A2_template_2023

July 15, 2023

1 Assignment 2: Classification

2 Using Machine Learning Tools

2.1 Overview

In this assignment, you will apply some popular machine learning techniques to the problem of classifying data from histological cell images for the diagnosis of malignant breast cancer. This will be presented as a practical scenario where you are approached by a client to solve a problem.

The main aims of this assignment are:

- to use the best practice machine learning workflow for producing a solution to a client's problem;
- to visualise data and determine the best pre-processing;
- to create the necessary datasets for training and testing purposes;
- to train and optimise a selection of models, then choose the best;
- to obtain an unbiased measurement of the final model's performance;
- to interpret results clearly and concisely.

This assignment relates to the following ACS CBOK areas: abstraction, design, hardware and software, data and information, HCI and programming.

2.2 General instructions

This assignment is divided into several tasks. Use the spaces provided in this notebook to answer the questions posed in each task. Note that some questions require writing a small amount of code, some require graphical results, and some require comments or analysis as text. It is your responsibility to make sure your responses are clearly labelled and your code has been fully executed (**with the correct results displayed**) before submission!

Do not manually edit the data set file we have provided! For marking purposes, it's important that your code runs correctly on the original data file.

Some of the parts of this assignment build on the workflow from the first assignment and that part of the course, and so less detailed instructions are provided for this, as you should be able to implement this workflow now without low-level guidance. A substantial portion of the marks for this assignment are associated with making the right choices and executing this workflow correctly and efficiently. Make sure you have clean, readable code as well as producing outputs, since your coding will also count towards the marks (however, excessive commenting is discouraged and will lose marks, so aim for a modest, well-chosen amount of comments and text in outputs).

This assignment can be solved using methods from sklearn, pandas, and matplotlib as presented in the workshops. Other libraries should not be used (even though they might have nice functionality) and certain restrictions on sklearn functions will be made clear in the instruction text. You are expected to search and carefully read the documentation for functions that you use, to ensure you are using them correctly.

3 Scenario

A client approaches you to solve a machine learning problem for them. They run a pathology lab that processes histological images for healthcare providers and they have created a product that measures the same features as in the *Wisconsin breast cancer data set* though using different acquisitions and processing methods. This makes their method much faster than existing ones, but it is also slightly noisier. They want to be able to diagnose *malignant* cancer (and distinguish them from *benign* growths) by employing machine learning techniques, and they have asked you to implement this for them.

Their requirements are: 1) have at least a 95% probability of detecting malignant cancer when it is present; 2) have no more than 1 in 10 healthy cases (those with benign tumours) labelled as positive (malignant).

They have hand-labelled 300 samples for you, which is all they have at the moment.

Please follow the instructions below, which will vary in level of detail, as appropriate to the marks given.

3.1 1. Investigate Dataset (10% = 3 marks)

[1]:
```
# This code imports some libraries that you will need.
# You should not need to modify it, though you are expected to make other␣
 ↪imports later in your code.

# Python 3.5 is required
import sys
assert sys.version_info >= (3, 5)

# Common imports
import numpy as np
import time

# Pandas for overview
import pandas as pd

# Scikit-Learn 0.20 is required
import sklearn
assert sklearn.__version__ >= "0.20"
from sklearn import tree
from sklearn import svm
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix

# Plot setup
import matplotlib as mpl
import matplotlib.pyplot as plt
mpl.rc('axes', labelsize=7)
mpl.rc('xtick', labelsize=6)
mpl.rc('ytick', labelsize=6)
mpl.rc('figure', dpi=240)
plt.close('all')

import seaborn as sns
```

3.1.1 1.1 Load the dataset [0.5 marks]

Do this from the csv file, `assignment2.csv`, as done in assignment 1 and workshops 2 and 3. Extract the feature names and label names for use later on. Note that we will be treating the *malignant* case as our *positive* case, as this is the standard convention in medicine.

Print out some information (in text) about the data, to verify that the loading has worked and to get a feeling for what is present in the dataset and the range of the values.

Also, graphically show the proportions of the labels in the whole dataset.

[2]:
```
# Your code here
data=pd.read_csv("assignment2.csv")
data.head()
```

[2]:
```
       label  mean radius  mean texture  mean perimeter   mean area  \
0  malignant    15.494654     15.902542      103.008265  776.437239
1  malignant    16.229871     18.785613      105.176755  874.712003
2  malignant    16.345671     20.114076      107.083804  872.563251
3  malignant    13.001009     19.876997       85.889775  541.281012
4  malignant    16.416060     17.397533      107.857386  891.516818

   mean smoothness  mean compactness  mean concavity  mean concave points  \
0         0.104239          0.168660        0.170572             0.085668
1         0.091843          0.092548        0.081681             0.053670
2         0.099924          0.123799        0.128788             0.078310
3         0.113423          0.173069        0.146214             0.069574
4         0.097321          0.111530        0.125971             0.068575

   mean symmetry  …  worst radius  worst texture  worst perimeter  \
0       0.205053  …     19.522957      22.427276       135.128520
1       0.180435  …     19.140235      24.905156       123.886045
```

```
2       0.189756  …     19.144816      25.601433       125.113036
3       0.212078  …     15.565911      26.145119       102.958265
4       0.179562  …     18.620376      22.306233       124.002529

    worst area  worst smoothness  worst compactness  worst concavity  \
0  1286.903131          0.142725           0.407483         0.445992
1  1234.499997          0.129135           0.223918         0.248846
2  1202.749973          0.135017           0.314402         0.332505
3   737.655082          0.161390           0.485912         0.430007
4  1139.490971          0.133950           0.230996         0.316620

   worst concave points  worst symmetry  worst fractal dimension
0              0.171662        0.353211                 0.097731
1              0.136735        0.284427                 0.085758
2              0.161497        0.313038                 0.084340
3              0.167254        0.432297                 0.117705
4              0.131715        0.269591                 0.080497

[5 rows x 31 columns]
```

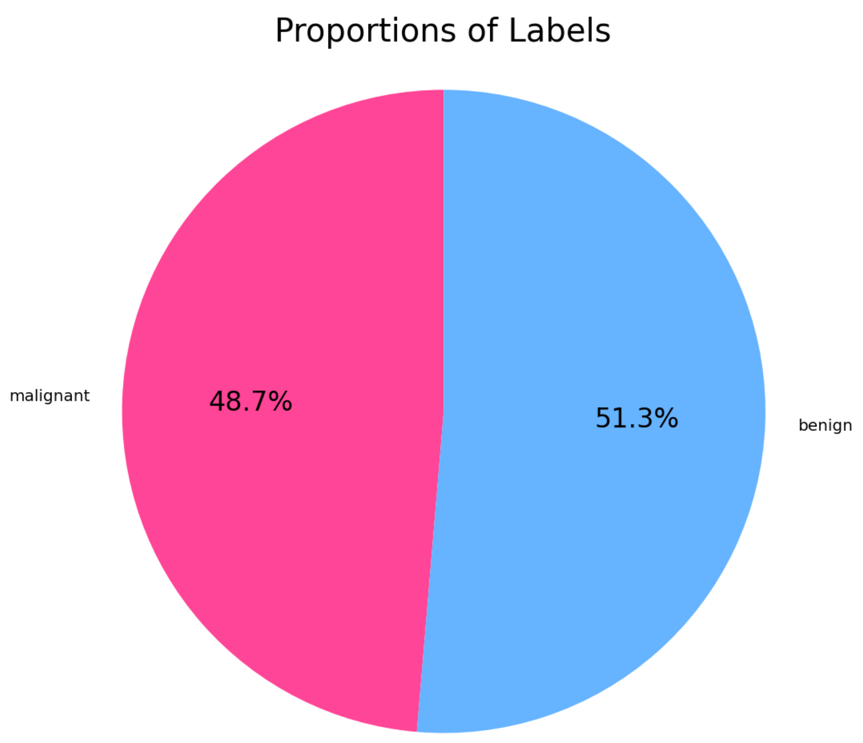
[3]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300 entries, 0 to 299
Data columns (total 31 columns):
 #   Column                   Non-Null Count  Dtype
---  ------                   --------------  -----
 0   label                    300 non-null    object
 1   mean radius              300 non-null    float64
 2   mean texture             300 non-null    float64
 3   mean perimeter           300 non-null    float64
 4   mean area                300 non-null    float64
 5   mean smoothness          300 non-null    float64
 6   mean compactness         300 non-null    float64
 7   mean concavity           300 non-null    float64
 8   mean concave points      300 non-null    float64
 9   mean symmetry            300 non-null    float64
 10  mean fractal dimension   300 non-null    float64
 11  radius error             300 non-null    float64
 12  texture error            300 non-null    float64
 13  perimeter error          300 non-null    float64
 14  area error               300 non-null    float64
 15  smoothness error         300 non-null    float64
 16  compactness error        300 non-null    float64
 17  concavity error          300 non-null    float64
 18  concave points error     300 non-null    float64
 19  symmetry error           300 non-null    float64
 20  fractal dimension error  300 non-null    float64
 21  worst radius             300 non-null    float64
 22  worst texture            300 non-null    float64
 23  worst perimeter          300 non-null    float64
 24  worst area               300 non-null    float64
 25  worst smoothness         300 non-null    float64
 26  worst compactness        300 non-null    float64
 27  worst concavity          300 non-null    float64
 28  worst concave points     300 non-null    float64
 29  worst symmetry           300 non-null    float64
 30  worst fractal dimension  300 non-null    float64
dtypes: float64(30), object(1)
memory usage: 72.8+ KB
```

[4]:
```
data.describe()
```

[4]:
```
       mean radius  mean texture  mean perimeter    mean area  \
count   300.000000    300.000000      300.000000   300.000000
mean     14.231808     19.312619       92.727687   664.367372
std       1.297393      1.572224        8.949937   129.515717
min      11.560025     15.349270       74.690886   477.371592
25%      13.356676     18.194791       86.659535   580.383274
50%      13.976933     19.220652       90.896982   628.004851
75%      15.103078     20.245660       99.093762   737.444716
max      19.090091     26.836291      126.168030  1300.788708

       mean smoothness  mean compactness  mean concavity  mean concave points  \
count       300.000000        300.000000      300.000000           300.000000
mean          0.096937          0.106615        0.092591             0.050820
std           0.005067          0.020819        0.030312             0.014350
min           0.084651          0.075184        0.050771             0.028701
25%           0.093305          0.091105        0.069071             0.039507
50%           0.096722          0.102401        0.084829             0.046744
75%           0.099995          0.117334        0.107994             0.060606
max           0.114500          0.192880        0.212704             0.105212

       mean symmetry  mean fractal dimension  …  worst radius  \
count     300.000000              300.000000  …    300.000000
mean        0.182546                0.062841  …     16.460566
std         0.010754                0.002736  …      1.798202
min         0.157059                0.057830  …     13.279265
25%         0.175353                0.060950  …     15.148044
50%         0.181685                0.062477  …     16.007171
75%         0.187789                0.064149  …     17.656889
max         0.226448                0.076091  …     22.676185

       worst texture  worst perimeter   worst area  worst smoothness  \
count     300.000000       300.000000   300.000000         300.000000
mean       25.772128       108.563914   900.644633           0.133424
std         2.346310        12.500033   209.738842           0.008678
min        20.144214        87.110184   633.771881           0.110342
25%        24.058893        99.229249   752.124790           0.127682
50%        25.689861       105.540619   828.667704           0.133064
75%        27.333610       116.274995  1011.628413           0.138650
max        34.614459       150.353232  1796.820974           0.164583

       worst compactness  worst concavity  worst concave points  \
count         300.000000       300.000000            300.000000
mean            0.261732         0.282075              0.118146
std             0.063535         0.079831              0.024552
min             0.167098         0.152272              0.066927
25%             0.215767         0.219671              0.098389
50%             0.247022         0.267894              0.115679
75%             0.298732         0.325278              0.136687
max             0.543118         0.635074              0.179794

       worst symmetry  worst fractal dimension
count      300.000000               300.000000
mean         0.293620                 0.084556
std          0.025620                 0.007427
min          0.240341                 0.072745
25%          0.277676                 0.079636
50%          0.288994                 0.082610
75%          0.305227                 0.087645
max          0.432297                 0.128288

[8 rows x 30 columns]
```

[5]:
```python
malignant=data['label'][data['label']=='malignant'].value_counts()
benign=data['label'][data['label']=='benign'].value_counts()
labels = ['malignant', 'benign']
datapro = [malignant[0], benign[0]]
colors = ['#ff4597', '#66b3ff']
plt.pie(datapro, labels=labels, colors=colors, autopct='%1.1f%%', startangle=90)
plt.title('Proportions of Labels')
plt.axis('equal')
plt.show()
```



3.1.2 1.2 Visualise the dataset [1.5 marks]

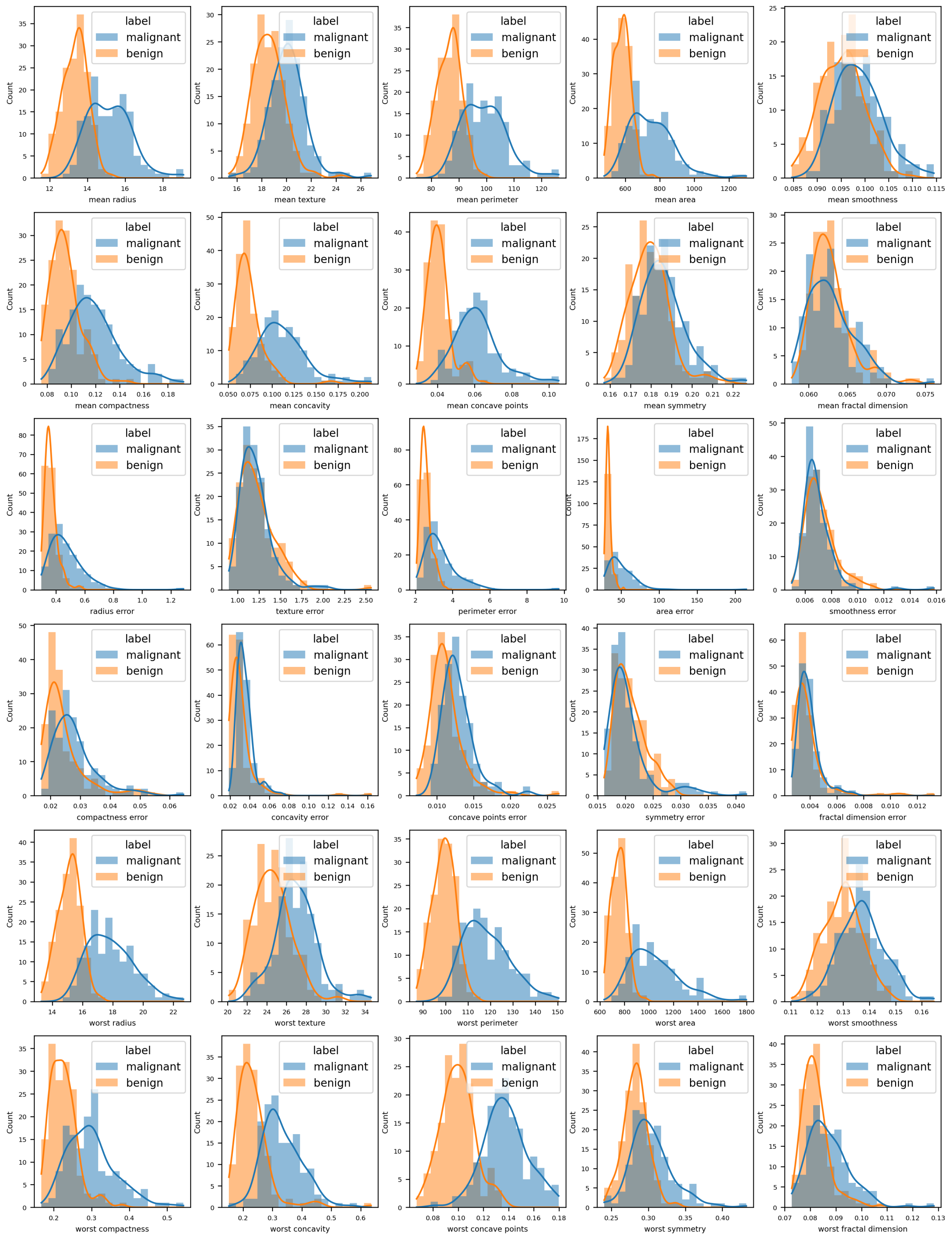
As this data is well curated by the client already, you do not need to worry about outliers, missing values or imputation in this case, but be aware that this is the exception, not the rule.

To familiarise yourself with the nature and information contained in the data, display histograms for the data according to the following instructions: - **display histograms** for each feature in the *mean* group, but on *each* histogram **have the two classes displayed together in one plot** (see example plot below and a code fragment to help you) - and note that your plot does not need to look exactly the example here; - **repeat this** for the *standard error* and *worst* groups; - make sure that in all cases you clearly label the plots and the classes in histograms.

[6]:
```
# Code fragment to help with plotting histograms combining matplotlib and␣
  ↪seaborn (and pandas)
#fig, axes = plt.subplots(Nrows, Ncols, figsize=(?, ?))
#...
#sns.histplot(data=df, x=??, hue="??", bins=??, kde=True, ax=axes[row,col],␣
  ↪edgecolor=None)
```

[6]:
```
# Your code here
Nrows = 6
Ncols = 5
fig, axes = plt.subplots(Nrows, Ncols, figsize=(15, 20))
```

```python
for i, column in enumerate(data.columns[1:]):
    row = i // Ncols
    col = i % Ncols
    sns.histplot(data=data, x=column, hue=data.iloc[:, 0], bins=20, kde=True, ax=axes[row, col], edgecolor=None)
```



3.1.3 1.3 Ranking the features [0.5 marks]

Based on the histograms, which do you think are the 3 strongest features for discriminating between the classes?

[8]:
```
# Your answer here
#Apparantly the less the common area of two lables have for each feature's␣
↪histrogram,the easier for us to discriminate the classes.
#So according to the histograms,I think the 3 strongest features should be␣
↪worst radius,worst area and worst perimeter.
```

3.1.4 1.4 Splitting the dataset [0.5 marks]

Split the dataset into appropriate subsets. You must choose what the subsets are and how big they are. However, we want to make sure the proportion of the two classes is consistent across all datasets, so use the *stratify* option, as used in workshops 5 and 6. Verify the size and label distribution in each dataset.

[7]:
```
data['label'] = data['label'].replace({'malignant': 1, 'benign': 0})
df = pd.DataFrame(data)
df
```

[7]:
```
     label  mean radius  mean texture  mean perimeter   mean area  \
0        1    15.494654     15.902542      103.008265  776.437239
1        1    16.229871     18.785613      105.176755  874.712003
2        1    16.345671     20.114076      107.083804  872.563251
3        1    13.001009     19.876997       85.889775  541.281012
4        1    16.416060     17.397533      107.857386  891.516818
..     …          …             …              …          …
295      0    14.048464     17.186671       90.974271  637.474225
296      0    12.879033     16.767790       83.123369  539.225356
297      1    13.123052     18.793057       84.897717  555.002209
298      0    14.411991     18.970674       93.423809  671.128126
299      0    12.704174     20.895143       82.227859  528.052132

     mean smoothness  mean compactness  mean concavity  mean concave points  \
0           0.104239          0.168660        0.170572             0.085668
1           0.091843          0.092548        0.081681             0.053670
2           0.099924          0.123799        0.128788             0.078310
3           0.113423          0.173069        0.146214             0.069574
4           0.097321          0.111530        0.125971             0.068575
..               …                 …               …                    …
295         0.094969          0.091549        0.063532             0.039494
296         0.092146          0.083986        0.059347             0.035404
297         0.098036          0.090178        0.066586             0.043711
```

```
298          0.086304          0.090118        0.070882              0.039482
299          0.098300          0.093698        0.068184              0.038141

     mean symmetry  …  worst radius  worst texture  worst perimeter  \
0         0.205053  …     19.522957      22.427276       135.128520
1         0.180435  …     19.140235      24.905156       123.886045
2         0.189756  …     19.144816      25.601433       125.113036
3         0.212078  …     15.565911      26.145119       102.958265
4         0.179562  …     18.620376      22.306233       124.002529
..             …   …           …             …               …
295       0.173324  …     15.790651      22.538529       103.423320
296       0.167690  …     14.358919      21.955513        93.620160
297       0.172389  …     14.991646      24.820718        97.933068
298       0.175789  …     16.555187      25.591332       108.978466
299       0.178533  …     14.199113      25.377961        93.143286

      worst area  worst smoothness  worst compactness  worst concavity  \
0    1286.903131          0.142725           0.407483         0.445992
1    1234.499997          0.129135           0.223918         0.248846
2    1202.749973          0.135017           0.314402         0.332505
3     737.655082          0.161390           0.485912         0.430007
4    1139.490971          0.133950           0.230996         0.316620
..           …               …                 …               …
295   819.408970          0.126466           0.206701         0.192139
296   684.694077          0.118165           0.191978         0.180949
297   726.695117          0.126203           0.201766         0.202433
298   893.818250          0.120338           0.246945         0.236415
299   681.453918          0.125313           0.195607         0.192059

     worst concave points  worst symmetry  worst fractal dimension
0                0.171662        0.353211                 0.097731
1                0.136735        0.284427                 0.085758
2                0.161497        0.313038                 0.084340
3                0.167254        0.432297                 0.117705
4                0.131715        0.269591                 0.080497
..                    …             …                       …
295              0.095350        0.287380                 0.078520
296              0.083989        0.263879                 0.078279
297              0.100361        0.256863                 0.079667
298              0.105354        0.280900                 0.081828
299              0.085053        0.265963                 0.078269

[300 rows x 31 columns]
```

[8]:
```python
# Your code here
from sklearn.model_selection import train_test_split
```

```
bigtrain_set, test_set = train_test_split(data, test_size=0.2, random_state=20,␣
 ↪stratify=data['label'])
train_set, val_set = train_test_split(bigtrain_set, test_size=0.25,␣
 ↪random_state=20, stratify=bigtrain_set['label'])
```

[9]:
```
X_train = train_set.iloc[:, 1:]
y_train = train_set.iloc[:, 0]
X_test = test_set.iloc[:, 1:]
y_test = test_set.iloc[:, 0]
X_val = val_set.iloc[:, 1:]
y_val = val_set.iloc[:, 0]
print(f'Shapes are {[X_train.shape,y_train.shape,X_test.shape,y_test.
 ↪shape,X_val.shape,y_val.shape]}')
```

```
Shapes are [(180, 30), (180,), (60, 30), (60,), (60, 30), (60,)]
```

3.2 2. Build, Train and Optimise Classifiers (60% = 18 marks)

3.2.1 2.1 Pipeline [0.5 marks]

Build a pre-processing pipeline that includes imputation (as even though we don't strictly need it here it is a good habit to always include it) and other appropriate pre-processing.

[10]:
```
# Your code here
from sklearn.impute import SimpleImputer
preproc_pl = Pipeline([ ('imputer', SimpleImputer(strategy="median")),
                        ('std_scaler', StandardScaler()) ])
```

3.2.2 2.2 Baseline measurements [1.5 marks]

For our classification task we will consider **three simple baseline cases**: 1) predicting all samples to be negative (class 1) 2) predicting all samples to be positive (class 2) 3) making a random prediction for each sample with equal probability for each class

For each case measure and display the following metrics: - balanced accuracy - recall - precision - auc - f1score - fbeta_score with beta=0.1 - fbeta_score with beta=10

Code is given below for the latter metrics (all metrics are discussed in lecture 4 and many are in workshop 4).

Also **calculate and display the confusion matrix** for each baseline case, using a heatmap and numbers (as in workshop 4).

[11]:
```
from sklearn.metrics import fbeta_score, make_scorer

f10_scorer = make_scorer(fbeta_score, beta=10)
f01_scorer = make_scorer(fbeta_score, beta=0.1)

def f10_score(yt,yp):
    return fbeta_score(yt, yp, beta=10)
```

```python
def f01_score(yt,yp):
    return fbeta_score(yt, yp, beta=0.1)
```

[12]:
```python
# Your code here
baseline1 = np.full(y_train.shape,0)
baseline2 = np.full(y_train.shape,1)
baseline3 = np.random.choice([0, 1], size=y_train.shape, p=[0.5, 0.5])
print(baseline1)
print(baseline2)
print(baseline3)
```

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]
[1 0 1 0 0 0 0 1 1 1 1 1 0 1 1 1 0 0 1 1 1 0 1 1 1 0 1 1 0 0 1 0 1 0 1 0 0
 0 0 1 1 0 0 0 1 1 0 0 1 1 1 1 1 0 0 1 0 1 0 0 1 0 1 0 0 0 1 1 0 1 1 1 0 0
 0 0 1 1 1 0 1 1 1 0 0 0 1 0 0 0 1 0 0 1 0 0 0 1 1 0 0 0 0 0 0 1 1 1 1 0 0
 1 0 1 0 0 0 1 0 1 1 1 0 1 0 0 1 0 1 0 0 0 1 0 0 1 1 1 1 0 1 0 0 1 0 1 0 0
 1 0 1 0 1 0 1 0 0 0 1 1 0 1 0 1 0 1 0 1 0 0 1 0 1 1 0 1 1 1 1 1]
```

[13]:
```python
#Accuracy
from sklearn.metrics import accuracy_score
accuracy1 = accuracy_score(y_train, baseline1)
accuracy2 = accuracy_score(y_train, baseline2)
accuracy3 = accuracy_score(y_train, baseline3)
print("Accuracy for baseline 1:", accuracy1)
print("Accuracy for baseline 2:", accuracy2)
print("Accuracy for baseline 3:", accuracy3)
```

```
Accuracy for baseline 1: 0.5111111111111111
Accuracy for baseline 2: 0.4888888888888889
Accuracy for baseline 3: 0.55
```

[14]:
```python
#recall
from sklearn.metrics import precision_score, recall_score
recall1 = recall_score(y_train, baseline1)
recall2 = recall_score(y_train, baseline2)
recall3 = recall_score(y_train, baseline3)
print("Recall for baseline 1:", recall1)
print("Recall for baseline 2:", recall2)
print("Recall for baseline 3:", recall3)
```

```
Recall for baseline 1: 0.0
Recall for baseline 2: 1.0
Recall for baseline 3: 0.5454545454545454
```

[15]:
```
#precision
precision1 = precision_score(y_train, baseline1)
precision2 = precision_score(y_train, baseline2)
precision3 = precision_score(y_train, baseline3)
print("Precision for baseline 1:", precision1)
print("Precision for baseline 2:", precision2)
print("Precision for baseline 3:", precision3)
```

```
Precision for baseline 1: 0.0
Precision for baseline 2: 0.4888888888888889
Precision for baseline 3: 0.5393258426966292
```

```
C:\Users\Acer\AppData\Roaming\Python\Python39\site-
packages\sklearn\metrics\_classification.py:1469: UndefinedMetricWarning:
Precision is ill-defined and being set to 0.0 due to no predicted samples. Use
`zero_division` parameter to control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

[16]:
```
#AUC
from sklearn.metrics import roc_auc_score
auc1 = roc_auc_score(y_train, baseline1)
auc2 = roc_auc_score(y_train, baseline2)
auc3 = roc_auc_score(y_train, baseline3)
print("AUC for baseline 1:",auc1)
print("AUC for baseline 2:",auc2)
print("AUC for baseline 3:",auc3)
```

```
AUC for baseline 1: 0.5
AUC for baseline 2: 0.5
AUC for baseline 3: 0.549901185770751
```

[17]:
```
#f1score
from sklearn.metrics import f1_score
f11=f1_score(y_train, baseline1)
f12=f1_score(y_train, baseline2)
f13=f1_score(y_train, baseline3)
print("f1score for baseline 1:",f11)
print("f1score for baseline 2:",f12)
print("f1score for baseline 3:",f13)
```

```
f1score for baseline 1: 0.0
```

```
f1score for baseline 2: 0.6567164179104478
f1score for baseline 3: 0.5423728813559321
```

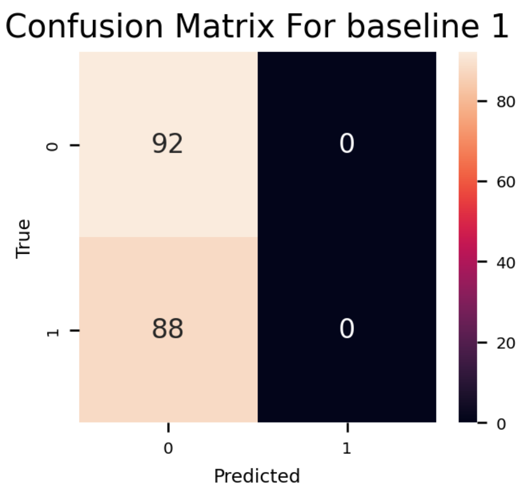
[18]:
```
#fbeta_score with beta=0.1
f01_1=f01_score(y_train, baseline1)
f01_2=f01_score(y_train, baseline2)
f01_3=f01_score(y_train, baseline3)
print("fbeta_score with beta=0.1 for baseline 1:",f01_1)
print("fbeta_score with beta=0.1 for baseline 2:",f01_2)
print("fbeta_score with beta=0.1 for baseline 3:",f01_3)
```

```
fbeta_score with beta=0.1 for baseline 1: 0.0
fbeta_score with beta=0.1 for baseline 2: 0.49137549756744797
fbeta_score with beta=0.1 for baseline 3: 0.5393858477970628
```

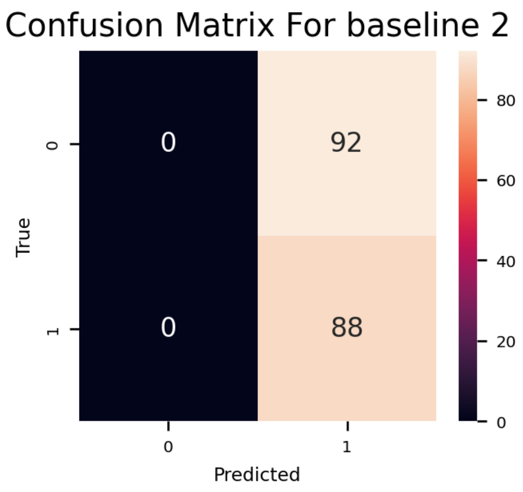
[19]:
```
#fbeta_score with beta=10
f10_1=f10_score(y_train, baseline1)
f10_2=f10_score(y_train, baseline2)
f10_3=f10_score(y_train, baseline3)
print("fbeta_score with beta=10 for baseline 1:",f10_1)
print("fbeta_score with beta=10 for baseline 2:",f10_2)
print("fbeta_score with beta=10 for baseline 3:",f10_3)
```

```
fbeta_score with beta=10 for baseline 1: 0.0
fbeta_score with beta=10 for baseline 2: 0.9897550111358574
fbeta_score with beta=10 for baseline 3: 0.5453931825852176
```

[20]:
```
#confusion matrix for baseline1
cmat = confusion_matrix(y_train, baseline1)
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix For baseline 1')
plt.show()
```

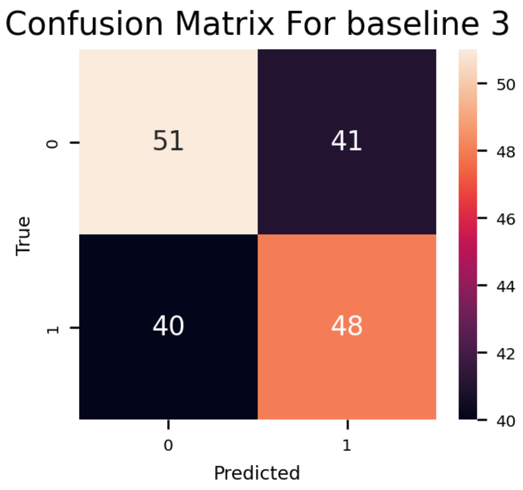


[21]:
```
#confusion matrix for baseline2
cmat = confusion_matrix(y_train, baseline2)
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix For baseline 2')
plt.show()
```



[22]:
```
#confusion matrix for baseline3
cmat = confusion_matrix(y_train, baseline3)
plt.figure(figsize=(3,2.5))
```

```
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Confusion Matrix For baseline 3')
plt.show()
```



3.2.3 2.3 Choose a performance metric [0.5 marks]

Based on the above baseline tests and the client's requirements, **choose a performance metric** to use for evaluating/driving your machine learning methods. **Give a reason for your choice.**

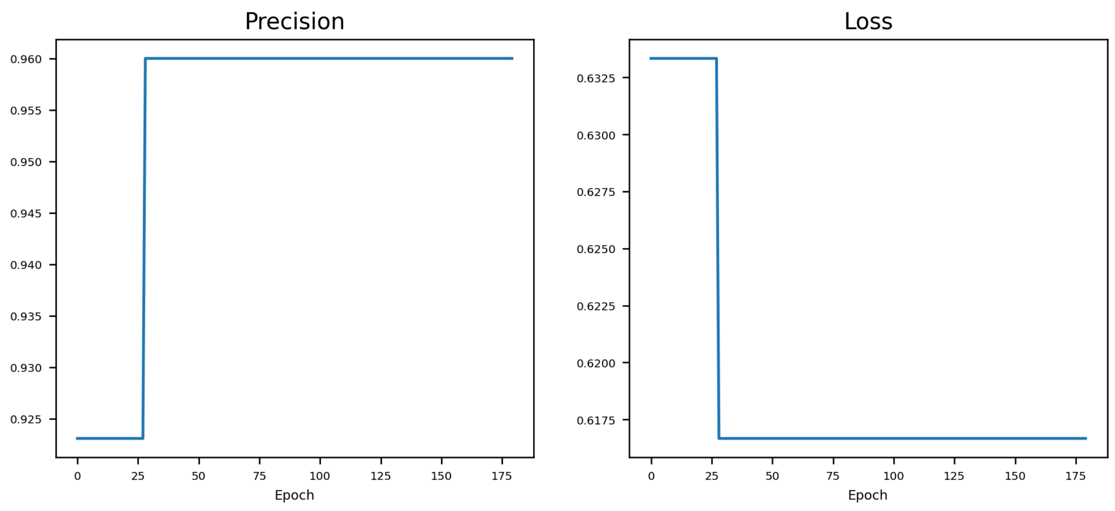
[23]:
```
# Your answer here
#Accoring to the client's requirements,we need to pay more attention to␣
↪detecting malignant cancer (>=0.95) than detecting negative
#samples(>=0.9) correctly,and apprantly finding all positive samples is pretty␣
↪important if we can meet the demand of 0.9 rate
#detecting negative samples(>=0.9) correctly.I will choose precision rate as my␣
↪performance metric.
```

3.2.4 2.4 SGD baseline [1 mark]

For a stronger baseline, **train and evaluate** the Stochastic Gradient Descent classifier (as seen in workshop 5). For this baseline case use the default settings for all the hyperparameters.

[24]:
```
# Your code here
from sklearn.linear_model import SGDClassifier
from sklearn.metrics import hinge_loss
def sgdfn(nsamp, learnrate, loop=True, learntype='constant',
          tol=1e-3, early_stopping=False, verbose=0, n_iter_no_change=5,␣
↪in_penalty=None, in_alpha=0.0001, in_figure=True):
    sgd = SGDClassifier(warm_start=True,
                        learning_rate=learntype,
                        eta0=learnrate,
                        early_stopping=early_stopping,
                        tol=tol,
                        verbose=verbose,
                        n_iter_no_change=n_iter_no_change,
                        penalty=in_penalty,
                        alpha=in_alpha)
    X_trainp = preproc_pl.fit_transform(X_train)
    X_valp = preproc_pl.transform(X_val)
    res=[]
    if loop:
        for n in range(nsamp):
            if early_stopping==False:
                sgd.partial_fit(X_trainp, y_train, classes=[0,1])
            else:
                sgd.fit(X_trainp, y_train)
            y_val_pred = sgd.predict(X_valp)
            sgd_acc = precision_score(y_val, y_val_pred)
            sgd_loss = hinge_loss(y_val, y_val_pred)
            res += [[sgd_acc, sgd_loss]]
    else:
        sgd.fit(X_trainp, y_train)
        y_val_pred = sgd.predict(X_valp)
        sgd_acc = precision_score(y_val, y_val_pred)
        sgd_loss = hinge_loss(y_val, y_val_pred)
        res += [[sgd_acc, sgd_loss]]

    res = np.array(res)

    if loop and in_figure:
        plt.figure(figsize=(10,4))
        plt.subplot(121)
        plt.plot(res[:,0])
        plt.title('Precision')
        plt.xlabel('Epoch')
        plt.subplot(122)
        plt.plot(res[:,1])
        plt.title('Loss')
        plt.xlabel('Epoch')
        plt.show()

    print('Precision:', res[-1,0], ', Loss: ', res[-1,1])
    return [res[-1,0], res[-1,1]],y_val_pred
```

[25]:
```
ntrain = X_train.shape[0]
res,y_val_pred=sgdfn(ntrain,0.00001)
```

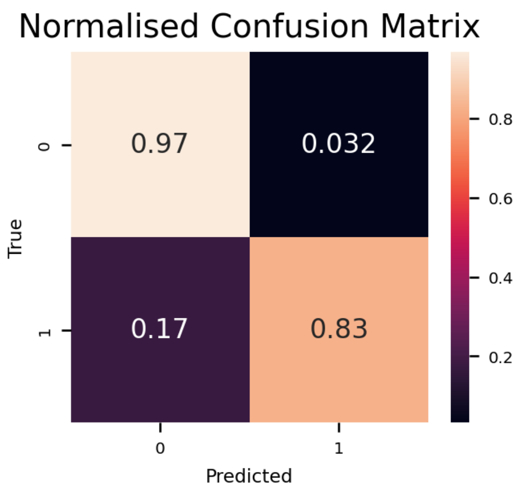


```
Precision: 0.96 , Loss:  0.6166666666666667
```

3.2.5 2.5 Confusion matrix [1 mark]

Calculate and display the normalized version of the confusion matrix. From this **calculate the *probability* that a sample from a person with a malignant tumour is given a result that they do not have cancer. Which of the client's two criteria does this relate to, and is this baseline satisfying this criterion or not?**

[26]:
```
# Your code here
cmat = confusion_matrix(y_val,y_val_pred, normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
```



[29]:
```
# Your answer here
#P=FN/(FN+TP)=0,this is related to requirment 1:which means the method has at␣
↪100% probability of detecting
#malignant cancer when it is present and it meet the demand of our client of␣
↪95% probablity.
```

3.2.6 2.6 Main classifier [11 marks]

Train and optimise the hyperparameters to give the best performance for **each of the following classifiers**: - KNN (K-Nearest Neighbour) classifier - Decision tree classifier - Support vector machine classifier - SGD classifier

Follow best practice as much as possible here. You must make all the choices and decisions yourself, and strike a balance between computation time and performance.

You can use any of the sci-kit learn functions in sklearn.model_selection.cross* and anything used in workshops 3, 4, 5 and 6. Other hyper-parameter optimisation functions apart from these cannot be used (even if they are good and can be part of best practice in other situations - for this assignment everyone should assume they only have very limited computation resources and limit themselves to these functions).

Display the performance of the different classifiers and the optimised hyperparameters.

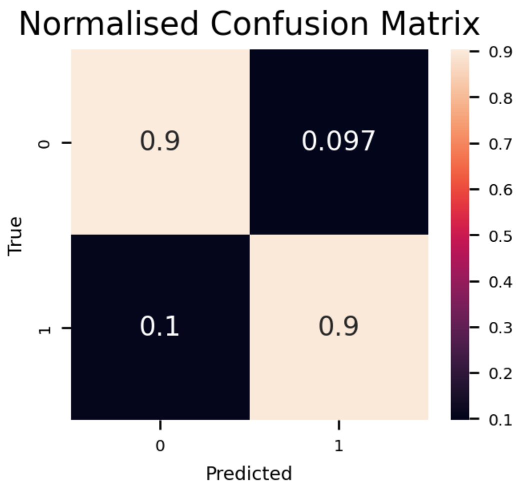
Based on these results, list the best 3 classifiers and indicate if you think any perform equivalently.

[27]:
```
#KNN
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
X_trainp = preproc_pl.fit_transform(X_train)
```

```
X_valp = preproc_pl.transform(X_val)
param_grid = {'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15]}
knn = KNeighborsClassifier()
grid_search = GridSearchCV(estimator=knn, param_grid=param_grid,␣
↪scoring='precision', cv=10)
grid_search.fit(X_trainp, y_train)
print("Best parameters:", grid_search.best_params_)
print("Best precision score:", grid_search.best_score_)
```

```
Best parameters: {'n_neighbors': 13}
Best precision score: 0.9888888888888889
```

[28]:
```
knn = KNeighborsClassifier(n_neighbors=13)
knn.fit(X_trainp,y_train)
pred_knn = knn.predict(X_valp)
#Confusion matrix and precision
cmat = confusion_matrix(y_val,pred_knn, normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
precisionm1 = precision_score(y_val,pred_knn)
print("precision for model 1:", precisionm1)
```



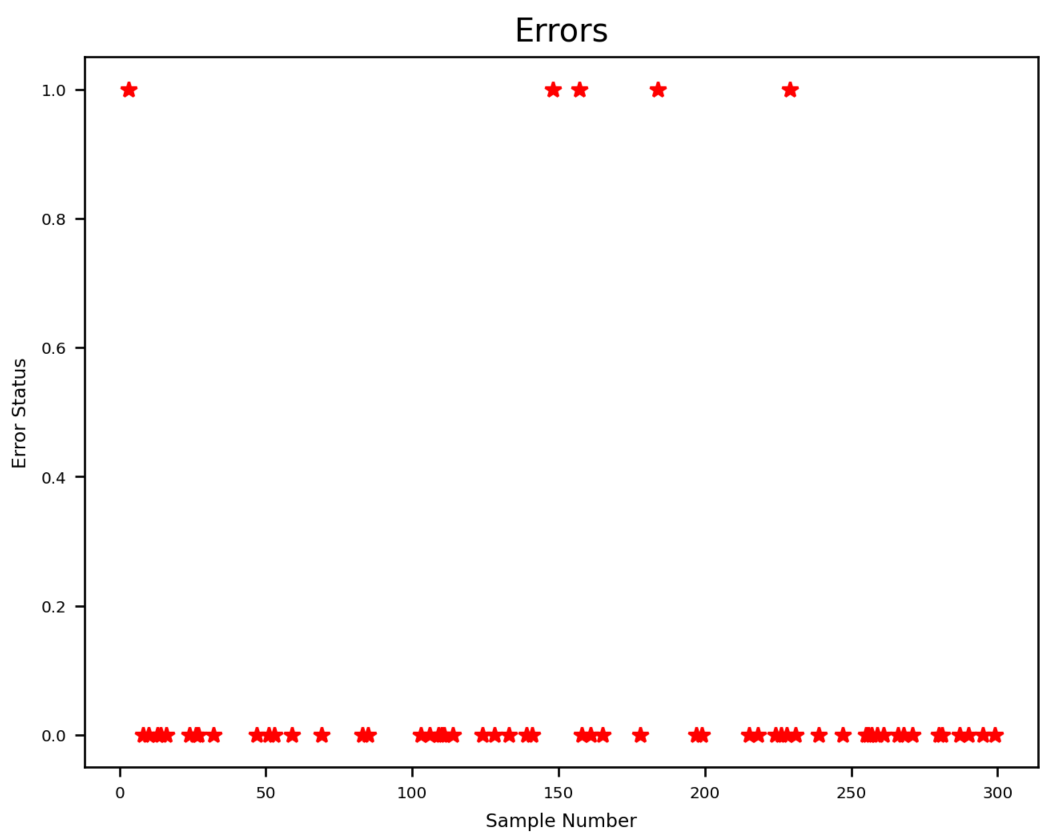
```
precision for model 1: 0.896551724137931
```

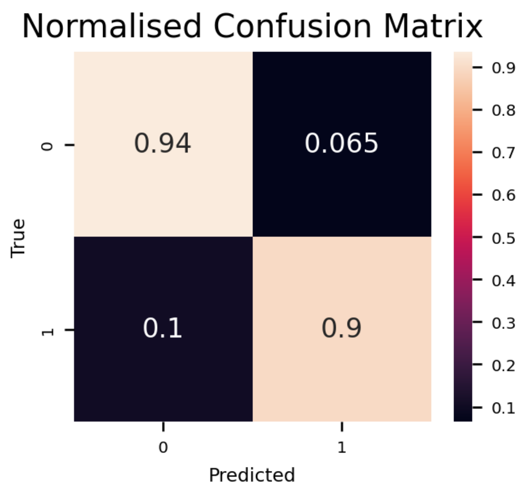
[36]:
```
#Decision tree
from sklearn.tree import DecisionTreeClassifier, plot_tree
```

```
dt = DecisionTreeClassifier()
param_grid = {
    'max_depth': [3, 5, 7],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 3],
    'max_features': [10, 5, 20],
}
grid_search = GridSearchCV(dt, param_grid, scoring='precision', cv=10)
grid_search.fit(X_trainp, y_train)
print("Best parameters:", grid_search.best_params_)
print("Best precision score:", grid_search.best_score_)
#This result is unstable,I find the best match through combing grid search and␣
↪trying by myself
```

```
Best parameters: {'max_depth': 5, 'max_features': 10, 'min_samples_leaf': 1,
'min_samples_split': 5}
Best precision score: 0.968888888888889
```

[37]:
```
dt_pl =␣
↪DecisionTreeClassifier(max_depth=5,min_samples_split=5,min_samples_leaf=2,max_features=20,␣
↪random_state=50)
dt_pl.fit(X_trainp,y_train)
y_val_pred_tree = dt_pl.predict(X_valp)
y_val_prob_tree = dt_pl.predict_proba(X_valp)
plt.plot(np.abs(y_val - y_val_pred_tree),'r*')
plt.xlabel('Sample Number')
plt.ylabel('Error Status')
plt.title('Errors')
plt.show()
#Confusion matrix and precision
cmat = confusion_matrix(y_val,y_val_pred_tree, normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
precisionm2 = precision_score(y_val,y_val_pred_tree)
print("precision for model 2:", precisionm2)
```



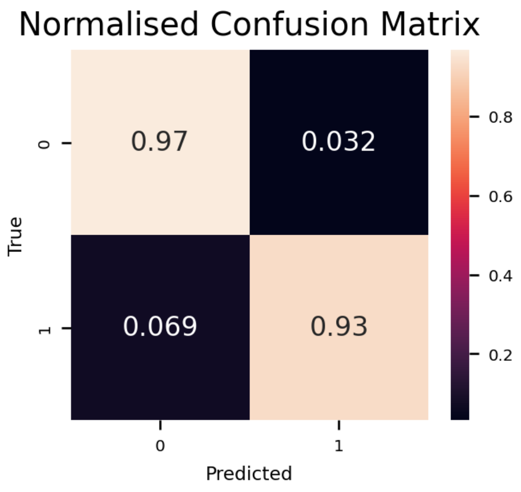


```
precision for model 2: 0.9285714285714286
```

[38]:
```
#SVM method
from sklearn.svm import SVC
parameters = {'C': [0.1, 1, 10, 100]}
svm = SVC(kernel='linear')
grid_search = GridSearchCV(svm, parameters, scoring='precision', cv=10)
grid_search.fit(X_trainp, y_train)
print("Best Parameters: ", grid_search.best_params_)
print("Best Precision: ", grid_search.best_score_)
```

```
Best Parameters:  {'C': 1}
Best Precision:  0.97
```

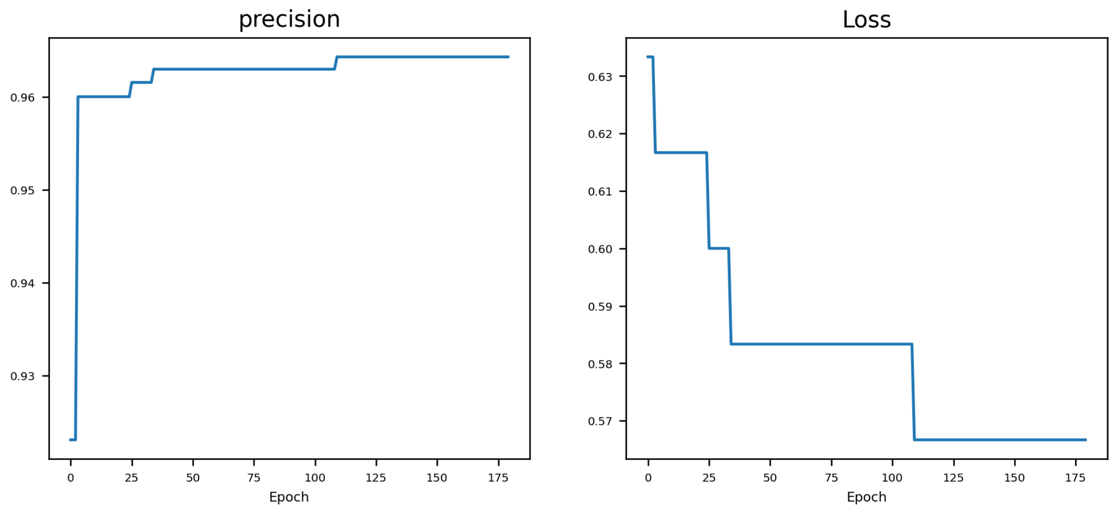
[39]:
```
svm_lin =SVC(kernel='linear',C=1)
svm_lin.fit(X_trainp, y_train)
y_val_predSVM = svm_lin.predict(X_valp)
#Confusion matrix and precision
cmat = confusion_matrix(y_val,y_val_predSVM, normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
precisionm3 = precision_score(y_val,y_val_predSVM)
print("precision for model 3:", precisionm3)
```



```
precision for model 3: 0.9642857142857143
```

[40]:
```
# SGD
def sgdfn(nsamp, learnrate, loop=True, learntype='constant',
        tol=1e-3, early_stopping=False, verbose=0, n_iter_no_change=5,␣
↪in_penalty=None, in_alpha=0.0001, in_figure=True):
    sgd = SGDClassifier(warm_start=True,
                        learning_rate=learntype,
                        eta0=learnrate,
                        early_stopping=early_stopping,
                        tol=tol,
                        verbose=verbose,
                        n_iter_no_change=n_iter_no_change,
                        penalty=in_penalty,
                        alpha=in_alpha)
    X_trainp = preproc_pl.fit_transform(X_train)
    X_valp = preproc_pl.transform(X_val)
    res=[]
    if loop:
        for n in range(nsamp):
            if early_stopping==False:
                sgd.partial_fit(X_trainp, y_train, classes=[0,1])
            else:
                sgd.fit(X_trainp, y_train)
            y_val_pred = sgd.predict(X_valp)
            sgd_acc = precision_score(y_val, y_val_pred)
            sgd_loss = hinge_loss(y_val, y_val_pred)
            res += [[sgd_acc, sgd_loss]]
    else:
        sgd.fit(X_trainp, y_train)
        y_val_pred = sgd.predict(X_valp)
        sgd_acc = precision_score(y_val, y_val_pred)
        sgd_loss = hinge_loss(y_val, y_val_pred)
        res += [[sgd_acc, sgd_loss]]

    res = np.array(res)

    if loop and in_figure:
        plt.figure(figsize=(10,4))
        plt.subplot(121)
        plt.plot(res[:,0])
        plt.title('precision')
        plt.xlabel('Epoch')
        plt.subplot(122)
        plt.plot(res[:,1])
        plt.title('Loss')
        plt.xlabel('Epoch')
        plt.show()

    print('Precision:', res[-1,0], ', Loss: ', res[-1,1])
    return [res[-1,0], res[-1,1]],y_val_pred
```

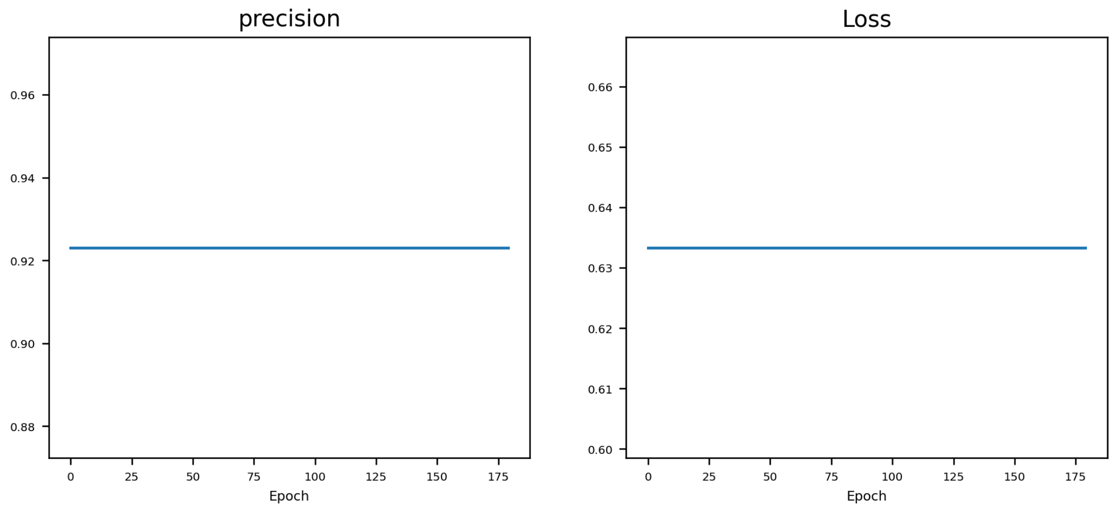
```
ntrain = X_train.shape[0]
res,y_val_pred=sgdfn(ntrain,0.00008)
```



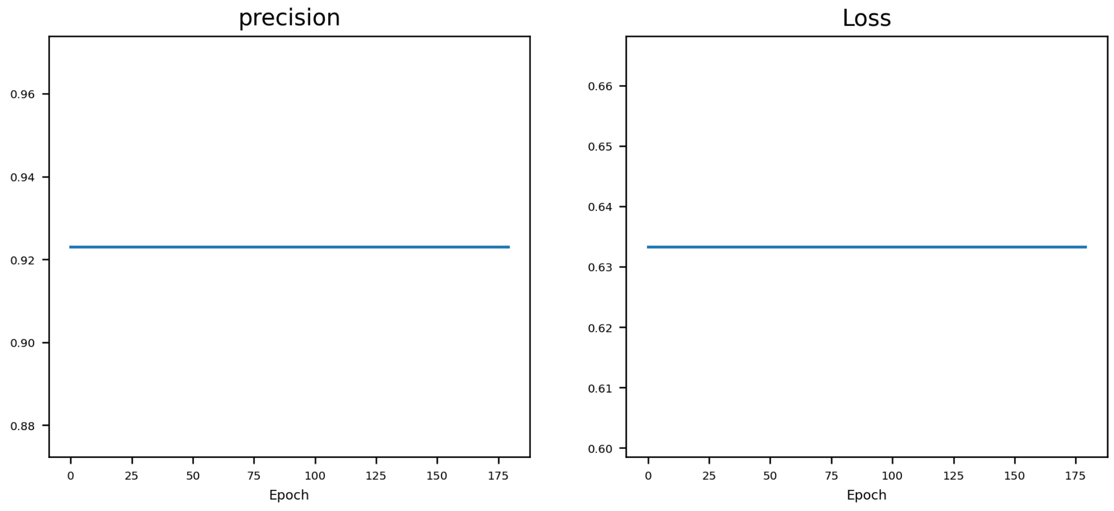
```
Precision: 0.9642857142857143 , Loss:  0.5666666666666667
```

[41]:
```
res=[]
for lr_exp in range(-8,+3):
    lr = 10.0**lr_exp
    print(f'Learning rate = {lr}')
    res += sgdfn(ntrain,lr)
print(res)
```

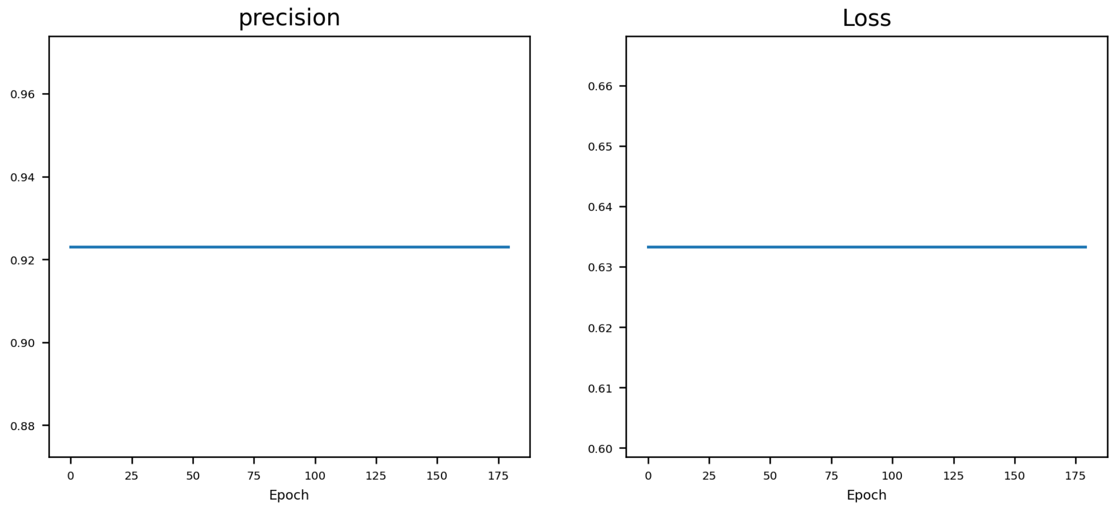
```
Learning rate = 1e-08
```



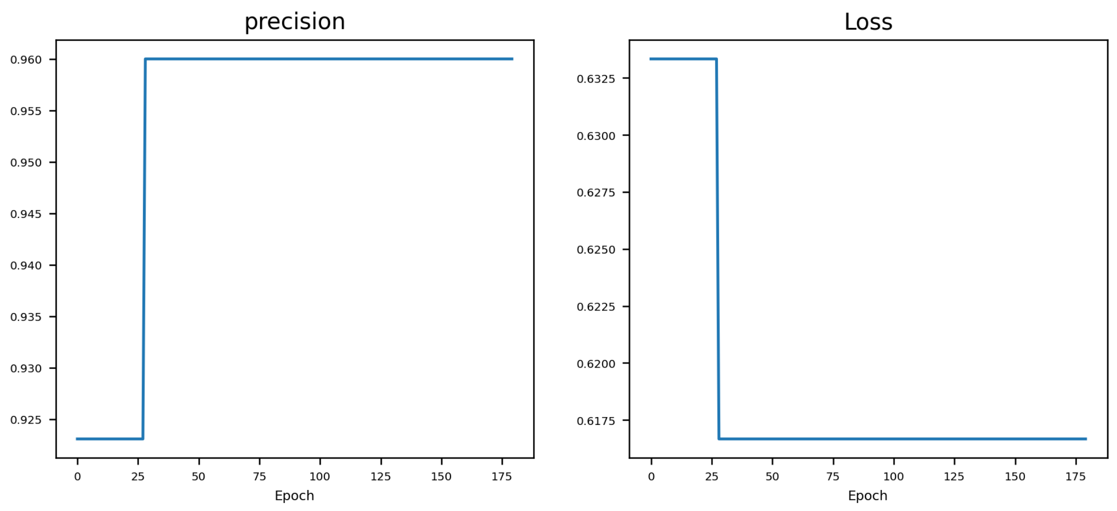
```
Precision: 0.9230769230769231 , Loss:  0.6333333333333333
Learning rate = 1e-07
```



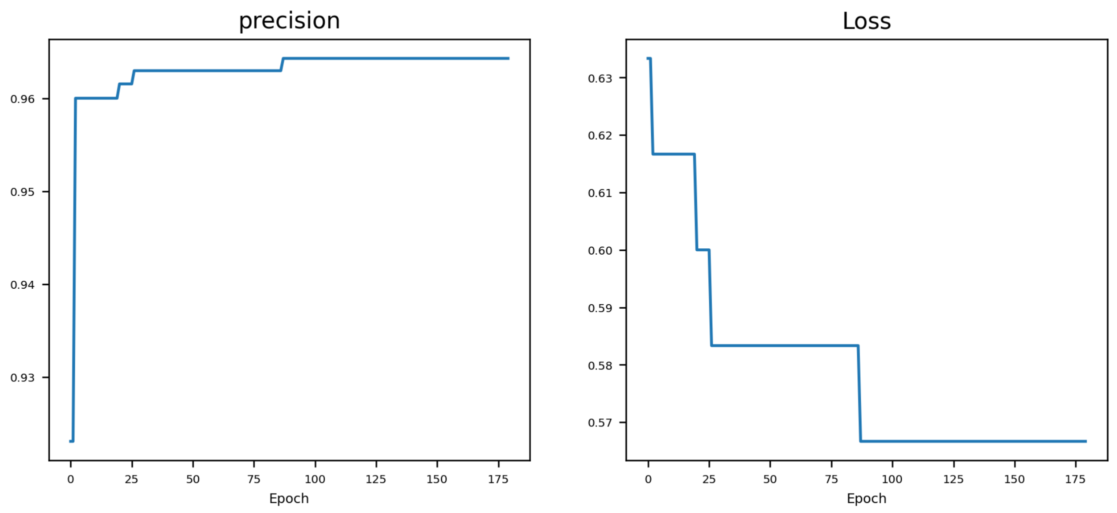
```
Precision: 0.9230769230769231 , Loss:  0.6333333333333333
Learning rate = 1e-06
```



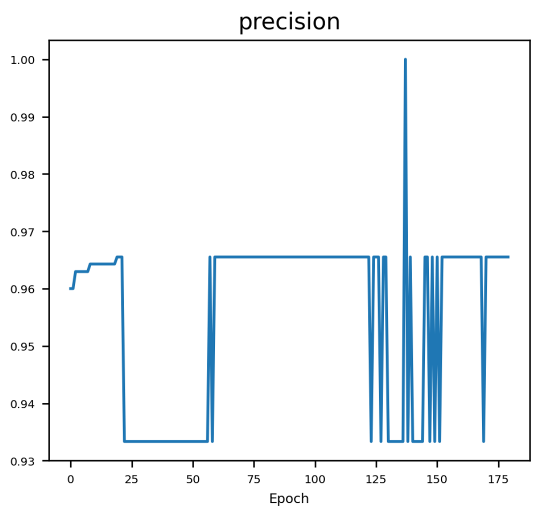
```
Precision: 0.9230769230769231 , Loss:  0.6333333333333333
Learning rate = 1e-05
```

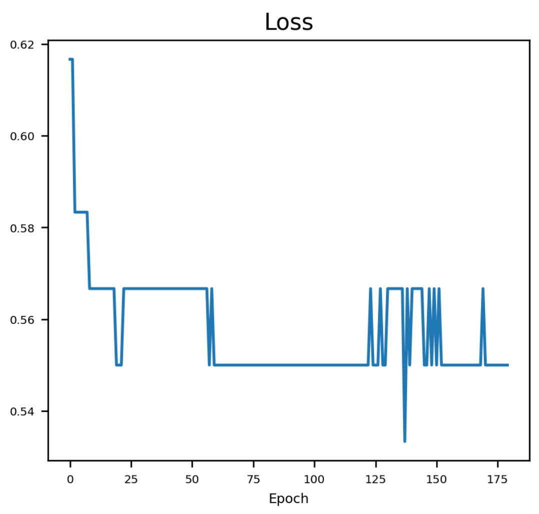


```
Precision: 0.96 , Loss:  0.6166666666666667
Learning rate = 0.0001
```

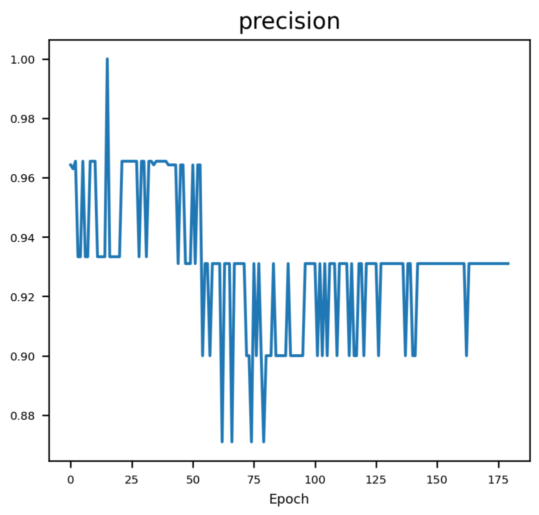


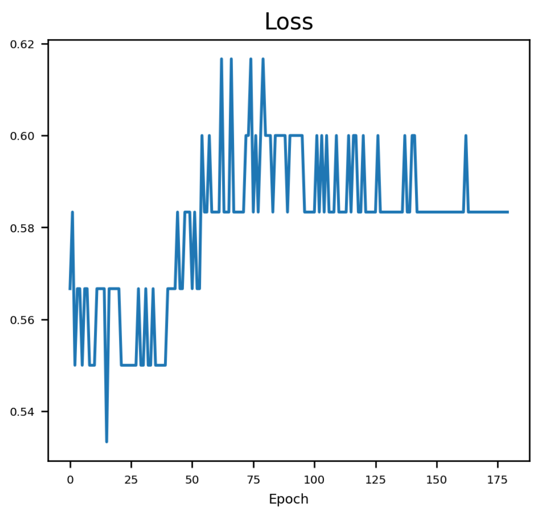
```
Precision: 0.9642857142857143 , Loss:  0.5666666666666667
Learning rate = 0.001
```



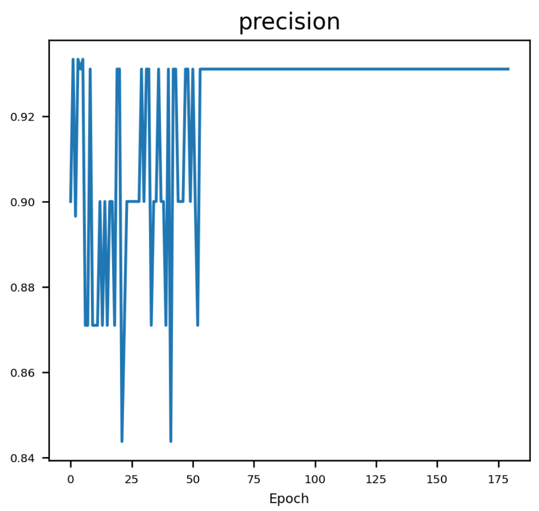


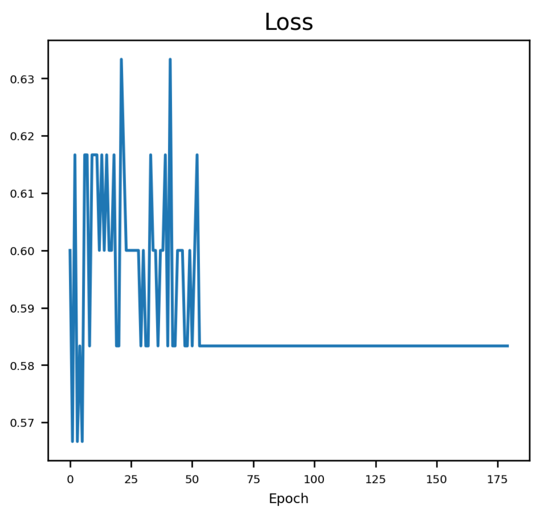
```
Precision: 0.9655172413793104 , Loss:  0.55
Learning rate = 0.01
```



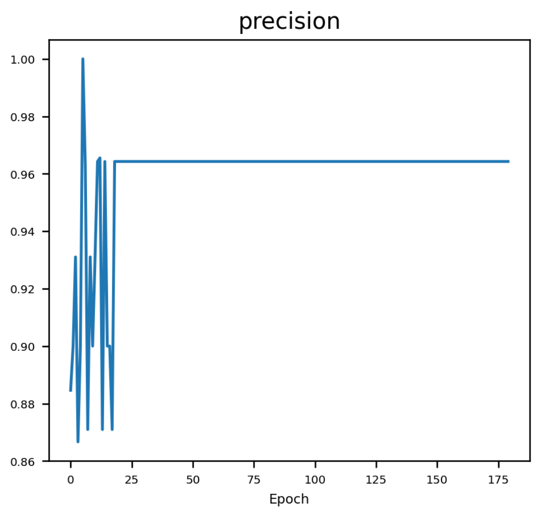


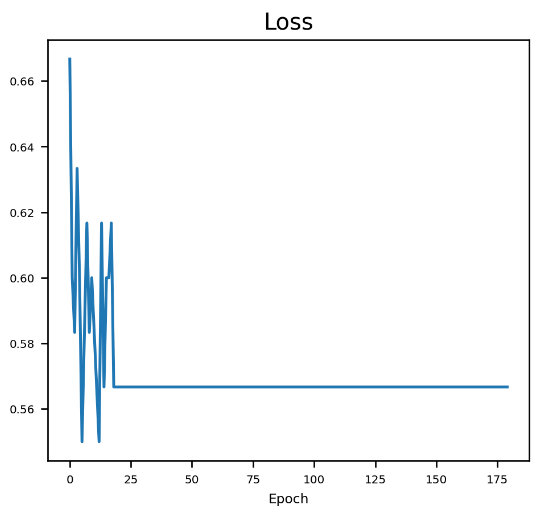
```
Precision: 0.9310344827586207 , Loss:  0.5833333333333334
Learning rate = 0.1
```



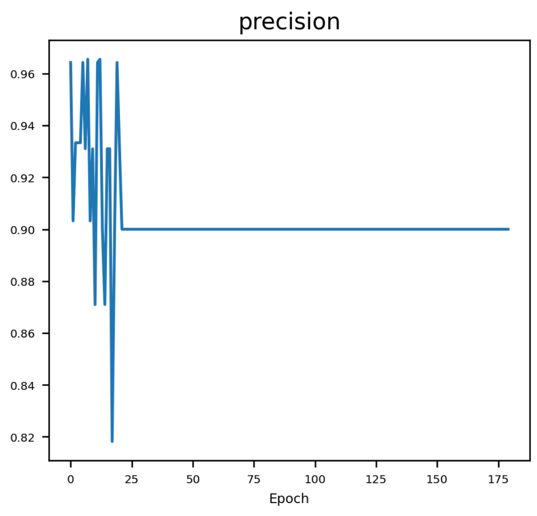


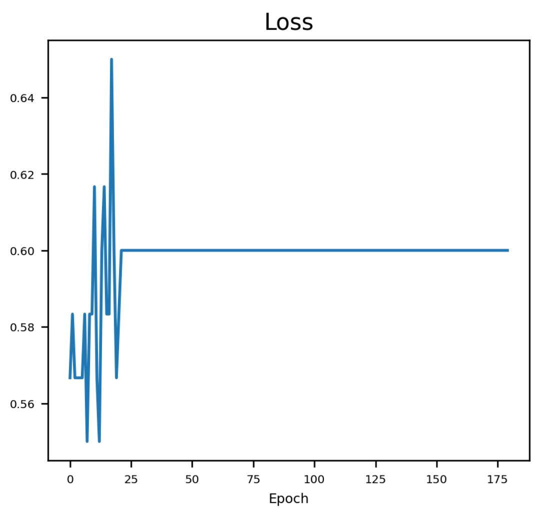
```
Precision: 0.9310344827586207 , Loss:  0.5833333333333334
Learning rate = 1.0
```



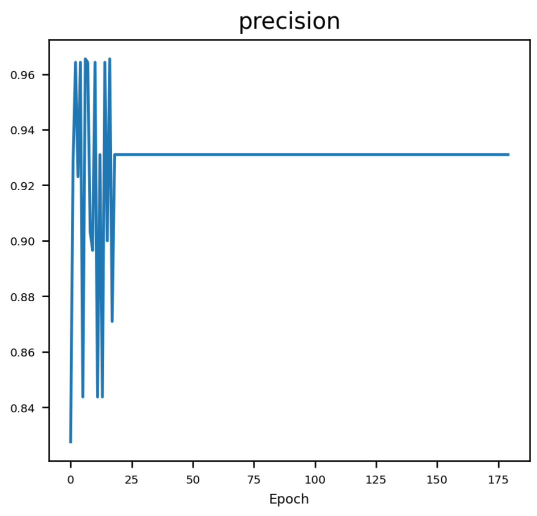


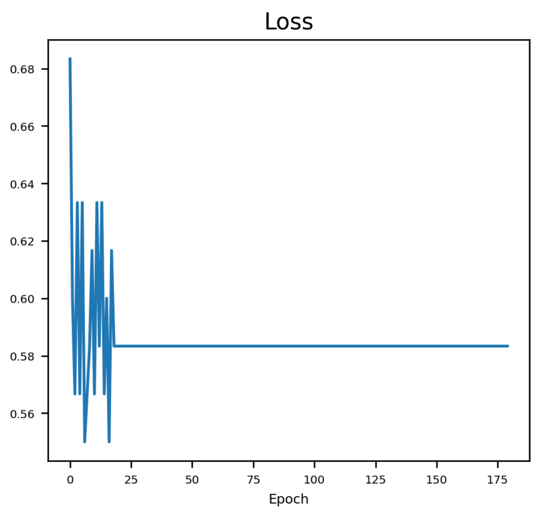
```
Precision: 0.9642857142857143 , Loss:  0.5666666666666667
Learning rate = 10.0
```





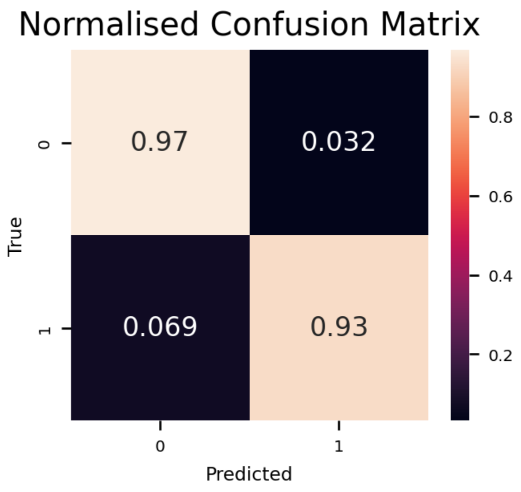
```
Precision: 0.9 , Loss:  0.6
Learning rate = 100.0
```





```
Precision: 0.9310344827586207 , Loss:  0.5833333333333334
[[0.9230769230769231, 0.6333333333333333], array([1, 0, 0, 0, 0, 1, 0, 1, 1, 0,
1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]), [0.9230769230769231,
0.6333333333333333], array([1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]), [0.9230769230769231,
0.6333333333333333], array([1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1]), [0.96,
0.6166666666666667], array([1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0]), [0.9642857142857143,
0.5666666666666667], array([1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0]), [0.9655172413793104,
0.55], array([1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0]), [0.9310344827586207,
0.5833333333333334], array([0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0]), [0.9310344827586207,
0.5833333333333334], array([0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0]), [0.9642857142857143,
0.5666666666666667], array([0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0]), [0.9, 0.6], array([0,
1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
       0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0]), [0.9310344827586207,
0.5833333333333334], array([0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
1, 1, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0])]
```

[38]:
```
cmat = confusion_matrix(y_val,y_val_pred, normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
precisionm4 = precision_score(y_val,y_val_pred)
print("precision for model 4:", precisionm4)
```



```
precision for model 4: 0.9642857142857143
```

[39]:
```
# Your answer here
#Accoring to the result,the 3 best models are SGD method,decision tree and SVM␣
↪method.Here I use precision rate and specificity
#to measure the performance and to determin if this model meet the demand.
↪According to the validation test,SVM and SGD models
#meet the criteria and behave equally on validation set,while decision tree␣
↪behave worse than to methods.
```

3.2.7 2.7 Model selection [1 mark]

Choose the best classifier (as seen in workshops 3 to 6) and give details of your hyperparameter settings. **Explain the reason for your choice.**

[40]:
```
# Your answer here
#Since SVM and SGD can meet the demand of our client's and behave equally on␣
↪precision and specificity, I think we can select either of
#the models,here I would choose SGD model as my model. Most of hyperparameter␣
↪settings are default setting like workshop exclude I set
#learning rate as 0.00008 through optimisation function and tune the parameter␣
↪by myself.
```

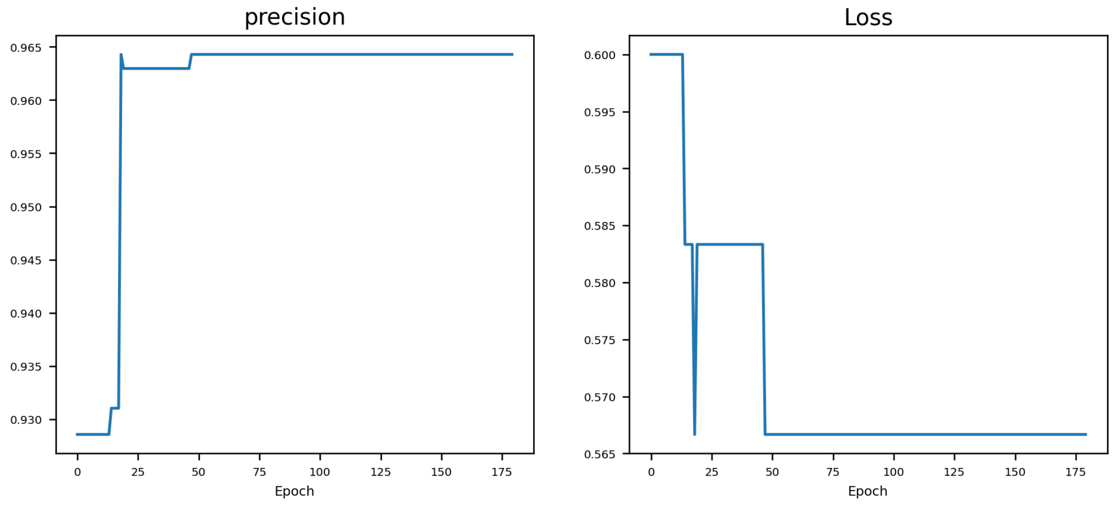
3.2.8 2.8 Final performance [1.5 marks]

Calculate and display an unbiased performance measure that you can present to the client.

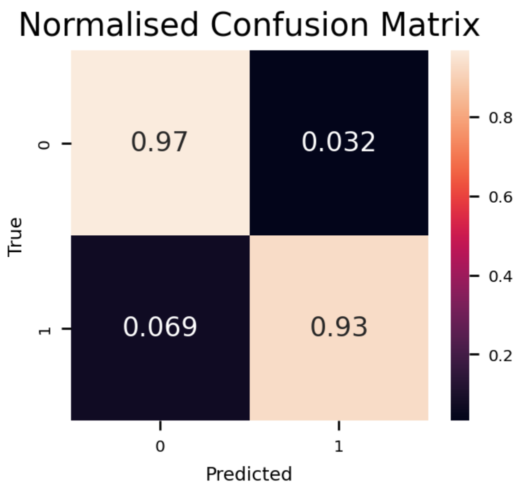
Is your chosen classifier underfitting or overfitting?

Does your chosen classifier meet the client's performance criteria?

```
[42]: # Your code here
def sgdfn(nsamp, learnrate, loop=True, learntype='constant',
          tol=1e-3, early_stopping=False, verbose=0, n_iter_no_change=5,␣
 ↪in_penalty=None, in_alpha=0.0001, in_figure=True):
    sgd = SGDClassifier(warm_start=True,
                        learning_rate=learntype,
                        eta0=learnrate,
                        early_stopping=early_stopping,
                        tol=tol,
                        verbose=verbose,
                        n_iter_no_change=n_iter_no_change,
                        penalty=in_penalty,
                        alpha=in_alpha)
    X_trainp = preproc_pl.fit_transform(X_train)
    X_testp = preproc_pl.transform(X_test)
    res=[]
    if loop:
        for n in range(nsamp):
            if early_stopping==False:
                sgd.partial_fit(X_trainp, y_train, classes=[0,1])
            else:
                sgd.fit(X_trainp, y_train)
            y_test_pred = sgd.predict(X_testp)
            sgd_acc = precision_score(y_test, y_test_pred)
            sgd_loss = hinge_loss(y_test, y_test_pred)
            res += [[sgd_acc, sgd_loss]]
    else:
        sgd.fit(X_trainp, y_train)
        y_test_pred = sgd.predict(X_testp)
        sgd_acc = precision_score(y_test, y_test_pred)
        sgd_loss = hinge_loss(y_test, y_test_pred)
        res += [[sgd_acc, sgd_loss]]

    res = np.array(res)

    if loop and in_figure:
        plt.figure(figsize=(10,4))
        plt.subplot(121)
        plt.plot(res[:,0])
        plt.title('precision')
        plt.xlabel('Epoch')
        plt.subplot(122)
        plt.plot(res[:,1])
        plt.title('Loss')
        plt.xlabel('Epoch')
        plt.show()
    print('precision:', res[-1,0], ', Loss: ', res[-1,1])
    return [res[-1,0], res[-1,1]],y_test_pred
ntrain = X_train.shape[0]
res,y_test_pred =sgdfn(ntrain,0.00008)
cmat = confusion_matrix(y_test,y_test_pred , normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
precisiont = precision_score(y_test, y_test_pred)
print("precision for model:", precisiont)
```



```
precision: 0.9642857142857143 , Loss:  0.5666666666666667
```



```
precision for model: 0.9642857142857143
```

[42]:
```
# Your answers here
#The SGD model after fitting all data from training set and apply it into test␣
↪set,can be find that the precicsion can be found to
#have 97% to detect the positive samples correctly and have nearly 7% rate to␣
↪mistakenly regard the healthy patient to be positive,
#which meets the client's criteria of 10%.According to the charts and compare␣
↪the confusion matrix with validation set,the consequence
#seems dosen't have problems of overfitting and underfitting.
#Here is another problem:It seems weird that the performance of SGD on test set␣
↪behave equally as on validation set,through setting
#different random seeds I'm sure that this is nothing wrong with code because I␣
↪can get different result of validation set and
#test set from other random seeds,so I think the reason is the dataset is very␣
↪small and this combo of random seed happens to
#cause this reslut.
```

3.3 3. Decision Boundaries (15% = 4.5 marks)

3.3.1 3.1 Rank features [1 mark]

Although it is only possible to know the true usefulness of a feature when you've combined it with others in a machine learning method, it is still helpful to have some measure for how discriminative each feature is on its own. One common method for doing this is to calculate a T-score (often used in statistics, and in the LDA machine learning method) for each feature.

The formula for the T-score is (mean(x2) - mean(x1))/(0.5*(stddev(x2) + stddev(x1))), where x1 and x2 are the datasets corresponding to the two classes. Large values for the T-score (either positive or negative) indicate discriminative ability.

Calculate the T-score for each feature and print out the best 4 features according to this score.

[43]:
```python
# Your code here
positive_data = data[data['label'] == 1]
negative_data = data[data['label'] == 0]
t_scores = {}
for column in data.columns[1:]:
    x1 = positive_data[column]
    x2 = negative_data[column]
    t_score = (x2.mean() - x1.mean()) / (0.5 * (x2.std() + x1.std()))
    t_scores[column] = t_score
i=0
for feature, score in t_scores.items():
    i+=1
    print(f"T-score for feature {i} {feature}: {score}")
#According to the result,the best 4 features are worst radius,worst␣
↪perimeter,concave points and worst area.
```

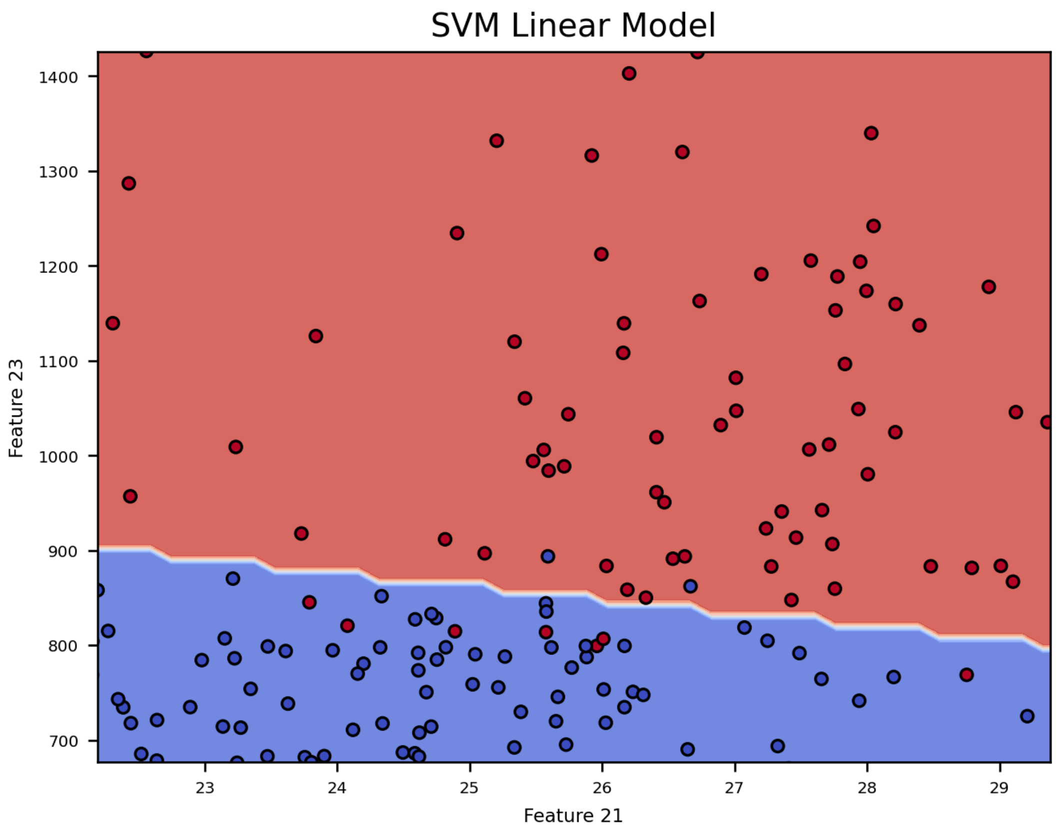
```
T-score for feature 1 mean radius: -2.0031836227577884
T-score for feature 2 mean texture: -1.0858009018741221
T-score for feature 3 mean perimeter: -2.0823044808776925
T-score for feature 4 mean area: -1.922528738419482
T-score for feature 5 mean smoothness: -0.7424559355220723
T-score for feature 6 mean compactness: -1.4255721566309447
T-score for feature 7 mean concavity: -1.6087855066521726
T-score for feature 8 mean concave points: -2.229435693128837
T-score for feature 9 mean symmetry: -0.6415664539538891
T-score for feature 10 mean fractal dimension: -0.011327316042990003
T-score for feature 11 radius error: -1.276429895728934
T-score for feature 12 texture error: 0.09237150211402838
T-score for feature 13 perimeter error: -1.2986045010645577
T-score for feature 14 area error: -1.4055110522365544
T-score for feature 15 smoothness error: 0.18916082599997003
T-score for feature 16 compactness error: -0.5073385648671872
T-score for feature 17 concavity error: -0.3360947488488207
T-score for feature 18 concave points error: -0.6678662933879579
T-score for feature 19 symmetry error: 0.12308665422832607
T-score for feature 20 fractal dimension error: -0.062280493097449l8
T-score for feature 21 worst radius: -2.4108393809617583
T-score for feature 22 worst texture: -1.1852547525346322
T-score for feature 23 worst perimeter: -2.473073212650544
T-score for feature 24 worst area: -2.226871351928768
T-score for feature 25 worst smoothness: -0.9666197403942813
T-score for feature 26 worst compactness: -1.4985867595165305
T-score for feature 27 worst concavity: -1.5858755353974208
T-score for feature 28 worst concave points: -2.4871607332953354
T-score for feature 29 worst symmetry: -0.9517608171723297
T-score for feature 30 worst fractal dimension: -0.7594656102681975
```

3.3.2 3.2 Visualise decision boundaries [2.5 marks]

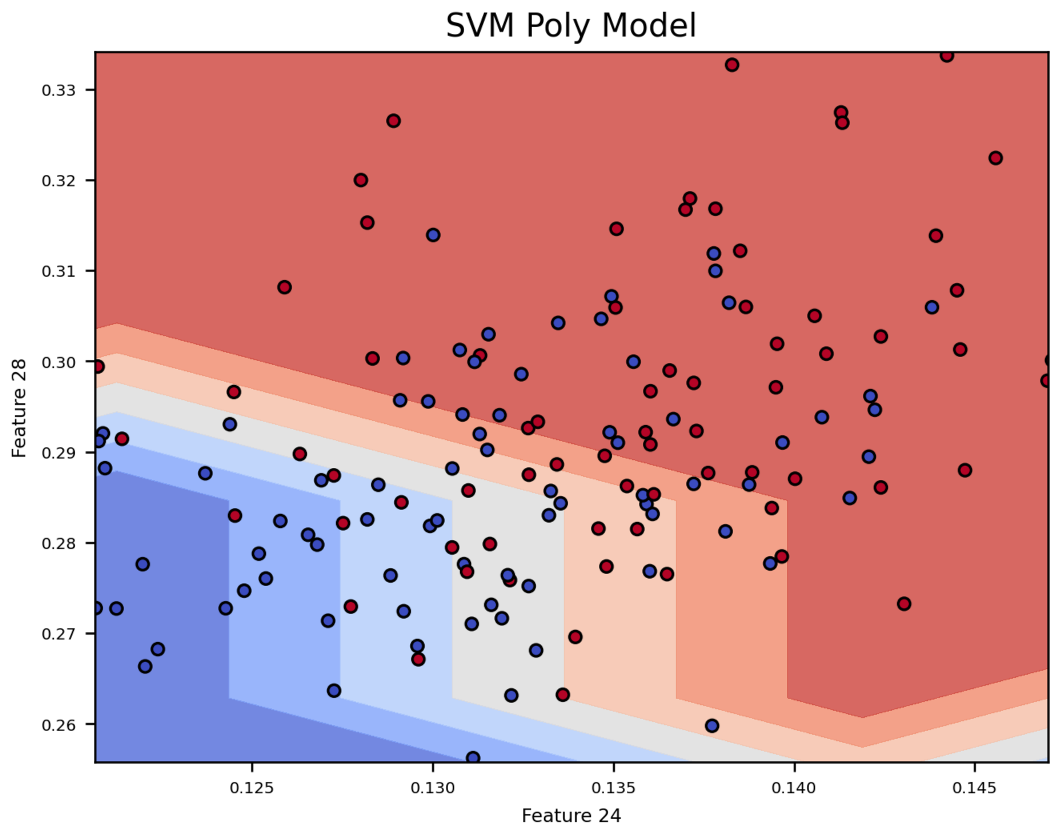
Display the decision boundaries for each pair of features from the best 4 chosen above. You can use the DecisionBoundaryDisplay function (as per workshop 6).

[58]:
```python
import matplotlib.pyplot as plt
from sklearn.inspection import DecisionBoundaryDisplay

feature1 = 21
feature2 = 23
feature3 = 24
feature4 = 28
```

```
a05, a95 = np.percentile(X_train.iloc[:, feature1], [5, 95])
b05, b95 = np.percentile(X_train.iloc[:, feature2], [5, 95])
c05, c95 = np.percentile(X_train.iloc[:, feature3], [5, 95])
d05, d95 = np.percentile(X_train.iloc[:, feature4], [5, 95])


svm_model = SVC(kernel='linear', C=1)


feat1 = X_train.iloc[:, feature1]
feat2 = X_train.iloc[:, feature2]
feat3 = X_train.iloc[:, feature3]
feat4 = X_train.iloc[:, feature4]


svm_model.fit(X_train.iloc[:, [feature1, feature2]], y_train)


disp = DecisionBoundaryDisplay.from_estimator(
    svm_model,
    X_train.iloc[:, [feature1, feature2]],
    response_method="predict",
    cmap=plt.cm.coolwarm,
    alpha=0.8,
    xlabel="Feature 21",
    ylabel="Feature 23"
)


plt.scatter(feat1, feat2, c=y_train, cmap=plt.cm.coolwarm, s=20, edgecolors="k")


plt.xlim([a05, a95])
plt.ylim([b05, b95])
plt.title("SVM Linear Model")
plt.show()
```



```
[68]: svm_model = SVC(kernel='poly', degree=10)
      svm_model.fit(X_train.iloc[:, [feature3, feature4]], y_train)

      disp = DecisionBoundaryDisplay.from_estimator(
          svm_model,
          X_train.iloc[:, [feature3, feature4]],      response_method="predict",
          cmap=plt.cm.coolwarm,
          alpha=0.8,
          xlabel="Feature 24",
          ylabel="Feature 28"
      )

      plt.scatter(feat3, feat4, c=y_train, cmap=plt.cm.coolwarm, s=20, edgecolors="k")

      plt.xlim([c05, c95])
      plt.ylim([d05, d95])
      plt.title("SVM Poly Model")
      plt.show()
```



3.3.3 3.3 Interpretation [1 mark]

From the decision boundaries displayed above, **would you expect the method to extrapolate well or not**? Give reasons for your answer.

[ ]:
```
# Your answer here
#To some degree I think using SVM method is OK to distinguish samples through␣
  ↪some features,but mat not behave well just like
#the models we trained before,from the chart 1 we can clearly find that the␣
  ↪feature 1 and 2 behave well on helping detecting different
#samples,and the linear boundary is perfect and simple.However,when it comes to␣
  ↪feature 3 and 4 on chart 2,the difference between
#samples seems not very clear and it's hard to distinguish different samples␣
  ↪through this decision boundary.
```

3.4 4. Second Round (15% = 4.5 marks)

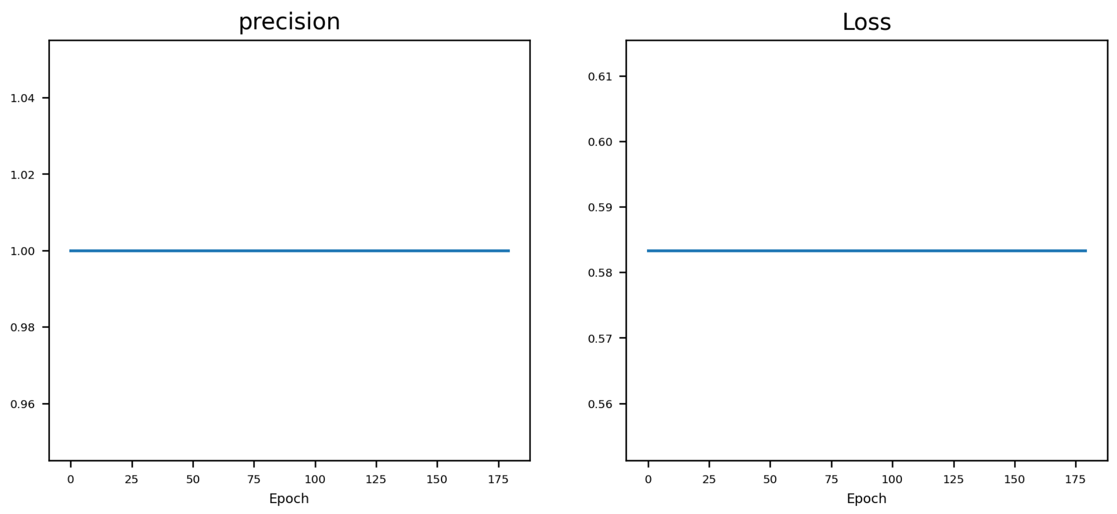
After presenting your initial results to the client they come back to you and say that they have done some financial analysis and it would save them a lot of time and money if they did not have to analyse every cell, which is needed to get the "worst" features. Instead, they can quickly get accurate estimates for the “mean” and “standard error” features from a much smaller, randomly selected set of cells.

They ask you to **give them a performance estimate for the same problem, but without using any of the “worst” features.**

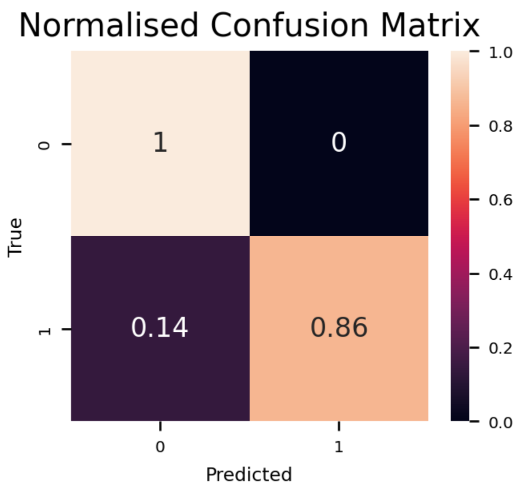
3.4.1 4.1 New estimate [3.5 marks]

Calculate an unbiased performance estimate for this new problem, as requested by the client.

```
[78]: # Your code here
data=pd.read_csv("assignment2.csv")
data['label'] = data['label'].replace({'malignant': 1, 'benign': 0})
columns_to_drop = [5,9,10,12,15,16,17,18,19,20,25,29,30]
data = data.drop(data.columns[columns_to_drop], axis=1)
bigtrain_set, test_set = train_test_split(data, test_size=0.2, random_state=20,␣
 ↪stratify=data['label'])
train_set, val_set = train_test_split(bigtrain_set, test_size=0.25,␣
 ↪random_state=20, stratify=bigtrain_set['label'])
X_train = train_set.iloc[:, 1:]
y_train = train_set.iloc[:, 0]
X_test = test_set.iloc[:, 1:]
y_test = test_set.iloc[:, 0]
X_val = val_set.iloc[:, 1:]
y_val = val_set.iloc[:, 0]
preproc_pl = Pipeline([ ('imputer', SimpleImputer(strategy="median")),
                        ('std_scaler', StandardScaler()) ])
def sgdfn(nsamp, learnrate, loop=True, learntype='constant',
          tol=1e-3, early_stopping=False, verbose=0, n_iter_no_change=5,␣
 ↪in_penalty=None, in_alpha=0.0001, in_figure=True):
    sgd = SGDClassifier(warm_start=True,
                        learning_rate=learntype,
                        eta0=learnrate,
                        early_stopping=early_stopping,
                        tol=tol,
                        verbose=verbose,
                        n_iter_no_change=n_iter_no_change,
                        penalty=in_penalty,
                        alpha=in_alpha)
    X_trainp = preproc_pl.fit_transform(X_train)
    X_testp = preproc_pl.transform(X_test)
    res=[]
    if loop:
        for n in range(nsamp):
            if early_stopping==False:
                sgd.partial_fit(X_trainp, y_train, classes=[0,1])
            else:
                sgd.fit(X_trainp, y_train)
            y_test_pred = sgd.predict(X_testp)
            sgd_acc = precision_score(y_test, y_test_pred)
            sgd_loss = hinge_loss(y_test, y_test_pred)
            res += [[sgd_acc, sgd_loss]]
    else:
        sgd.fit(X_trainp, y_train)
        y_test_pred = sgd.predict(X_testp)
        sgd_acc = precision_score(y_test, y_test_pred)
        sgd_loss = hinge_loss(y_test, y_test_pred)
        res += [[sgd_acc, sgd_loss]]

    res = np.array(res)

    if loop and in_figure:
        plt.figure(figsize=(10,4))
        plt.subplot(121)
        plt.plot(res[:,0])
        plt.title('precision')
        plt.xlabel('Epoch')
        plt.subplot(122)
        plt.plot(res[:,1])
        plt.title('Loss')
        plt.xlabel('Epoch')
        plt.show()

    print('precision:', res[-1,0], ', Loss: ', res[-1,1])
    return [res[-1,0], res[-1,1]],y_test_pred
ntrain = X_train.shape[0]
res,y_test_pred =sgdfn(ntrain,0.000001)
cmat = confusion_matrix(y_test,y_test_pred , normalize='true')
plt.figure(figsize=(3,2.5))
sns.heatmap(cmat,annot=True)
plt.xlabel('Predicted')
plt.ylabel('True')
plt.title('Normalised Confusion Matrix')
plt.show()
precisiont = precision_score(y_test, y_test_pred)
print("precision for model:", precisiont)
```



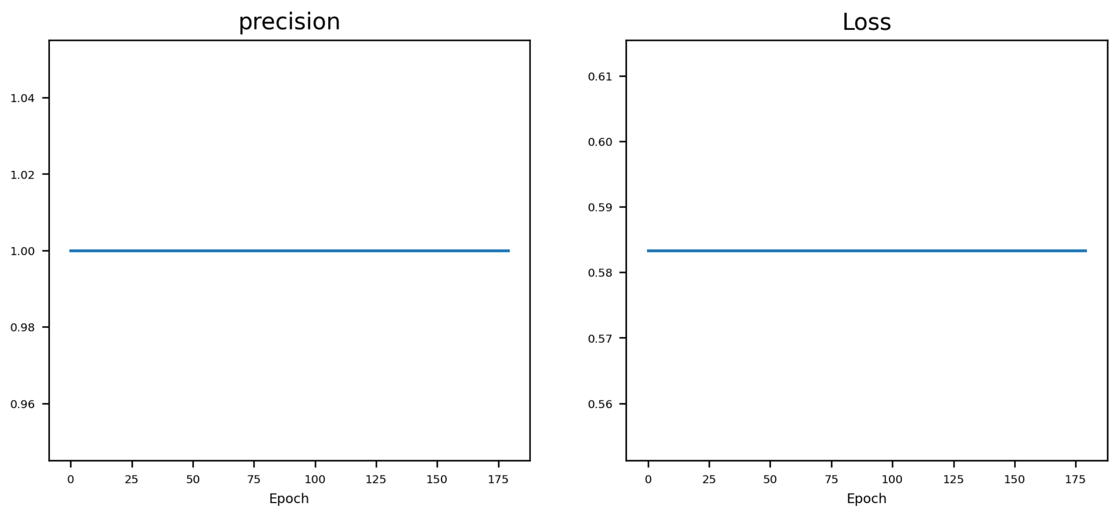
```
precision: 1.0 , Loss:  0.5833333333333334
```



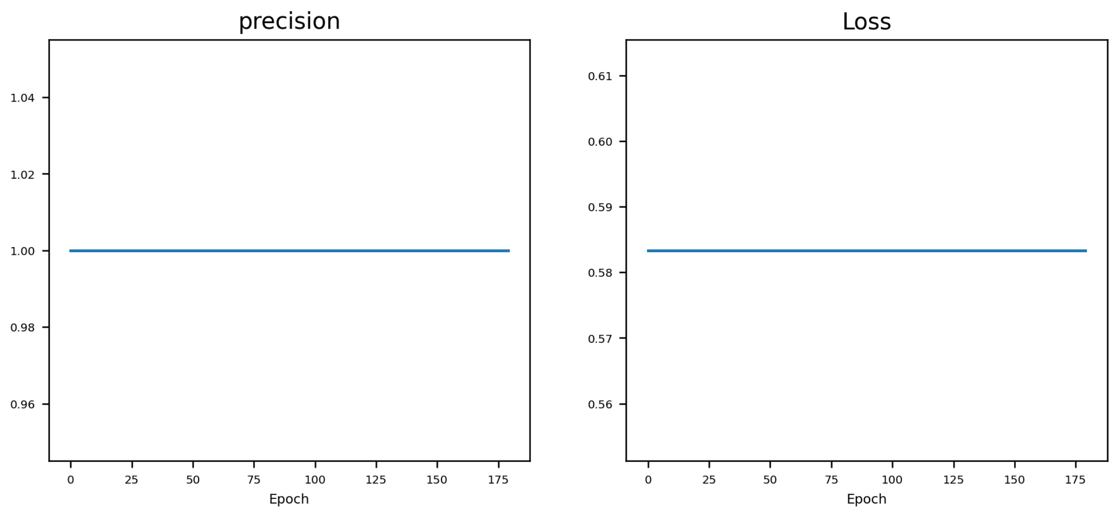
```
precision for model: 1.0
```

[76]:
```
res=[]
for lr_exp in range(-8,+3):
    lr = 10.0**lr_exp
    print(f'Learning rate = {lr}')
    res += sgdfn(ntrain,lr)
print(res)
```

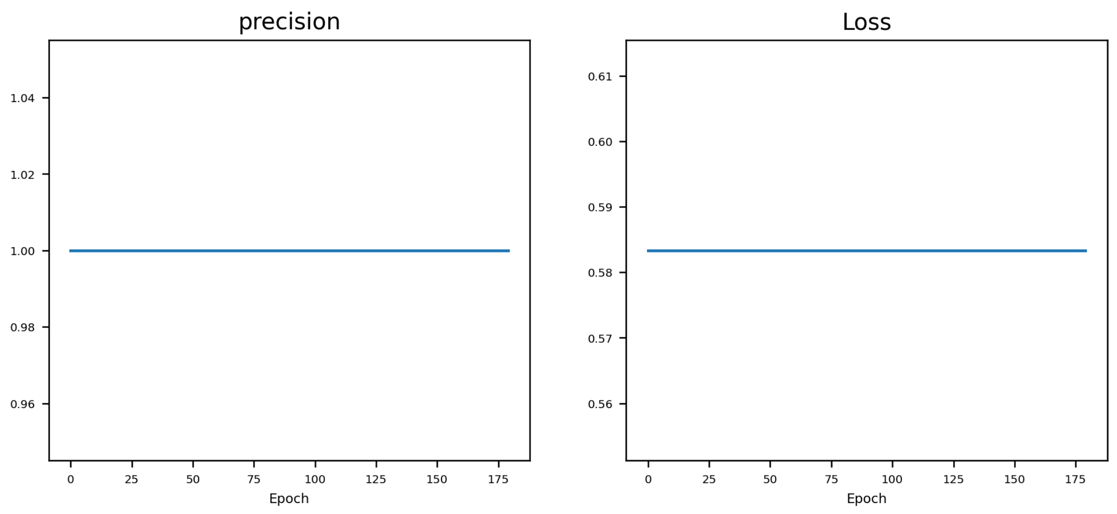
```
Learning rate = 1e-08
```



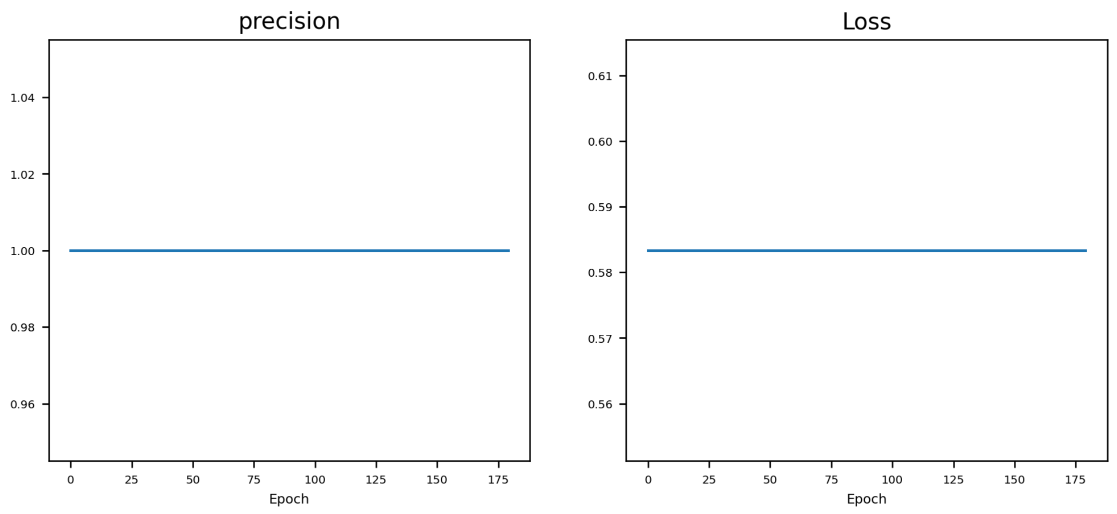
```
precision: 1.0 , Loss:  0.5833333333333334
Learning rate = 1e-07
```



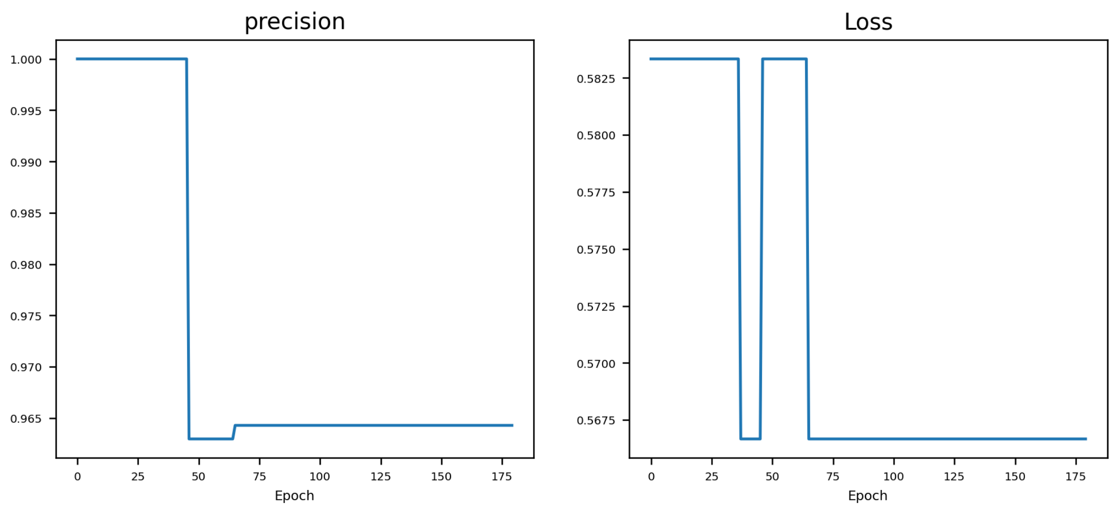
```
precision: 1.0 , Loss:  0.5833333333333334
Learning rate = 1e-06
```



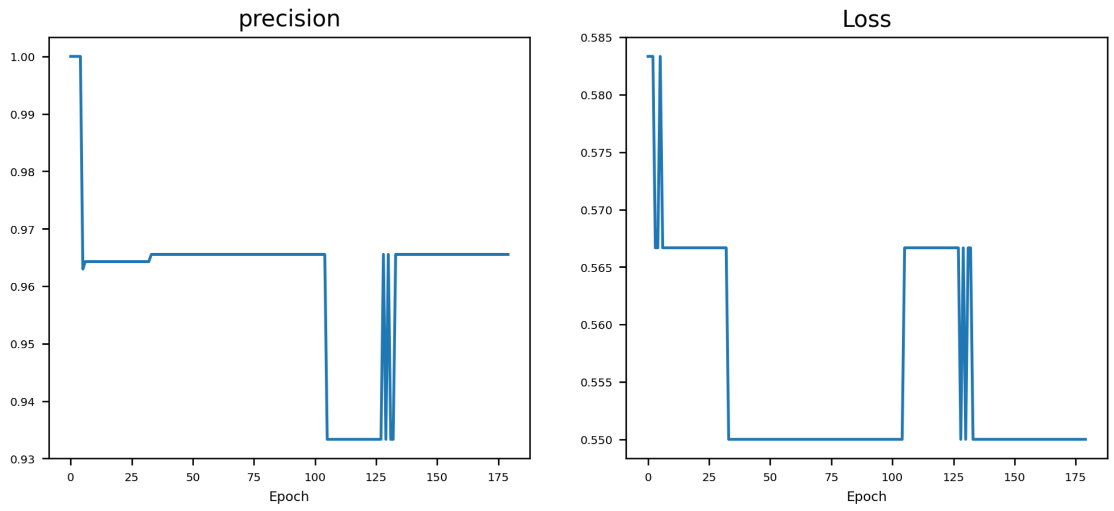
```
precision: 1.0 , Loss:  0.5833333333333334
Learning rate = 1e-05
```



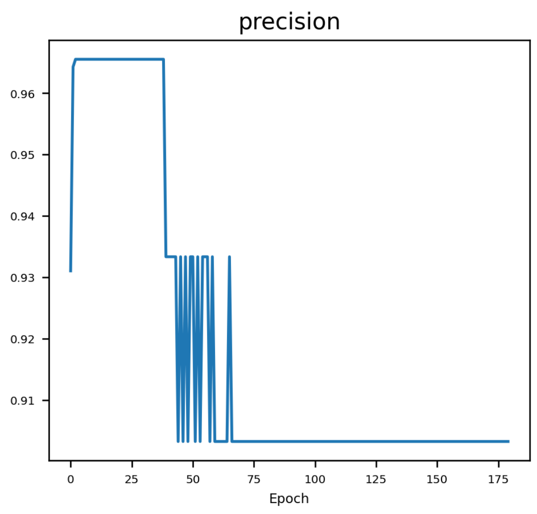
```
precision: 1.0 , Loss:  0.5833333333333334
Learning rate = 0.0001
```

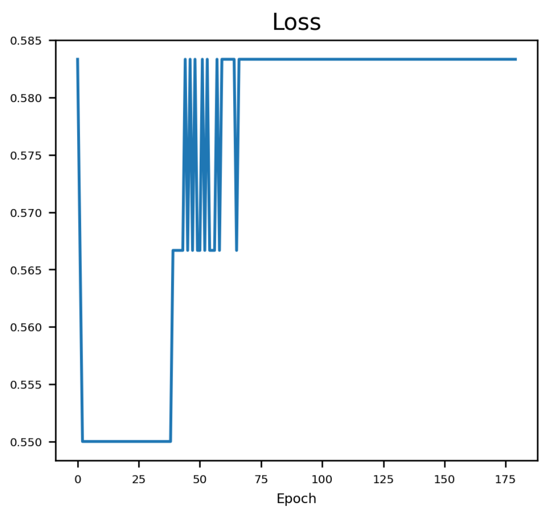


```
precision: 0.9642857142857143 , Loss:  0.5666666666666667
Learning rate = 0.001
```

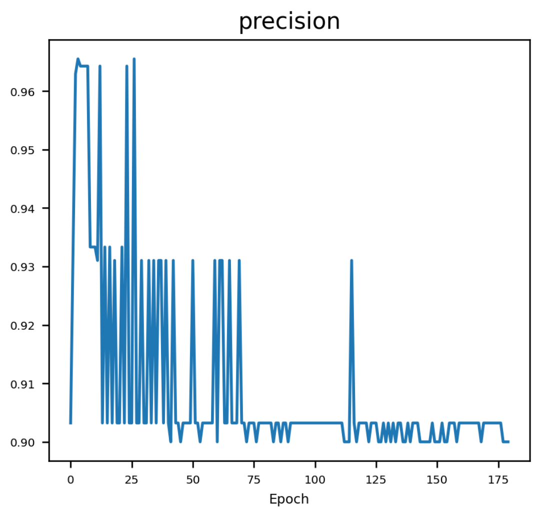


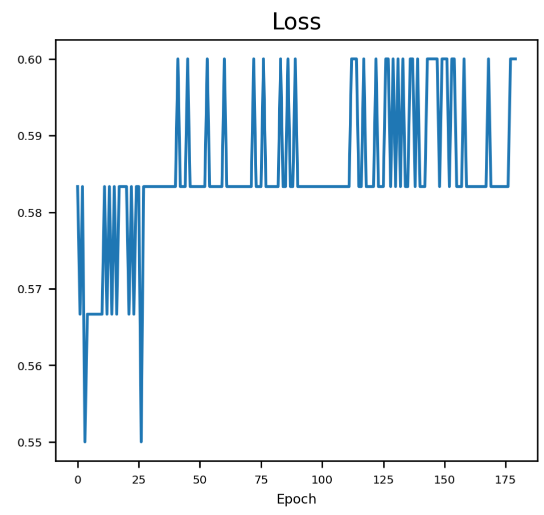
```
precision: 0.9655172413793104 , Loss:  0.55
Learning rate = 0.01
```



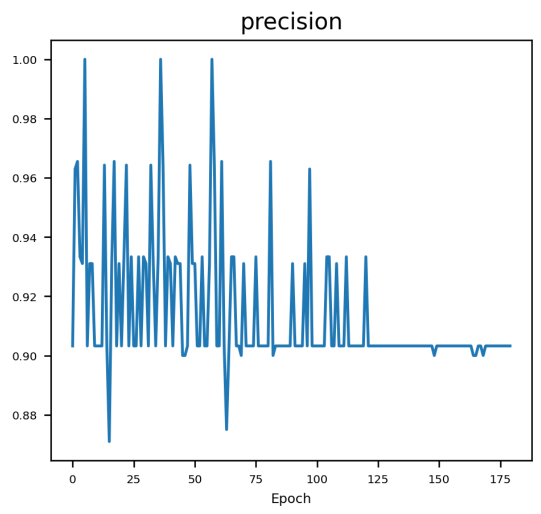


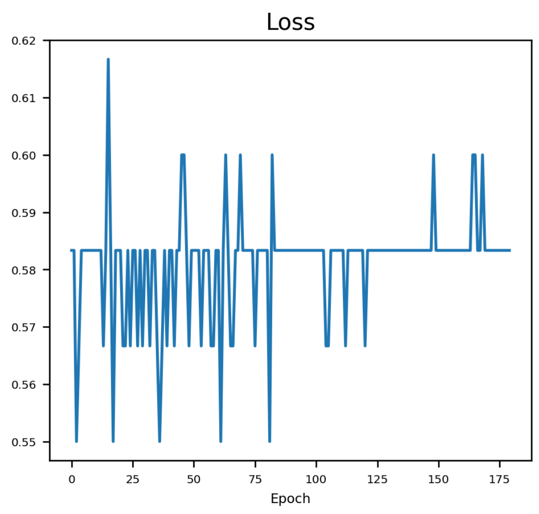
```
precision: 0.9032258064516129 , Loss:  0.5833333333333334
Learning rate = 0.1
```



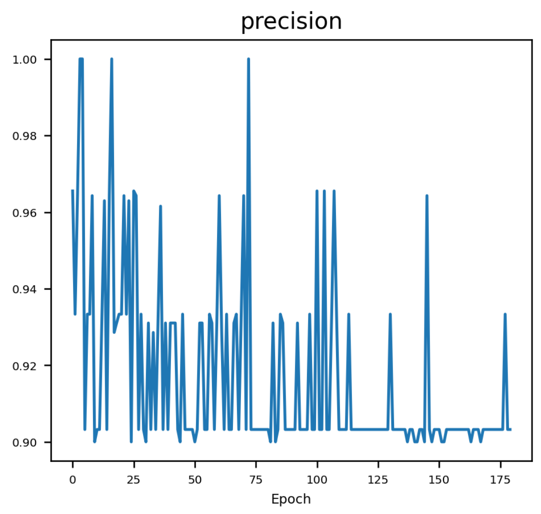


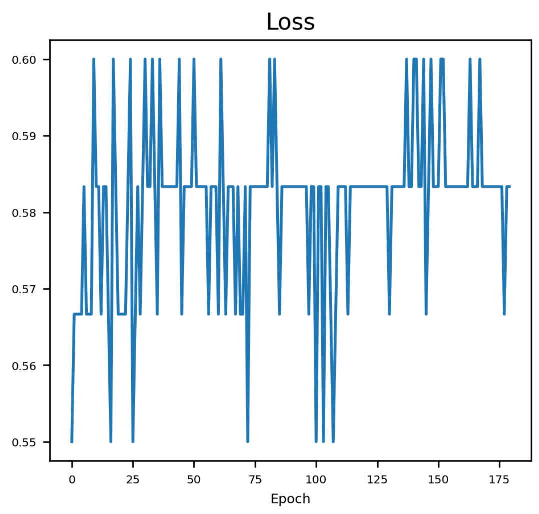
```
precision: 0.9 , Loss:  0.6
Learning rate = 1.0
```



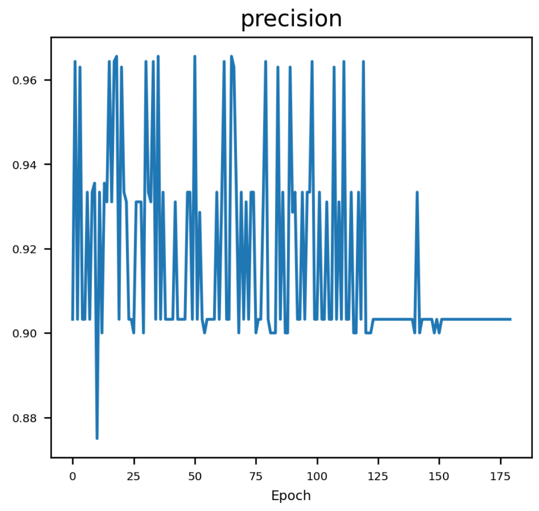


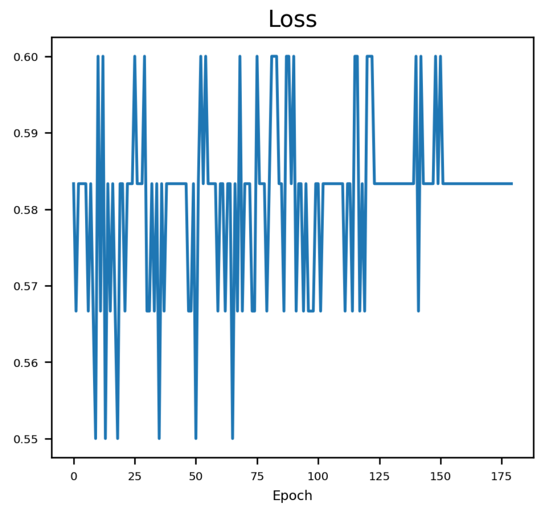
```
precision: 0.9032258064516129 , Loss:  0.5833333333333334
Learning rate = 10.0
```





```
precision: 0.9032258064516129 , Loss:  0.5833333333333334
Learning rate = 100.0
```





```
precision: 0.9032258064516129 , Loss:  0.5833333333333334
[[1.0, 0.5833333333333334], array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1,
0, 1, 0, 0, 0, 1, 0,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [1.0,
0.5833333333333334], array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [1.0,
0.5833333333333334], array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [1.0,
0.5833333333333334], array([1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9642857142857143,
0.5666666666666667], array([1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 0,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9655172413793104,
0.55], array([1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9032258064516129,
0.5833333333333334], array([1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9, 0.6], array([1,
1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1,
       0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9032258064516129,
0.5833333333333334], array([1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9032258064516129,
0.5833333333333334], array([1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0]), [0.9032258064516129,
0.5833333333333334], array([1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1,
0, 0, 0, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1,
       0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0])]
```

3.4.2 4.2 Performance difference [1 mark]

Do you think the new classifier, that does not use the "worst" features, is: - **as good as the previous classifier** (that uses all the features) - **better than the previous classifier** - **worse than the previous classifier**

Give reasons for your answer.

[74]:
```
data
```

[74]:
```
     label  mean radius  mean texture  mean perimeter   mean area  \
0        1    15.494654     15.902542      103.008265  776.437239
1        1    16.229871     18.785613      105.176755  874.712003
2        1    16.345671     20.114076      107.083804  872.563251
3        1    13.001009     19.876997       85.889775  541.281012
4        1    16.416060     17.397533      107.857386  891.516818
..     …          …            …              …         …
295      0    14.048464     17.186671       90.974271  637.474225
296      0    12.879033     16.767790       83.123369  539.225356
297      1    13.123052     18.793057       84.897717  555.002209
298      0    14.411991     18.970674       93.423809  671.128126
299      0    12.704174     20.895143       82.227859  528.052132

     mean compactness  mean concavity  mean concave points  radius error  \
0            0.168660        0.170572             0.085668      0.653654
1            0.092548        0.081681             0.053670      0.445451
2            0.123799        0.128788             0.078310      0.549625
3            0.173069        0.146214             0.069574      0.430693
4            0.111530        0.125971             0.068575      0.525532
..                …              …                   …            …
295          0.091549        0.063532             0.039494      0.355219
296          0.083986        0.059347             0.035404      0.314989
297          0.090178        0.066586             0.043711      0.474658
298          0.090118        0.070882             0.039482      0.356964
```

```
299          0.093698        0.068184            0.038141      0.364040

     perimeter error  area error  worst radius  worst texture  \
0           4.962255   80.619370     19.522957      22.427276
1           3.005373   50.407958     19.140235      24.905156
2           3.643671   62.732851     19.144816      25.601433
3           3.051434   33.614356     15.565911      26.145119
4           3.747194   59.164555     18.620376      22.306233
..               …          …            …              …
295         2.481640   33.861241     15.790651      22.538529
296         2.230067   28.250520     14.358919      21.955513
297         3.238155   40.474522     14.991646      24.820718
298         2.560170   35.435273     16.555187      25.591332
299         2.694336   33.293080     14.199113      25.377961

     worst perimeter   worst area  worst compactness  worst concavity  \
0         135.128520  1286.903131           0.407483         0.445992
1         123.886045  1234.499997           0.223918         0.248846
2         125.113036  1202.749973           0.314402         0.332505
3         102.958265   737.655082           0.485912         0.430007
4         124.002529  1139.490971           0.230996         0.316620
..               …          …                 …                …
295       103.423320   819.408970           0.206701         0.192139
296        93.620160   684.694077           0.191978         0.180949
297        97.933068   726.695117           0.201766         0.202433
298       108.978466   893.818250           0.246945         0.236415
299        93.143286   681.453918           0.195607         0.192059

     worst concave points
0                0.171662
1                0.136735
2                0.161497
3                0.167254
4                0.131715
..                    …
295              0.095350
296              0.083989
297              0.100361
298              0.105354
299              0.085053

[300 rows x 18 columns]
```

[81]:
```
# Your answer here
#Here I drop the worst 13 features--with T-score start with 0,and reuse the SGD␣
↪method again to evaluate the performance.
```

```
#We can see that the precision is even better than the model with all␣
 ↪features,but the specificity rate behave worse than previous model.
#That means the new classifier with less worst features even behave better than␣
 ↪model with all features to some degree,but still may
#behave worse to some fields,which is acceptable to some certain circumstances.
```