Name: Aman Sharma

Rollno.: A25

Regno: 11810810

Github link: https://github.com/amanshr07/INT247

Echocardiogram

i) Abstract

About the dataset:

https://archive.ics.uci.edu/ml/datasets/echocardiogram

- echocardiogram_data.txt: dataset of ECG results
- echocardiogram_description.txt: description from UCI website
- columns.txt: List of columns in the echocardiogram_data.txt dataset

Data Set Characteristics:	Multivariate	**Number of Instances:**	132	**Area:**	Life
Attribute Characteristics:	Categorical, Integer, Real	**Number of Attributes:**	12	**Date Donated**	1989-02-28
Associated Tasks:	Classification	**Missing Values?**	Yes	**Number of Web Hits:**	208897

a) Data Set Information:

All the patients suffered heart attacks at some point in the past. Some are still alive and some are not. The survival and still-alive variables, when taken together, indicate whether a patient survived for at least one year following the heart attack.

The problem addressed by past researchers was to predict from the other variables whether or not the patient will survive at least one year. The most difficult part of this problem is correctly predicting that the patient will NOT survive. (Part of the difficulty seems to be the size of the data set.)

b) Attribute Information:

1. survival -- the number of months patient survived (has survived, if patient is still alive). Because all the patients had their heart attacks at different times, it is possible that some patients have survived less than one year but they are still alive. Check the second variable to confirm this. Such patients cannot be used for the prediction task mentioned above.
2. still-alive -- a binary variable. 0=dead at end of survival period, 1 means still alive
3. age-at-heart-attack -- age in years when heart attack occurred
4. pericardial-effusion -- binary. Pericardial effusion is fluid around the heart. 0=no fluid, 1=fluid
5. fractional-shortening -- a measure of contracility around the heart lower numbers are increasingly abnormal
6. epss -- E-point septal separation, another measure of contractility. Larger numbers are increasingly abnormal.
7. lvdd -- left ventricular end-diastolic dimension. This is a measure of the size of the heart at end-diastole. Large hearts tend to be sick hearts.
8. wall-motion-score -- a measure of how the segments of the left ventricle are moving
9. wall-motion-index -- equals wall-motion-score divided by number of segments seen. Usually 12-13 segments are seen in an echocardiogram. Use this variable INSTEAD of the wall motion score.
10. mult -- a derivate var which can be ignored
11. name -- the name of the patient (I have replaced them with "name")
12. group -- meaningless, ignore it
13. alive-at-1 -- Boolean-valued. Derived from the first two attributes. 0 means patient was either dead after 1 year or had been followed for less than 1 year. 1 means patient was alive at 1 year.

ii) INTRODUCTION

We are working on Echocardiogram data, so first we need to understand what does Echocardiogram means,

An **echocardiogram (echo)** is a graphic outline of the heart's movement. During an echo test, ultrasound (high-frequency sound waves) from a hand-held wand placed on your chest provides pictures of the heart's valves and chambers and helps the sonographer evaluate the pumping action of the heart. Echo is often combined with Doppler ultrasound and color Doppler to evaluate blood flow across the heart's valves.

Why is an echocardiogram performed?

The test is used to:

- Assess the overall function of your heart
- Determine the presence of many types of heart disease, such as valve disease, myocardial disease, pericardial disease, infective endocarditis, cardiac masses and congenital heart disease
- Follow the progress of valve disease over time
- Evaluate the effectiveness of your medical or surgical treatments

iii) Implementation:

Before implementing the project, we need to understand the problems with the data:

- One bad row
- Many missing values
- Missing Values mostly consistent with dataset description, but not completely.

a) So first we need to clean the data set(initial_exploration.ipynb)

```
dataFile = "data/echocardiogram_data.txt"
try:
  ecg = pandas.read_csv(dataFile, na_values=["?"],
                        header=None)
except Exception as e:
    print (e)
```

```
Error tokenizing data. C error: Expected 13 fields in line 50, saw 14
```

So there's something wrong with line 50.

What is it?

```
# Grab that one line
line50 = open(dataFile).readlines()[49]
print (line50)
# Chop it up
fields = line50.split(",")
print (fields)
print (len(fields))
```

```
,?,?,77,?,?,?,?,?,2,?,name,2,?

['', '?', '?', '77', '?', '?', '?', '?', '?', '2', '?', 'name', '2', '?\n']
14
```

Alright, that's indeed bad. Worse, the name column is two entries from the end, as it should be. This means that extra field is somewhere in the middle.

Well, at least the rest of the information looks sparse enough that this record is useless.

So, pandas has an option to skip lines with too many fields. By default it will warn us for each bad line.

After adding the data to ecg and adding header to it, we have cleaned our data little bit. But there are still some missing variables:

Three are a number of missing values

The dataset description gives this listing:

Attribute #:	Number of Missing Values: (total: 132)
1	2
2	1
3	5
4	1
5	8
6	15
7	11
8	4
9	1
10	4
11	0
12	22
13	58

After few more operation we came to know that There's not enough data to attempt classification to identify patients who will survive for one year. Instead, try a regression problem with the survival duration as target variable.
The function above can be used to clean the data, and will be put into a module called ecg_tools.

For details process of exploration and cleaning of data you can visit:

https://github.com/amanshr07/INT247/blob/main/initial_exploration.ipynb

b)Data Loading/Cleaning functionality moved to module(ecg_tools.py)

https://github.com/amanshr07/INT247/blob/main/ecg_tools.py

c)Now for regression Model

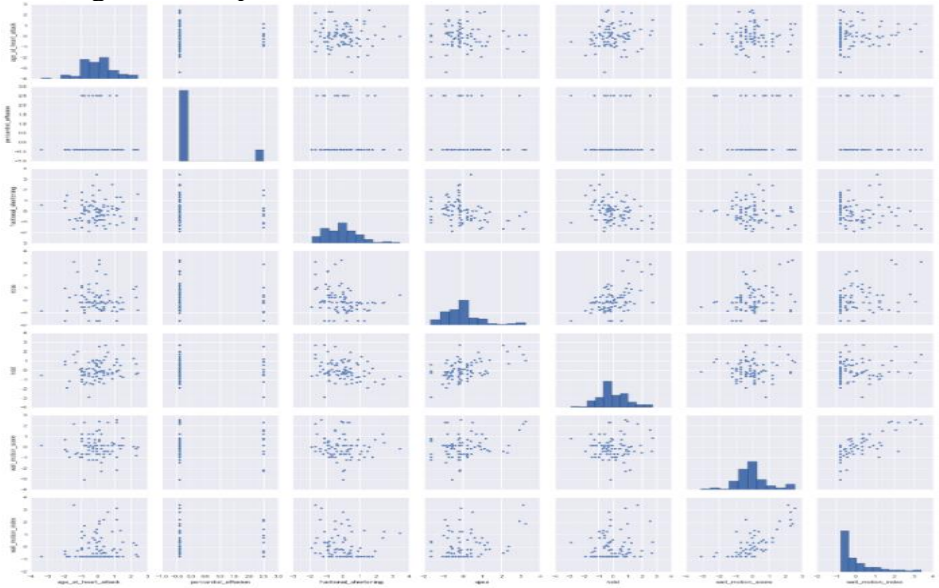
https://github.com/amanshr07/INT247/blob/main/survival_duration_regression.ipynb

In original data set we need to predict which patients survive one year after heart attack. We will try to predict survival duration.

After cleaning function ported from notebook to module(ecg_tools.py)

1)Scatter Matrix (see the diagram below):

- Top row: target variable vs. each other variable
- Not much correlation visible anywhere really
- Originally imagined doing dimensionality reduction, but didn't seem worth it after seeing the graphs
- Missing values replaced with medium of distribution for that variable



2) After Regression:

- Split data 80/20 train test
- Tired several algorithms: least-squares, lasso, ridge, support vector nearest-neighbor
- Grid search for hyperparameters, %-fold cross validation at each grid point

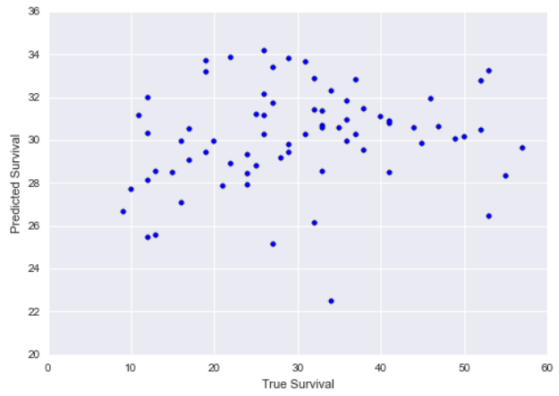
3) Graph After Each regression:

Basic Linear Regression

```
lr = LinearRegression()

trainShow(lr, trainX, trainY)
```

```
Best params not implemented for model: <class 'sklearn.linear_mod
el.base.LinearRegression'>
```

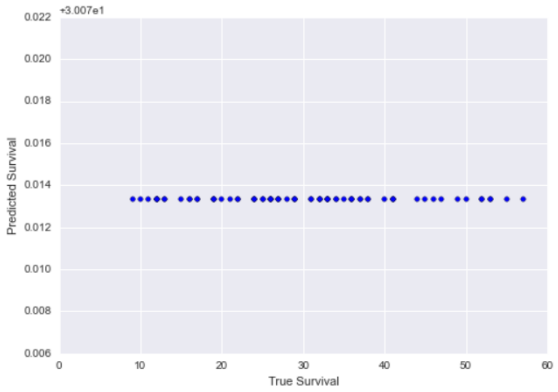


That fails to fit the training set very well.

Lasso regression

```
lasso = GridSearchCV(
                    Lasso(),
                    cv=5,
                    param_grid={
                       "alpha": numpy.logspace(-3, 3, 1000),
                      }
                    )
trainShow(lasso, trainX, trainY)
```

```
{'alpha': 3.3076497807442427}
-0.0278420857174
```

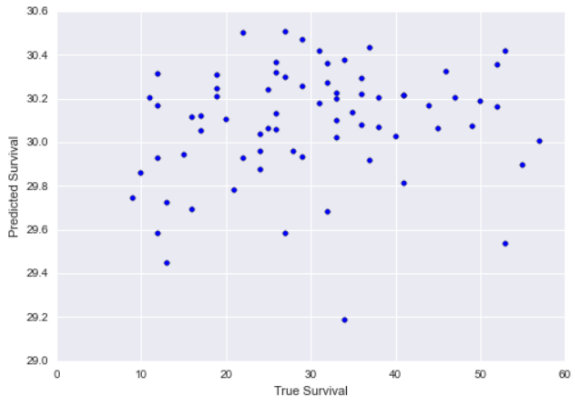


This just went towards real hard regularization which has no shape. That's no good.

Ridge regression

```
ridge = GridSearchCV(
                    Ridge(),
                    cv=5,
                    param_grid={
                       "alpha": numpy.logspace(-3, 3, 1000),
                      }
                    )
trainShow(ridge, trainX, trainY)
```

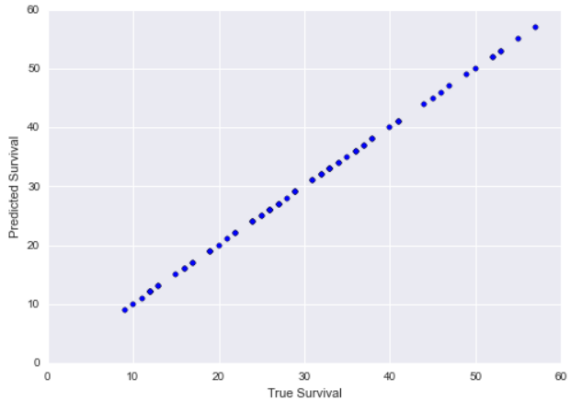
```
{'alpha': 1000.0}
-0.0274745825411
```



Nearest neigbors regression

```
neighbor = GridSearchCV(
                    KNeighborsRegressor(),
                    cv=5,
                    param_grid={
                       "n_neighbors": [5, 10, 25, 40],
                       "weights": ["uniform", "distance"],
                       "algorithm":["ball_tree", "kd_tree", "brut
e"]
                      }
                    )
trainShow(neighbor, trainX, trainY)
```

```
{'n_neighbors': 40, 'weights': 'distance', 'algorithm': 'ball_tre
e'}
-0.0469257647475
```

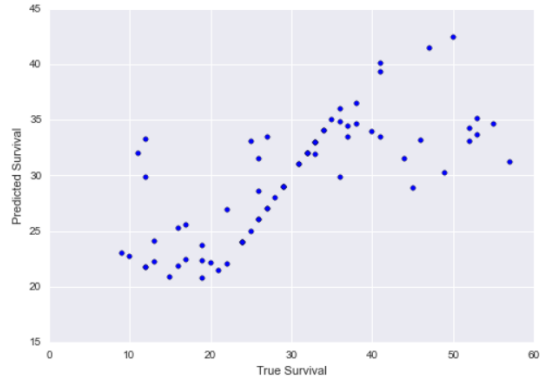


Support Vector Regression

```
svr = GridSearchCV(
                    SVR(),
                    cv=5,
                    param_grid={
                       "kernel":["rbf"],
                       "C": numpy.logspace(-3, 6, 20),
                       "gamma": numpy.logspace(-3, 3, 10),
                       "epsilon": numpy.logspace(-6, 0, 10)
                      }
                    )

trainShow(svr, trainX, trainY)
```

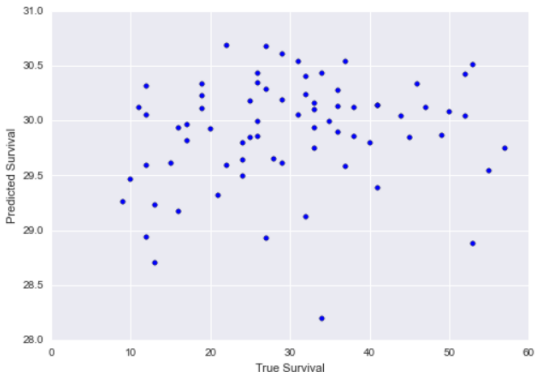
```
{'epsilon': 0.0001, 'C': 6.1584821106602607, 'gamma': 0.464158883
36127775, 'kernel': 'rbf'}
0.00711553215524
```



Kernel Ridge Regression

```
kridge = GridSearchCV(
                    KernelRidge(),
                    cv=5,
                    param_grid={
                       "alpha": numpy.logspace(-3, 3, 10),
                       "kernel":["rbf", "linear", "poly"],
                       "gamma": numpy.logspace(-3, 3, 10),
                       "degree":[1,2,3]
                      }
                    )
trainShow(kridge, trainX, trainY)
```

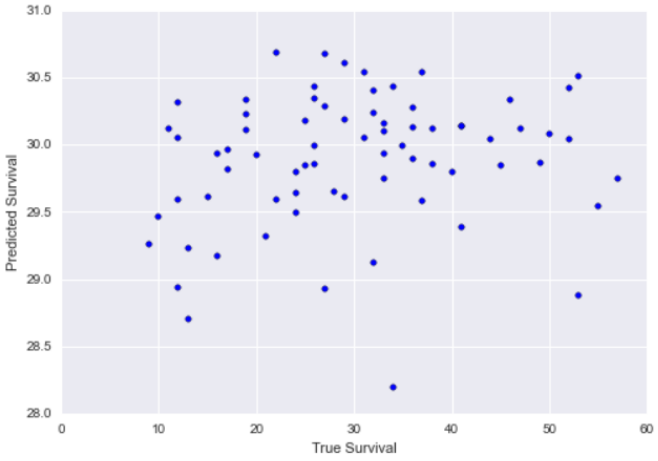
```
{'alpha': 0.46415888336127775, 'gamma': 0.001, 'degree': 1, 'kern
el': 'poly'}
-0.0275136502935
```



Kernel Ridge Regression

```
kridge = GridSearchCV(
                    KernelRidge(),
                    cv=5,
                    param_grid={
                       "alpha": numpy.logspace(-3, 3, 10),
                       "kernel":["rbf", "linear", "poly"],
                       "gamma": numpy.logspace(-3, 3, 10),
                       "degree":[1,2,3]
                      }
                    )
trainShow(kridge, trainX, trainY)
```

```
{'alpha': 0.46415888336127775, 'gamma': 0.001, 'degree': 1, 'kern
el': 'poly'}
-0.0275136502935
```



4) Now performing the test,

Test MSE

Take a look at the mean square error on the test sample for these models.

```
models = OrderedDict()
models["lr"] = lr
models["lasso"] = lasso
models["ridge"] = ridge
models["kridge"] = kridge
models["svr"] = svr
models["neighbor"] = neighbor

for name, model in models.items():
  print "%s: %s" % (name, mse(model, testX, testY))
```

```
lr: 98.9232488776
lasso: 97.4340277778
ridge: 97.0083634704
kridge: 96.5386013402
svr: 113.334842691
neighbor: 101.57879808
```

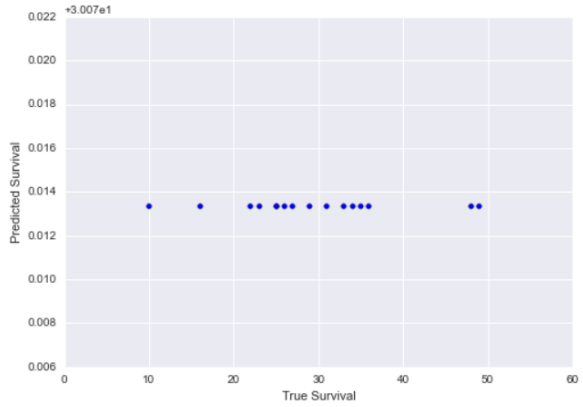
Those are all pretty terrible.

Look at lasso and ridge

Lasso test

```
testShow(lasso, testX, testY)
```

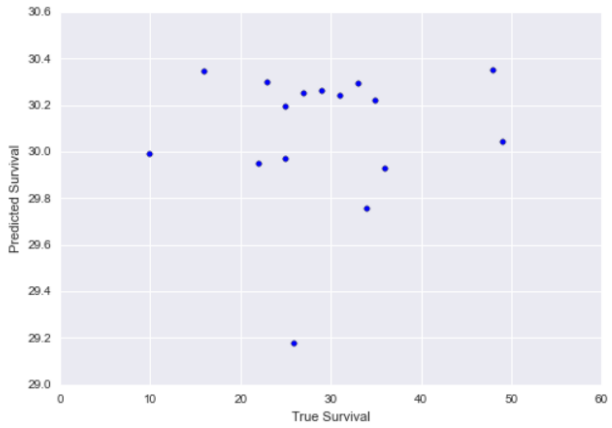
```
97.4340277778
```



Ridge test

```
testShow(ridge, testX, testY)
```

```
97.0083634704
```



Conclusion:

- The data set had some missing values so we had to rectify it.
- We used two notebooks, one module.
- No algorithm successfully fit he training set.
- Nearest-neighbour just let every point be its own neighbour.