```
In [99]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import scikitplot as skplt
from sklearn import neighbors, datasets, tree, linear_model, metrics,svm
from sklearn.model_selection import cross_val_score, train_test_split,KFold
import itertools
from itertools import permutations
from sklearn.metrics import recall_score
from sklearn.model_selection import GridSearchCV
import warnings
warnings.filterwarnings('ignore')
```

Question 4

1. Import Data

```
In [26]: #1. Import Data
data =  pd.read_csv('wdbc.data', sep=",",header=None)
#2. Fix the column names
features=['radius','texture','perimeter','area','smoothness','compactness','concavity','concave points'
,'symmetry','fractal dimension']
atr=['mean','standard error','largest']
column1 = ['ID','Diagnosis']
column2 = list(itertools.product(features,atr))
for i in range(len(column2)):
    column2[i]='-'.join(column2[i])
column1.extend(column2)
data.columns=column1
data=data.iloc[:,1:]
data.head(3)
```

Out[26]:

	Diagnosis	radius-mean	radius-standard error	radius-largest	texture-mean	texture-standard error	texture-largest	perimeter-mean	perimeter-standard error	perimeter-largest	...	concavity-largest	concave points-mean	concave points-standard error
0	M	17.99	10.38	122.8	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	25.38	17.33	184.6
1	M	20.57	17.77	132.9	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	24.99	23.41	158.8
2	M	19.69	21.25	130.0	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	23.57	25.53	152.5

3 rows × 31 columns

```
In [20]: #data.isna().sum()
data.describe()
```

Out[20]:

	radius-mean	radius-standard error	radius-largest	texture-mean	texture-standard error	texture-largest	perimeter-mean	perimeter-standard error	perimeter-largest	area-mean	...	co
count	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	...	569
mean	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	0.048919	0.181162	0.062798	...	16
std	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	0.038803	0.027414	0.007060	...	4
min	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	0.000000	0.106000	0.049960	...	7
25%	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	0.020310	0.161900	0.057700	...	13
50%	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	0.033500	0.179200	0.061540	...	14
75%	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	0.074000	0.195700	0.066120	...	18
max	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	0.201200	0.304000	0.097440	...	36

8 rows × 30 columns

2. Performance Metric

Given that the consequence of not identifying maligant case is also serious, I want to also take **Recall** into account and therefore choose the **F-measure(maligant)** as our performance matric.

3. Data Normalization

```
In [27]: from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
data.iloc[:,1:]=scaler.fit_transform(data.iloc[:,1:])
data.head(10)
```

Out[27]:

	Diagnosis	radius-mean	radius-standard error	radius-largest	texture-mean	texture-standard error	texture-largest	perimeter-mean	perimeter-standard error	perimeter-largest	...	concavity-largest	concave points-mean	co p sta
0	M	0.521037	0.022658	0.545989	0.363733	0.593753	0.792037	0.703140	0.731113	0.686364	...	0.620776	0.141525	0.6
1	M	0.643144	0.272574	0.615783	0.501591	0.289880	0.181768	0.203608	0.348757	0.379798	...	0.606901	0.303571	0.5
2	M	0.601496	0.390260	0.595743	0.449417	0.514309	0.431017	0.462512	0.635686	0.509596	...	0.556386	0.360075	0.5
3	M	0.210090	0.360839	0.233501	0.102906	0.811321	0.811361	0.565604	0.522863	0.776263	...	0.248310	0.385928	0.2
4	M	0.629893	0.156578	0.630986	0.489290	0.430351	0.347893	0.463918	0.518390	0.378283	...	0.519744	0.123934	0.5
5	M	0.258839	0.202570	0.267984	0.141506	0.678613	0.461996	0.369728	0.402038	0.518687	...	0.268232	0.312633	0.2
6	M	0.533343	0.347311	0.523875	0.380276	0.379164	0.274891	0.264058	0.367793	0.370707	...	0.531839	0.416844	0.5
7	M	0.318472	0.376057	0.320710	0.184263	0.598267	0.445126	0.219447	0.297465	0.573737	...	0.324795	0.429638	0.2
8	M	0.284869	0.409537	0.302052	0.159618	0.674099	0.533157	0.435567	0.464861	0.651515	...	0.268943	0.498667	0.2
9	M	0.259312	0.484613	0.277659	0.140997	0.595558	0.675480	0.532568	0.424602	0.489899	...	0.254714	0.763859	0.2

10 rows × 31 columns

4. Nested CV

Via NestedCV, I want to first find out which method has the best performance. After finding the best method, I will conduct grid search to find the best hyper-parameters for the model.

```
In [69]: # Split the target variable
y_data=np.where(data['Diagnosis']=='M',1,0)
x_data=data.iloc[:,1:]

# Create the Classifier
t=tree.DecisionTreeClassifier(random_state=9)
knn=neighbors.KNeighborsClassifier()
log=linear_model.LogisticRegression(penalty='elasticnet',l1_ratio=0.5,solver='saga')
s=svm.SVC(kernel='rbf',random_state=9)

# Create the grid search
tree_grid={'criterion':['gini','entropy'],
           'max_depth':list(range(30))}
knn_grid={'weights':['uniform','distance'],
          'n_neighbors':list(range(5,31))}
log_grid={'C':[0.1,1,5,10,50,100,500,1000]}
svm_grid={'C':[0.1,1,5,10,50,100],
          'gamma':[1,5,10,15,20,25,30,50,100]}

# Create the CV
inner_cv = KFold(n_splits=5, shuffle=True, random_state=9)
outer_cv = KFold(n_splits=5, shuffle=True, random_state=9)

#Nested CV for SVM
clf = GridSearchCV(estimator=s, param_grid=svm_grid, cv=inner_cv,scoring='f1')
nested_score = cross_val_score(clf, X=x_data, y=y_data, cv=outer_cv,scoring='f1')
svm_result=nested_score.mean()

#Nested CV for Decision Tree
clf = GridSearchCV(estimator=t, param_grid=tree_grid, cv=inner_cv,scoring='f1')
nested_score = cross_val_score(clf, X=x_data, y=y_data, cv=outer_cv,scoring='f1')
tree_result=nested_score.mean()

#Nested CV for Logistic Regression
clf = GridSearchCV(estimator=log, param_grid=log_grid, cv=inner_cv,scoring='f1')
nested_score = cross_val_score(clf, X=x_data, y=y_data, cv=outer_cv,scoring='f1')
log_result=nested_score.mean()

#Nested CV for KNN
clf = GridSearchCV(estimator=knn, param_grid=knn_grid, cv=inner_cv,scoring='f1')
nested_score = cross_val_score(clf, X=x_data, y=y_data, cv=outer_cv,scoring='f1')
knn_result=nested_score.mean()
```

```
In [80]: print('Average Performance of SVM Classifier: {}%'.format(round(svm_result*100,2)))
print('Average Performance of Decision Tree Classifier: {}%'.format(round(tree_result*100,2)))
print('Average Performance of Logistic Regression Classifier: {}%'.format(round(log_result*100,2)))
print('Average Performance of KNN Classifier: {}%'.format(round(knn_result*100,2)))
print("According to NestedCV's result, the SVM Classifier has the best performance.")
```

```
Average Performance of SVM Classifier: 97.52%
Average Performance of Decision Tree Classifier: 92.16%
Average Performance of Logistic Regression Classifier: 96.32%
Average Performance of KNN Classifier: 95.77%
According to NestedCV's result, the SVM Classifier has the best performance.
```

5. Grid Search CV on SVM Model to find the best hyper-parameters

```
In [88]: y_data=np.where(data['Diagnosis']=='M',1,0)
x_data=data.iloc[:,1:]
x_train,x_test,y_train,y_test = train_test_split(x_data,y_data,test_size=0.2,random_state=9)

svm_grid={'C':[0.1,1,5,10,50,100],
          'gamma':[1,5,10,15,20,25,30,50,100]}
s=svm.SVC(kernel='rbf',random_state=9,probability=True)
model = GridSearchCV(estimator=s, param_grid=svm_grid, cv=5,scoring='f1')
model.fit(x_train,y_train)
print ("With CV grid search, I found the best hyperparameter is C={} and gamma={}.".format(model.best_p
arams_['C'],model.best_params_['gamma']))
print("Prediction F1-Positive Score on Test Data: {}%".format(round(metrics.f1_score(y_test, model.pred
ict(x_test))*100,2)))
```

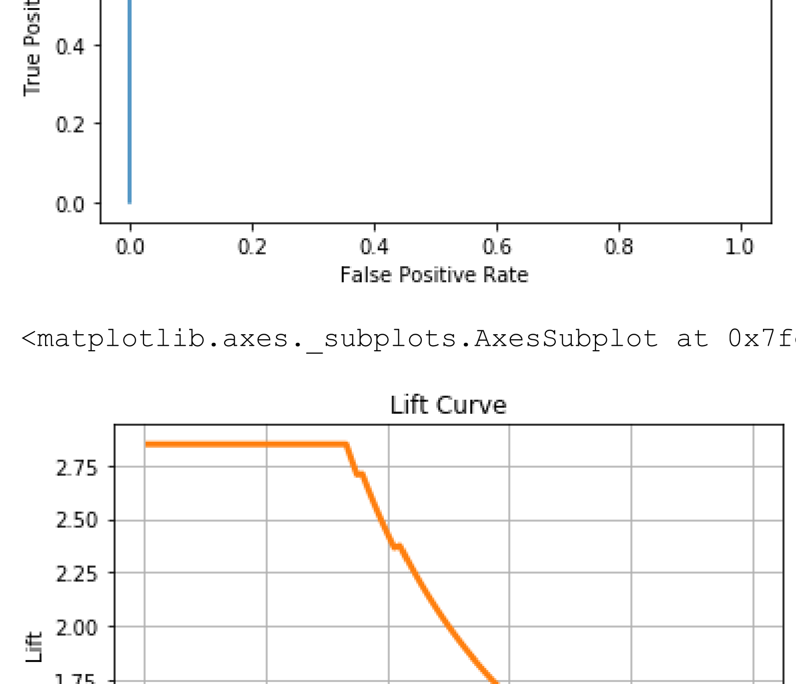
```
With CV grid search, I found the best hyperparameter is C=10 and gamma=1.
Prediction F1-Positive Score on Test Data: 97.44%
```

6. Plot the ROC Curve & Lift Curve

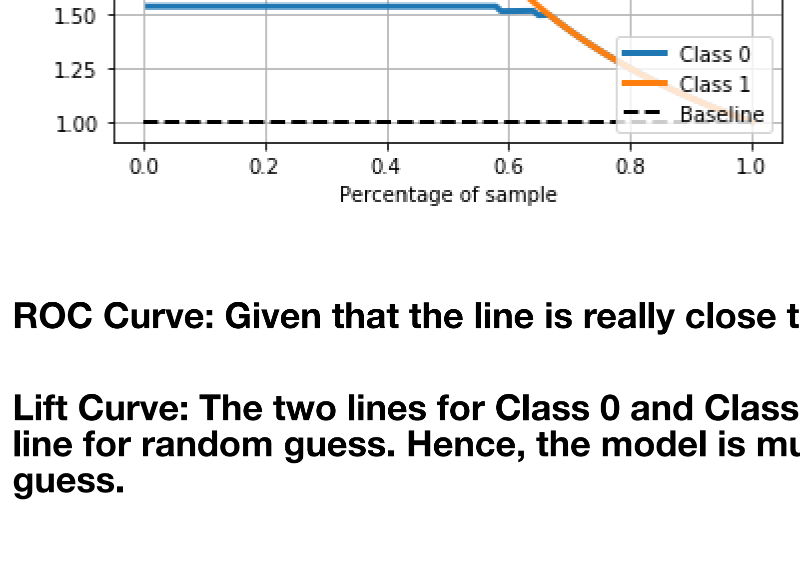
```
In [112]: prob=model.predict_proba(x_test)[:,1]

#create ROC curve
fpr, tpr, _ = metrics.roc_curve(y_test, prob)
plt.plot(fpr,tpr)
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.title('ROC Curve')
plt.show()

#create Lift curve
skplt.metrics.plot_lift_curve(y_test,model.predict_proba(x_test))
```




```
Out[112]: <matplotlib.axes._subplots.AxesSubplot at 0x7fe73744ec90>
```



ROC Curve: Given that the line is really close to the top left corner, the model is really accurate.

Lift Curve: The two lines for Class 0 and Class 1 are all much higher than the baseline which is the line for random guess. Hence, the model is much better at predicting both class 1 and 0 than random guess.