CTG Case Study

December 6, 2021

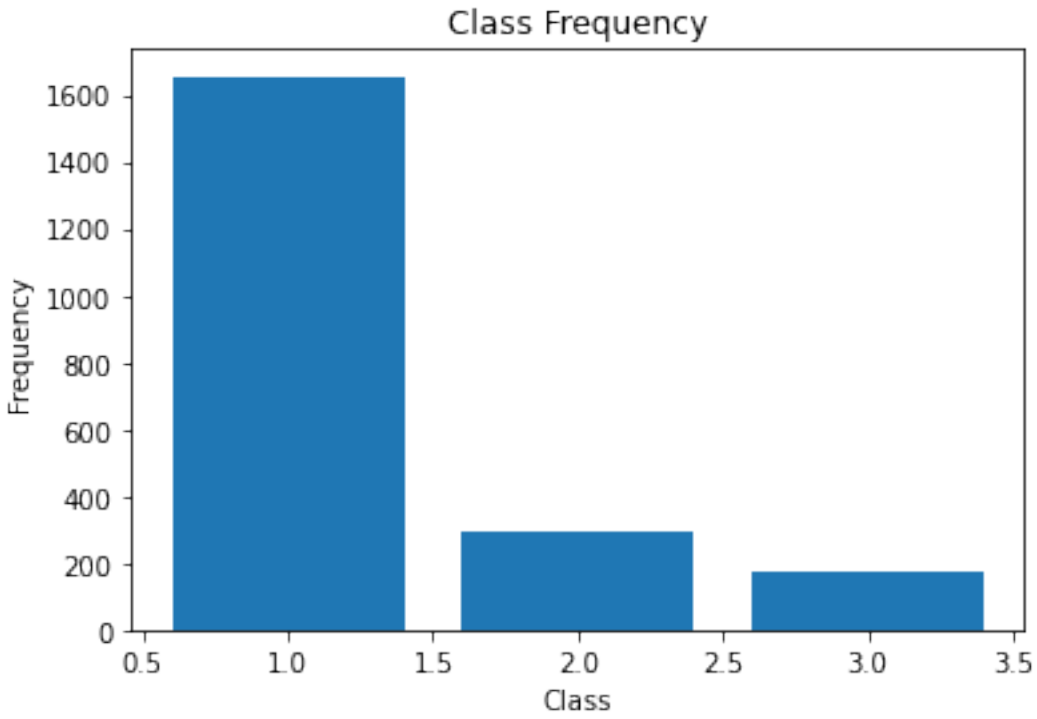
1 Data Description for CTG Data

1.0.1 fetal_health (1,2,3)

- 1: Normal
- 2: Suspect
- 3: Pathological

2 Visual Distribution of 3 classes using bar chart:

[13]:
```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

data = pd.read_csv('data/fetal_health-1.csv')
N = len(data['fetal_health']) # Data set size

unique, counts = np.unique(data.iloc[:,-1], return_counts=True)
#print(unique, counts)
plt.bar(unique, counts)

plt.title('Class Frequency')
plt.xlabel('Class')
plt.ylabel('Frequency')

plt.show()

print('Relative Frequencies of each class in whole dataset:')
print(data['fetal_health'].value_counts()/N)

#print(data.describe())
```




```
Relative Frequencies of each class in whole dataset:
1.0    0.778457
2.0    0.138758
3.0    0.082785
Name: fetal_health, dtype: float64
```

Frequency of Each class (1,2,3) respectively 1 -> 1655 (78%)

2 -> 295 (14%)

3 -> 176 (8%)

2.0.1 Now it is obvious the dataset is imbalanced and heavily leans toward Normal Outputs (1)

2.0.2 Plan of Attack

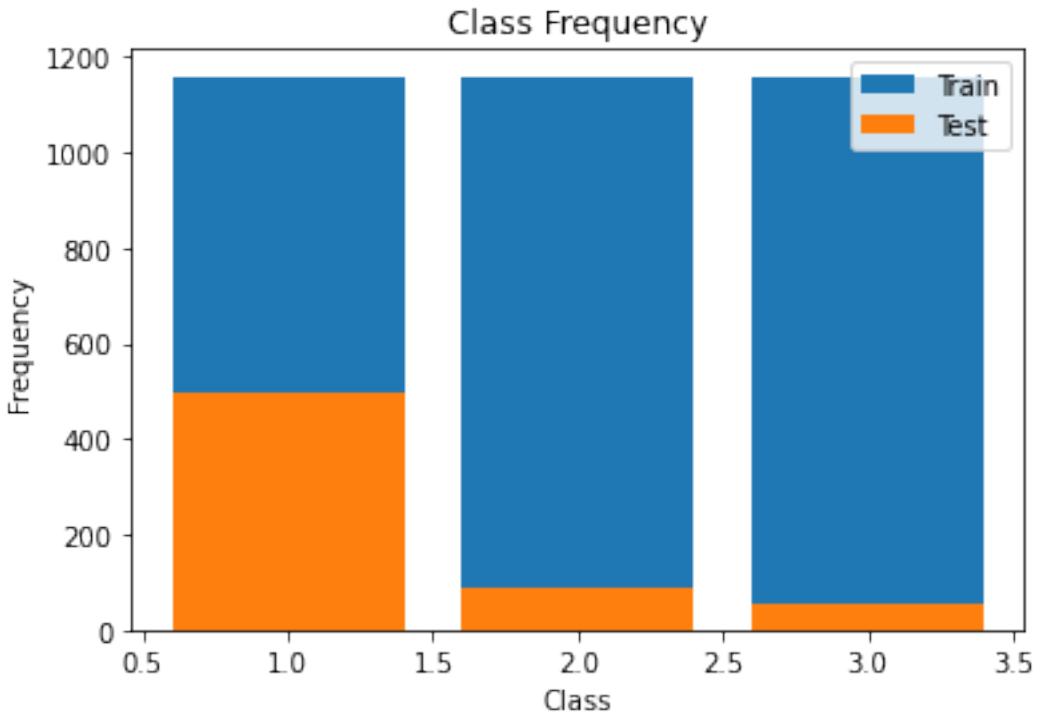
We will test 3 methods and choose the one with the best results.

- Simply stratify the data when splitting.
- Split randomly and oversample our train data.
- Split data with stratification and then oversample our train data.

Note: We are stratifing with respect to the fetal_health attribute

Note: We should never over sample our test data.

[14]:
```
# Oversample data
def oversample_data(data):
    count_1,count_2,count_3 = data['fetal_health'].value_counts()
    class_1 = data[data['fetal_health'] == 1.0]
    class_2 = data[data['fetal_health'] == 2.0]
    class_3 = data[data['fetal_health'] == 3.0]

    class_2_over = class_2.sample(count_1,replace =True)
    class_3_over = class_3.sample(count_1,replace =True)
    return pd.concat([class_1, class_2_over,class_3_over], axis=0)


#Random Data Split
#train_data, test_data = train_test_split(data,test_size=0.3)

#Straified data split on fetal_health parameter
train_data, test_data = train_test_split(data,test_size=0.3,␣
 ↪stratify=data['fetal_health'])

#OverSample
train_data = oversample_data(train_data)


#Plot Frequencies
unique, counts = np.unique(train_data.iloc[:,-1], return_counts=True)
#print(unique, counts)
plt.bar(unique, counts)

unique, counts = np.unique(test_data.iloc[:,-1], return_counts=True)
#print(unique, counts)
plt.bar(unique, counts)

plt.title('Class Frequency')
plt.xlabel('Class')
plt.ylabel('Frequency')
plt.legend(['Train','Test'])
plt.show()

print('Relative Frequencies of each class in train set:')
print(train_data['fetal_health'].value_counts()/len(train_data['fetal_health']))

print('Relative Frequencies of each class in test set:')
print(test_data['fetal_health'].value_counts()/len(test_data['fetal_health']))
```




```
Relative Frequencies of each class in train set:
3.0    0.333333
2.0    0.333333
1.0    0.333333
Name: fetal_health, dtype: float64
Relative Frequencies of each class in test set:
1.0    0.778997
2.0    0.137931
3.0    0.083072
Name: fetal_health, dtype: float64
```

2.1 Result

Now that we used the stratify parameter, our test and training split has consistent amount of data relative to the global percentages.

3 10 Features most reflective of fetal health

3.1 Method

We decided to use pearson correlation to do this. The closer the absolute value of the output is to 1, the stronger the correlation.

In the code, we took the last row of data which is gives me the correlation of everything with fetal_health.

[15]:
```
corr_table = data.corr()
fh_corr = corr_table.iloc[-1,:].abs().sort_values(ascending=False).to_frame()
best_attrib = fh_corr.index.values[1:11]
#Print Top 10 attributes
print(fh_corr.iloc[1:11])
```

```
                                                 fetal_health
prolongued_decelerations                             0.484859
abnormal_short_term_variability                      0.471191
percentage_of_time_with_abnormal_long_term_vari…     0.426146
accelerations                                        0.364066
histogram_mode                                       0.250412
histogram_mean                                       0.226985
mean_value_of_long_term_variability                  0.226797
histogram_variance                                   0.206630
histogram_median                                     0.205033
uterine_contractions                                 0.204894
```

3.2 Result

As a result of pearson correlation, we found the set above to be the most highly correlated to fetal healt.

4 Significance Testing

We now will test if these attributes are not just randomly significant in our sample.

We will use both 0.05 and 0.1 as our significance values.

4.0.1 Assumption

We are currently assuming the attributes are independent.

[16]:
```
from scipy.stats import ttest_ind

sig_levels = (0.05,0.1)

significance_test = {}
#Last tuple in dict, will contain the results of the hypothesis testing, If␣
 ↪True we reject the null hypothesis.
for attrib in best_attrib:
    t_score,p_score = ttest_ind(data[attrib],data['fetal_health'])
    significance_test[attrib] = (t_score,p_score,p_score<=sig_levels)
```

```
    print(significance_test[attrib][2])
#print(significance_test)
```

```
[ True  True]
[ True  True]
[ True  True]
[ True  True]
[ True  True]
[ True  True]
[ True  True]
[ True  True]
[ True  True]
[ True  True]
```

4.1 Result

We can now see that for all 10 attributes, they are statistically significant on both critical values.

5 Model Development

We will now create 2 models, a linear regression model and a naive bayes model.

[17]:
```
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LinearRegression
from sklearn.metrics import accuracy_score

#Convert from continous values to class values
def reg_to_class(pred):
    tmp = [ round(x) for x in pred ]
    for i in range(len(tmp)):
        if tmp[i] < 1.0:
            tmp[i] = 1.0
        elif tmp[i]>3.0:
            tmp[i] = 3.0
    return tmp

#split up train data
train_x = train_data.iloc[:,:-1]
train_y = train_data.iloc[:,-1]

#split up test data
test_x = test_data.iloc[:,:-1]
test_y = test_data.iloc[:,-1]

# Naive Bayes
```

```
NB_model = GaussianNB().fit(train_x,train_y)
NB_pred = NB_model.predict(test_x)
print(accuracy_score(NB_pred,test_y))

#Linear Regression
LR_model = LinearRegression().fit(train_x,train_y)
LR_pred = LR_model.predict(test_x)
LR_pred_classes = reg_to_class(LR_pred)

print(accuracy_score(LR_pred_classes,test_y))
```

```
0.7398119122257053
0.700626959247649
```

5.1 Code Overview

Up above, we created linear regression and naive bayes models for our data.

Here we are defining a helper function we found online

[18]:
```
#function found online for multi-class confusion
def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    import itertools
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    #print(cm)

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, rotation=45)
    plt.yticks(tick_marks, classes)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.
```

```
    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt),
                 horizontalalignment="center",
                 color="white" if cm[i, j] > thresh else "black")

    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.tight_layout()
```

6 Metrics

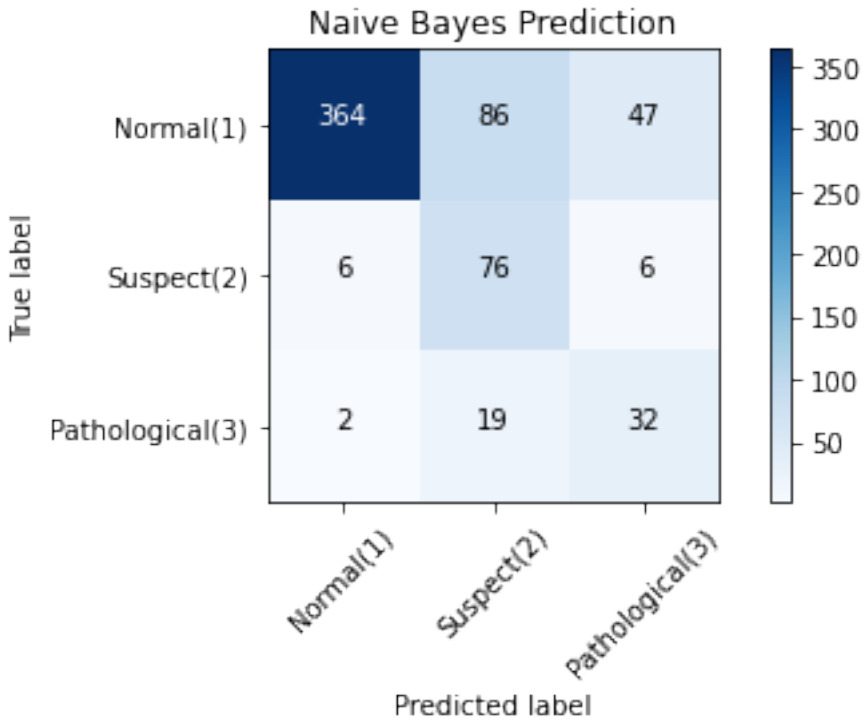
Below we will show the confusion matrices, ROC curve, F1 score, and P-R curve for both models

6.1 Naive Bayes:

[19]:
```
from sklearn.metrics import confusion_matrix,roc_auc_score, f1_score
from sklearn.metrics import average_precision_score, precision_recall_curve
from sklearn.metrics import auc, plot_precision_recall_curve,␣
 ↪classification_report
from sklearn.preprocessing import label_binarize

#Binarizing Predictions and Expecteds
binarized_test_y = label_binarize(test_y,classes=[1.0,2.0,3.0])
NB_prob_pred = NB_model.predict_proba(test_x)

#Calculate many metrics
metrics = classification_report(test_y, NB_pred, digits=3,output_dict=True)
roc_score =␣
 ↪roc_auc_score(test_y,NB_prob_pred,multi_class='ovr',average='weighted')
pr_auc =␣
 ↪average_precision_score(binarized_test_y,NB_prob_pred,average='weighted')

print('Weighted Average F1 Score: {:.2f}'.format(metrics['weighted␣
 ↪avg']['f1-score']))
print('Weighted Average ROC-AUC:  {:.2f}'.format(roc_score))
print('Weighted Average PR-AUC:   {:.2f}\n'.format(pr_auc))

cm = confusion_matrix(test_y, NB_pred,labels=[1.0,2.0,3.0])
plot_confusion_matrix(cm,classes=['Normal(1)','Suspect(2)','Pathological(3)'],title='Naive␣
 ↪Bayes Prediction')
```

```
Weighted Average F1 Score: 0.77
Weighted Average ROC-AUC:  0.90
Weighted Average PR-AUC:   0.87
```

```
Confusion matrix, without normalization
```

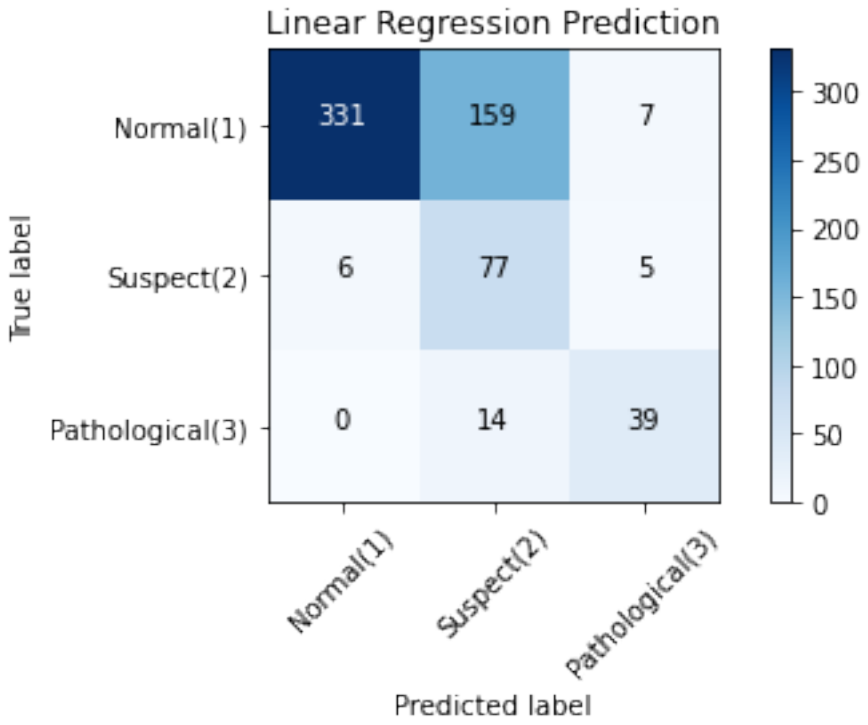


6.2 Linear Regression:

[20]:
```
#Binarizing Predictions and Expecteds
binarized_pred = label_binarize(LR_pred_classes,classes=[1.0,2.0,3.0])

#Calculate many metrics
metrics = classification_report(test_y, LR_pred_classes,␣
 ↪digits=3,output_dict=True)
roc_score =␣
 ↪roc_auc_score(test_y,binarized_pred,multi_class='ovr',average='weighted')
pr_auc =␣
 ↪average_precision_score(binarized_test_y,binarized_pred,average='weighted')

print('Weighted Average F1 Score: {:.2f}'.format(metrics['weighted␣
 ↪avg']['f1-score']))
print('Weighted Average ROC-AUC:  {:.2f}'.format(roc_score))
print('Weighted Average PR-AUC:   {:.2f}\n'.format(pr_auc))

cm = confusion_matrix(test_y, LR_pred_classes,labels=[1.0,2.0,3.0])
```

```
plot_confusion_matrix(cm,classes=['Normal(1)','Suspect(2)','Pathological(3)'],title='Linear Regression Prediction')
```

```
Weighted Average F1 Score: 0.74
Weighted Average ROC-AUC:  0.81
Weighted Average PR-AUC:   0.80

Confusion matrix, without normalization
```



7 A Different Approach

7.1 Clustering

We can take a new unsupervised approach as well. Since our data is continous we can use k-means clustering.

7.1.1 K Parameter

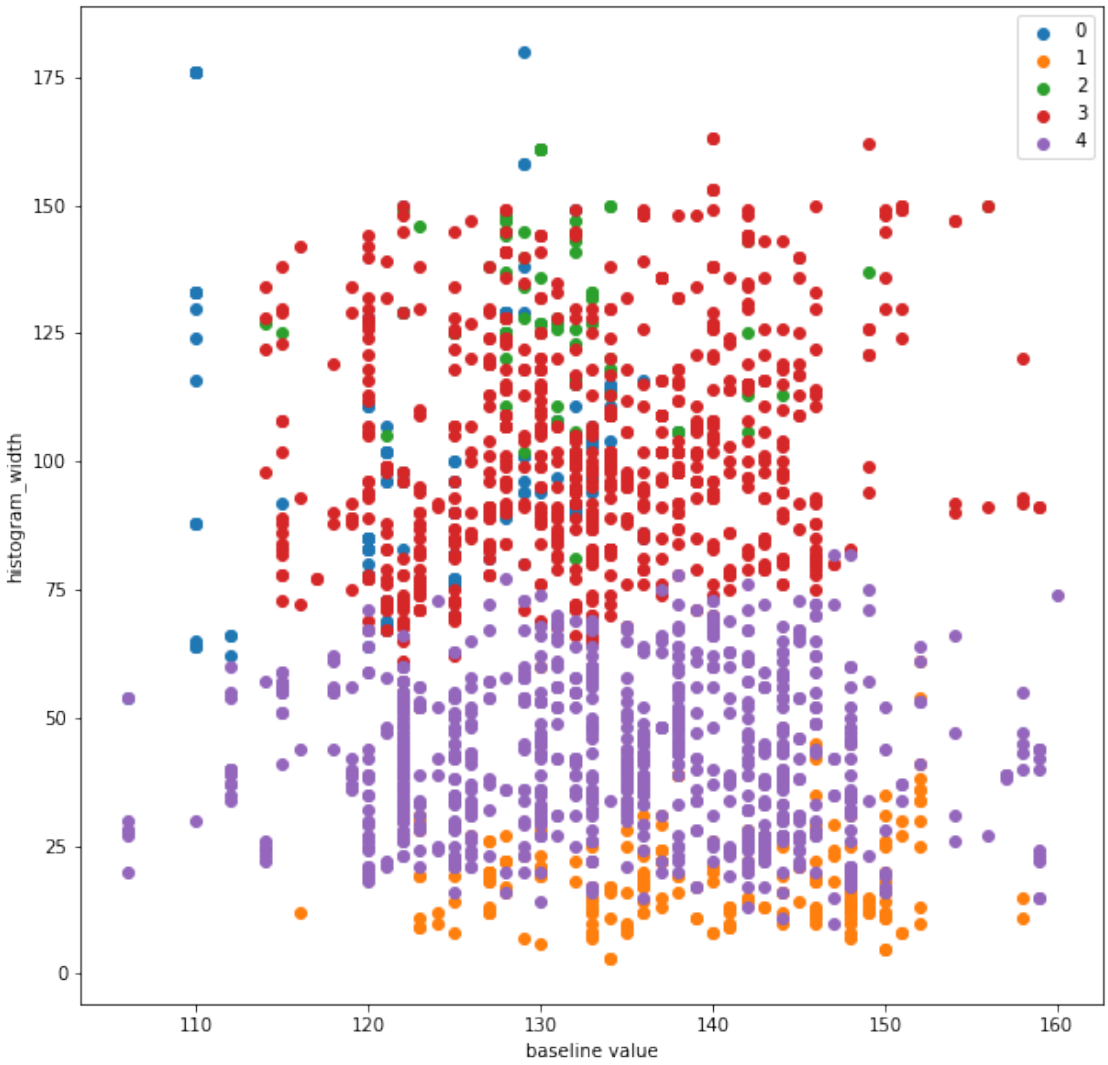
We choose to test multiple cluster sizes. k=5,10,15

8 Let's Begin

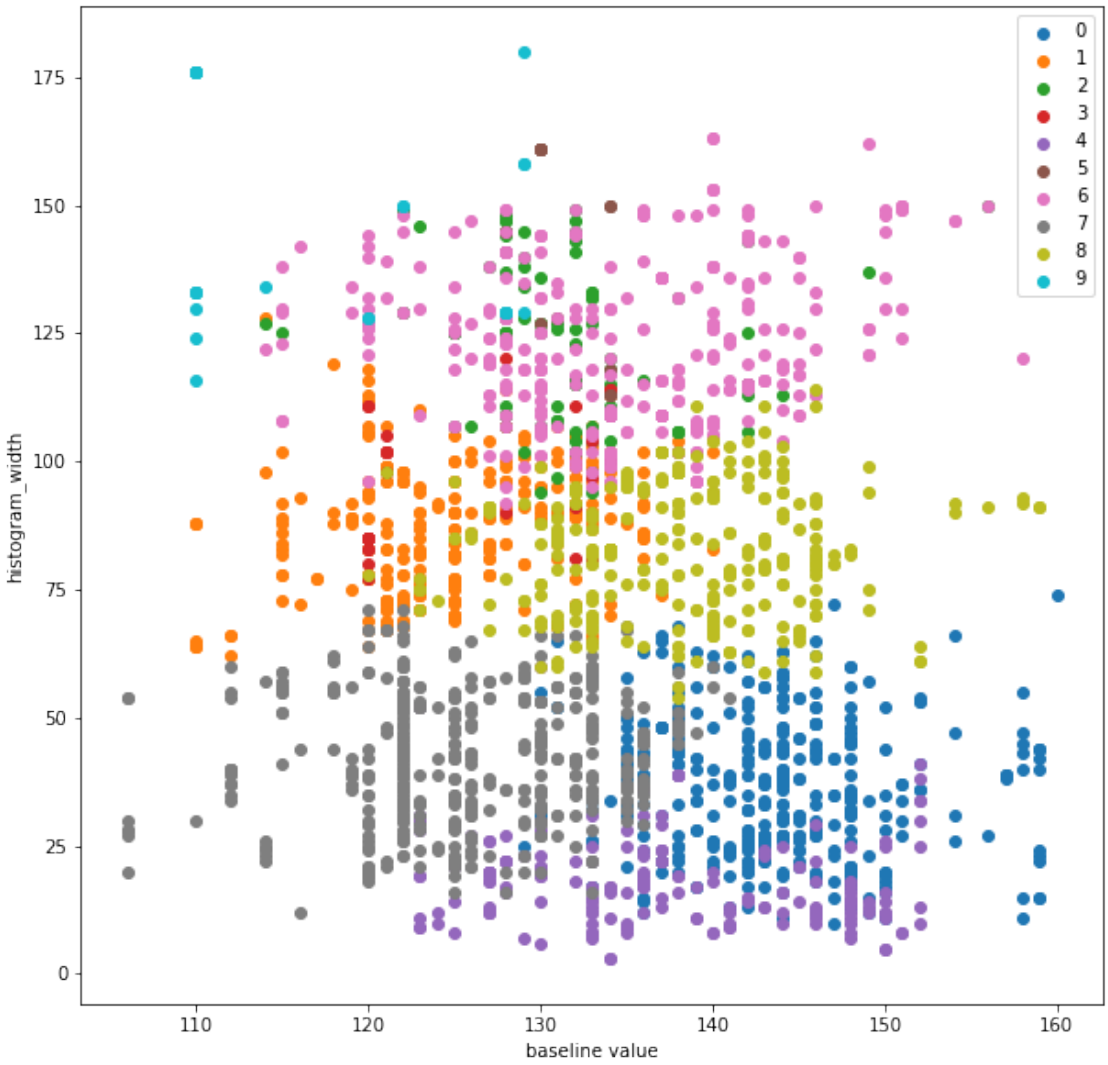
[21]:
```
from sklearn.cluster import KMeans

k_lst = [5,10,15]

km_models = {}
cluster_preds = {}
complete_test_sets = {}

for k in k_lst:
    km_models[k] = KMeans(k).fit(train_x)
    cluster_preds[k] = km_models[k].predict(data.iloc[:,:-1])
    complete_test_data = data.copy()
    complete_test_data['cluster'] = cluster_preds[k]
    complete_test_sets[k] = complete_test_data
   #print(complete_test_sets[k].head)

def cluster_metrics(k):
    #Getting unique labels

    u_labels = [i for i in range(k)]
    #print(complete_test_sets[km_models[k].cluster_centers_[]].head.label)

    #plotting the results:
    plt.figure(figsize=(10,10))
    for i in u_labels:
        xs = complete_test_sets[k].loc[complete_test_sets[k]['cluster'] ==␣
↪i]['baseline value']
        ys = complete_test_sets[k].loc[complete_test_sets[k]['cluster'] ==␣
↪i]['histogram_width']
        plt.scatter(xs , ys , label = i)
    plt.xlabel('baseline value')
    plt.ylabel('histogram_width')
    plt.legend()
    plt.show()
```

9 Let's get our metrics!

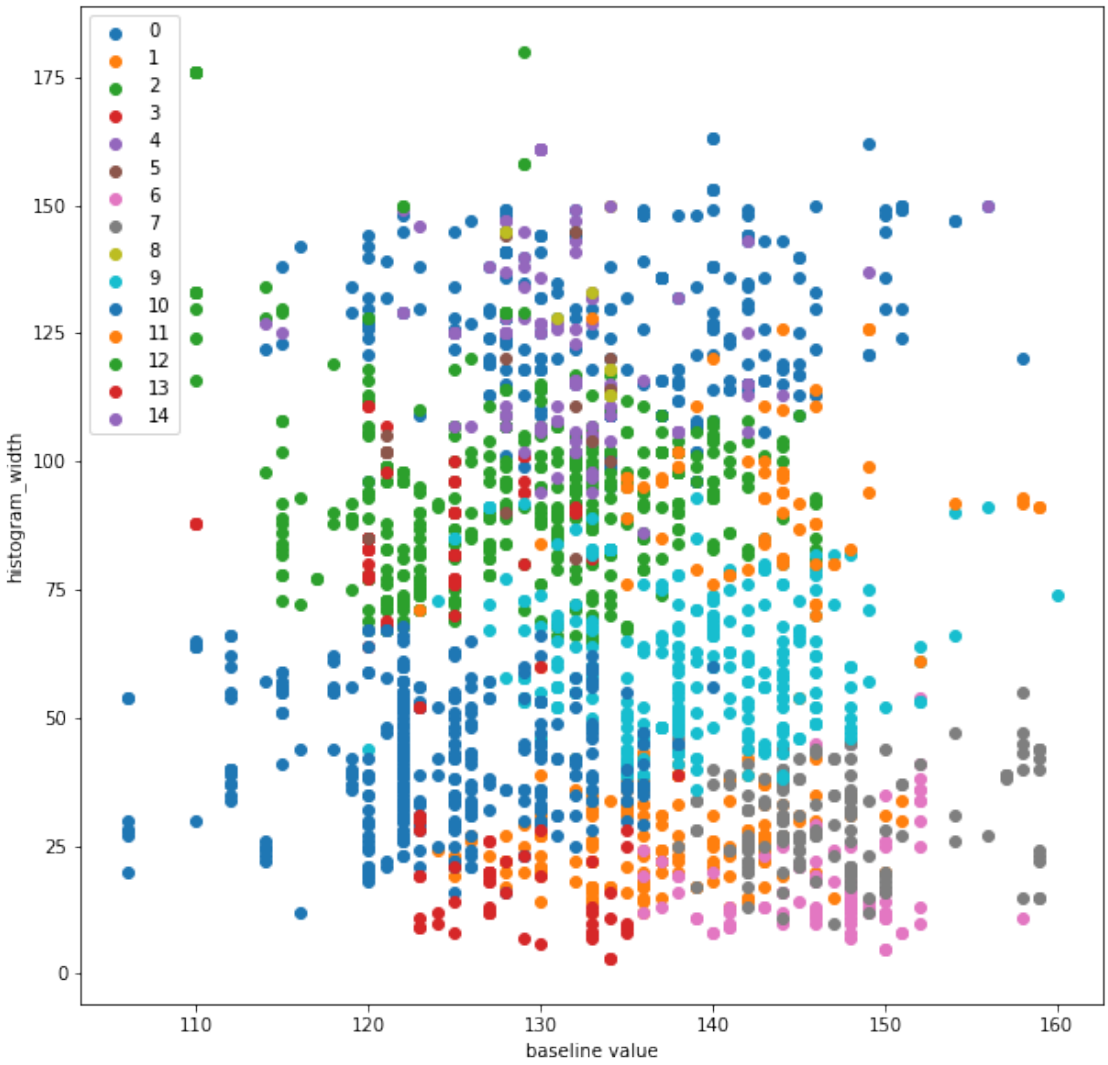
[22]:
```
cluster_metrics(5)
#print(classification_report(test_y, cluster_preds[5], digits=3,labels=[1.0,2.
↪0,3.0]))
```



[23]:
```
cluster_metrics(10)
#print(classification_report(test_y, cluster_preds[10], digits=3,labels=[1.0,2.
↪0,3.0]))
```



[24]:
```
cluster_metrics(15)
#print(classification_report(test_y, cluster_preds[15], digits=3,labels=[1.0,2.
↪0,3.0]))
```



10 Observations and Conclusions

10.1 Stratify the data

10.1.1 Naive Bayes

Weighted Average F1 Score: 0.84

Weighted Average ROC-AUC: 0.93

Weighted Average PR-AUC: 0.89

10.1.2 Linear Regression

Weighted Average F1 Score: 0.83

Weighted Average ROC-AUC: 0.82

Weighted Average PR-AUC: 0.79

10.2 Over sample, not stratified

10.2.1 Naive Bayes

Weighted Average F1 Score: 0.78

Weighted Average ROC-AUC: 0.92

Weighted Average PR-AUC: 0.88

10.2.2 Linear Regression

Weighted Average F1 Score: 0.78

Weighted Average ROC-AUC: 0.83

Weighted Average PR-AUC: 0.82

10.3 Over sample, stratified

10.3.1 Naive Bayes

Weighted Average F1 Score: 0.79

Weighted Average ROC-AUC: 0.92

Weighted Average PR-AUC: 0.89

10.3.2 Linear Regression

Weighted Average F1 Score: 0.76

Weighted Average ROC-AUC: 0.83

Weighted Average PR-AUC: 0.81

10.4 Analyzing Results

In this case, it seems like oversampling the dataset was not necessary. Just by looking at our metrics, the Naive Bayes classifier on the stratified dataset is the best model of the 9 tested.

Surprisingly when oversampling the data, we saw that the metrics slightly decreased per model. This could be attributed to potential overfitting.

When Looking at the clustering data, we can definitly see there are proper clusters that seem to corrolate well to each other but futher analysis will be needed to learn what each cluster means for fetal_health

10.5 Best Model and Preprocessing

Out of everything we tested, we can infer that there is no need to oversample out data and we can stratify against the fetal_health attribute when splitting up the dataset.

The best model is the Naive Bayes model and that will be the model of choice for any predictive work.