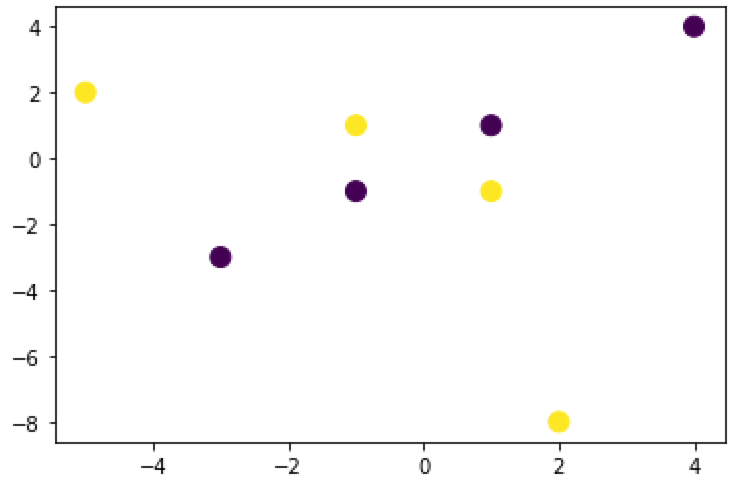
In [1]:
```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from IPython.display import Markdown as md
from IPython.core.display import Image, display
import sklearn
from sklearn import linear_model
from collections import defaultdict
import time
import warnings

warnings.filterwarnings("ignore")
sed = 2023
```

Problem 1

1------------

In [2]:
```
df1 = pd.DataFrame({
    'label':[1,1,1,1,2,2,2,2],
    'x_1':[-1,1,-3,4,-1,1,-5,2],
    'x_2':[-1,1,-3,4,1,-1,2,-8]
})
```

In [3]:
```
plt.scatter(df1["x_1"], df1["x_2"],c = df1['label'], s = 100)
plt.show()
```

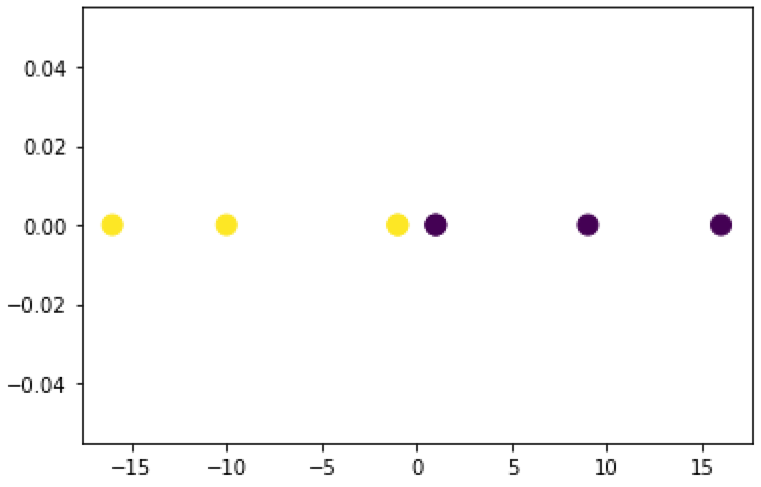


- (1), According to the scatter plot, the dataset is not linear separable.
- (2), Only X_1 and X_2 has k and Ws that $\forall x \in X_1, \sum_{i=1}^{n} w_i > k$ and $\forall x \in X_2, \sum_{i=1}^{n} w_i < k$ are linear separatable. How ever in our dataset we don't have such solution.

2------------

- Let z = $x_1 * x_2$

In [4]:
```
df1['z'] = df1['x_1']*df1['x_2']
plt.scatter(df1["z"], y = np.zeros_like(df1["z"]),c = df1['label'], s = 100)
plt.show()
```

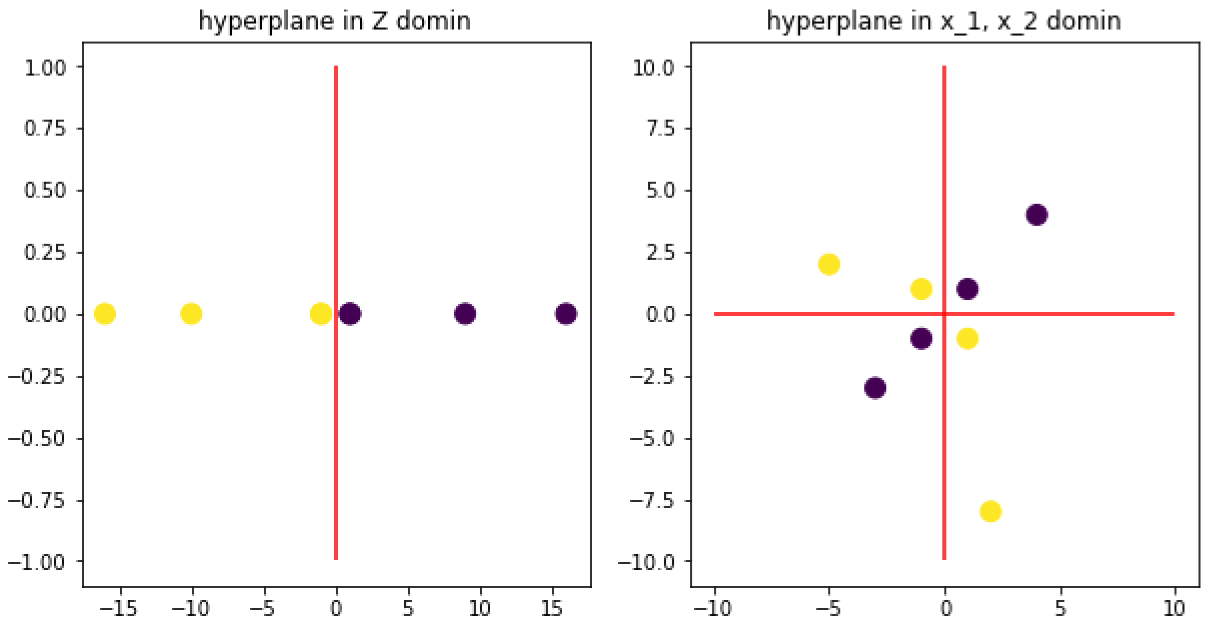


- According to the plot $Class_1$ and $Class_2$ are now linearly separable in Z domin.

3------------

In [5]:
```
fig, ax = plt.subplots(1,2,figsize = (10,5))
ax[0].scatter(x = df1["z"], y = np.zeros_like(df1["z"]),c = df1['label'], s = 100)
ax[0].vlines(0,-1,1, color = 'r')
ax[0].set_title('hyperplane in Z domin')

ax[1].scatter(x = df1["x_1"], y = df1["x_2"],c = df1['label'], s = 100)
ax[1].vlines(0,-10,10, color = 'r')
ax[1].hlines(0,-10,10, color = 'r')
ax[1].set_title('hyperplane in x_1, x_2 domin')
```

Out[5]:
```
Text(0.5, 1.0, 'hyperplane in x_1, x_2 domin')
```



4------------

- In most real problems, Data sets are not linearly separable. Only using non-linear transformation can we accurately classify objects.
- By introducing non-linearity, we can usually gain a more flexible model and thus decrease bias.

problem 2

1------------

In [6]:
```python
display(Image('problem2_1.png'))
```

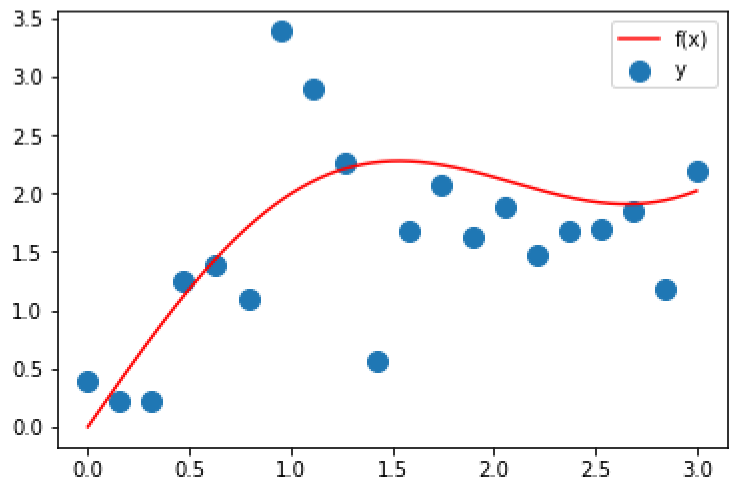
$$MSE = \frac{1}{t}\sum_{i=1}^{t}(\hat{y}_i - y_i)^2 = \frac{1}{t}\sum_{i=1}^{t}(g(\bar{x}_i, D) - f(\bar{x}_i) - \varepsilon_i)^2$$

$$\begin{aligned}
E[MSE] &= \frac{1}{t}\sum_{i=1}^{t}E[(g(\bar{x}_i, D) - f(\bar{x}_i) - \varepsilon_i)^2] \\
&= \frac{1}{t}\sum_{i=1}^{t}E[(g(\bar{x}_i, D) - f(\bar{x}_i))]^2 + \frac{\sum_{i=1}^{t}E[\varepsilon_i^2]}{t} + \frac{2}{t}\sum_{i=1}^{n}\underbrace{E[(g(\bar{x}_i, D) - f(\bar{x}_i))]}_{=0}\cdot E(\varepsilon) \\
&= \frac{1}{t}\sum_{i=1}^{t}E[(g(\bar{x}_i, D) - f(\bar{x}_i))]^2 + \frac{\sum_{i=1}^{t}E[\varepsilon_i^2]}{t} \\
&= \frac{1}{t}\sum_{i=1}^{t}E[\{(f(\bar{x}_i) - E[g(\bar{x}_i, D)]) + (E[g(\bar{x}_i, D)] - g(\bar{x}_i, D))\}^2] + \frac{\sum_{i=1}^{t}E(\varepsilon_i^2)}{t} \\
&= \underbrace{\frac{1}{t}\sum_{i=1}^{t}\{f(\bar{x}_i) - E[g(\bar{x}_i, D)]\}^2}_{Bias^2} + \underbrace{\frac{1}{t}\sum_{i=1}^{t}E[\{g(\bar{x}_i, D) - E[g(\bar{x}_i, D)]\}^2]}_{Variance} + \underbrace{\frac{\sum_{i=1}^{t}E[\varepsilon_i^2]}{t}}_{Noise}
\end{aligned}$$

2------------

In [7]:
```python
# generate samples
std = np.sqrt(0.3)
x = np.linspace(0,3, 20)
np.random.seed(seed=sed)
noise = np.random.normal(0,std,20)
y = x + np.sin(1.5*x) + noise
display_x = np.linspace(0,3, 2000)
plt.plot(display_x, display_x+np.sin(1.5*display_x), c = 'r')
plt.scatter(x,y,s = 100)
plt.legend(['f(x)','y'])
```

Out[7]:
```
<matplotlib.legend.Legend at 0x7ff21233e670>
```



3-----------

In [8]:

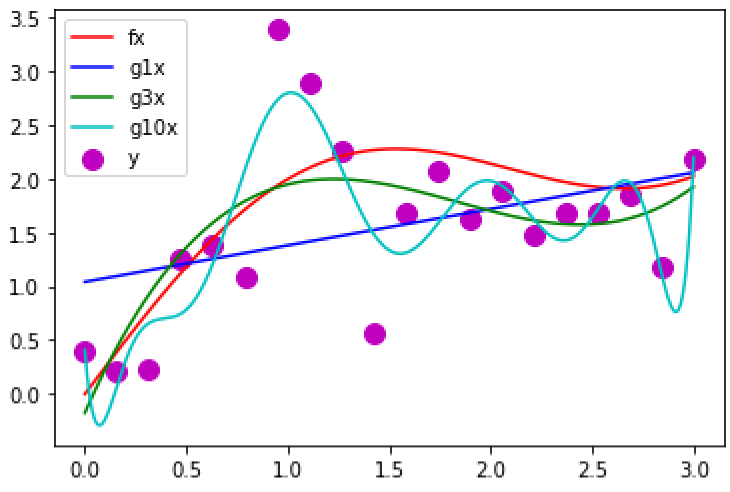
```
# generate data
df2 = pd.DataFrame({
    'x_1':x,
    'x_2':x**2,
    'x_3':x**3,
    'x_4':x**4,
    'x_5':x**5,
    'x_6':x**6,
    'x_7':x**7,
    'x_8':x**8,
    'x_9':x**9,
    'x_10':x**10,
    'y': y
})
display_xs = []
for i in range(1,11):
    display_xs.append(display_x**i)
display_xs = np.array(display_xs).transpose()

g_1 = sklearn.linear_model.LinearRegression()
g_1.fit(X = df2.iloc[:,[0]], y = df2['y'])
g1x = g_1.intercept_ + np.dot(display_xs[:,0:1], g_1.coef_)

g_3 = sklearn.linear_model.LinearRegression()
g_3.fit(X = df2.iloc[:,0:3], y = df2['y'])
g3x = g_3.intercept_ + np.dot(display_xs[:,0:3],g_3.coef_)

g_10 = sklearn.linear_model.LinearRegression()
g_10.fit(X = df2.iloc[:,0:10], y = df2['y'])
g10x = g_10.intercept_ + np.dot(display_xs[:,0:10],g_10.coef_);
```

In [9]:

```
plt.scatter(x,y,s = 100,c = 'm', label = 'y')
plt.plot(display_x, display_x+np.sin(1.5*display_x), c = 'r', label = 'fx')
plt.plot(display_x, g1x, c = 'b', label = 'g1x')
plt.plot(display_x, g3x, c = 'g', label = 'g3x')
plt.plot(display_x, g10x, c = 'c', label = 'g10x')
plt.legend()
```

Out[9]:

```
<matplotlib.legend.Legend at 0x7ff2123ad400>
```

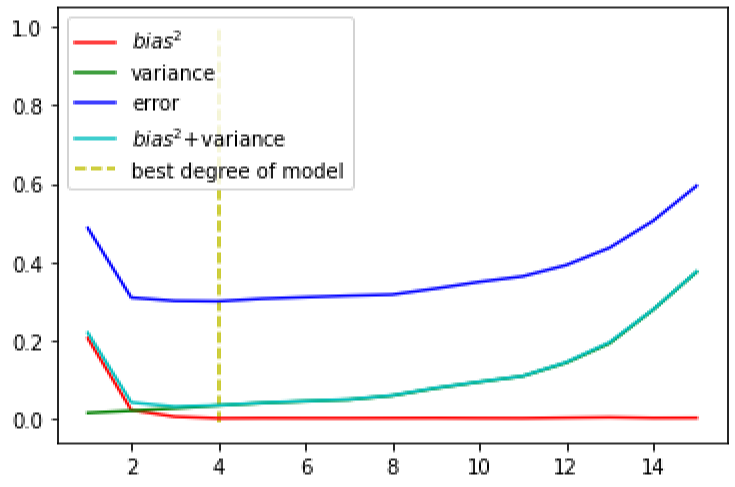


- According to the plot, $g_1(x)$ is underfitting, $g_{10}(x)$ is overfitting.

4-----------

In [10]:

```
x = np.linspace(0,3,50)
x = np.random.permutation(x)
x_train = x[:40]
x_test = x[40:]

pred_trains = defaultdict(list)
pred_tests = defaultdict(list)

train_errors = defaultdict(list)
test_errors = defaultdict(list)

for i in range(100):
    y_train = x_train + np.sin(1.5 * x_train) + np.random.normal(0,std,40)
    y_test = x_test + np.sin(1.5 * x_test) + np.random.normal(0,std,10)
    for degree in range(1,16):
        model = np.polyfit(x_train, y_train, degree)

        pred_train = np.polyval(model, x_train)
        pred_trains[degree].append(pred_train)

        pred_test = np.polyval(model, x_test)
        pred_tests[degree].append(pred_test)

        train_errors[degree].append(np.mean((pred_train - y_train)**2))
        test_errors[degree].append(np.mean((pred_test - y_test)**2))


def calculate_estimator_bias_squared(pred_test):
    pred_test = np.array(pred_test)
    average_model_prediction = pred_test.mean(0)
    fx = x_test + np.sin(1.5*x_test)
    return np.mean((average_model_prediction - fx) ** 2)


def calculate_estimator_variance(pred_test):
    pred_test = np.array(pred_test)
    average_model_prediction = pred_test.mean(0)
    return np.mean((pred_test - average_model_prediction) ** 2)

complexity_train_error = []
complexity_test_error = []
bias_squared = []
variance = []
for degree in range(1,16):
    complexity_train_error.append(np.mean(train_errors[degree]))
    complexity_test_error.append(np.mean(test_errors[degree]))
    bias_squared.append(calculate_estimator_bias_squared(pred_tests[degree]))
    variance.append(calculate_estimator_variance(pred_tests[degree]))

best_model_degree = np.argmin(complexity_test_error)+1

degrees = [i for i in range(1,16)]
plt.plot(degrees,bias_squared, c = 'r',label = '$bias^2$')
plt.plot(degrees,variance, c = 'g',label = 'variance')
plt.plot(degrees,complexity_test_error, c = 'b',label = 'error')
plt.plot(degrees,np.array(variance)+ np.array(bias_squared), c = 'c',label = '$bias^2$+va
plt.vlines(best_model_degree,-0.01,1,linestyles= '--', color = 'y', label = 'best degree
plt.legend()
plt.show()
```



In [11]:

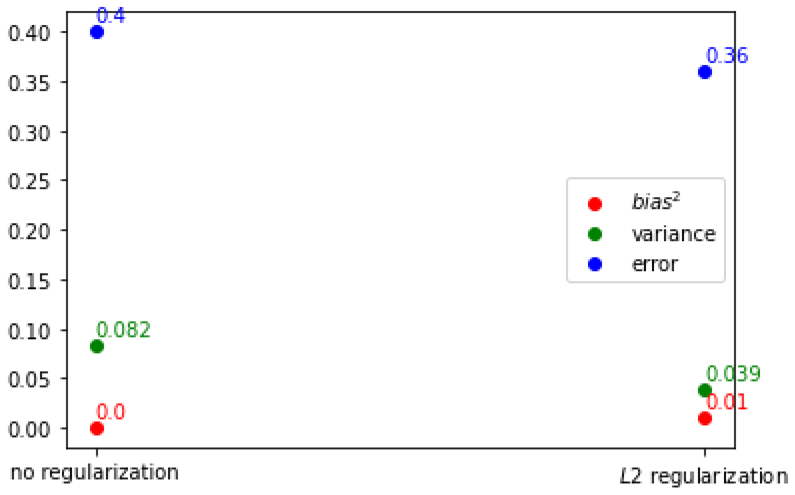
```
display(md('According to the plot the best degree to fit the model is: '+str(best_model_de
```

According to the plot the best degree to fit the model is: 4 !

5------------------

In [12]:

```
# return test squared bias, variance and error
from sklearn.linear_model import Ridge
x = np.linspace(0,3,50)
x = np.random.permutation(x)
x_train = x[:40]
x_test = x[40:]

x_train_10 = []
x_test_10 = []
for i in range(1,11):
    x_train_10.append(x_train**i)
    x_test_10.append(x_test**i)
x_train_10 = np.transpose(np.array(x_train_10))
x_test_10 = np.transpose(np.array(x_test_10))


pred_trains = defaultdict(list)
pred_tests = defaultdict(list)

train_errors = defaultdict(list)
test_errors = defaultdict(list)

for i in range(100):
    y_train = x_train + np.sin(1.5 * x_train) + np.random.normal(0,std,40)
    y_test = x_test + np.sin(1.5 * x_test) + np.random.normal(0,std,10)
    for type_model in range(1,3):
        if type_model == 1:
            model = np.polyfit(x_train, y_train, 10)

            pred_train = np.polyval(model, x_train)
            pred_trains[type_model].append(pred_train)

            pred_test = np.polyval(model, x_test)
            pred_tests[type_model].append(pred_test)

            train_errors[type_model].append(np.mean((pred_train - y_train)**2))
            test_errors[type_model].append(np.mean((pred_test - y_test)**2))

        else:
            model = Ridge()
            model.fit(x_train_10,y_train)
            pred_train = model.predict(x_train_10)
            pred_trains[type_model].append(pred_train)

            pred_test = model.predict(x_test_10)
            pred_tests[type_model].append(pred_test)

            train_errors[type_model].append(np.mean((pred_train - y_train)**2))
            test_errors[type_model].append(np.mean((pred_test - y_test)**2))


complexity_train_error = []
complexity_test_error = []
bias_squared = []
variance = []
for degree in range(1,3):
    complexity_train_error.append(np.mean(train_errors[degree]))
    complexity_test_error.append(np.mean(test_errors[degree]))
    bias_squared.append(calculate_estimator_bias_squared(pred_tests[degree]))
    variance.append(calculate_estimator_variance(pred_tests[degree]))

best_model_degree = np.argmin(complexity_test_error)+1

degrees = [i for i in range(1,3)]
labels = ['no regularization', '$L2$ regularization']
plt.scatter(degrees,bias_squared, c = 'r',label = '$bias^2$')
plt.scatter(degrees,variance, c = 'g',label = 'variance')
plt.scatter(degrees,complexity_test_error, c = 'b',label = 'error')
plt.text(2,bias_squared[1]+0.01,str(round(bias_squared[1],3)), c = 'r')
plt.text(1,bias_squared[0]+0.01,str(round(bias_squared[0],3)), c = 'r')
plt.text(2,complexity_test_error[1]+0.01,str(round(complexity_test_error[1],3)), c = 'b')
plt.text(1,complexity_test_error[0]+0.01,str(round(complexity_test_error[0],3)), c = 'b')
plt.text(2,variance[1]+0.01,str(round(variance[1],3)), c = 'g')
plt.text(1,variance[0]+0.01,str(round(variance[0],3)), c = 'g')
plt.xticks(degrees, labels)
plt.legend()
plt.show()
```



- Compared with model without regularization, $L2$(Ridge) model has panelty on the predictors, thus reduces model complexity.
- From previous analysis, we know that model with lower complexity have higher bias and lower variance. Thus compared with Ridge regression, $g_{10}x$ have lower bias and higher variance
- Since fit model with 10 polynomial terms is already overfitting, when conduct $L2$ regularization, we are expected to have lower MSE.

Problem 3

In [13]:
```
# load Data From opem ML
from scipy.io import arff
import urllib.request
import io # for io.StringIO()

url_1 = 'https://www.openml.org/data/download/37/dataset_37_diabetes.arff'
ftpstream = urllib.request.urlopen(url_1)
diabetes_data, _ = arff.loadarff(io.StringIO(ftpstream.read().decode('utf-8')))

url_2 = 'https://www.openml.org/data/download/61/dataset_61_iris.arff'
ftpstream = urllib.request.urlopen(url_2)
iris_data, _ = arff.loadarff(io.StringIO(ftpstream.read().decode('utf-8')))

diabetes_data = pd.DataFrame(diabetes_data)
iris_data = pd.DataFrame(iris_data)
```

1---------------

Show diabetes dataset

In [14]:
```
diabetes_data.head()
```

Out[14]:

	preg	plas	pres	skin	insu	mass	pedi	age	class
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0	b'tested_positive'
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0	b'tested_negative'
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0	b'tested_positive'
3	1.0	89.0	66.0	23.0	94.0	28.1	0.167	21.0	b'tested_negative'
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0	b'tested_positive'

- Diabetes_data is has binary outcome, test_positive or test_negative.
- It has 8 predictors and all of them are numerical features.
- It has 768 instances.
- It has no categorical features.

Show iris dataset

In [15]:
```
iris_data.head()
```

Out[15]:

	sepallength	sepalwidth	petallength	petalwidth	class
0	5.1	3.5	1.4	0.2	b'Iris-setosa'
1	4.9	3.0	1.4	0.2	b'Iris-setosa'
2	4.7	3.2	1.3	0.2	b'Iris-setosa'
3	4.6	3.1	1.5	0.2	b'Iris-setosa'
4	5.0	3.6	1.4	0.2	b'Iris-setosa'

- iris_data has 3 class of out come which are setosa, versicolor, and virginica.
- It has 4 predictors and all of them are numerical features.
- It has 150 instances.
- It has no categorical features.

2---------------

train pipline for diabetes dataset

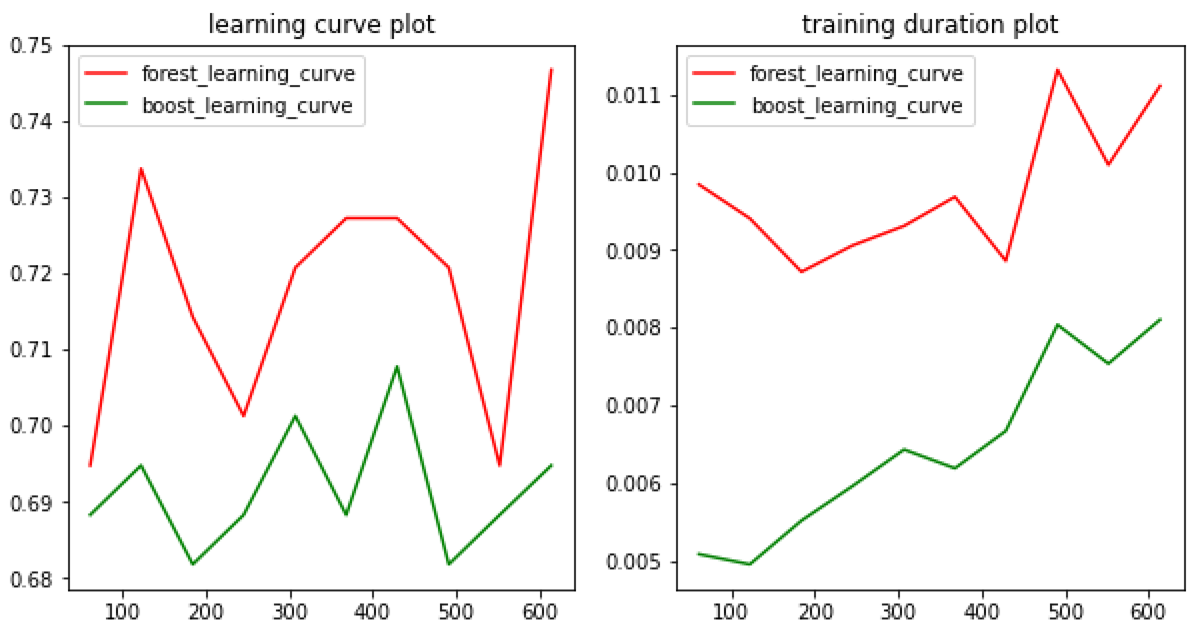
In [16]:

```
# convert target dtype to binary
diabetes_data['class'] = diabetes_data['class'] == b'tested_positive'
# 80% train test split
x_train, x_test, y_train, y_test = sklearn.model_selection.train_test_split(diabetes_data.
                                                                             diabetes_data
                                                                            test_size = 0.2
                                                                            random_state=se
```

In [17]:

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn import metrics
percent = [i for i in range(1,11)]
test_samples = len(y_test)
n_estimators = 8


# I didn't tune the hyper parameters just randomly picked 8 as n_estimators for both mode
time_duation_forests = []
time_duation_boosts = []
accuracy_forests = []
accuracy_boosts = []
training_size_of_data = []
for per in percent:
    if per != 10:
        x_train_per, _, y_train_per, _ = sklearn.model_selection.train_test_split(
                                                                                 x_train,
                                                                                 y_train,
                                                                                 train_size
                                                                                 random_sta

    else:
        x_train_per, y_train_per = x_train, y_train

    training_size_of_data.append(len(x_train_per))
    # fit forest model and get fitting time
    time_start_forest = time.time()
    forest_regressor = RandomForestClassifier(n_estimators = n_estimators, random_state =
    forest_regressor.fit(x_train_per, y_train_per)
    time_end_forest = time.time()
    time_duation_forest = time_end_forest - time_start_forest
    time_duation_forests.append(time_duation_forest)
    # get forest accuracy on test model
    matrix_forest = metrics.confusion_matrix(y_test, forest_regressor.predict(x_test))
    accuracy_forest = (matrix_forest[0,0]+matrix_forest[1,1])/test_samples
    accuracy_forests.append(accuracy_forest)


    # fit boost model and get fitting time
    time_start_boost = time.time()
    gradient_boosting_regressor = GradientBoostingClassifier(n_estimators = n_estimators,
    gradient_boosting_regressor.fit(x_train_per, y_train_per)
    time_end_boost = time.time()
    time_duation_boost = time_end_boost - time_start_boost
    time_duation_boosts.append(time_duation_boost)
    # get boost accuracy on test model
    matrix_boost = metrics.confusion_matrix(y_test, gradient_boosting_regressor.predict(x_
    accuracy_boost = (matrix_boost[0,0]+matrix_boost[1,1])/test_samples
    accuracy_boosts.append(accuracy_boost)

fix,ax = plt.subplots(1,2,figsize = (10,5))

ax[0].plot(training_size_of_data,accuracy_forests, label = 'forest_learning_curve', c = 'r
ax[0].plot(training_size_of_data,accuracy_boosts, label = 'boost_learning_curve', c = 'g')
ax[0].legend()
ax[0].set_title('learning curve plot')

ax[1].plot(training_size_of_data,time_duation_forests, label = 'forest_learning_curve', c
ax[1].plot(training_size_of_data,time_duation_boosts, label = 'boost_learning_curve', c =
ax[1].legend()
ax[1].set_title('training duration plot')
```

Out[17]:
```
Text(0.5, 1.0, 'training duration plot')
```



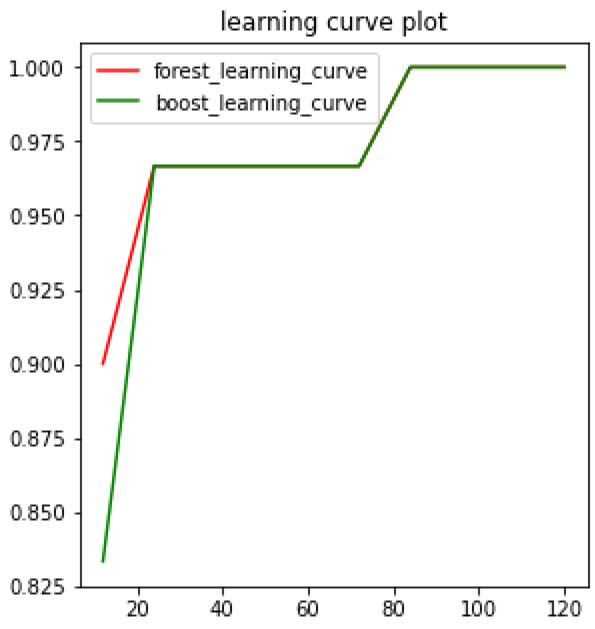
train pipline for iris dataset

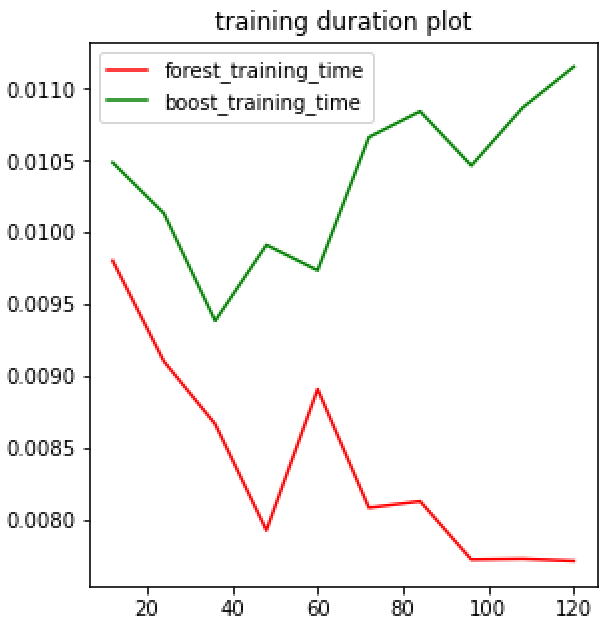
In [18]:
```
# convert iris_data target value dtype
iris_data['class'][iris_data['class'] == b'Iris-setosa'] = 'setosa'
iris_data['class'][iris_data['class'] == b'Iris-versicolor'] = 'versicolor'
iris_data['class'][iris_data['class'] == b'Iris-virginica'] = 'vairginica'
# 80% train test split
x_train, x_test, y_train, y_test = sklearn.model_selection.train_test_split(iris_data.iloc
                                                                            iris_data['cla
                                                                           test_size = 0.2
                                                                           random_state=se
```

In [19]:
```
percent = [i for i in range(1,11)]
test_samples = len(y_test)
n_estimators = 8


# I didn't tune the hyper parameters just randomly picked 8 as n_estimators for both mode
time_duation_forests = []
time_duation_boosts = []
accuracy_forests = []
```

```
accuracy_boosts = []
training_size_of_data = []
for per in percent:
    if per != 10:
        x_train_per, _, y_train_per, _ = sklearn.model_selection.train_test_split(
                                                                                    x_train,
                                                                                    y_train,
                                                                                    train_size
                                                                                    random_sta
    else:
        x_train_per, y_train_per = x_train, y_train

    training_size_of_data.append(len(x_train_per))
    # fit forest model and get fitting time
    time_start_forest = time.time()
    forest_regressor = RandomForestClassifier(n_estimators = n_estimators, random_state =
    forest_regressor.fit(x_train_per, y_train_per)
    time_end_forest = time.time()
    time_duation_forest = time_end_forest - time_start_forest
    time_duation_forests.append(time_duation_forest)
    # get forest accuracy on test model
    matrix_forest = metrics.confusion_matrix(y_test, forest_regressor.predict(x_test))
    accuracy_forest = (matrix_forest[0,0]+matrix_forest[1,1]+matrix_forest[2,2])/test_samp
    accuracy_forests.append(accuracy_forest)


    # fit boost model and get fitting time
    time_start_boost = time.time()
    gradient_boosting_regressor = GradientBoostingClassifier(n_estimators = n_estimators,
    gradient_boosting_regressor.fit(x_train_per, y_train_per)
    time_end_boost = time.time()
    time_duation_boost = time_end_boost - time_start_boost
    time_duation_boosts.append(time_duation_boost)
    # get boost accuracy on test model
    matrix_boost = metrics.confusion_matrix(y_test, gradient_boosting_regressor.predict(x_
    accuracy_boost = (matrix_boost[0,0]+matrix_boost[1,1]+matrix_boost[2,2])/test_samples
    accuracy_boosts.append(accuracy_boost)

fix,ax = plt.subplots(1,2,figsize = (10,5))

ax[0].plot(training_size_of_data,accuracy_forests, label = 'forest_learning_curve', c = 'r
ax[0].plot(training_size_of_data,accuracy_boosts, label = 'boost_learning_curve', c = 'g')
ax[0].legend()
ax[0].set_title('learning curve plot')

ax[1].plot(training_size_of_data,time_duation_forests, label = 'forest_training_time', c =
ax[1].plot(training_size_of_data,time_duation_boosts, label = 'boost_training_time', c = '
ax[1].legend()
ax[1].set_title('training duration plot')
```

```
Out[19]: Text(0.5, 1.0, 'training duration plot')
```





- Here the training duration for iris dataset is decreasing is simply because training samples is too small training duration is dominate by other facts, I also enlarge the iris data set and rerun the experiment, the training duration is now increasing with larger training size.

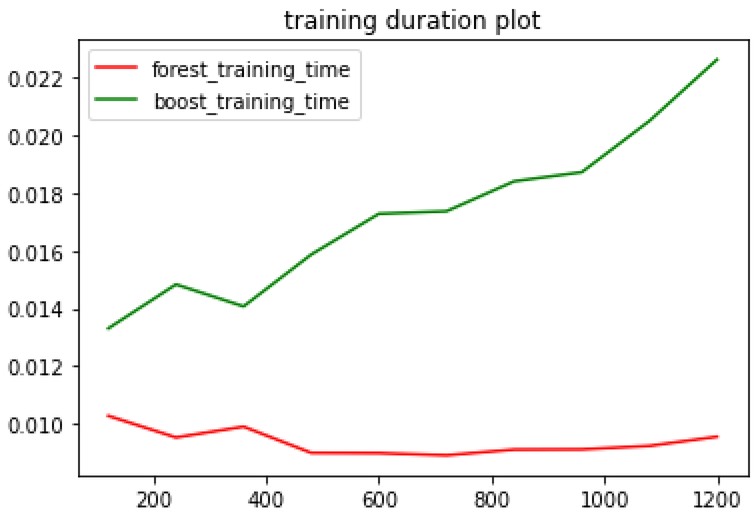
Rerun iris with larger samples

In [20]:

```
iris_data_list = []
for i in range(10):
    iris_data_list.append(iris_data)
iris_data = pd.concat(iris_data_list, axis = 0)

# 80% train test split
x_train, x_test, y_train, y_test = sklearn.model_selection.train_test_split(iris_data.iloc
                                                                             iris_data['cla
                                                                            test_size = 0.2
                                                                            random_state=se
```

In [21]:

```
percent = [i for i in range(1,11)]
test_samples = len(y_test)
n_estimators = 8

# I didn't tune the hyper parameters just randomly picked 8 as n_estimators for both mode
time_duation_forests = []
time_duation_boosts = []
accuracy_forests = []
accuracy_boosts = []
training_size_of_data = []
for per in percent:
    if per != 10:
        x_train_per, _, y_train_per, _ = sklearn.model_selection.train_test_split(
                                                                                 x_train,
                                                                                 y_train,
                                                                                 train_size
                                                                                 random_sta

    else:
        x_train_per, y_train_per = x_train, y_train

    training_size_of_data.append(len(x_train_per))
    # fit forest model and get fitting time
    time_start_forest = time.time()
    forest_regressor = RandomForestClassifier(n_estimators = n_estimators, random_state =
    forest_regressor.fit(x_train_per, y_train_per)
    time_end_forest = time.time()
    time_duation_forest = time_end_forest - time_start_forest
    time_duation_forests.append(time_duation_forest)
    # get forest accuracy on test model
    matrix_forest = metrics.confusion_matrix(y_test, forest_regressor.predict(x_test))
    accuracy_forest = (matrix_forest[0,0]+matrix_forest[1,1]+matrix_forest[2,2])/test_samp
    accuracy_forests.append(accuracy_forest)


    # fit boost model and get fitting time
    time_start_boost = time.time()
    gradient_boosting_regressor = GradientBoostingClassifier(n_estimators = n_estimators,
    gradient_boosting_regressor.fit(x_train_per, y_train_per)
    time_end_boost = time.time()
    time_duation_boost = time_end_boost - time_start_boost
    time_duation_boosts.append(time_duation_boost)
    # get boost accuracy on test model
    matrix_boost = metrics.confusion_matrix(y_test, gradient_boosting_regressor.predict(x_
    accuracy_boost = (matrix_boost[0,0]+matrix_boost[1,1]+matrix_boost[2,2])/test_samples
    accuracy_boosts.append(accuracy_boost)

plt.plot(training_size_of_data,time_duation_forests, label = 'forest_training_time', c = '
plt.plot(training_size_of_data,time_duation_boosts, label = 'boost_training_time', c = 'g'
plt.legend()
plt.title('training duration plot')
plt.show()
```



3-------------------------------

Diabetes data set

three Observations

- The accuracy on test set is unstable, and don't show any trend when increasing training samples.
- The training time is increasing when increasing the training size.
- Forest model have higher accuracy and longer training time compared with gradient boost.

Compare training time and accuracy

- **Train a forest mode with whole trainig set will give the highest accuracy (75%).**
- **Using boost model and 10% of training set has the shorest training time**

Iris data set

Observations

- The accuracy on the test set is increasing when increasing the training size.
- Since the whole data set only has 150 samples the training time is dominated by other factors, can't tell the relationship between training size and training time. **(But when duplicate the data set 10 times and rerun the pipeline, we can see the training time is increasing when we have a larger training size.)**
- Accuracy of the two models is about the same level, but the forest will have less training time than the boost model.
- Compared with Diabetes data, the training time relationship between the two models is different, which might be because the forest model's training time is more sensitive to a number of predictors.

Compare training time and accuracy

- **Train a both mode with more than 100 trainig samples will give the highest accuracy (100%).**
- **Using forest model with around 80 training samples has the shortest training time.**

Problem 4

1-----------------------------

- According to 'The Relationship Between Precision-Recall and ROC Curves' Paper: **Note that a point in ROC space defines aunique confusion matrix when the dataset is fixed.Since in PR space we ignore T N, one might worrythat each point may correspond to multiple confusionmatrices. However, with a fixed number of positiveand negative examples, given the other three entriesin a matrix, T N is uniquely determined. If Recall =0, we are unable to recover FP, and thus cannot finda unique confusion matrix.Consequently, we have a one-to-one mapping betweenconfusion matrices and points in PR space. This implies that we also have a one-to-one mapping betweenpoints (each defined by a confusion matrix) in ROCspace and PR space; hence, we can translate a curvein ROC space to PR space and vice-versa.One important definition we need for our next theoremis the notion that one curve dominates another curve,“meaning that all other...curves are beneath it or equal to it (Provost et al., 1998).”**
- With a fixed data set, a point in ROC space defines a unique confusion matrix, Thus each point define a unique TN, TP, FN, FP, thus True negative matter for both ROC and PR curve. and a point in ROC space corresponds to a unique point on PR curve.

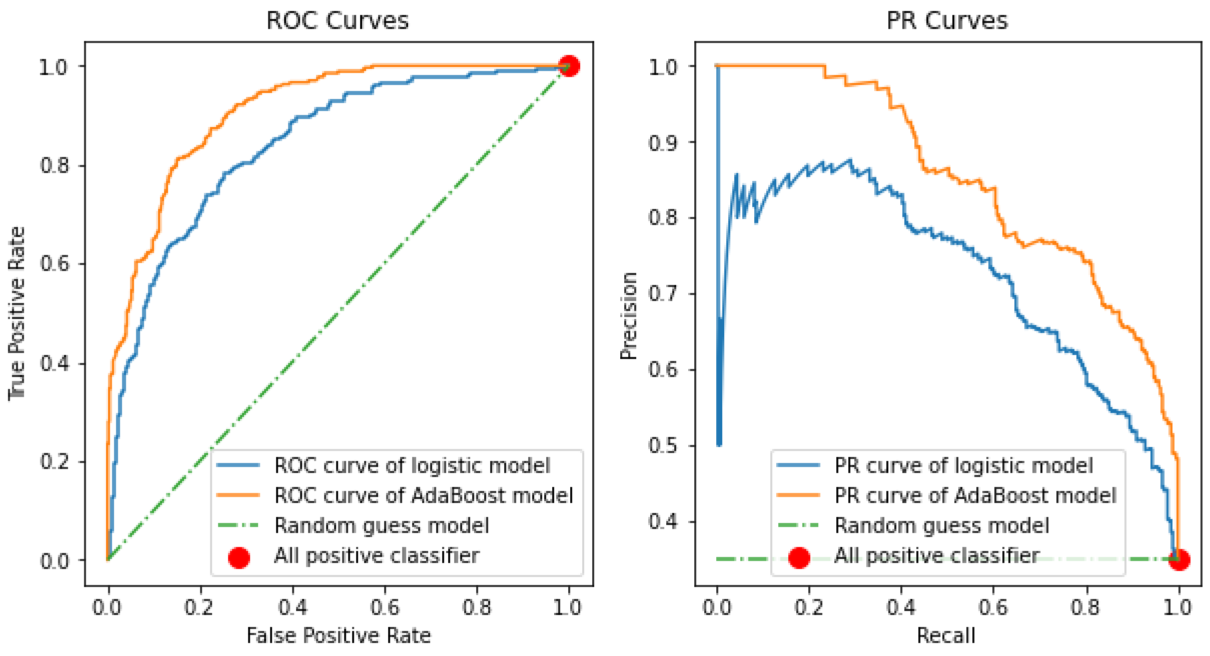
2-----------------------------

In [22]:

```
# still using Diabetes data sets in problem 3
from sklearn.metrics import roc_curve, precision_recall_curve
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import AdaBoostClassifier
import plotly.express as px
x_train = diabetes_data.iloc[:,0:8]
y_train = diabetes_data['class']

logi_model = LogisticRegression()
logi_model.fit(np.array(x_train), np.array(y_train))
```

```
y_score_logi = logi_model.predict_proba(x_train)[:, 1]
fpr_logi, tpr_logi, _ = roc_curve(y_train, y_score_logi)
precision_logi, recall_logi, _ = precision_recall_curve(y_train, y_score_logi)

ada_model = AdaBoostClassifier()
ada_model.fit(np.array(x_train), np.array(y_train))
y_score_ada = ada_model.predict_proba(x_train)[:, 1]
fpr_ada, tpr_ada, _ = roc_curve(y_train, y_score_ada)
precision_ada, recall_ada, _ = precision_recall_curve(y_train, y_score_ada)

fix,ax = plt.subplots(1,2,figsize = (10,5))
ax[0].plot(fpr_logi, tpr_logi, label = 'ROC curve of logistic model')
ax[0].plot(fpr_ada, tpr_ada, label = 'ROC curve of AdaBoost model')
ax[0].plot([0,1],[0,1], label = 'Random guess model',linestyle= '-.')
ax[0].set_xlabel('False Positive Rate')
ax[0].set_ylabel('True Positive Rate')
ax[0].set_title('ROC Curves')
ax[0].scatter(1,1,c = 'r',s = 100, label = 'All positive classifier')
ax[0].legend()

ax[1].plot(recall_logi, precision_logi, label = 'PR curve of logistic model')
ax[1].plot(recall_ada, precision_ada, label = 'PR curve of AdaBoost model')
ax[1].plot([0,1],[precision_ada[0],precision_ada[0]], label = 'Random guess model',linesty
ax[1].scatter(1,precision_ada[0],c = 'r',s = 100, label = 'All positive classifier')
ax[1].set_xlabel('Recall')
ax[1].set_ylabel('Precision')
ax[1].set_title('PR Curves')
ax[1].legend()
```

Out[22]:
```
<matplotlib.legend.Legend at 0x7ff215d78940>
```



- Since the data is not balanced, precision for all positive classifier is around 0.35

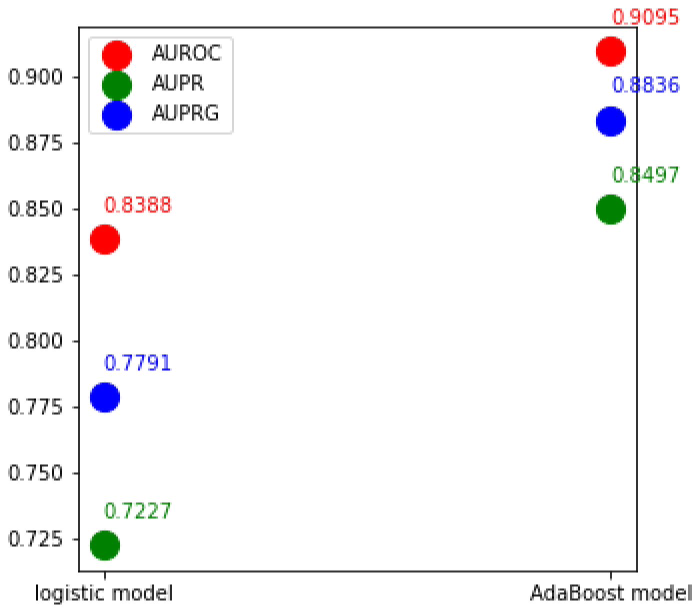
3-------------------------------------

- According to the paper,
- $precG = \frac{prec-\pi}{(1-\pi)prec}$
- $recG = \frac{rec-\pi}{(1-\pi)rec}$

- For simplify I just use the package from 'https://github.com/meeliskull/prg/tree/master/Python_package'

In [23]:
```python
# pip install prg
from sklearn.metrics import auc
from prg import prg

prg_curve_logi = prg.create_prg_curve(y_train, y_score_logi)
prg_curve_ada = prg.create_prg_curve(y_train, y_score_ada)

AUPRG_logi= prg.calc_auprg(prg_curve_logi)
AUPRG_ada= prg.calc_auprg(prg_curve_ada)

AUROC_logi = auc(fpr_logi, tpr_logi)
AUPR_logi = auc(recall_logi, precision_logi)

AUROC_ada = auc(fpr_ada, tpr_ada)
AUPR_ada = auc(recall_ada, precision_ada)


fix,ax = plt.subplots(1,1,figsize = (5,5))

labels = ['logistic model', 'AdaBoost model']
ax.scatter(0,AUROC_logi, label = 'AUROC', c = 'r',s = 200)
ax.text(0,AUROC_logi+0.01, str(round(AUROC_logi,4)),c = 'r')
ax.scatter(0,AUPR_logi, label = 'AUPR', c = 'g',s = 200)
ax.text(0,AUPR_logi+0.01, str(round(AUPR_logi,4)),c = 'g')
ax.scatter(0,AUPRG_logi, label = 'AUPRG', c = 'b',s = 200)
ax.text(0,AUPRG_logi+0.01, str(round(AUPRG_logi,4)),c = 'b')

ax.scatter(1,AUROC_ada,  c = 'r',s = 200)
ax.text(1,AUROC_ada+0.01, str(round(AUROC_ada,4)),c = 'r')
ax.scatter(1,AUPR_ada,  c = 'g',s = 200)
ax.text(1,AUPR_ada+0.01, str(round(AUPR_ada,4)),c = 'g')
ax.scatter(1,AUPRG_ada,  c = 'b',s = 200)
ax.text(1,AUPRG_ada+0.01, str(round(AUPRG_ada,4)),c = 'b')
ax.legend()
ax.set_xticks([0,1])
ax.set_xticklabels(labels)
```

Out[23]:
```
[Text(0, 0, 'logistic model'), Text(1, 0, 'AdaBoost model')]
```



- According to the paper, Precision-Recall analysis differs from classification accuracy in that thebaseline to beat is the always-positive classifier rather than any random classifier. This baseline hasprec=π andrec=1,anditiseasilyseenthatanymodelwithprec<π orrec<π losesagainstthis baseline. Hence it makes sense to consider only precision and recall values in the interval [π,1]

- I agree with the statement and I think PR-gain is better.