr_model_2.predict(X_test))) #better
```

```
Mean squared error: 0.00
Variance score: 0.54
```

Random forest regression

The same process for Random Forest Regression, after tune, the R2 score can achieve 0.52.

Random forest

```
rfg_model = RandomForestRegressor(max_depth=2, max_features='auto', n_estimators=10, random_state=0)
rfg_model.fit(X_train, y_train)
```

```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=2,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=10, n_jobs=1,
           oob_score=False, random_state=0, verbose=0, warm_start=False)
```

```
print(MSE(y_test, rfg_model.predict(X_test)))
print(R2(y_test, rfg_model.predict(X_test))) #fine
```

```
7.40006257101e-05
0.455232750582
```

```
def rf_param_selection(X, y, nfolds):
    tuned_parameters = {'max_depth': range(2,6),
                        'n_estimators':[1,5, 10,100],
                        'max_features' : ['auto', 'log2', 'sqrt'],
                        'random_state' : [0,1]}
    grid_search = GridSearchCV(RandomForestRegressor(), tuned_parameters, cv=nfolds, scoring=two_scorer())
    grid_search.fit(X, y)
    grid_search.best_params_
    return grid_search.best_params_
```

```
rf_param_selection(X_train, y_train, 10)
```

```
{'max_depth': 5, 'max_features': 'log2', 'n_estimators': 1, 'random_state': 1}
```

```
rfg_model_2 = RandomForestRegressor(max_depth=6, max_features='auto', n_estimators=100, random_state=0)
rfg_model_2.fit(X_train, y_train)
```

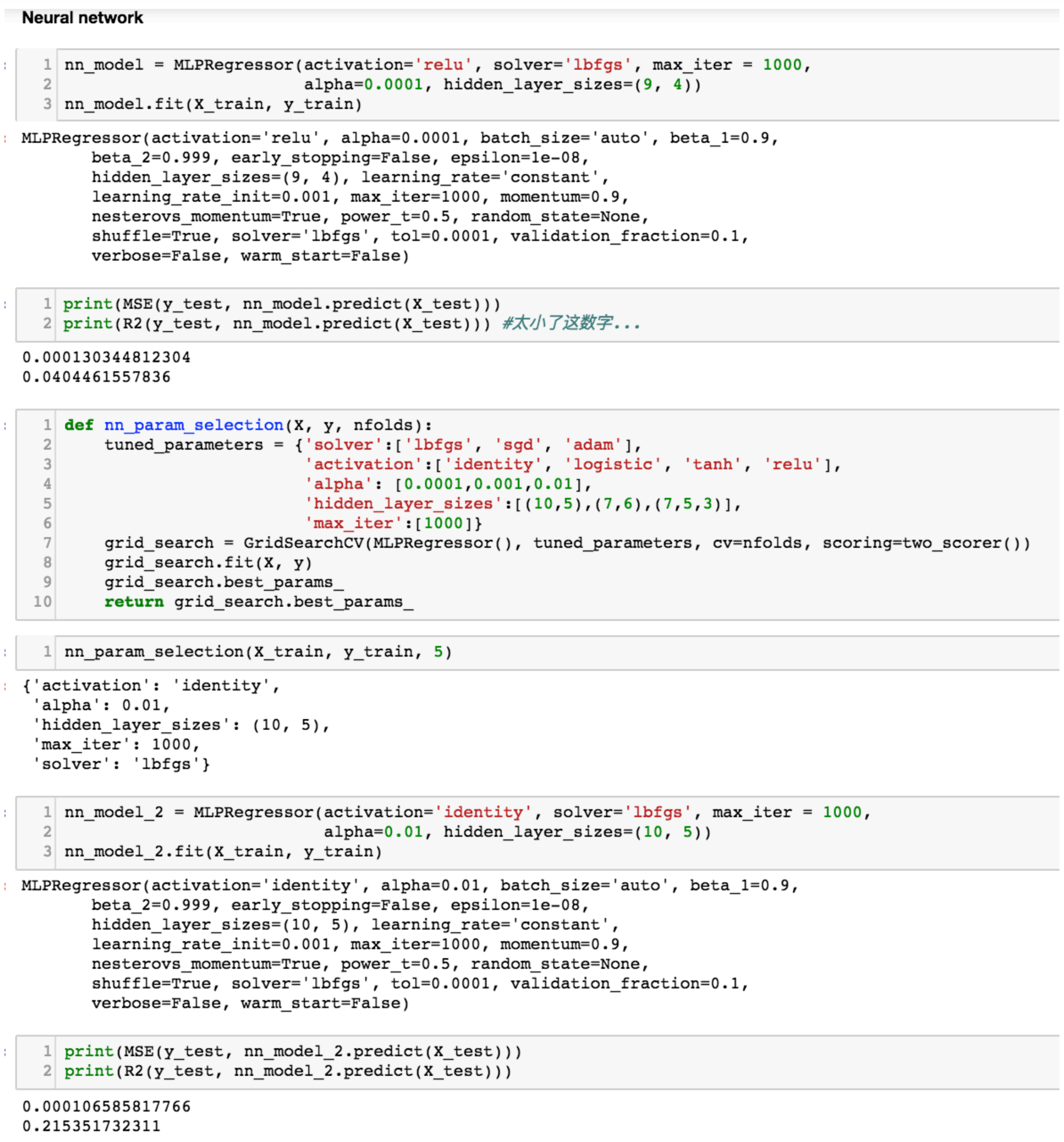
```
RandomForestRegressor(bootstrap=True, criterion='mse', max_depth=6,
           max_features='auto', max_leaf_nodes=None,
           min_impurity_decrease=0.0, min_impurity_split=None,
           min_samples_leaf=1, min_samples_split=2,
           min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
           oob_score=False, random_state=0, verbose=0, warm_start=False)
```

```
print(MSE(y_test, rfg_model_2.predict(X_test)))
print(R2(y_test, rfg_model_2.predict(X_test))) #better
```

```
6.37191745766e-05
0.530921270784
```

Neural network

And for neural network. I tried more than 10 combines, howevre this is already the best score I can get.



Neural network

```python
nn_model = MLPRegressor(activation='relu', solver='lbfgs', max_iter = 1000,
                        alpha=0.0001, hidden_layer_sizes=(9, 4))
nn_model.fit(X_train, y_train)
```

```
MLPRegressor(activation='relu', alpha=0.0001, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(9, 4), learning_rate='constant',
       learning_rate_init=0.001, max_iter=1000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=None,
       shuffle=True, solver='lbfgs', tol=0.0001, validation_fraction=0.1,
       verbose=False, warm_start=False)
```

```python
print(MSE(y_test, nn_model.predict(X_test)))
print(R2(y_test, nn_model.predict(X_test))) #太小了这数字...
```

```
0.000130344812304
0.0404461557836
```

```python
def nn_param_selection(X, y, nfolds):
    tuned_parameters = {'solver':['lbfgs', 'sgd', 'adam'],
                        'activation':['identity', 'logistic', 'tanh', 'relu'],
                        'alpha': [0.0001,0.001,0.01],
                        'hidden_layer_sizes':[(10,5),(7,6),(7,5,3)],
                        'max_iter':[1000]}
    grid_search = GridSearchCV(MLPRegressor(), tuned_parameters, cv=nfolds, scoring=two_scorer())
    grid_search.fit(X, y)
    grid_search.best_params_
    return grid_search.best_params_
```

```python
nn_param_selection(X_train, y_train, 5)
```

```
{'activation': 'identity',
 'alpha': 0.01,
 'hidden_layer_sizes': (10, 5),
 'max_iter': 1000,
 'solver': 'lbfgs'}
```

```python
nn_model_2 = MLPRegressor(activation='identity', solver='lbfgs', max_iter = 1000,
                          alpha=0.01, hidden_layer_sizes=(10, 5))
nn_model_2.fit(X_train, y_train)
```

```
MLPRegressor(activation='identity', alpha=0.01, batch_size='auto', beta_1=0.9,
       beta_2=0.999, early_stopping=False, epsilon=1e-08,
       hidden_layer_sizes=(10, 5), learning_rate='constant',
       learning_rate_init=0.001, max_iter=1000, momentum=0.9,
       nesterovs_momentum=True, power_t=0.5, random_state=None,
       shuffle=True, solver='lbfgs', tol=0.0001, validation_fraction=0.1,
       verbose=False, warm_start=False)
```

```python
print(MSE(y_test, nn_model_2.predict(X_test)))
print(R2(y_test, nn_model_2.predict(X_test)))
```

```
0.000106585817766
0.215351732311
```

Results

All four algorithms perform well if only taking MSE as loss function, however, if take R2 into consideration, then support vector regression has a much better performance.

Reference

Kaggle: https://www.kaggle.com/

Wikipedia: https://www.wikipedia.org/

Stackoverflow: https://stackoverflow.com/

StackExchange: https://stackexchange.com/