Classification Model Analysis for Machine Learning

Main objective of analysis

The objective of this analysis is to create, evaluate and recommend proper classification models to predict the target.
The analysis will cover major classification models and approaches such as Linear Regression, K - Nearest Neighbors, Support Vector Machine, etc.
The analysis will also list out the follow - up actions that could be followed after some models are recommended.

Brief description of the data set and a summary of its attributes

The idea of this analysis is to recommend classification models with good potentials to predict public trading stocks in the U.S market. Thus, the data set to be analyzed is the collection of the U.S stock statistics from 2014 to 2018.

The source of this data set is as following:

https://www.kaggle.com/cnic92/200-financial-indicators-of-us-stocks-20142018

The data set contains more than 200 financial indicators covering revenue, profitability, cash flow, inventory, expense, liability and other perspectives. In reality, it would be too complex to include all indicators into the classification model. Therefore, it is necessary to determine and extract the most valuable indicators as our model features.

The detail of associate data exploratory, feature determination and screening is to be described in the following paragraphs.

Data load, pre - processing and exploration

The data set contains 5 data files at the beginning:

- 2014_Financial_Data.csv
- 2015_Financial_Data.csv
- 2016_Financial_Data.csv
- 2017_Financial_Data.csv
- 2018_Financial_Data.csv

Each file contains 3 to 4 thousand records and 224 columns. The information is summarized as following:

```
In [14]: print('2014_Financial_Data.csv shape: {}'.format(df1.shape))
         print('2015_Financial_Data.csv shape: {}'.format(df2.shape))
         print('2016_Financial_Data.csv shape: {}'.format(df3.shape))
         print('2017_Financial_Data.csv shape: {}'.format(df4.shape))
         print('2018_Financial_Data.csv shape: {}'.format(df5.shape))

         2014_Financial_Data.csv shape: (3808, 224)
         2015_Financial_Data.csv shape: (4120, 224)
         2016_Financial_Data.csv shape: (4797, 224)
         2017_Financial_Data.csv shape: (4960, 224)
         2018_Financial_Data.csv shape: (4392, 224)
```

In order to facilitate the follow - up tasks, I would like to concatenate these data files into one, add an extra column (“Data Year”) to identify yearly data and rename yearly PRICE VAR [%] column to “Next Year VAR%”.

```
In [14]: # Add Year column
         df1['Data Year'] = 2014
         df2['Data Year'] = 2015
         df3['Data Year'] = 2016
         df4['Data Year'] = 2017
         df5['Data Year'] = 2018
         # Rename Price VAR column
         df1.rename(columns={'2015 PRICE VAR [%]':'Next Year VAR%'}, inplace=True)
         df2.rename(columns={'2016 PRICE VAR [%]':'Next Year VAR%'}, inplace=True)
         df3.rename(columns={'2017 PRICE VAR [%]':'Next Year VAR%'}, inplace=True)
         df4.rename(columns={'2018 PRICE VAR [%]':'Next Year VAR%'}, inplace=True)
         df5.rename(columns={'2019 PRICE VAR [%]':'Next Year VAR%'}, inplace=True)
```

```
In [15]: # concate 5-year data files
         stock_data = pd.concat([df1,df2,df3,df4,df5], axis=0)
         print(stock_data.shape)
         print('Total rows are equal? ' + str(stock_data.shape[0] == df1.shape[0]+df2.shape[0]+df3.shape[0]+df4.shape[0]+df5.shape[0]))

         (22077, 225)
         Total rows are equal? True
```

```
In [16]: # Re-order data columns
         columns = stock_data.columns.to_list()
         new_columns = []
         new_columns.append(columns[-1])
         new_columns.append(columns[0])
         new_columns.append(columns[-2])
         new_columns.append(columns[-3])
         new_columns.extend(columns[1:-3])

         stock_data = stock_data[new_columns]
         stock_data
```

Out[16]:

	Data Year	Stock	Class	Next Year VAR%	Revenue	Revenue Growth	Cost of Revenue	Gross Profit	R&D Expenses	SG&A Expense	...	10Y Dividend per Share Growth (per Share)	5Y Dividend per Share Growth (per Share)	3Y Dividend per Share Growth (per Share)	Rece
0	2014	PG	0	-9.323276	7.440100e+10	-0.0713	3.903000e+10	3.537100e+10	0.000000e+00	2.146100e+10	...	0.1013	0.0834	0.0751	
1	2014	VIPS	0	-25.512193	3.734148e+09	1.1737	2.805625e+09	9.285226e+08	1.083303e+08	3.441414e+08	...	NaN	NaN	NaN	
2	2014	KR	1	33.118297	9.837500e+10	0.0182	7.813800e+10	2.023700e+10	0.000000e+00	1.519600e+10	...	0.0000	0.1215	0.1633	
3	2014	RAD	1	2.752291	2.552641e+10	0.0053	1.820268e+10	7.323734e+09	0.000000e+00	6.561162e+09	...	0.0000	0.0000	0.0000	
4	2014	GIS	1	12.897715	1.790960e+10	0.0076	1.153980e+10	6.369800e+09	0.000000e+00	3.474300e+09	...	0.1092	0.1250	0.1144	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
4387	2018	YRIV	0	-90.962099	0.000000e+00	0.0000	0.000000e+00	0.000000e+00	0.000000e+00	3.755251e+06	...	NaN	NaN	0.0000	
4388	2018	YTEN	0	-77.922077	5.560000e+05	-0.4110	0.000000e+00	5.560000e+05	4.759000e+06	5.071000e+06	...	0.0000	0.0000	0.0000	
4389	2018	ZKIN	0	-17.834400	5.488438e+07	0.2210	3.659379e+07	1.829059e+07	1.652633e+06	7.020320e+06	...	NaN	NaN	0.0000	
4390	2018	ZOM	0	-73.520000	0.000000e+00	0.0000	0.000000e+00	0.000000e+00	1.031715e+07	4.521349e+06	...	NaN	NaN	NaN	
4391	2018	ZYME	1	209.462222	5.301900e+07	0.0243	0.000000e+00	5.301900e+07	5.668400e+07	2.945700e+07	...	NaN	NaN	0.0000	

22077 rows × 225 columns

Due to the massive volume of data columns and records, the next step that I am going to do is to screen and define independent variables and then to do the data cleaning.

Data screening and variable definition

Based on the data set, here I extract the most representative financial indicators as the independent variables:

1. Profit Margin
2. Net Profit Margin
3. Operating Profit Margin
4. EPS
5. Return on Assets (ROA)
6. Return on Equity (ROE)
7. Price to Free Cash Flows Ratio
8. Price Earnings Ratio
9. Price Earnings to Growth Ratio
10. Asset Turnover
11. Current Ratio
12. Quick Ratio
13. Debt Equity Ratio
14. Interest Coverage
15. Receivables Turnover
16. Invertory Turnover
17. Working Capital

The logic is to leverage these well - known financial figures as the key indicators to evaluate if the stock is worth buying or not. In addition to these representative financial figures, I also put “Next Year VAR%” into the variable list.

Out[20]:

	Data Year	Stock	Class	Next Year VAR%	Profit Margin	Net Profit Margin	EPS	returnOnAssets	returnOnEquity	priceToFreeCashFlowsRatio	priceEarningsRatio	priceEarning
0	2014	PG	0	-9.323276	0.1560	0.1565	4.1900	0.5765	0.1664	21.0348	18.7566	
1	2014	VIPS	0	-25.512193	0.0058	0.0364	0.2396	0.0403	0.3294	1.3589	81.5526	
2	2014	KR	1	33.118297	0.0150	0.0154	1.4700	0.1011	0.2821	14.6302	12.0340	
3	2014	RAD	1	2.752291	0.0080	0.0098	4.6000	0.0668	-0.1180	17.2736	28.6087	
4	2014	GIS	1	12.897715	0.1020	0.1019	2.9000	0.6265	0.2792	17.6902	18.7034	
...	...	...	...	...	...	...	...	...	...	...	...	
4387	2018	YRIV	0	-90.962099	-1.2310	0.0000	-0.0800	NaN	-0.0800	0.0000	0.0000	
4388	2018	YTEN	0	-77.922077	-16.4930	-16.4928	-0.9200	-0.8423	-1.6093	0.0000	0.0000	
4389	2018	ZKIN	0	-17.834400	0.1280	0.1279	0.5200	0.2228	0.1895	0.0000	6.1538	
4390	2018	ZOM	0	-73.520000	NaN	0.0000	-0.1800	-7.5619	-4.5523	0.0000	0.0000	
4391	2018	ZYME	1	209.462222	-0.6890	-0.6895	-1.2600	-0.2021	-0.2025	18.2699	0.0000	

22077 rows × 20 columns

Now the data set is ready to be further examined and cleaned. The data set schema is described as following:

Column	Type	Description
Data Year	int64	
Stock	object	Stock code
Class	int64	The target variable (dependent variable) that the model is going to predict. “0” means not worth buying and “1” is worth buying.
Next Year VAR%	float64	This is the next - year stock price variation.
Profit Margin	float64	Profit (% of revenue)
Net Profit Margin	float64	The profit amount over company revenue.
EPS	float64	Earning per share
returnOnAssets	float64	The profitability of the company’s assets.
returnOnEquity	float64	The profitability of the company’s equity.
priceToFreeCashFlowsRatio	float64	The relationship between stock share price and company’s free cash.
priceEarningsRatio	float64	P/E ratio.
priceEarningsToGrowthRatio	float64	PEG ratio. It is similar to P/E and takes company growth into consideration.
assetTurnover	float64	How much revenue is generated from company owned assets.
currentRatio	float64	This ratio indicates the cash amount that a company owns in hand.
quickRatio	float64	How fast to turn assets into cash.
debtEquityRatio	float64	Company debt and equity percentage.
Interest Coverage	float64	How much interest expense that a

		company has to pay per year.
Receivables Turnover	float64	How fast (per year) a company can collect money from the goods sold.
Inventory Turnover	float64	The frequency a company can “sell and refresh” its inventory per year. The higher the better.
Working Capital	float64	The difference between company current assets and current liabilities. The higher the better.

Data cleaning and feature engineering

- Deal with NA value
  Generally the NA value can be dropped or filled in with substitute values. In this analysis, I simply drop all NA values found in the data set. This is intuitive and reasonable since these important financial indicators are crucial to company and stock performance analysis and should not be left empty. Also we have no information to create make - up value and substitute the empty value. As a result I would rather drop all found NA values in the data set.

```
In [40]: def get_na_columns(df):
             columns = df.columns
             columns_has_na = []
             for column in columns:
                 has_na = df[column].isna().any().sum()
                 if(has_na>0):
                     columns_has_na.append(column)
             return columns_has_na
```

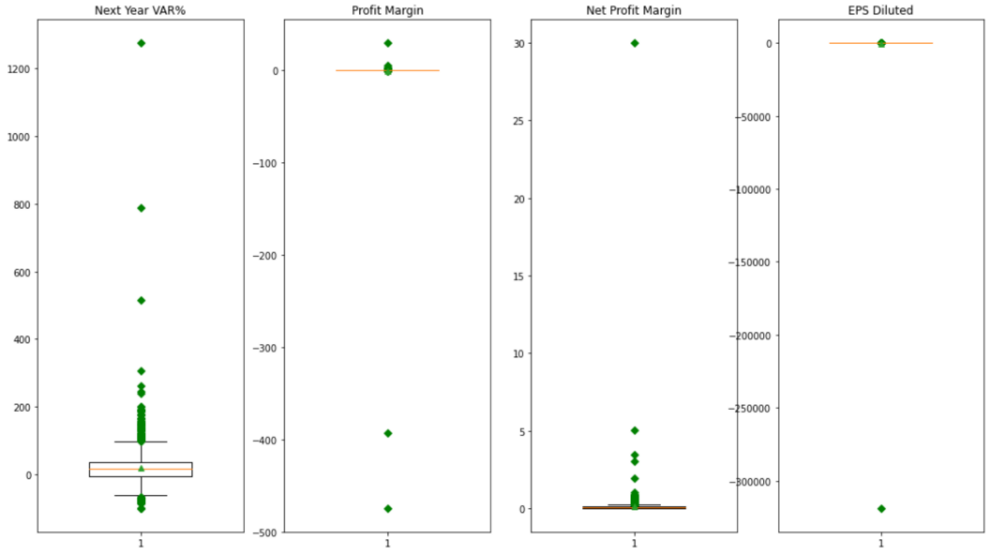
```
In [41]: # Deal with NA data
         # Focus on financial indicator dataset.
         # To maintain model quality, I drop the records with NA values instead of filling 0.
         indicator_data = indicator_data.dropna()
         indicator_data.shape

Out[41]: (9174, 20)
```

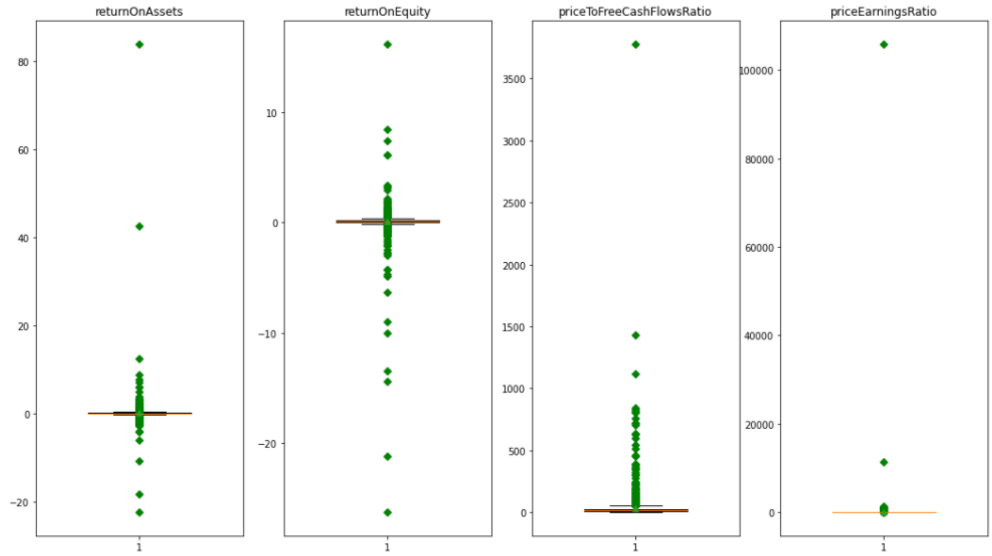
```
In [42]: # Verify if there is any na exists.
         s = get_na_columns(indicator_data)
         len(s)

Out[42]: 0
```

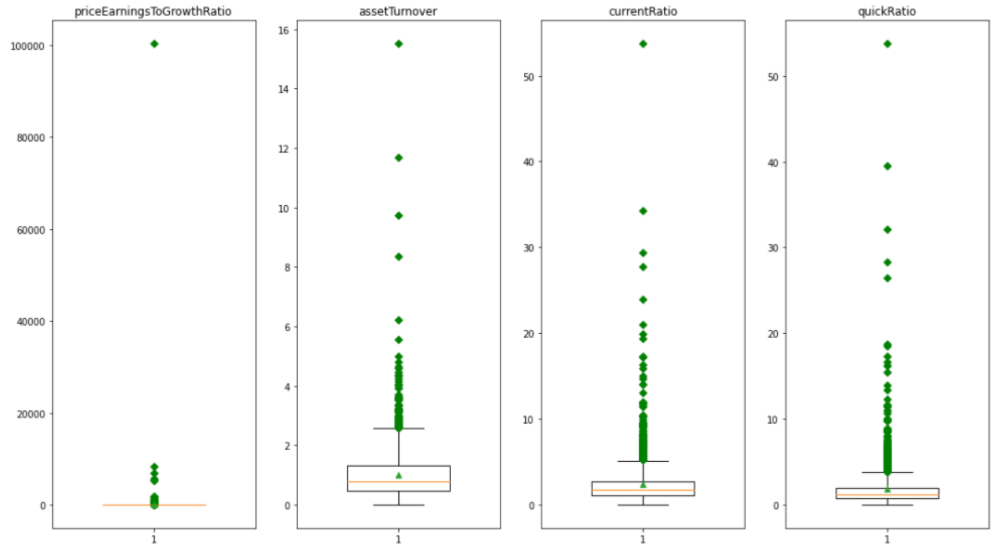
- Outlier
  There are 9174 records after dropping NA values. The next task is to see whether there are outliers in the data set.

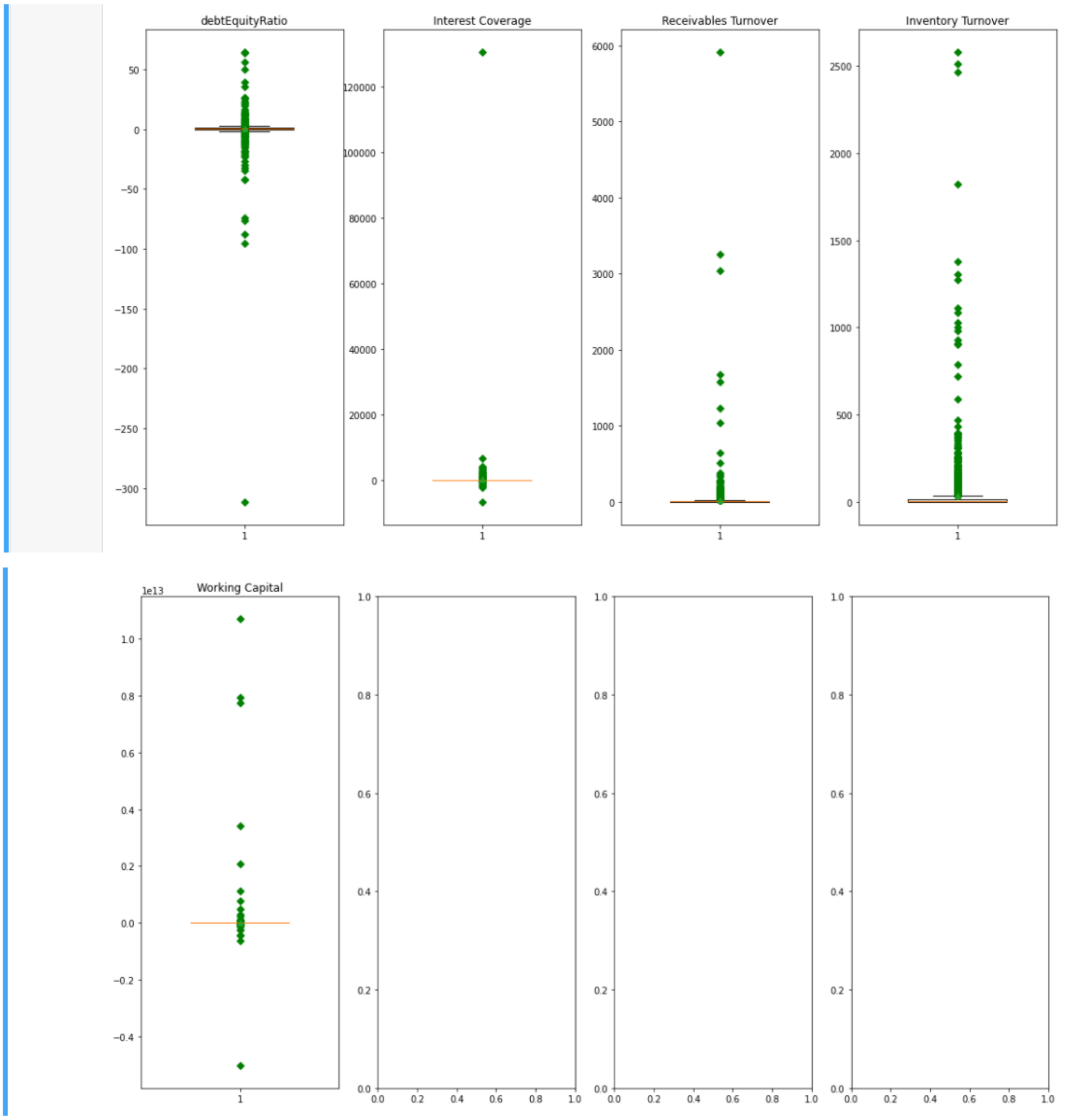
  I would like to use the box plot to visualize the data distribution of each feature (by using 2016 year data as example):

In [376]:
```
feature_column = indicator_data.columns.values.tolist()[3:]
get_box(indicator_data.loc[indicator_data['Data Year']==2016], feature_column)
# indicator_data.info()
```









It is obvious that there are some outliers in each feature. I would like to use the “1.5 IQR” method to find out the values of these outliers.
After reading some real outlier data as below, it comes to an interesting question of how to deal with the outlier data. Take stock code “TOPS” 2016 data (as below) as an example, the company had a very terrible EPS number (-319,100):

```
In [436]: def check_IQR(df, column):
              scale = 1.5
              describe_df = df[column].describe()
              Q1 = describe_df['25%']
              Q3 = describe_df['75%']
              IQR = Q3 - Q1
              lower_bound = Q1 - scale*IQR
              upper_bound = Q3 + scale*IQR
              return df[column].loc[(df[column]<lower_bound)|(df[column]>upper_bound)]
```

```
In [438]: a = check_IQR(indicator_data, 'EPS Diluted')
          a.sort_values(ascending=False)
          indicator_data.iloc[4339]

Out[438]: Data Year                              2016
          Stock                                  TOPS
          Class                                     0
          Next Year VAR%                     -99.9994
          Profit Margin                        -0.012
          Net Profit Margin                     0.037
          EPS Diluted                         -319100
          returnOnAssets                       0.0094
          returnOnEquity                       0.0231
          priceToFreeCashFlowsRatio                 0
          priceEarningsRatio                        0
          priceEarningsToGrowthRatio          0.40038
          assetTurnover                      0.198392
          currentRatio                          0.227
          quickRatio                        0.0717316
          debtEquityRatio                      1.8571
          Interest Coverage                    1.3401
          Receivables Turnover                355.413
          Inventory Turnover                   57.732
          Working Capital                 -1.5492e+07
          Name: 4339, dtype: object
```

I had also checked out the raw data it was not very clear if the EPS number is correct (since it showed very low weighted average shares outstanding):

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
	Revenue	Revenue Growth	Cost of Revenue	Gross Profit	R&D Ex	SG&A Expense	Operating Expenses	Operating Income	Interest Expense	Earnings before Tax	Income Tax Expense	Net Income - Non-C	Net Income - Di	Net Income	Preferre	Net Income Com	EPS	EPS Diluted	Weighted Average Shs Out
TOPS	28433000	1.1746	16948000	11485000	0	2906000	6637000	4848000	3093000	1052000	0	0	0	1052000	1E+06	-351000	-319100	-319100	1

Therefore, instead of applying substitute numbers such as mean or median to replace the outlier values, at this moment I would like to get rid of all the records that contain outliers. As a result, there are 2379 records left. This would be the baseline for the modeling stage later on.

In [439]:

```
# Use 1.5 IQR to isolate and drop outliers
check_columns = indicator_list[3:]
check_columns
tmp_index_list=[]
for check_column in check_columns:
    column_outlier_df = check_IQR(indicator_data, check_column)
    print('Column {} has {} outlier data'.format(check_column, column_outlier_df.shape[0]))
#     add index to temp list
    tmp_index_list.extend(column_outlier_df.index.values.tolist())

outlier_index = set(tmp_index_list)
# Try to drop the outliers
indicator_data.drop(index=outlier_index, inplace=True)
indicator_data.shape
indicator_data['Data Year'].value_counts()
```

```
Column Next Year VAR% has 249 outlier data
Column Profit Margin has 618 outlier data
Column Net Profit Margin has 621 outlier data
Column EPS Diluted has 583 outlier data
Column returnOnAssets has 937 outlier data
Column returnOnEquity has 963 outlier data
Column priceToFreeCashFlowsRatio has 738 outlier data
Column priceEarningsRatio has 957 outlier data
Column priceEarningsToGrowthRatio has 967 outlier data
Column assetTurnover has 386 outlier data
Column currentRatio has 683 outlier data
Column quickRatio has 769 outlier data
Column debtEquityRatio has 1041 outlier data
Column Interest Coverage has 1329 outlier data
Column Receivables Turnover has 1242 outlier data
Column Inventory Turnover has 1397 outlier data
Column Working Capital has 1614 outlier data
```

```
C:\Users\mikee\Anaconda3\lib\site-packages\pandas\core\frame.py:4174: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-ve
rsus-a-copy
  errors=errors,
```

Out[439]:

```
2018    524
2017    509
2016    462
2014    460
2015    424
Name: Data Year, dtype: int64
```

- Feature engineering
  Before modeling, the last thing that I would like to do is to standardize all the features into the same scale to avoid error interpretation from some modeling approach (such as KNN). I am using MinMaxScaler to re - scale the all minimum and maximum feature values between 0 and 1.

In [384]:

```
# Scale the rest features
scaler = MinMaxScaler()
# scaler = RobustScaler()
example_standardized = scaler.fit_transform(indicator_data[feature_column])
# example_standardized.shape
example_standardized_df = pd.DataFrame(data=example_standardized, columns=feature_column)
```

In [441]:

```
example_standardized_df.describe().T
```

Out[441]:

	count	mean	std	min	25%	50%	75%	max
Next Year VAR%	2379.0	0.501616	0.180708	0.0	0.381852	0.497295	0.629547	1.0
Profit Margin	2379.0	0.481987	0.133076	0.0	0.384824	0.452575	0.543767	1.0
Net Profit Margin	2379.0	0.270249	0.180212	0.0	0.136678	0.230098	0.351327	1.0
EPS Diluted	2379.0	0.393533	0.186757	0.0	0.247742	0.348387	0.487097	1.0
returnOnAssets	2379.0	0.484031	0.150597	0.0	0.369415	0.443326	0.561776	1.0
returnOnEquity	2379.0	0.426322	0.151998	0.0	0.317673	0.390844	0.499000	1.0
priceToFreeCashFlowsRatio	2379.0	0.294861	0.229194	0.0	0.124628	0.277793	0.419631	1.0
priceEarningsRatio	2379.0	0.389787	0.200353	0.0	0.251174	0.348801	0.491019	1.0
priceEarningsToGrowthRatio	2379.0	0.367108	0.206184	0.0	0.236792	0.326939	0.475298	1.0
assetTurnover	2379.0	0.369248	0.199712	0.0	0.217954	0.329662	0.491497	1.0
currentRatio	2379.0	0.405840	0.190381	0.0	0.264549	0.374619	0.512302	1.0
quickRatio	2379.0	0.353383	0.198243	0.0	0.211413	0.308005	0.456682	1.0
debtEquityRatio	2379.0	0.527066	0.144843	0.0	0.418203	0.497958	0.603957	1.0
Interest Coverage	2379.0	0.485409	0.149147	0.0	0.384609	0.435999	0.545623	1.0
Receivables Turnover	2379.0	0.347662	0.167352	0.0	0.241621	0.316843	0.418463	1.0
Inventory Turnover	2379.0	0.211373	0.198862	0.0	0.086309	0.175474	0.274395	1.0
Working Capital	2379.0	0.490422	0.167108	0.0	0.375892	0.439085	0.564245	1.0

Variation of models and model selection

Here to check again the data set that is going to be analyzed:

In [442]:
```
indicator_data.shape
a = indicator_data[['Data Year', 'Stock', 'Class']]
# a.dropna(inplace=True)
a.reset_index(inplace=True, drop=True)
a
example_standardized_df
indicator_data = pd.concat([a, example_standardized_df], axis=1)
indicator_data
```

Out[442]:

	Data Year	Stock	Class	Next Year VAR%	Profit Margin	Net Profit Margin	EPS Diluted	returnOnAssets	returnOnEquity	priceToFreeCashFlowsRatio	priceEarningsRatio	priceEarnin
0	2014	KO	1	0.492223	0.701897	0.560887	0.361290	0.922090	0.642943	0.389840	0.431504	
1	2014	NWL	1	0.564045	0.463415	0.239913	0.329032	0.821153	0.582967	0.395260	0.460333	
2	2014	CAG	1	0.580404	0.355014	0.093057	0.245161	0.360039	0.290084	0.159976	0.564784	
3	2014	SJM	1	0.602101	0.558266	0.366049	0.854194	0.686156	0.399640	0.302980	0.295337	
4	2014	DG	1	0.478266	0.444444	0.213013	0.563871	0.686983	0.554378	0.335102	0.294162	
...	...	...	...	...	...	...	...	...	...	...	...	
2374	2018	SXI	1	0.552404	0.414634	0.172664	0.523871	0.420436	0.337265	0.554549	0.595017	
2375	2018	TRNS	1	0.807879	0.387534	0.138859	0.259355	0.504137	0.405438	0.487975	0.308200	
2376	2018	WHLM	0	0.306354	0.314363	0.039985	0.175484	0.450910	0.240504	0.166397	0.603296	
2377	2018	XELB	1	0.610150	0.368564	0.111596	0.162581	0.416988	0.196721	0.071646	0.311823	
2378	2018	ZKIN	0	0.366746	0.631436	0.464922	0.221935	0.587286	0.553778	0.000000	0.101888	

2379 rows × 20 columns

The dimension is 2379 records with 20 columns.

1. **Define dependent and independent variables**

Among these columns, Data Year and Stock are for reference usage. The Class column is the dependent variable (X) and the rest 17 columns (y in numeric data type) are independent variables.

2. **Define train and test data set**

Before modeling, I would split the data set into train and test groups. In order to maintain the proportion of X in both train and test groups, I would “shuffle” the “X” by using StratifiedShuffleSplit. As a result, the train set contains 1665 records and the test set contains 714 records by setting 30% testing size.

In [444]:
```
# 1. Define X, y
# y = Class (recommend or not recommend)
# X = columns
y = indicator_data['Class'].values
X = indicator_data[feature_column].values
# 2. Check if we need shuffle the independent variables

indicator_data['Class'].value_counts()
splitter = StratifiedShuffleSplit(n_splits=1, test_size=0.3, random_state=42)

train_idx, test_idx = next(splitter.split(X,y))
# Create the data sets
X_train = indicator_data.loc[train_idx, feature_column]
y_train = indicator_data.loc[train_idx, 'Class']

X_test = indicator_data.loc[test_idx, feature_column]
y_test = indicator_data.loc[test_idx, 'Class']

print(X_train.shape)
print(y_train.shape)
print(X_test.shape)
print(y_test.shape)
```

```
(1665, 17)
(1665,)
(714, 17)
(714,)
```

3. Model varification

Basically I would go over major classification models in this analysis and evaluate the scoring through the report and confusion matrix.

a. Linear regression

The first model that I am going to create is the linear regression model. In addition to a simple linear regression model, I would also try the models that implement L1 and L2 penalties.

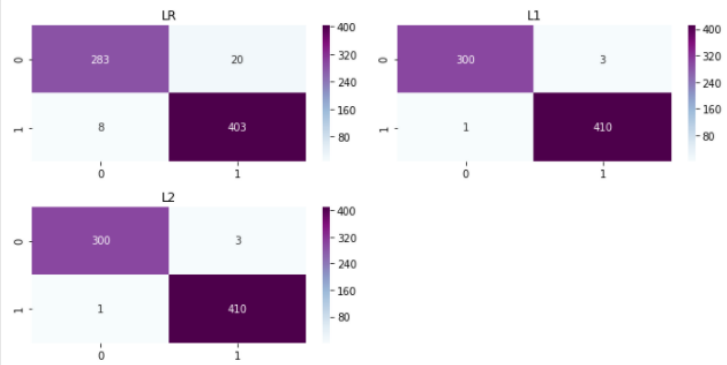
```
In [445]: # Standard Logistic regression
lr = LogisticRegression(solver='liblinear').fit(X_train, y_train)
lr_l1 = LogisticRegressionCV(Cs=10, cv=4, penalty='l1', solver='liblinear', n_jobs=-1, max_iter=1000).fit(X_train, y_train)
lr_l2 = LogisticRegressionCV(Cs=10, cv=4, penalty='l2', solver='liblinear', n_jobs=-1, max_iter=1000).fit(X_train, y_train)
```

```
In [515]: # Run simple Linear Regression, and with L1, L2 penalty
lr_list = [lr, lr_l1, lr_l2]
lr_type = ['LR', 'L1', 'L2']
lr_cm = {}
lr_cm_list = []
auc_list = []
for model, model_type in zip(lr_list, lr_type):
    y_hat = model.predict(X_test)
    y_hat_proba = model.predict_proba(X_test)
    cm = confusion_matrix(y_test, y_hat)
    lr_cm[model_type] = cm
    result = classification_report(y_test, y_hat
                                   , output_dict=True
                                  )
    lr_cm_list.append(result)
    arr = model.predict_proba(X_test)
    auc_score = round(roc_auc_score(y_test, arr[:,1]),4)
    auc_list.append(auc_score)
```

```
In [516]: report_columns = ['precision', 'recall', 'f1-score', 'support']
index = ['0', '1']
for cm, model_type, auc in zip(lr_cm_list, lr_type, auc_list):
    df_0 = pd.DataFrame(data=cm['0'], index=['0'])
    df_1 = pd.DataFrame(data=cm['1'], index=['1'])
    df = pd.concat([df_0, df_1], axis=0)
    print('The classification report for {}: '.format(model_type))
    print(df)
    print('The accuracy for {} is {}'.format(model_type, cm['accuracy']))
    print('The roc_auc_score for {} is {}'.format(model_type, auc))
    print('\n')
```

```
The classification report for LR:
   precision    recall  f1-score  support
0   0.972509  0.933993  0.952862      303
1   0.952719  0.980535  0.966427      411
The accuracy for LR is 0.9607843137254902
The roc_auc_score for LR is 0.9961


The classification report for L1:
   precision    recall  f1-score  support
0   0.996678  0.990099  0.993377      303
1   0.992736  0.997567  0.995146      411
The accuracy for L1 is 0.9943977591036415
The roc_auc_score for L1 is 0.9998


The classification report for L2:
   precision    recall  f1-score  support
0   0.996678  0.990099  0.993377      303
1   0.992736  0.997567  0.995146      411
The accuracy for L2 is 0.9943977591036415
The roc_auc_score for L2 is 0.9998
```

In [517]:
```
fig, axList = plt.subplots(nrows=2, ncols=2)
axList = axList.flatten()
fig.set_size_inches(10, 5)
axList[-1].axis('off')
for ax, model_type in zip(axList[:-1], lr_type):
    sns.heatmap(lr_cm[model_type], ax=ax, annot=True, fmt='d',cmap='BuPu');
    ax.set(title=model_type);
plt.tight_layout()
```



The linear regression approach provides beautiful prediction results. Applying penalties (L1 or L2) could slightly improve the scoring and accuracy. In general, the prediction generates good TP and TN so that the major scores (precision, recall, F1 score and accuracy) are very high.

For simple linear regression:

- TP: 283
- TN: 403
- FP: 20
- FN: 8

	precision	recall	f1_score	support
0 (Not worth buying)	0.972509	0.933993	0.952862	303
1 (Worth buying)	0.952719	0.980535	0.966427	411

The roc_auc_score of simple linear regression is 0.9961.

For linear regression with L1 penalty:

- TP: 300
- TN: 410
- FP: 3
- FN: 1

	precision	recall	f1_score	support
0 (Not worth buying)	0.996678	0.990099	0.993377	303
1 (Worth buying)	0.992736	0.997567	0.995146	411

The roc_auc_score of simple linear regression is 0.9998.

For linear regression with L2 penalty (same to L1):

- TP: 300
- TN: 410
- FP: 3
- FN: 1

	precision	recall	f1_score	support
0 (Not worth buying)	0.996678	0.990099	0.993377	303
1 (Worth buying)	0.992736	0.997567	0.995146	411

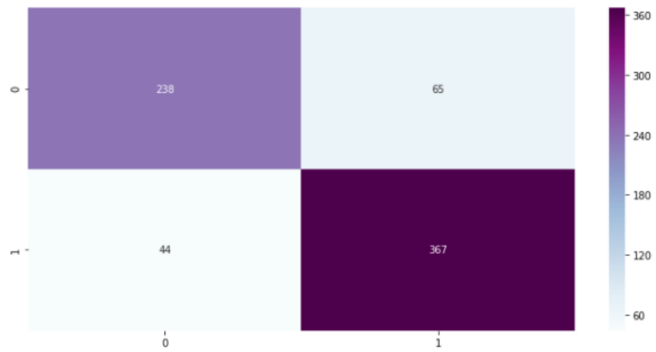
The roc_auc_score of simple linear regression is 0.9998.

b. K - Nearest Neighbors (KNN)

The next step is to predict with KNN approach. To find out the optimal hyperparameter, I would use the grid search method to run the KNN classification.

```
In [520]: # Test KNN with GridSearchCV
          param_dict = {'n_neighbors':range(1,10),
                        'weights':['uniform', 'distance'],
                        'n_jobs':[-1],
                        'p':[1,2]
                       }
          gcv = GridSearchCV(KNeighborsClassifier(), param_dict)
          gcv.fit(X_train, y_train)
          y_hat_gcv = gcv.predict(X_test)
          # print(gcv.get_params())
          print(gcv.best_estimator_)
          print(gcv.best_params_)
          print(classification_report(y_test, y_hat_gcv))
          print('Accuracy score: ', round(accuracy_score(y_test, y_hat_gcv), 4))
          print('F1 Score: ', round(f1_score(y_test, y_hat_gcv), 4))

          plotCM('KNN', y_test, y_hat_gcv)
```

```
KNeighborsClassifier(n_jobs=-1, n_neighbors=8, p=1)
{'n_jobs': -1, 'n_neighbors': 8, 'p': 1, 'weights': 'uniform'}
              precision    recall  f1-score   support

           0       0.84      0.79      0.81       303
           1       0.85      0.89      0.87       411

    accuracy                           0.85       714
   macro avg       0.85      0.84      0.84       714
weighted avg       0.85      0.85      0.85       714

Accuracy score:  0.8473
F1 Score:  0.8707
```



Still, the KNN result is good enough but is not as good as linear regression. It provides over 80% accuracy and F1 score, however it performs worse in predicting stocks not worth buying than those worth buying.

- TP: 238
- TN: 367
- FP: 65
- FN: 44

	precision	recall	f1_score	support
0 (Not worth buying)	0.84	0.79	0.81	303
1 (Worth buying)	0.85	0.89	0.87	411

The roc_auc_score of KNN is 0.9180.

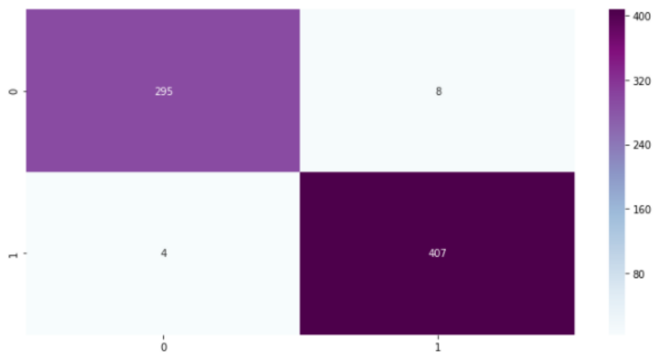
```
In [469]: arr = gcv.predict_proba(X_test)
          round(roc_auc_score(y_test, arr[:,1]),4)

Out[469]: 0.918
```

c. Support Vector Machine (SVM)

Then following the SVM approach. To find out the optimal hyperparameter, I would use the grid search method to run the SVM classification.

```
In [471]: param_dict = {'kernel':['poly', 'rbf', 'sigmoid'],
                        'degree':range(1,5),
                        'gamma':[.5, 1, 2, 10],
                        'probability':[True]
                       }
          gcv_svc = GridSearchCV(SVC(), param_dict)
          gcv_svc.fit(X_train, y_train)
          y_hat_gcv_svc = gcv_svc.predict(X_test)
          print(gcv_svc.best_estimator_)
          print(gcv_svc.best_params_)
          # print(gcv_svc.get_params())
          print(classification_report(y_test, y_hat_gcv_svc))
          print('Accuracy score: ', round(accuracy_score(y_test, y_hat_gcv_svc), 4))
          print('F1 Score: ', round(f1_score(y_test, y_hat_gcv_svc), 4))

          plotCM('GCV SVC', y_test, y_hat_gcv_svc)
```

```
SVC(degree=1, gamma=10, kernel='poly', probability=True)
{'degree': 1, 'gamma': 10, 'kernel': 'poly', 'probability': True}
              precision    recall  f1-score   support

           0       0.99      0.97      0.98       303
           1       0.98      0.99      0.99       411

    accuracy                           0.98       714
   macro avg       0.98      0.98      0.98       714
weighted avg       0.98      0.98      0.98       714

Accuracy score:  0.9832
F1 Score:  0.9855
```



The SVM result is almost as good as linear regression. With the optimal hyperparameter It provides over 98% accuracy (0.9832) and F1 score (0.9855) on the test set.

- TP: 295
- TN: 407
- FP: 8
- FN: 4

	precision	recall	f1_score	support
0 (Not worth buying)	0.99	0.97	0.98	303
1 (Worth buying)	0.98	0.99	0.99	411

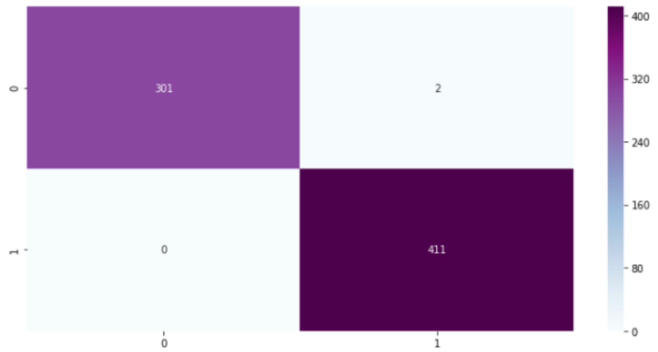
The roc_auc_score of SVM is 0.9991.

```
In [472]: arr = gcv_svc.predict_proba(X_test)
          round(roc_auc_score(y_test, arr[:,1]),4)

Out[472]: 0.9991
```

d. Decision Tree

Then following the Decision Tree approach. To find out the optimal hyperparameter, I would use the grid search method to run the Decision Tree classification.

```
In [456]: # Decision Tree
          # Try to get the tree structure
          tree = DecisionTreeClassifier(random_state=42)
          tree = tree.fit(X_train, y_train)
          tree_depth = tree.tree_.max_depth
          tree_features_length = len(tree.feature_importances_)
          # Use grid search for the best hyperparameter
          param_grid = {'max_depth':range(1, tree_depth+1, 2),
                        'max_features': range(1, tree_features_length+1)}
          gcv_dt = GridSearchCV(DecisionTreeClassifier(random_state=42),
                                param_grid=param_grid,
                                n_jobs=-1)
          gcv_dt.fit(X_train, y_train)
          y_hat_gcv_dt = gcv_dt.predict(X_test)
          print(gcv_dt.best_estimator_)
          print(gcv_dt.best_params_)
          print(classification_report(y_test, y_hat_gcv_dt))
          print('Accuracy score: ', round(accuracy_score(y_test, y_hat_gcv_dt), 4))
          print('F1 Score: ', round(f1_score(y_test, y_hat_gcv_dt), 4))

          plotCM('GCV DT', y_test, y_hat_gcv_dt)

          DecisionTreeClassifier(max_depth=1, max_features=10, random_state=42)
          {'max_depth': 1, 'max_features': 10}
                        precision    recall  f1-score   support

                     0       1.00      0.99      1.00       303
                     1       1.00      1.00      1.00       411

              accuracy                           1.00       714
             macro avg       1.00      1.00      1.00       714
          weighted avg       1.00      1.00      1.00       714

          Accuracy score:  0.9972
          F1 Score:  0.9976
```



The Decision Tree result is so far the best. With the optimal hyperparameter It provides nearly 100% accuracy (0.9972) and F1 score (0.9976) on the test set.

- TP: 301
- TN: 411
- FP: 2
- FN: 0

	precision	recall	f1_score	support
0 (Not worth buying)	1	0.99	1	303
1 (Worth buying)	1	1	1	411

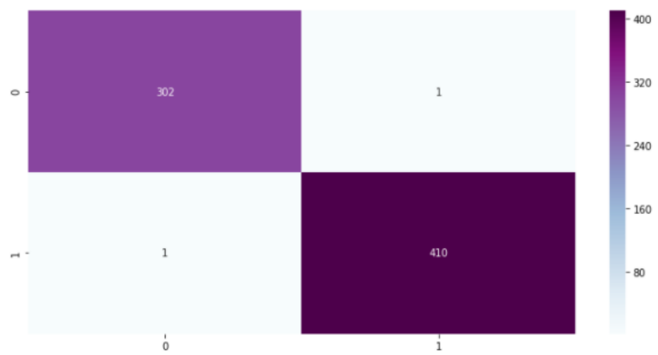
The roc_auc_score of Decision Tree is 0.9967.

```
In [473]: arr = gcv_dt.predict_proba(X_test)
          round(roc_auc_score(y_test, arr[:,1]),4)

Out[473]: 0.9967
```

e. Random Forest

The next classification approach is Random Forest to ensemble multiple trees through bagging to predict. Again I would use the grid search method to find out the best hyperparameter combination.

```
In [458]: # rfc = RandomForestClassifier(random_state=42)
          param_grid = {'n_estimators':[15, 20, 30, 40, 50, 100, 150, 200, 300, 400],
          #               'oob_score': [True],
                        'max_features': ['sqrt', 'log2'],
                        'warm_start':[True]
                       }
          gcv_rfc = GridSearchCV(RandomForestClassifier(random_state=42),
                                 param_grid=param_grid,
                                 n_jobs=-1)
          gcv_rfc.fit(X_train, y_train)
          y_hat_gcv_rfc = gcv_rfc.predict(X_test)

          print(gcv_rfc.best_estimator_)
          print(gcv_rfc.best_params_)

          # print(gcv_rfc.get_params())
          print(classification_report(y_test, y_hat_gcv_rfc))
          print('Accuracy score: ', round(accuracy_score(y_test, y_hat_gcv_rfc), 4))
          print('F1 Score: ', round(f1_score(y_test, y_hat_gcv_rfc), 4))

          plotCM('GCV Random Forest', y_test, y_hat_gcv_rfc)

          RandomForestClassifier(max_features='sqrt', n_estimators=15, random_state=42,
                                 warm_start=True)
          {'max_features': 'sqrt', 'n_estimators': 15, 'warm_start': True}
                        precision    recall  f1-score   support

                     0       1.00      1.00      1.00       303
                     1       1.00      1.00      1.00       411

              accuracy                           1.00       714
             macro avg       1.00      1.00      1.00       714
          weighted avg       1.00      1.00      1.00       714

          Accuracy score:  0.9972
          F1 Score:  0.9976
```



The Random Forest result is very close to the Decision Tree. It also generates nearly 100% accuracy (0.9972) and F1 score (0.9976) on the test set through 15 estimators.

- TP: 302
- TN: 410
- FP: 1
- FN: 1

	precision	recall	f1_score	support
0 (Not worth buying)	1	1	1	303
1 (Worth buying)	1	1	1	411

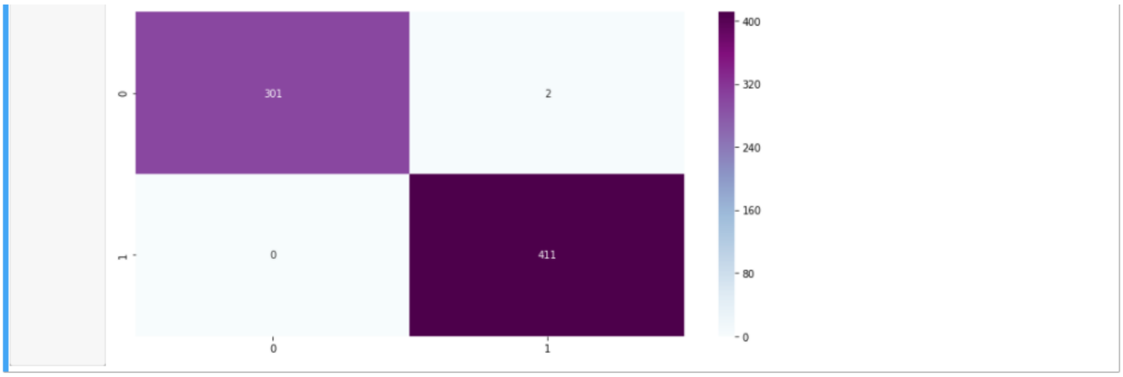
The roc_auc_score of Random Forest is 1.0.

```
In [474]: arr = gcv_rfc.predict_proba(X_test)
          round(roc_auc_score(y_test, arr[:,1]),4)

Out[474]: 1.0
```

f. Boosting

The final classification approach to be reviewed is ensembled with boosting. Again I would use the grid search method with GradientBoosting to find out the best hyperparameter combination.

```
In [459]: # The parameters to be fit
          tree_list = [15, 25, 50, 100, 200, 400]
          param_grid = {'n_estimators': tree_list,
                        'learning_rate': [0.1, 0.01, 0.001, 0.0001],
                        'subsample': [1.0, 0.5],
                        'max_features': [1, 2, 3, 4]}

          # The grid search object
          gcv_boosting = GridSearchCV(GradientBoostingClassifier(random_state=42),
                                      param_grid=param_grid,
                                      scoring='accuracy',
                                      n_jobs=-1)

          gcv_boosting.fit(X_train, y_train)
          y_hat_gcv_boosting = gcv_boosting.predict(X_test)

          print(gcv_boosting.best_estimator_)
          print(gcv_boosting.best_params_)

          print(classification_report(y_test, y_hat_gcv_boosting))
          print('Accuracy score: ', round(accuracy_score(y_test, y_hat_gcv_boosting), 4))
          print('F1 Score: ', round(f1_score(y_test, y_hat_gcv_boosting), 4))

          plotCM('GCV Boosting', y_test, y_hat_gcv_boosting)

          GradientBoostingClassifier(max_features=1, random_state=42)
          {'learning_rate': 0.1, 'max_features': 1, 'n_estimators': 100, 'subsample': 1.0}
                        precision    recall  f1-score   support

                     0       1.00      0.99      1.00       303
                     1       1.00      1.00      1.00       411

              accuracy                           1.00       714
             macro avg       1.00      1.00      1.00       714
          weighted avg       1.00      1.00      1.00       714

          Accuracy score:  0.9972
          F1 Score:  0.9976
```



The Boosting result is very close to the Decision Tree and Random Forest. It generates nearly 100% accuracy (0.9972) and F1 score (0.9976) on the test set through 100 estimators.

- TP: 301
- TN: 411
- FP: 2
- FN: 0

	precision	recall	f1_score	support
0 (Not worth buying)	1	0.99	1	303
1 (Worth buying)	1	1	1	411

The roc_auc_score of Boosting is 0.9978.

```
In [475]: arr = gcv_boosting.predict_proba(X_test)
          round(roc_auc_score(y_test, arr[:,1]),4)

Out[475]: 0.9978
```

Key Findings and Insights, which walks your reader through the main drivers of your model and insights from your data derived from your classification model.

First of all, let's summarize the statistics among all classifiers that I have gone through:

	TP	TN	FP	FN	Accuracy	ROC
LR	283	403	20	8	0.9608	0.9961
L1	300	410	3	1	0.9944	0.9998
L2	300	410	3	1	0.9944	0.9998
KNN	238	367	65	44	0.8473	0.9180
SVM	295	407	8	4	0.9832	0.9991
Decision Tree	301	411	2	0	0.9972	0.9967
Random Forest	302	410	1	1	0.9972	1
Boosting	301	411	2	0	0.9972	0.9978

Here are some insights that lead to model selection:

- In general, all 8 classifiers that have been tested provide good prediction results.
- Among linear regression, modeling with penalty can generate better predictions than modeling without penalty.
- “Tree Family” and Boosting modeling provide the best outcomes. Random Forest, in particular, can even reach the full ROC score.
- KNN performs fairly but not as well as others.
- Interestingly, even SVM generates a slightly less precise outcome but it earns higher ROC score than Decision Tree and Boosting.

So according to the result matrix and insights, I would recommend Random Forest as the first choice, following are Boosting and Decision Tree. The recommendation is made not only on the accuracy and ROC, but also on the consideration of maximizing the TP/TN (correctly identifies stocks not worth buying and worth buying) and minimizing FP/FN.

Suggestions for next steps

So far the Random Forest is the best choice based on the stock performance data from 2014 to 2018.

Even though the prediction on the test set is extremely outstanding, there are still some tasks that can be conducted to see if the current model is too good to believe:

1. The 17 independent features out of 225 are mainly correspondent to well - known financial indicators, and maybe not the most correlative with the

dependent variable. It is good to see whether the outcome will be different if the independent variables are selected on the most correlative basis.

2. During the data exploratory phase, records with NA values and outlier values are all dropped, thus the data set volume decreases from 22077 to 2379. It is a nearly 90% volume decrease and may be the risk of overfitting. Therefore it could be further discussed if it is necessary to examine the outliers to figure out the logic to generate substitute values to restrict the volume decrease.
3. To extract 2019 and 2020 stock data from other public resources as the new test sets and run the classifiers to see if the similar outcomes can be found.