Capstone - Breast cancer prediction using Deep Nueral Network

Background

Breast cancer is a type of cancer with high mortality rates among women, and it is one of the most common causes of death in women. According to the National Cancer Institue statistics in USA, one out of eight women suffers from breast cancer and 6% of all deaths worldwide are caused by this type of cancer.

Early diagnosis and accurate diagnosis of breast cancer is of prime importance as it will increase the survival chances. Thus, a precise and reliable system is essential for the timely diagnosis of benign or malignant breast tumors.

Objective

The key objective is to develop a model to predict breast cancer as benign or malignant using the data set from the digitized image of FNA sample.

Radiologists conduct Fine Needle Aspirate (FNA) procedure of breast tumor. FNA is a non invasive technique for detecting breast cancer. This procedure reveals features such as tumor radius, area, perimeter, concavity, texture and fractal dimensions. These features are further studied by medical experts to classify tumor as benign or malignant. Pathologists require a lot of skill and expertise to perform the analysis on the FNA sample. Applyting the suitable features of the FNA results in the most important diagnostic problem in early stages of breast cancer. Hence, development of algorithms which provide accurate predictions is of great interest.

Dataset used - Breast Cancer Wisconsin Diagnostic Data set

Features in the dataset are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

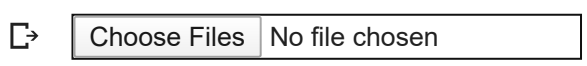
Predictive Model - Deep Nueral Network using Tensor Flow

```
# Predict whether the cancer is malignant or benign

# Load all the required packages

import tensorflow as tf
import pandas as pd
from sklearn.utils import shuffle
from sklearn import preprocessing
from sklearn.metrics import roc_auc_score, roc_curve, auc, confusion_matrix
import matplotlib.gridspec as gridspec
import seaborn as sns
import matplotlib.pyplot as plt
import random as rn
import os
import numpy as np
%matplotlib inline
```

```
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:541: FutureWarning: Passing (type, 1) or '1t
  _np_qint8 = np.dtype([("qint8", np.int8, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:542: FutureWarning: Passing (type, 1) or '1t
  _np_quint8 = np.dtype([("quint8", np.uint8, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:543: FutureWarning: Passing (type, 1) or '1t
  _np_qint16 = np.dtype([("qint16", np.int16, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:544: FutureWarning: Passing (type, 1) or '1t
  _np_quint16 = np.dtype([("quint16", np.uint16, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:545: FutureWarning: Passing (type, 1) or '1t
  _np_qint32 = np.dtype([("qint32", np.int32, 1)])
/usr/local/lib/python3.6/dist-packages/tensorboard/compat/tensorflow_stub/dtypes.py:550: FutureWarning: Passing (type, 1) or '1t
  np_resource = np.dtype([("resource", np.ubyte, 1)])
/usr/local/lib/python3.6/dist-packages/statsmodels/tools/_testing.py:19: FutureWarning: pandas.util.testing is deprecated. Use t
  import pandas.util.testing as tm
```

Loading the dataset to the Colab notebook

```
# load the dataset from the local directory to the Colab jupyter notebook

from google.colab import files
uploaded = files.upload()
```




```
Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cel
Saving breast cancer data.csv to breast cancer data (5).csv
```

```
# Load the data
```

```
train_filename = "breast cancer data.csv"


# Set column keys

idKey = "id"
diagnosisKey = "diagnosis"
radiusMeanKey = "radius_mean"
textureMeanKey = "texture_mean"
perimeterMeanKey = "perimeter_mean"
areaMeanKey = "area_mean"
smoothnessMeanKey = "smoothness_mean"
compactnessMeanKey = "compactness_mean"
concavityMeanKey = "concavity_mean"
concavePointsMeanKey = "concave points_mean"
symmetryMeanKey = "symmetry_mean"
fractalDimensionMean = "fractal_dimension_mean"
radiusSeKey = "radius_se"
textureSeKey = "texture_se"
perimeterSeKey = "perimeter_se"
areaSeKey = "area_se"
smoothnessSeKey = "smoothness_se"
compactnessSeKey = "compactness_se"
concavitySeKey = "concavity_se"
concavePointsSeKey = "concave points_se"
symmetrySeKey = "symmetry_se"
fractalDimensionSeKey = "fractal_dimension_se"
radiusWorstKey = "radius_worst"
textureWorstKey = "texture_worst"
perimeterWorstKey = "perimeter_worst"
areaWorstKey = "area_worst"
smoothnessWorstKey = "smoothness_worst"
compactnessWorstKey = "compactness_worst"
concavityWorstKey = "concavity_worst"
concavePointsWorstKey = "concave points_worst"
symmetryWorstKey = "symmetry_worst"
fractalDimensionWorstKey = "fractal_dimension_worst"


train_columns = [idKey, diagnosisKey, radiusMeanKey, textureMeanKey, perimeterMeanKey, areaMeanKey,
                 smoothnessMeanKey, compactnessMeanKey, concavityMeanKey, concavePointsMeanKey,
                 symmetryMeanKey, fractalDimensionMean, radiusSeKey, textureSeKey, perimeterSeKey,
                 areaSeKey, smoothnessSeKey, compactnessSeKey, concavitySeKey, concavePointsSeKey,
                 symmetrySeKey, fractalDimensionSeKey, radiusWorstKey, textureWorstKey, perimeterWorstKey,
                 areaWorstKey, smoothnessWorstKey, compactnessWorstKey, concavityWorstKey,
                 concavePointsWorstKey, symmetryWorstKey, fractalDimensionWorstKey]


def get_train_data():
    df = pd.read_csv(train_filename, names= train_columns, delimiter=',', skiprows=1)
    return df


train_data = get_train_data()
```

```
# Exploring the data

train_data.head()
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

```
train_data.describe()
```

	id	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	p
count	5.690000e+02	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	569.000000	
mean	3.037183e+07	14.127292	19.289649	91.969033	654.889104	0.096360	0.104341	0.088799	
std	1.250206e+08	3.524049	4.301036	24.298981	351.914129	0.014064	0.052813	0.079720	
min	8.670000e+03	6.981000	9.710000	43.790000	143.500000	0.052630	0.019380	0.000000	
25%	8.692180e+05	11.700000	16.170000	75.170000	420.300000	0.086370	0.064920	0.029560	
50%	9.060240e+05	13.370000	18.840000	86.240000	551.100000	0.095870	0.092630	0.061540	
75%	8.813129e+06	15.780000	21.800000	104.100000	782.700000	0.105300	0.130400	0.130700	
max	9.113205e+08	28.110000	39.280000	188.500000	2501.000000	0.163400	0.345400	0.426800	

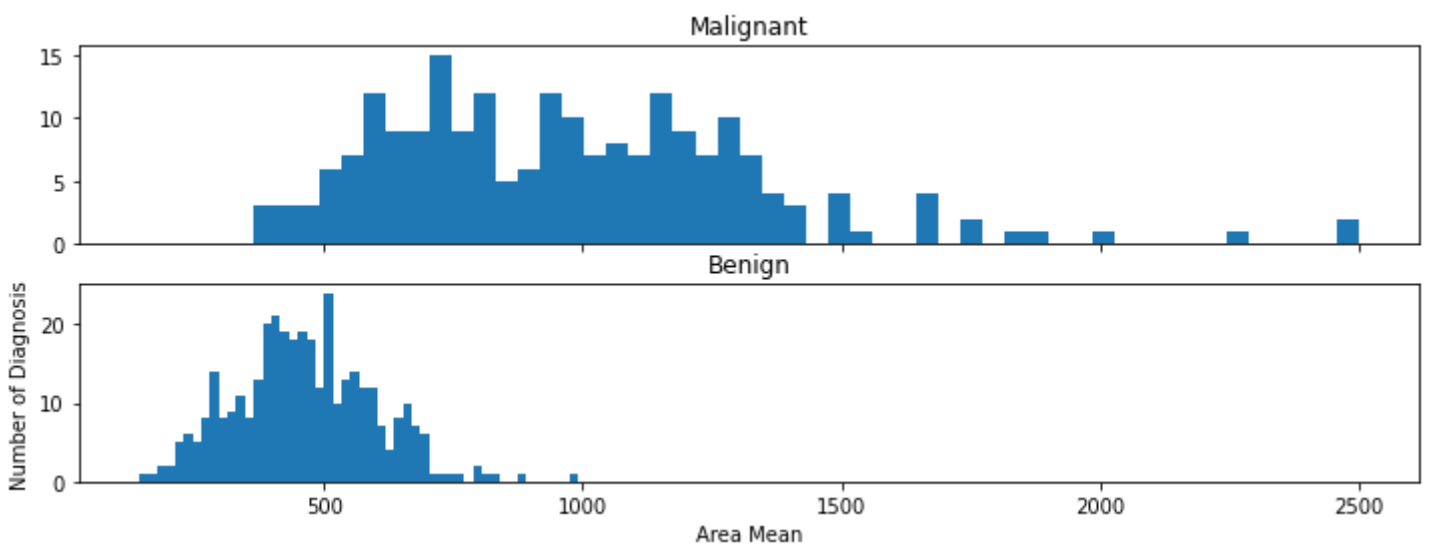
```
# Checking for null values

# No missing values found in the dataset

train_data.isnull().sum()
```

```
id                         0
diagnosis                  0
radius_mean                0
texture_mean               0
perimeter_mean             0
area_mean                  0
smoothness_mean            0
compactness_mean           0
concavity_mean             0
concave points_mean        0
symmetry_mean              0
fractal_dimension_mean     0
radius_se                  0
texture_se                 0
perimeter_se               0
area_se                    0
smoothness_se              0
compactness_se             0
concavity_se               0
concave points_se          0
symmetry_se                0
fractal_dimension_se       0
radius_worst               0
texture_worst              0
perimeter_worst            0
area_worst                 0
smoothness_worst           0
compactness_worst          0
concavity_worst            0
concave points_worst       0
symmetry_worst             0
fractal_dimension_worst    0
dtype: int64
```

```
# how area_mean compares across malignant and benign diagnosis.

print ("Malignant")
print (train_data.area_mean[train_data.diagnosis == "M"].describe())
print ()
print ("Benign")
print (train_data.area_mean[train_data.diagnosis == "B"].describe())
```

```
Malignant
count     212.000000
mean      978.376415
std       367.937978
min       361.600000
25%       705.300000
50%       932.000000
75%      1203.750000
max      2501.000000
Name: area_mean, dtype: float64

Benign
count    357.000000
mean     462.790196
std      134.287118
min      143.500000
25%      378.200000
50%      458.400000
75%      551.100000
max      992.100000
Name: area_mean, dtype: float64
```

```
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(12,4))

bins = 50

ax1.hist(train_data.area_mean[train_data.diagnosis == "M"], bins = bins)
ax1.set_title('Malignant')

ax2.hist(train_data.area_mean[train_data.diagnosis == "B"], bins = bins)
ax2.set_title('Benign')

plt.xlabel('Area Mean')
plt.ylabel('Number of Diagnosis')
plt.show()
```

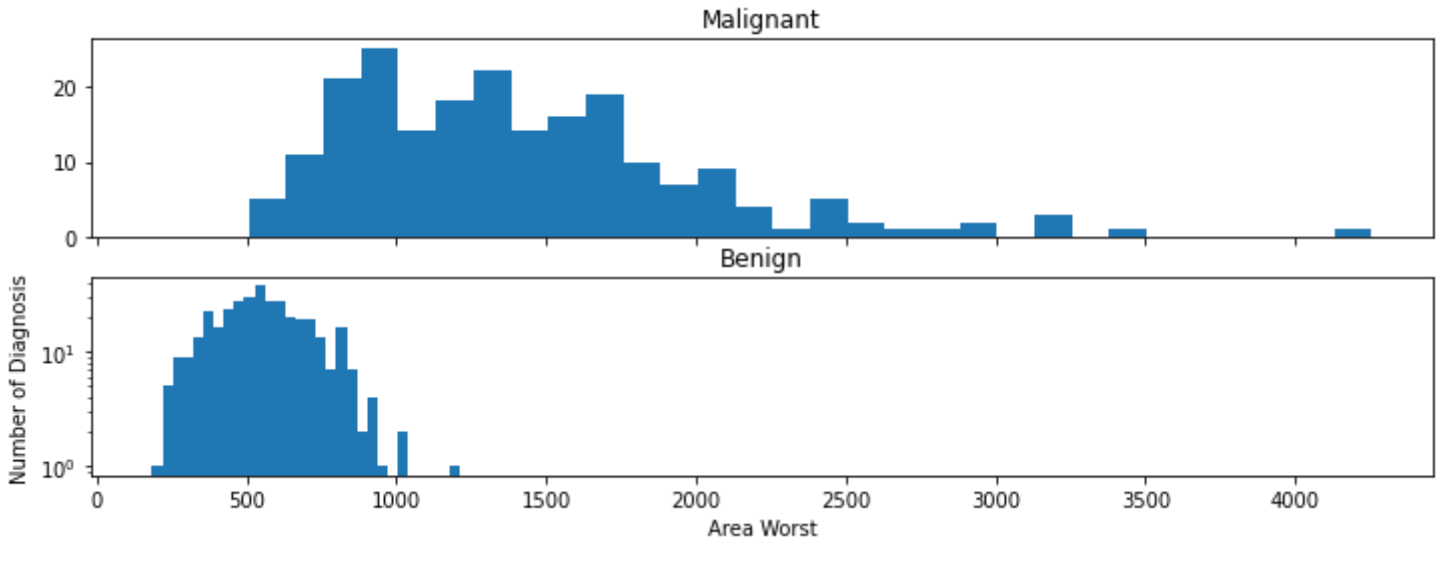


The 'area_mean' feature looks different as it increases its value across both types of diagnosis. It appears that, the malignant diagnosis are more uniformly distributed, while benign diagnosis have a normal distribution. This could make it easier to detect a malignant diagnosis when the area_mean is above the 750 value.

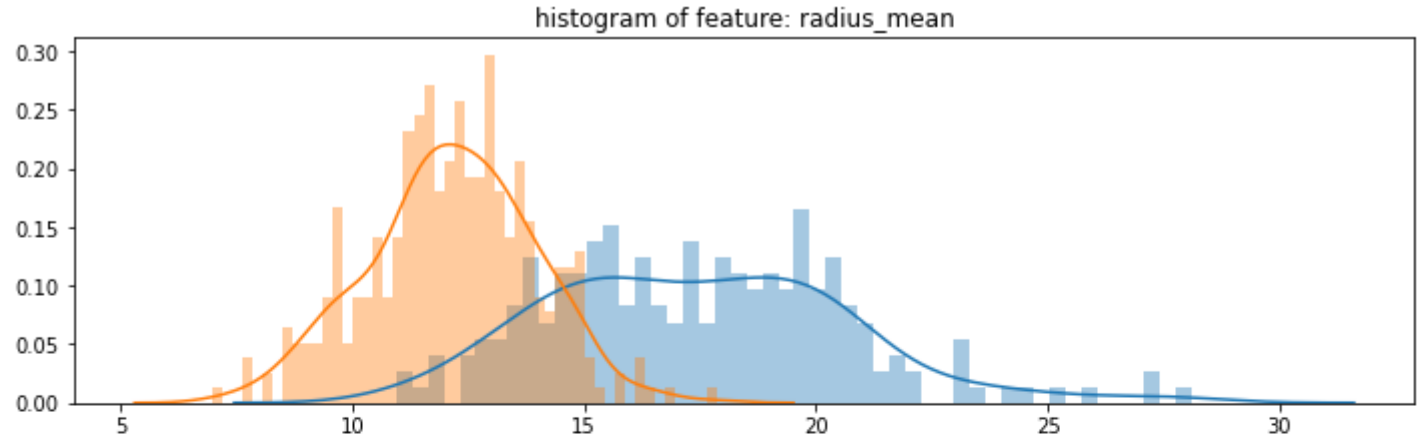
```
# how the diagnosis area_worst differs between the two types.

print ("Malignant")
print (train_data.area_worst[train_data.diagnosis == "M"].describe())
print ()
print ("Benign")
print (train_data.area_worst[train_data.diagnosis == "B"].describe())
```

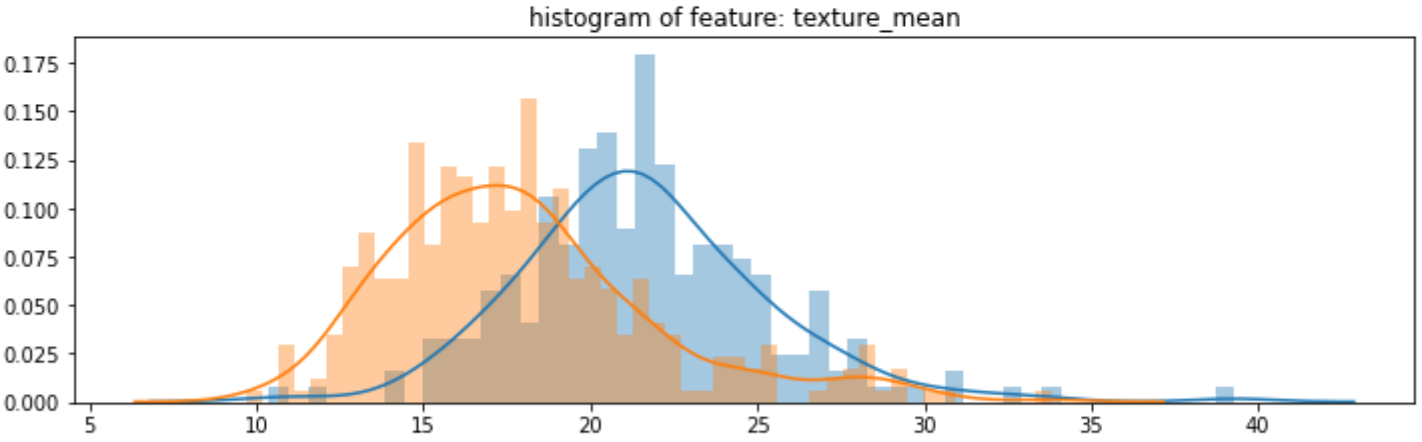
```
Malignant
count     212.000000
mean     1422.286321
std       597.967743
min       508.100000
25%       970.300000
50%      1303.000000
75%      1712.750000
max      4254.000000
Name: area_worst, dtype: float64

Benign
count     357.000000
mean      558.899440
std       163.601424
min       185.200000
25%       447.100000
50%       547.400000
75%       670.000000
max      1210.000000
Name: area_worst, dtype: float64
```

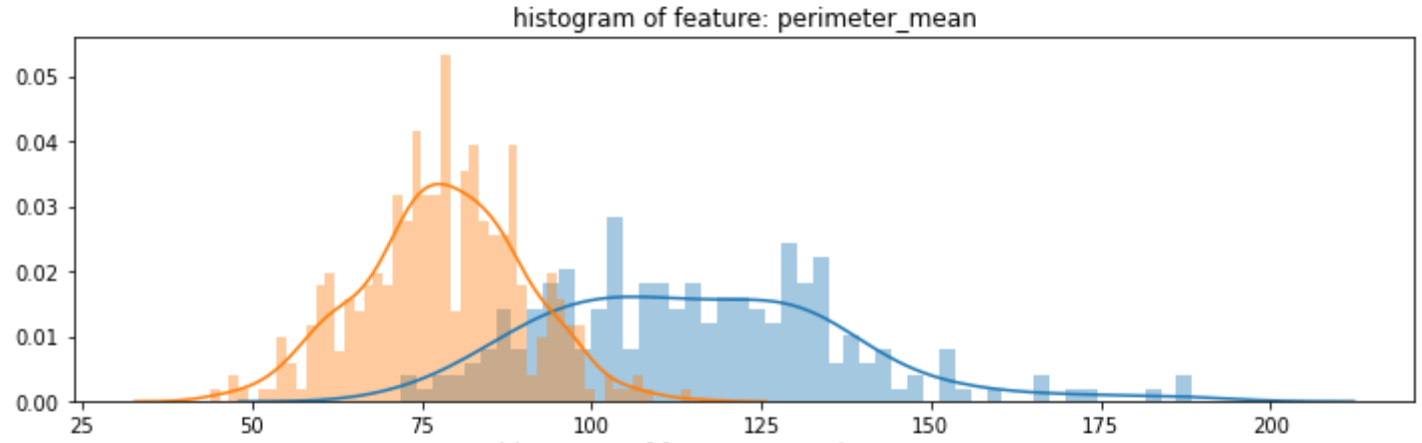
```
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(12,4))

bins = 30

ax1.hist(train_data.area_worst[train_data.diagnosis == "M"], bins = bins)
ax1.set_title('Malignant')

ax2.hist(train_data.area_worst[train_data.diagnosis == "B"], bins = bins)
ax2.set_title('Benign')

plt.xlabel('Area Worst')
plt.ylabel('Number of Diagnosis')
plt.yscale('log')
plt.show()

# Doesn't look much different than the last graph
```

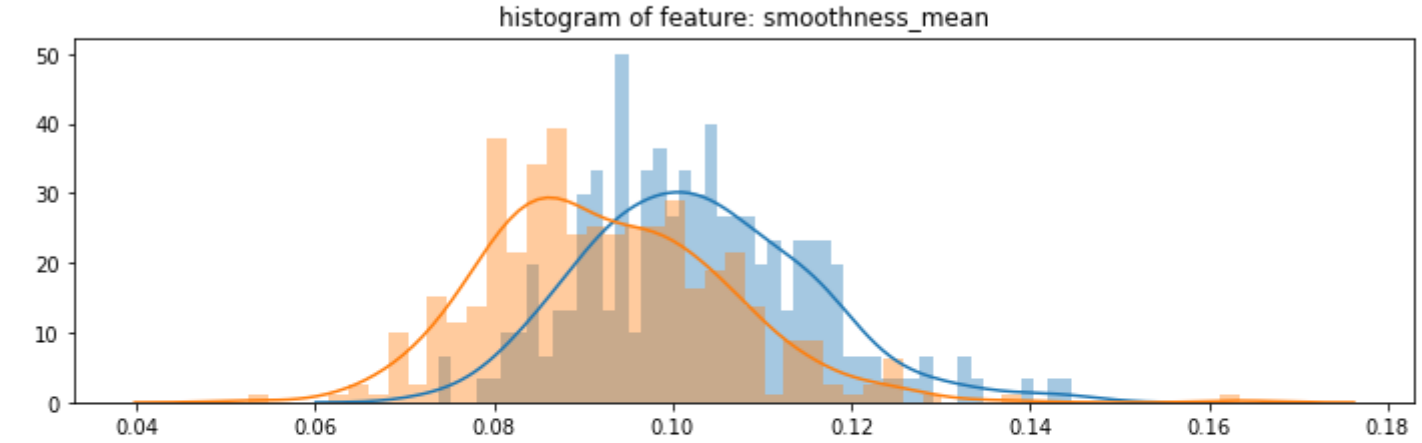


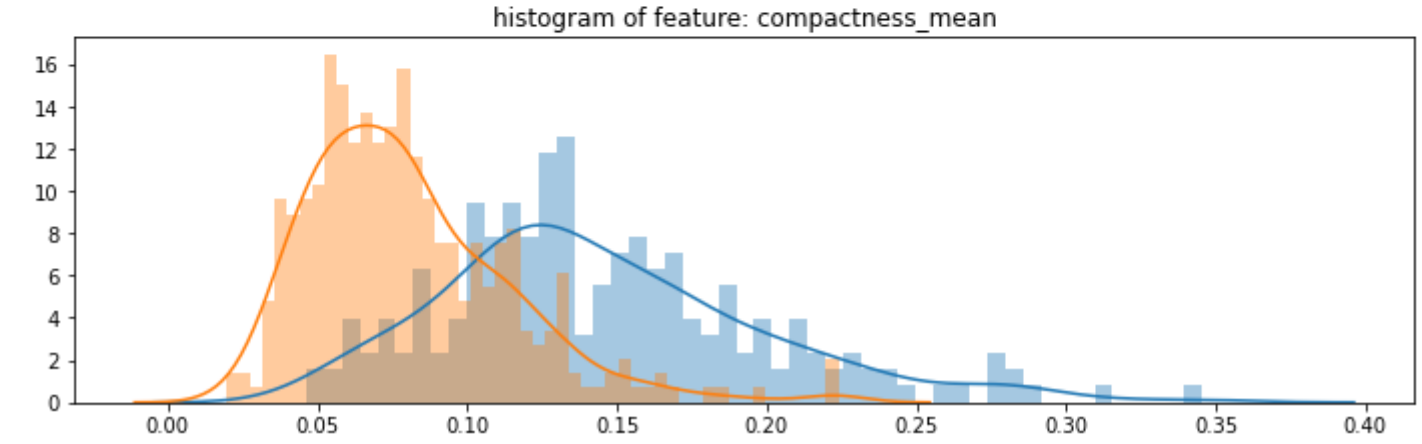

```
#Selecting only the rest of the features

r_data = train_data.drop([idKey, areaMeanKey, areaWorstKey, diagnosisKey], axis=1)
r_features = r_data.columns


plt.figure(figsize=(12,28*4))
gs = gridspec.GridSpec(28, 1)
for i, cn in enumerate(r_data[r_features]):
    ax = plt.subplot(gs[i])
    sns.distplot(train_data[cn][train_data.diagnosis == "M"], bins=50)
    sns.distplot(train_data[cn][train_data.diagnosis == "B"], bins=50)
    ax.set_xlabel('')
    ax.set_title('histogram of feature: ' + str(cn))
plt.show()
```

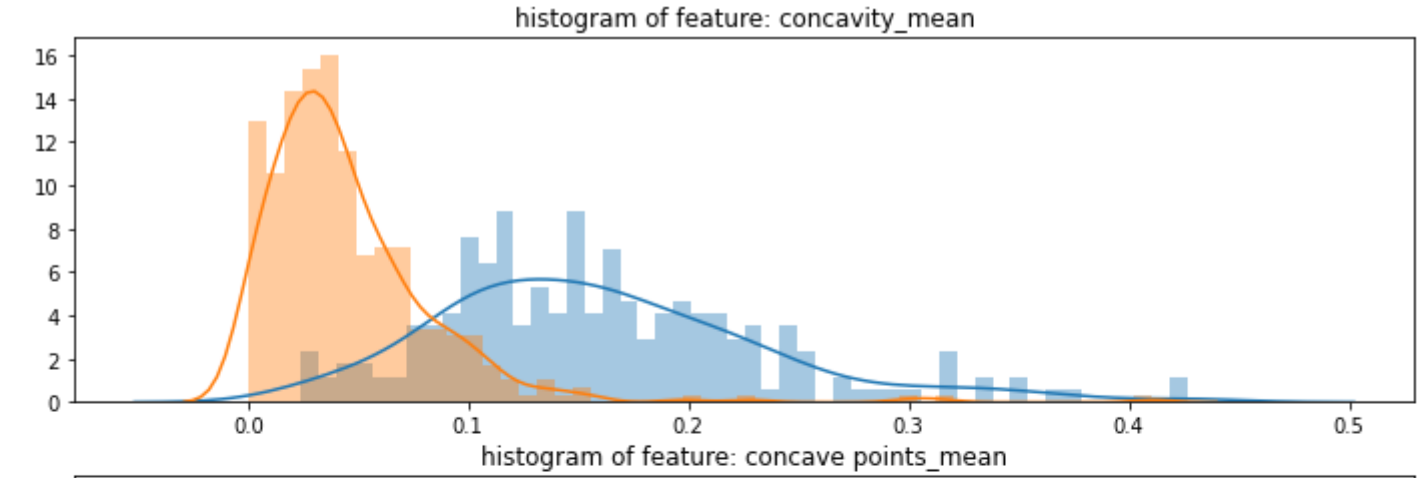

histogram of feature: radius_mean

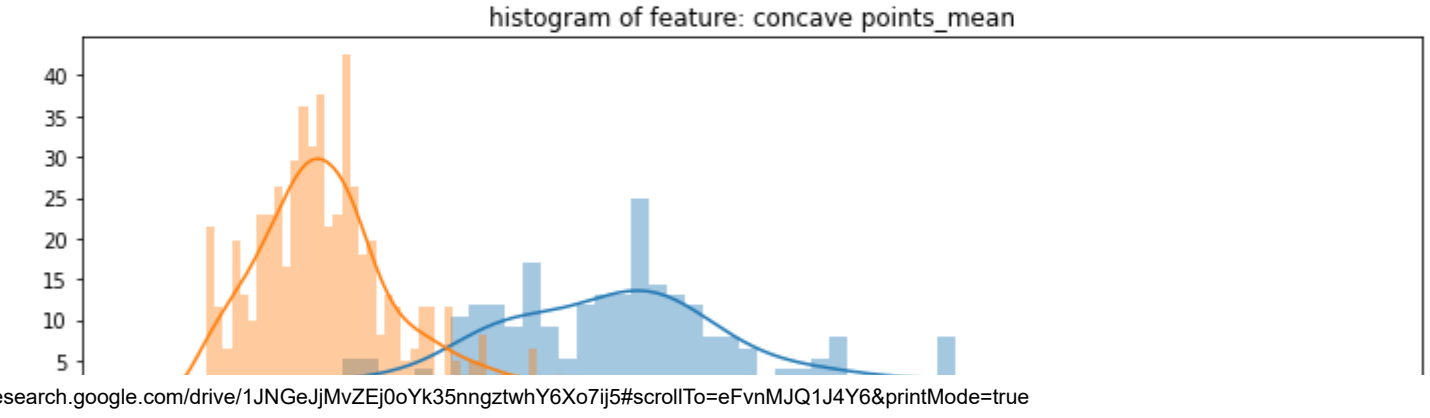

histogram of feature: texture_mean

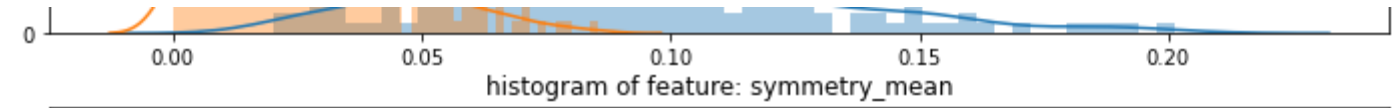

histogram of feature: perimeter_mean


histogram of feature: smoothness_mean

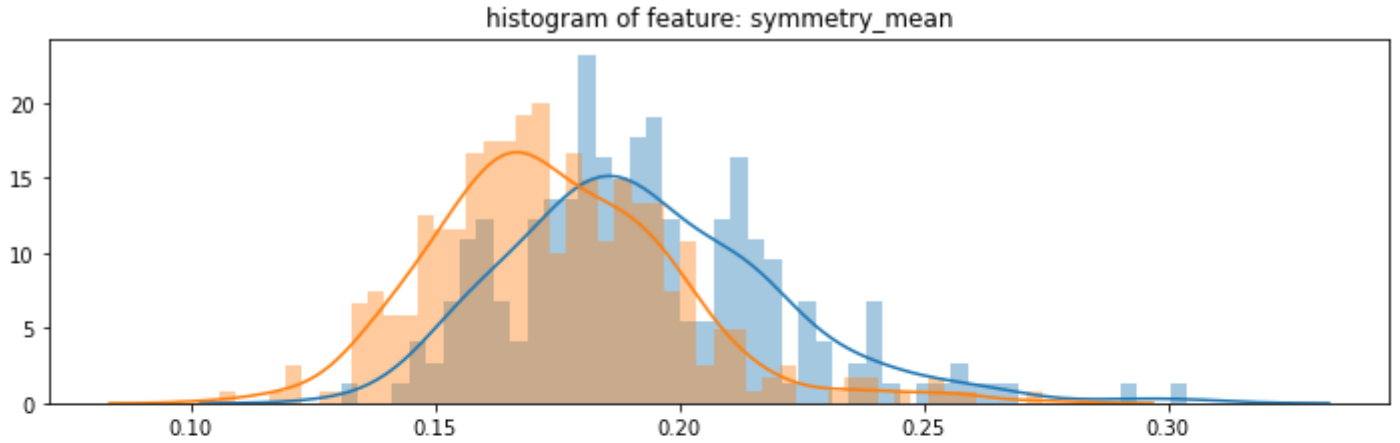

histogram of feature: compactness_mean


histogram of feature: concavity_mean

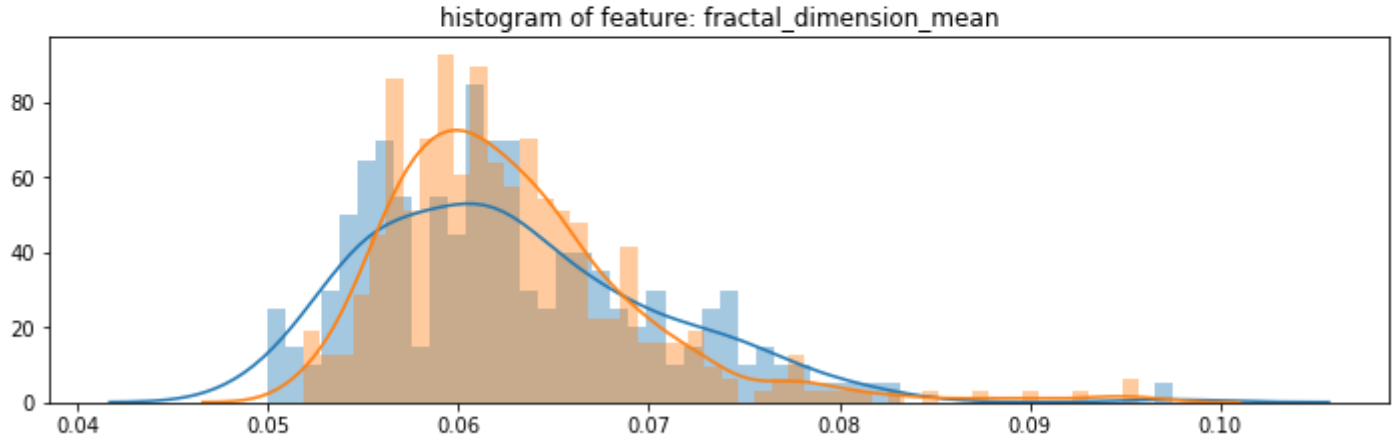

histogram of feature: concave points_mean



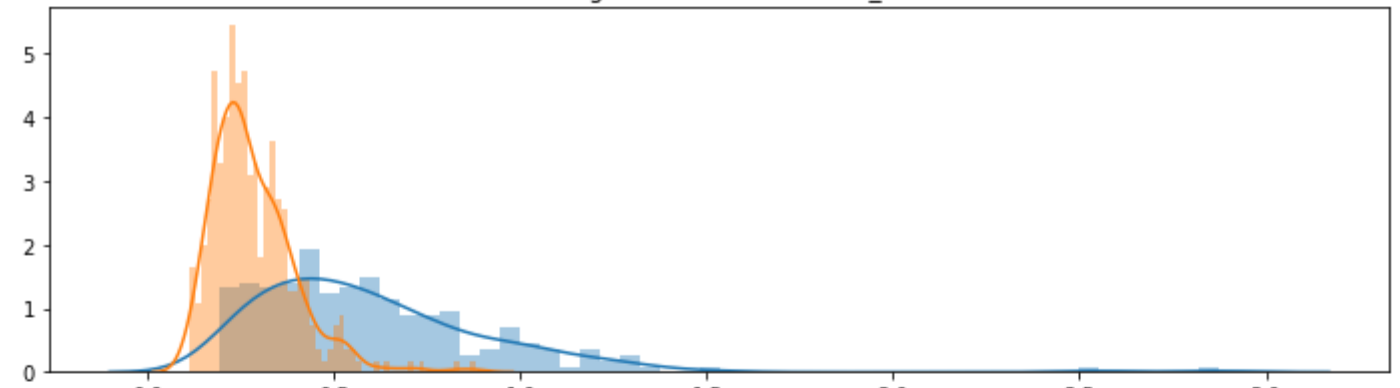
histogram of feature: symmetry_mean



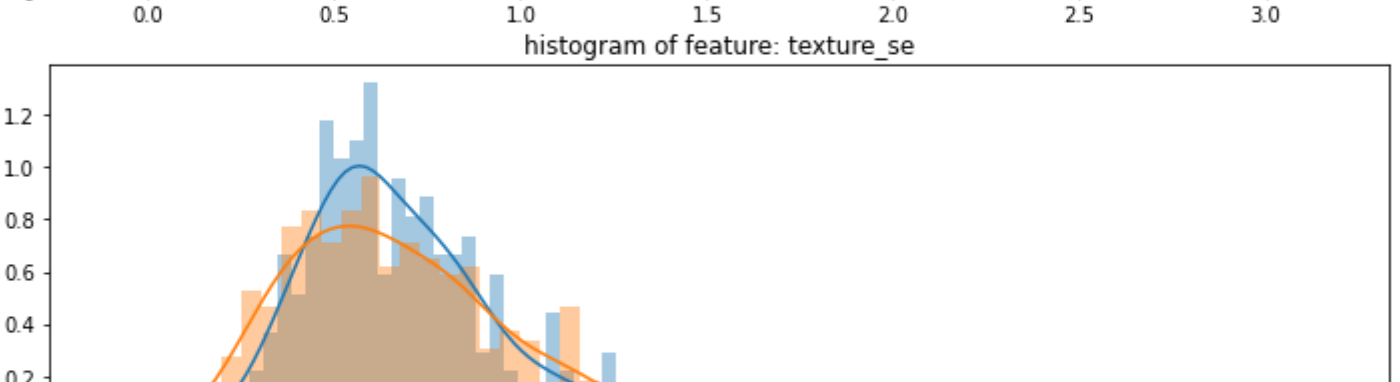
histogram of feature: fractal_dimension_mean



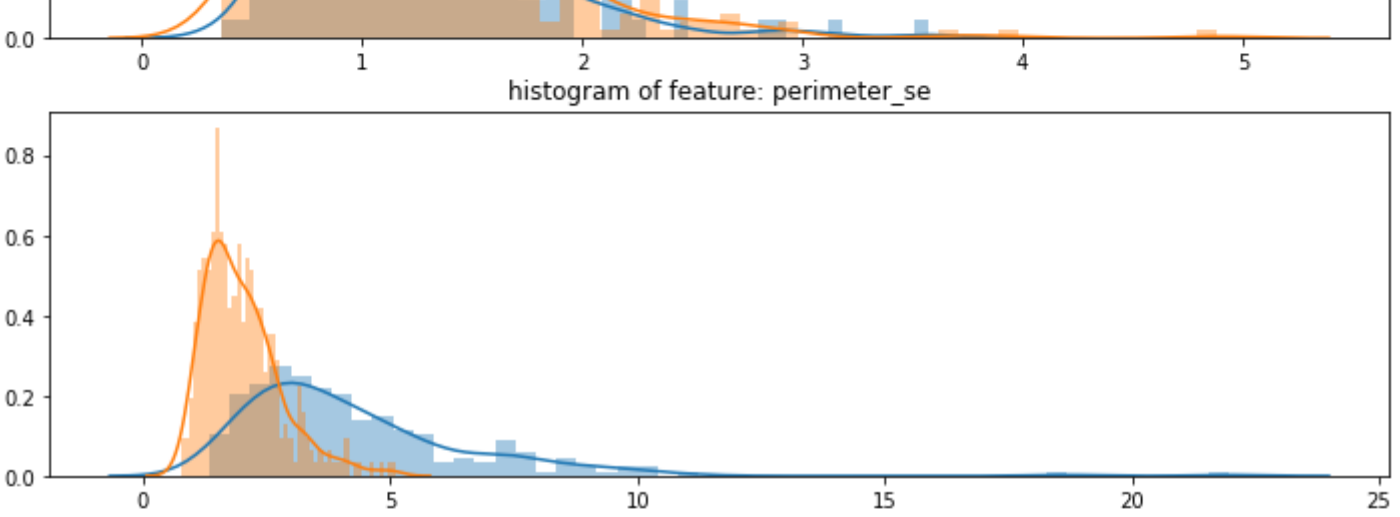
histogram of feature: radius_se



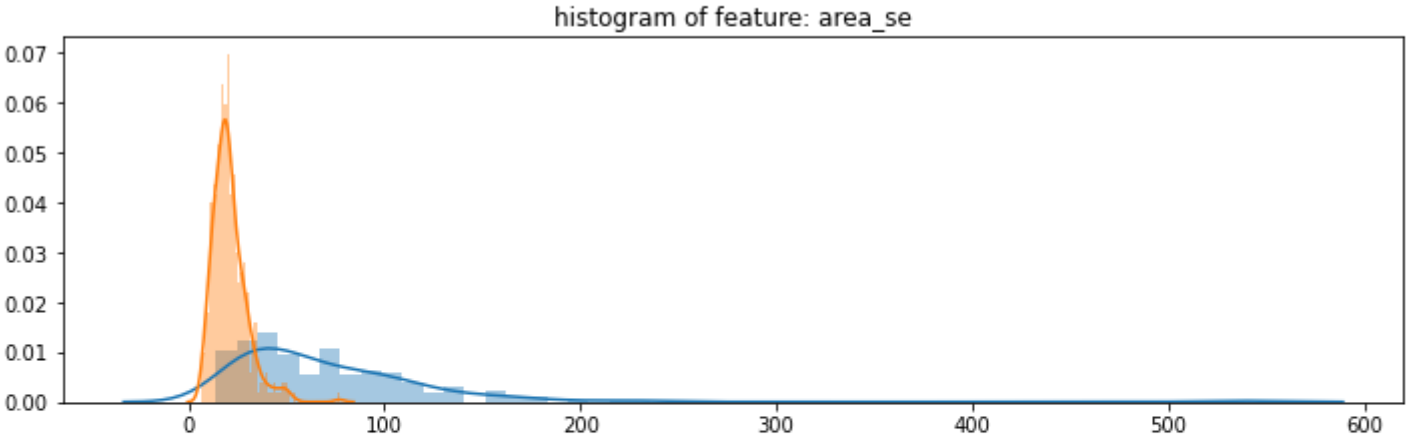
histogram of feature: texture_se



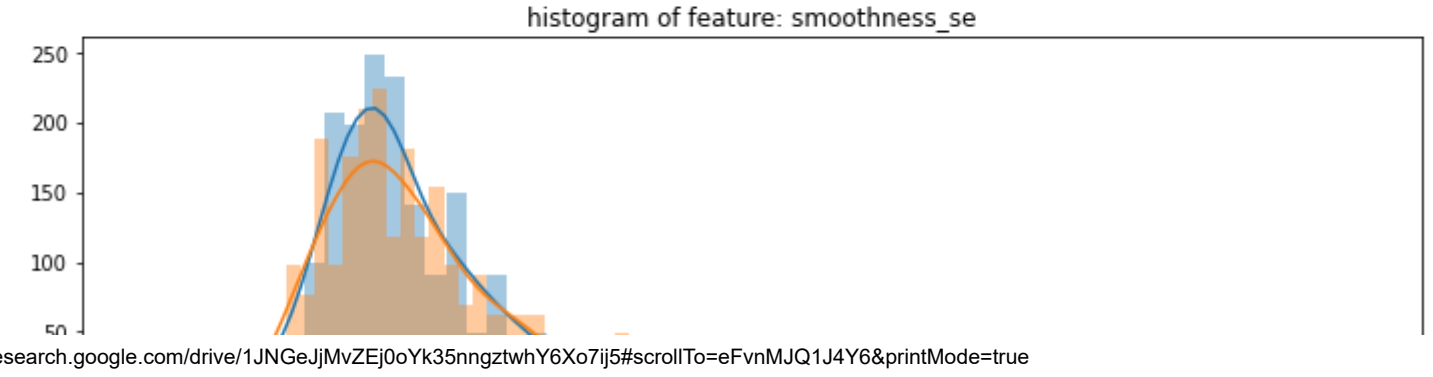
histogram of feature: perimeter_se

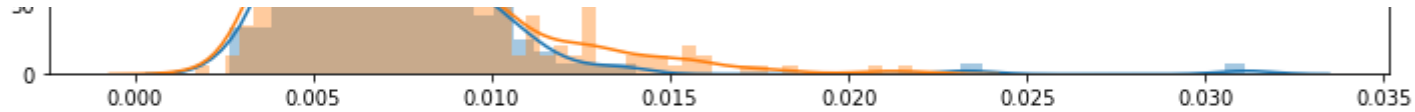


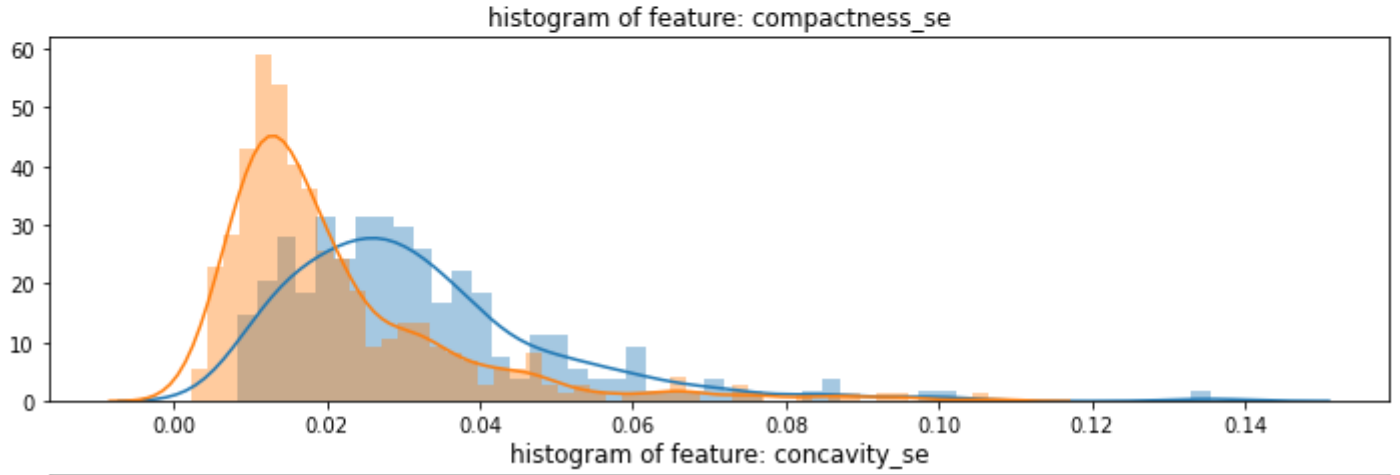
histogram of feature: area_se

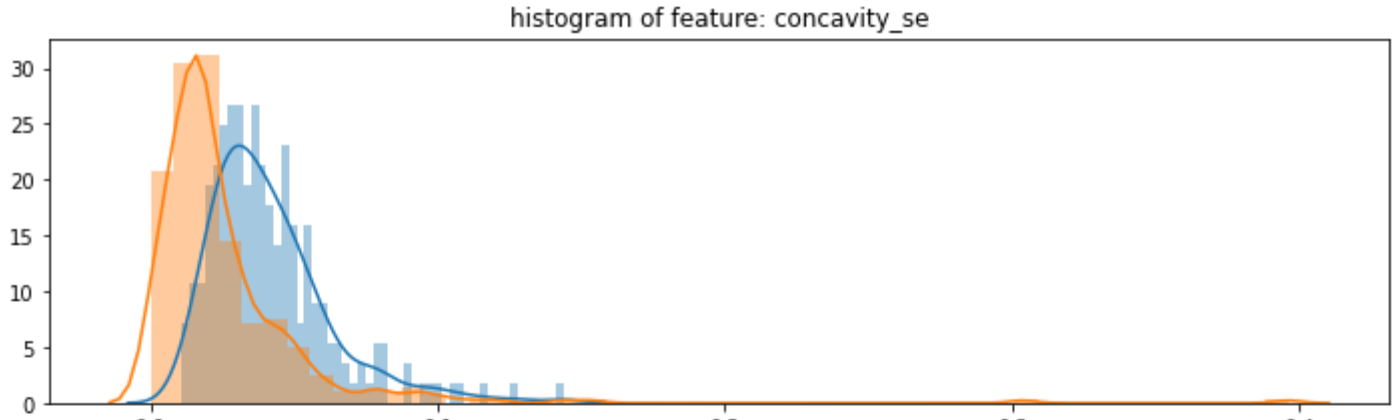


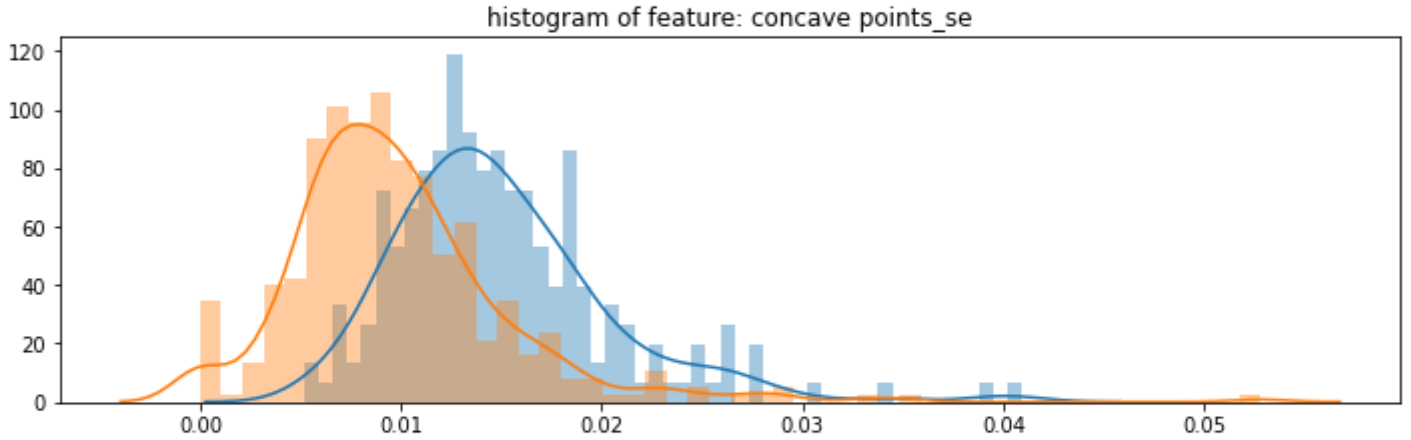
histogram of feature: smoothness_se

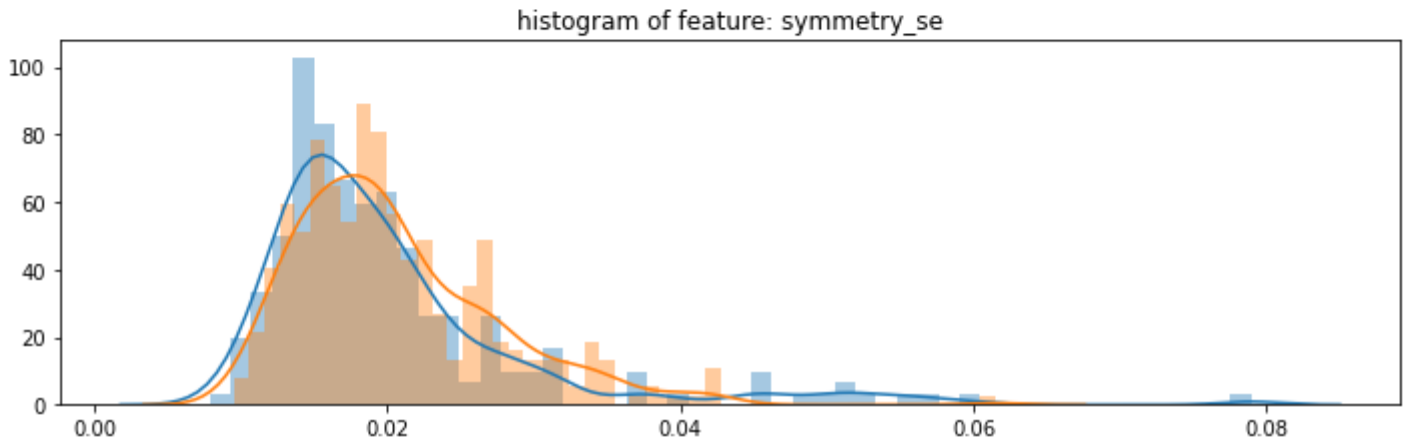


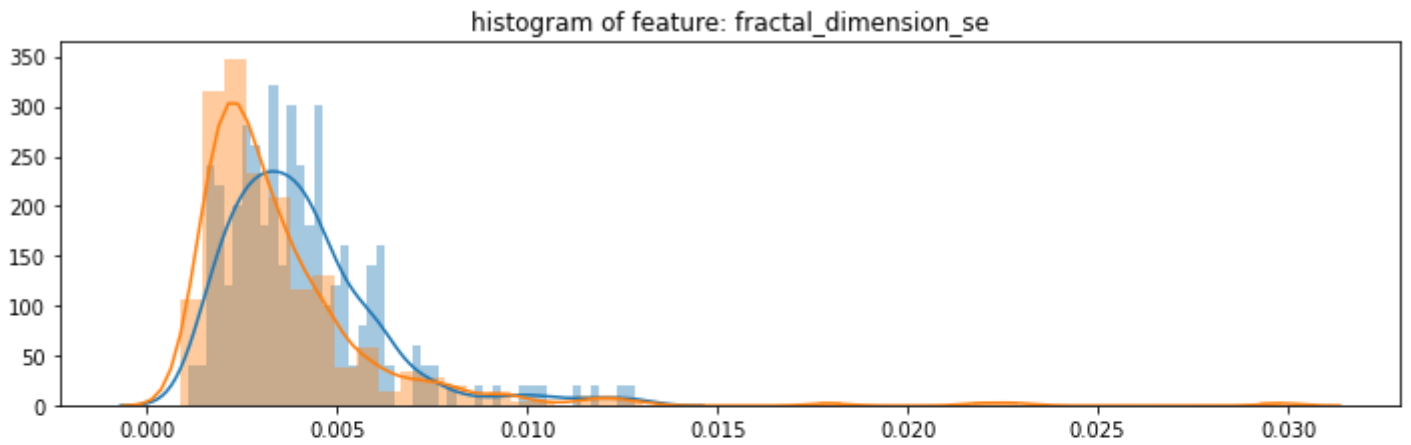

0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035

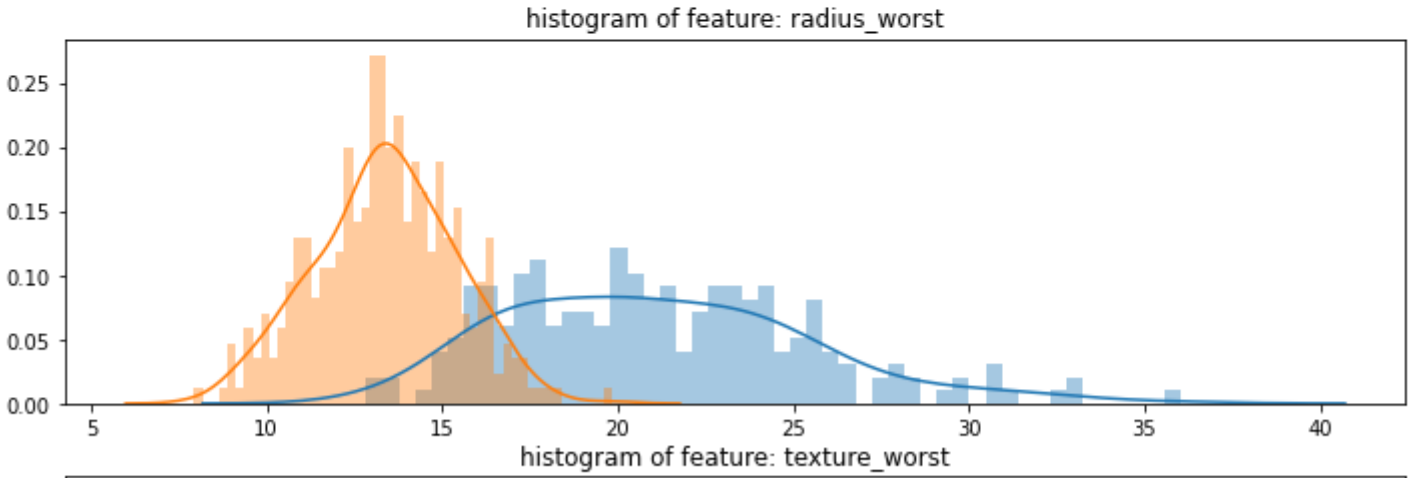

histogram of feature: compactness_se
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14

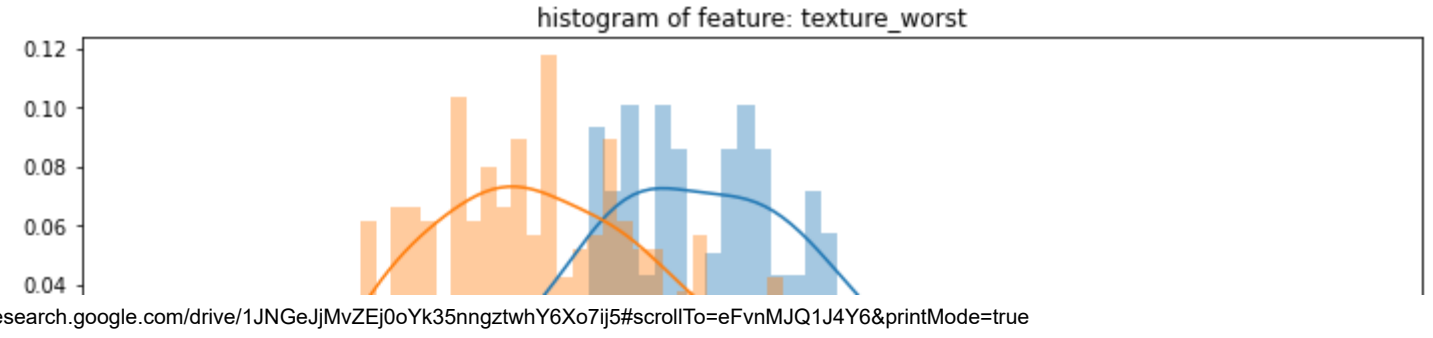

histogram of feature: concavity_se
0.0
0.1
0.2
0.3
0.4

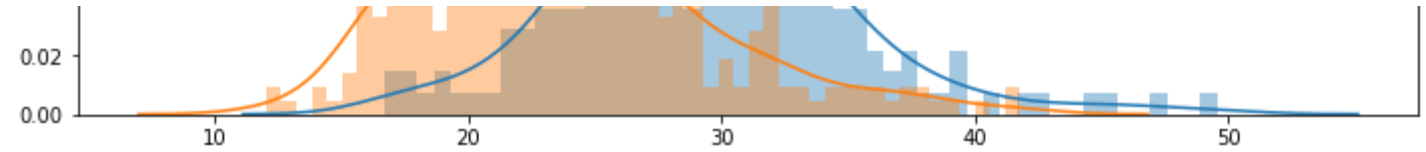

histogram of feature: concave points_se
0.00
0.01
0.02
0.03
0.04
0.05

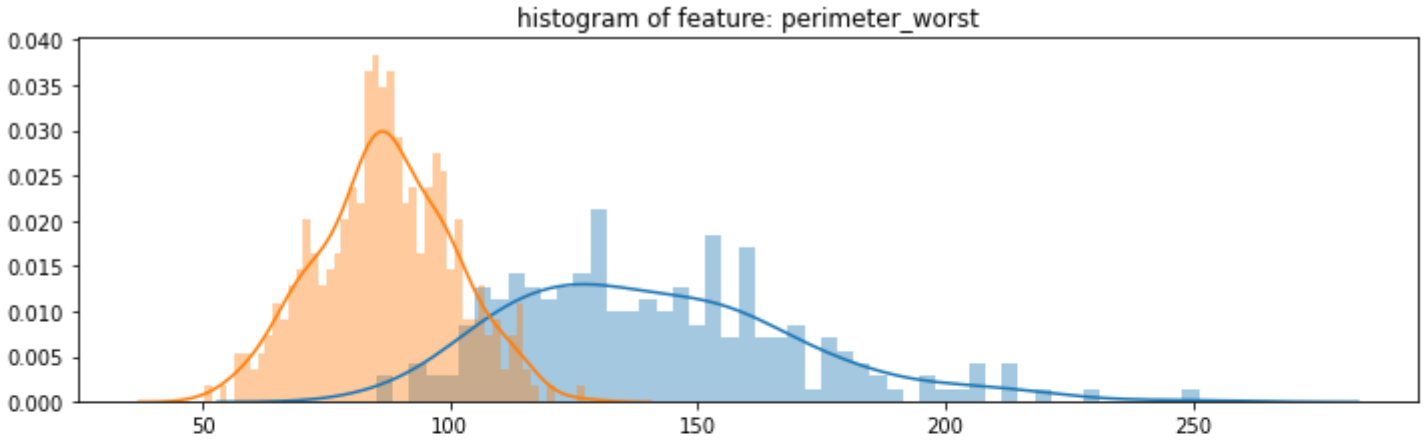

histogram of feature: symmetry_se
0.00
0.02
0.04
0.06
0.08


histogram of feature: fractal_dimension_se
0.000
0.005
0.010
0.015
0.020
0.025
0.030

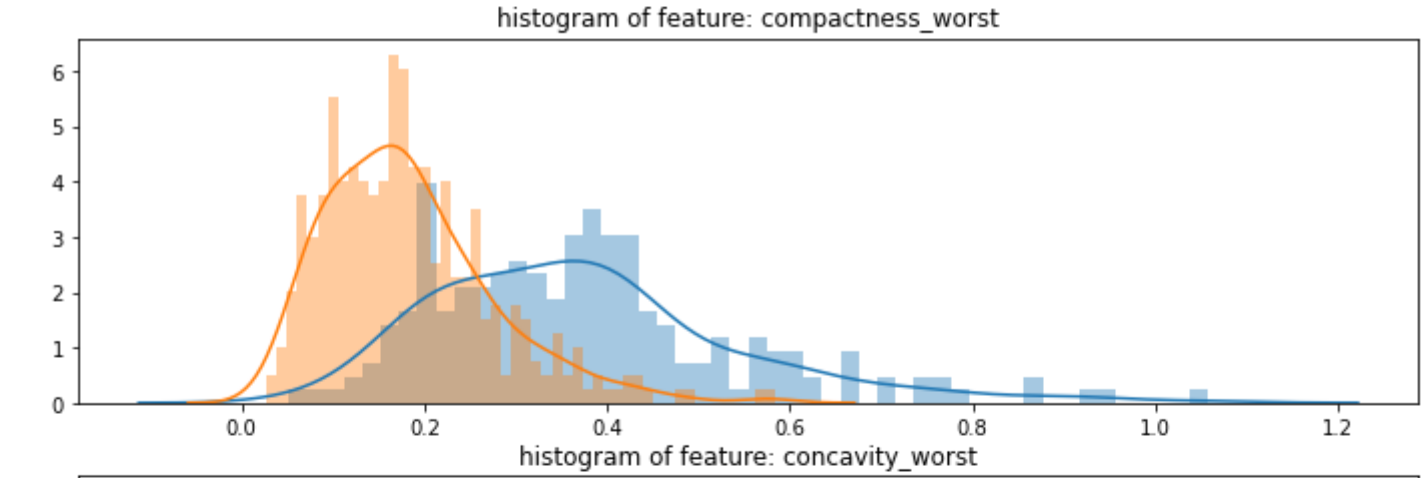

histogram of feature: radius_worst
5
10
15
20
25
30
35
40

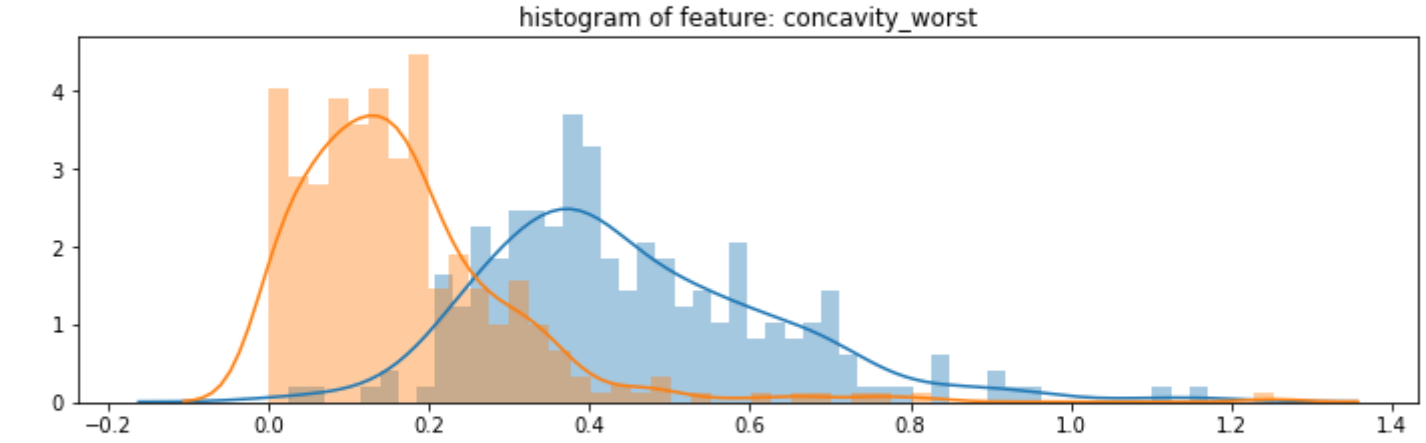

histogram of feature: texture_worst

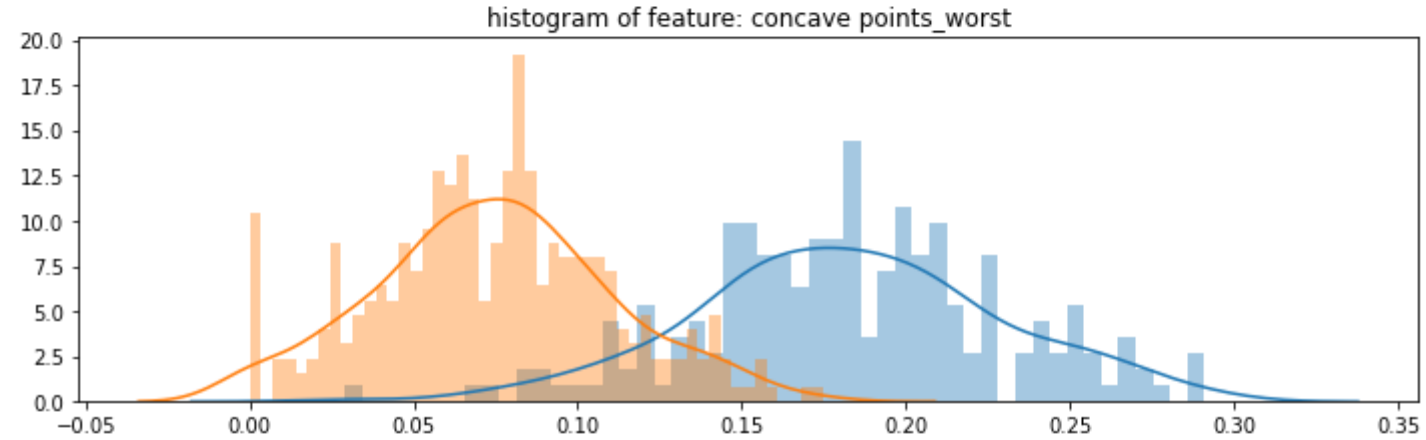


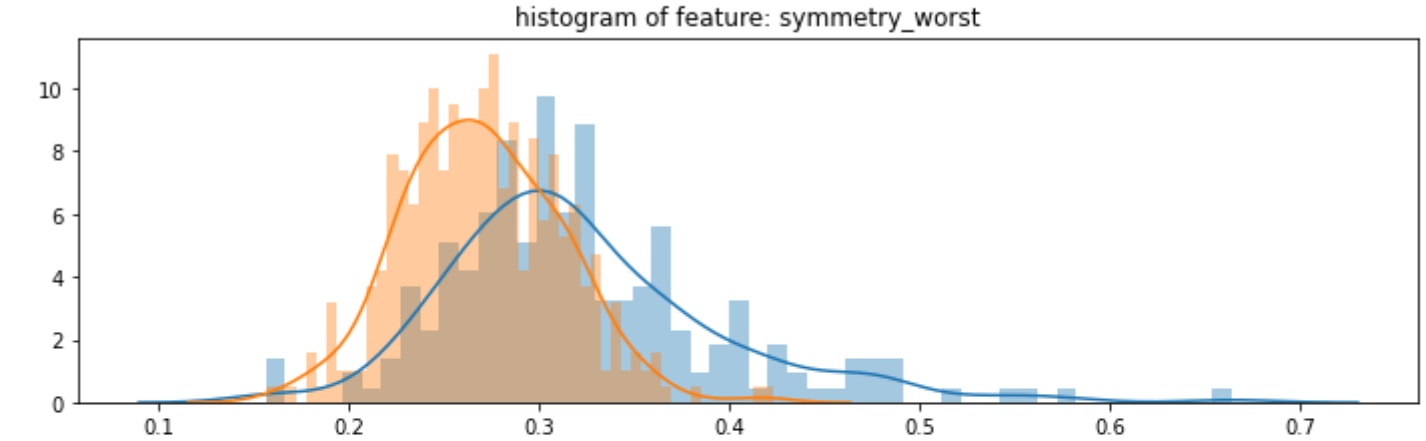

histogram of feature: perimeter_worst

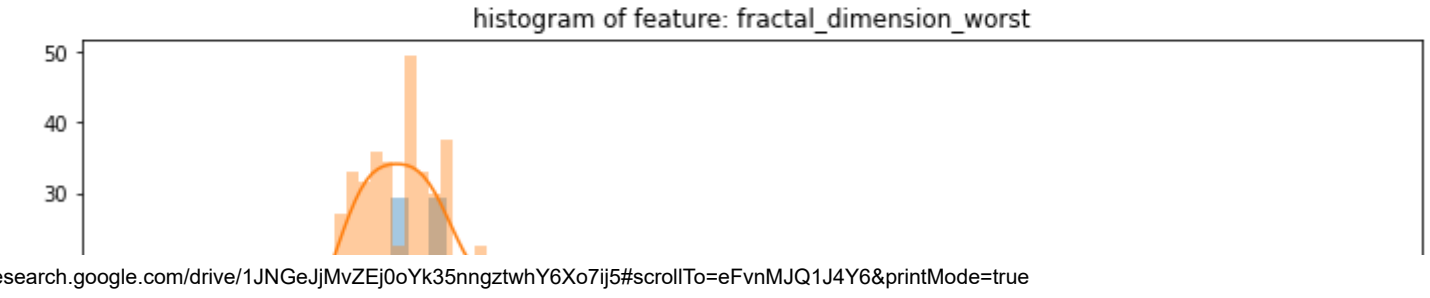
histogram of feature: smoothness_worst

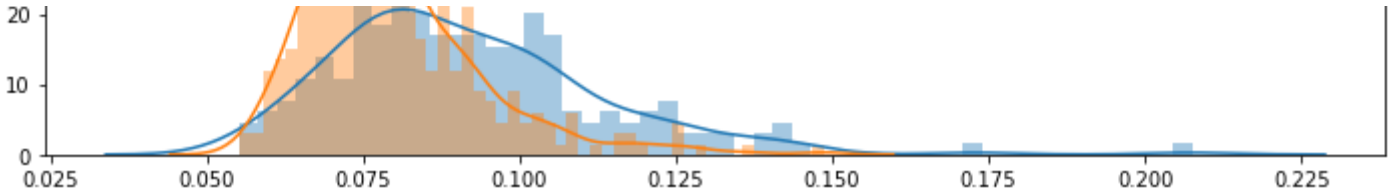

histogram of feature: compactness_worst


histogram of feature: concavity_worst


histogram of feature: concave points_worst


histogram of feature: symmetry_worst


histogram of feature: fractal_dimension_worst




```
# Update the value of diagnosis. 1 for Malignant and 0 for Benign

train_data.loc[train_data.diagnosis == "M", 'diagnosis'] = 1
train_data.loc[train_data.diagnosis == "B", 'diagnosis'] = 0


# Create a new feature for benign (non-malignant) diagnosis.

train_data.loc[train_data.diagnosis == 0, 'benign'] = 1
train_data.loc[train_data.diagnosis == 1, 'benign'] = 0


# Convert benign column type to integer

train_data['benign'] = train_data.benign.astype(int)


# Rename 'Class' to 'Malignant'

train_data = train_data.rename(columns={'diagnosis': 'malignant'})


# 212 malignant diagnosis, 357 benign diagnosis. 37.25% of diagnostics were malignant.

print(train_data.benign.value_counts())
print()
print(train_data.malignant.value_counts())
```

```
1    357
0    212
Name: benign, dtype: int64

0    357
1    212
Name: malignant, dtype: int64
```

```
pd.set_option("display.max_columns",101)
train_data.head()
```

	id	malignant	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean
0	842302	1	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001
1	842517	1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869
2	84300903	1	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974
3	84348301	1	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414
4	84358402	1	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980

```
# Create dataframes of only Malignant and Benign diagnosis.

Malignant = train_data[train_data.malignant == 1]
Benign = train_data[train_data.benign == 1]


# Set train_X equal to 80% of the malignant diagnosis.

train_X = Malignant.sample(frac=0.8)
count_Malignants = len(train_X)


# Add 80% of the benign diagnosis to train_X.

train_X = pd.concat([train_X, Benign.sample(frac = 0.8)], axis = 0)


# test_X contains all the diagnostics not in train_X.

test_X = train_data.loc[~train_data.index.isin(train_X.index)]


# Shuffle the dataframes so that the training is done in a random order.
```

```
train_X = shuffle(train_X)
test_X = shuffle(test_X)


# Add the target features to train_Y and test_Y

train_Y = train_X.malignant
train_Y = pd.concat([train_Y, train_X.benign], axis=1)

test_Y = test_X.malignant
test_Y = pd.concat([test_Y, test_X.benign], axis=1)


# Drop target features from train_X and test_X

train_X = train_X.drop(['malignant','benign'], axis = 1)
test_X = test_X.drop(['malignant','benign'], axis = 1)


# Check to ensure all of the training/testing dataframes are of the correct length

print(len(train_X))
print(len(train_Y))
print(len(test_X))
print(len(test_Y))
```

```
456
113
113
```

```
# Names of all of the features in train_X

features = train_X.columns.values


# Transform each feature in features so that it has a mean of 0 and standard deviation of 1
# This helps with training the softmax algorithm

for feature in features:
    mean, std = train_data[feature].mean(), train_data[feature].std()
    train_X.loc[:, feature] = (train_X[feature] - mean) / std
    test_X.loc[:, feature] = (test_X[feature] - mean) / std
```

Train the Neural Network

```
# Parameters

learning_rate = 0.005
training_dropout = 0.9
display_step = 1
training_epochs = 5
batch_size = 100
accuracy_history = []
cost_history = []
valid_accuracy_history = []
valid_cost_history = []


# Number of input nodes

input_nodes = train_X.shape[1]

# Number of labels (malignant and benign)

num_labels = 2


# Split the testing data into validation and testing sets

split = int(len(test_Y)/2)

train_size = train_X.shape[0]
n_samples = train_Y.shape[0]

input_X = train_X.to_numpy()
input_Y = train_Y.to_numpy()
input_X_valid = test_X.to_numpy()[:split]
```

```
input_Y_valid = test_Y.to_numpy()[:split]
input_X_test = test_X.to_numpy()[split:]
input_Y_test = test_Y.to_numpy()[split:]


def calculate_hidden_nodes(nodes):
    return (((2 * nodes)/3) + num_labels)


# Number of nodes in each hidden layer

hidden_nodes1 = round(calculate_hidden_nodes(input_nodes))
hidden_nodes2 = round(calculate_hidden_nodes(hidden_nodes1))
hidden_nodes3 = round(calculate_hidden_nodes(hidden_nodes2))
print(input_nodes, hidden_nodes1, hidden_nodes2, hidden_nodes3)
```

```
31 23 17 13
```

```
# Percent of nodes to keep during dropout.

import tensorflow.compat.v1 as tf
tf.disable_v2_behavior() # solution to overcome the placeholder issue in Tensorflow 2.0

pkeep = tf.placeholder(tf.float32)
```

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow_core/python/compat/v2_compat.py:65: disable_resource_v
Instructions for updating:
non-resource variables are not supported in the long term
```

```
# Input

x = tf.placeholder(tf.float32, [None, input_nodes])

# Layer 1

W1 = tf.Variable(tf.truncated_normal([input_nodes, hidden_nodes1], stddev = 0.1))
b1 = tf.Variable(tf.zeros([hidden_nodes1]))
y1 = tf.nn.relu(tf.matmul(x, W1) + b1)

# Layer 2

W2 = tf.Variable(tf.truncated_normal([hidden_nodes1, hidden_nodes2], stddev = 0.1))
b2 = tf.Variable(tf.zeros([hidden_nodes2]))
y2 = tf.nn.relu(tf.matmul(y1, W2) + b2)

# Layer 3

W3 = tf.Variable(tf.truncated_normal([hidden_nodes2, hidden_nodes3], stddev = 0.1))
b3 = tf.Variable(tf.zeros([hidden_nodes3]))
y3 = tf.nn.relu(tf.matmul(y2, W3) + b3)
y3 = tf.nn.dropout(y3, pkeep)

# Layer 4

W4 = tf.Variable(tf.truncated_normal([hidden_nodes3, 2], stddev = 0.1))
b4 = tf.Variable(tf.zeros([2]))
y4 = tf.nn.softmax(tf.matmul(y3, W4) + b4)
```

```
WARNING:tensorflow:From <ipython-input-39-6b22a9dfa67f>:21: calling dropout (from tensorflow.python.ops.nn_ops) with keep_prob i
Instructions for updating:
Please use `rate` instead of `keep_prob`. Rate should be set to `rate = 1 - keep_prob`.
```

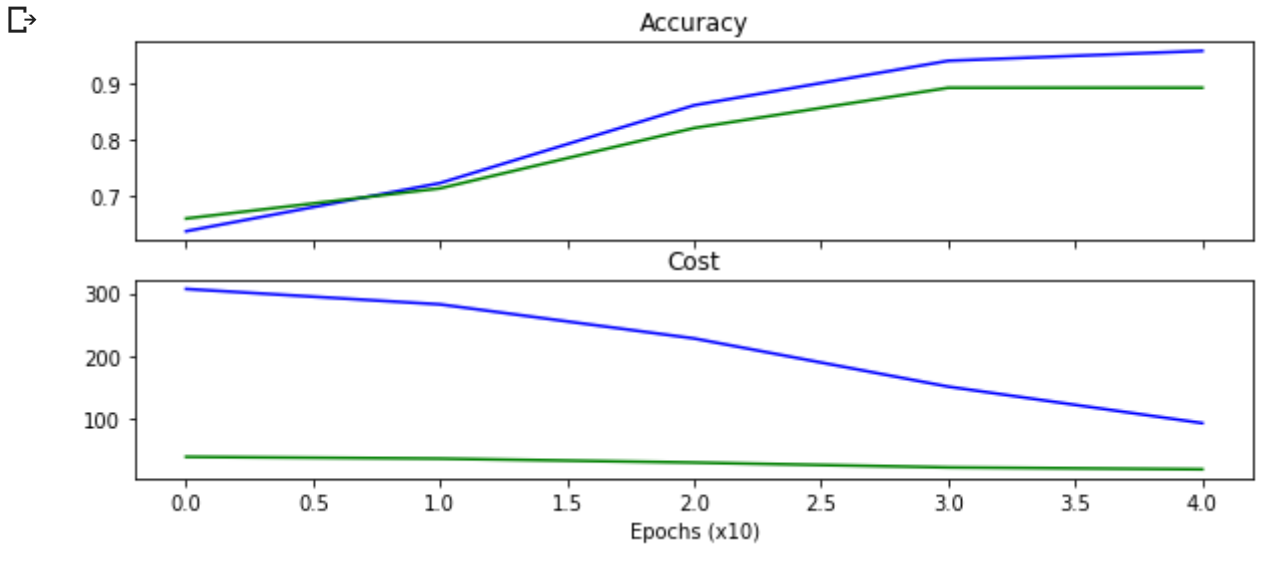
```
# Output

y = y4
y_ = tf.placeholder(tf.float32, [None, num_labels])


# Minimize error using cross entropy
# Adam optimiser

import datetime

cost = -tf.reduce_sum(y_ * tf.log(y))

optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost)
```

```
log_dir = "logs/fit/" + datetime.datetime.now().strftime("%Y%m%d-%H%M%S")
tensorboard_callback = tf.keras.callbacks.TensorBoard(log_dir=log_dir, histogram_freq=1)


# Test the model

correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))

# Calculate accuracy

accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))




# Initializing the variables

init = tf.global_variables_initializer()


with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

for epoch in range(training_epochs):
  for batch in range(int(n_samples/batch_size)):
    batch_x = input_X[batch * batch_size : (1 + batch) * batch_size]
    batch_y = input_Y[batch * batch_size : (1 + batch) * batch_size]

sess.run([optimizer], feed_dict={x: batch_x,
                                 y_: batch_y,
                                 pkeep: training_dropout})

# Display logs

if (epoch) % display_step == 0:
  train_accuracy, newCost = sess.run([accuracy, cost],
                                     feed_dict={x: input_X, y_: input_Y,
                                     pkeep: training_dropout})

valid_accuracy, valid_newCost = sess.run([accuracy, cost],
                                         feed_dict={x: input_X_valid,
                                         y_: input_Y_valid, pkeep: 1})

print ("Epoch:", epoch, "Acc =", "{:.5f}".format(train_accuracy),
       "Cost =", "{:.5f}".format(newCost),
       "Valid_Acc =", "{:.5f}".format(valid_accuracy),
       "Valid_Cost = ", "{:.5f}".format(valid_newCost))

# Record the results of the model

accuracy_history.append(train_accuracy)
cost_history.append(newCost)
valid_accuracy_history.append(valid_accuracy)
valid_cost_history.append(valid_newCost)

# If the model does not improve after 15 logs, stop the training

if valid_accuracy < max(valid_accuracy_history) and epoch > 100:
  stop_early += 1
      if stop_early == 15:
        break
            else:
              stop_early = 0


print("Optimization Finished!")
```

```
Epoch: 0 Acc = 0.63816 Cost = 307.06464 Valid_Acc = 0.66071 Valid_Cost =  37.65114
Epoch: 1 Acc = 0.72368 Cost = 282.31601 Valid_Acc = 0.71429 Valid_Cost =  34.78037
Epoch: 2 Acc = 0.86184 Cost = 227.63763 Valid_Acc = 0.82143 Valid_Cost =  28.42972
Epoch: 3 Acc = 0.94079 Cost = 150.32190 Valid_Acc = 0.89286 Valid_Cost =  20.68654
Epoch: 4 Acc = 0.95833 Cost = 91.75507 Valid_Acc = 0.89286 Valid_Cost =  17.64664
Optimization Finished!
```

```
# Plot the accuracy and cost summaries
f, (ax1, ax2) = plt.subplots(2, 1, sharex=True, figsize=(10,4))

ax1.plot(accuracy_history, color='b') # blue
ax1.plot(valid_accuracy_history, color='g') # green
ax1.set_title('Accuracy')
```

```
ax1.set_title('Accuracy')

ax2.plot(cost_history, color='b')
ax2.plot(valid_cost_history, color='g')
ax2.set_title('Cost')

plt.xlabel('Epochs (x10)')
plt.show()
```



Conclusion

This is the best fit model in terms of accuracy. Using the 'AdamOptimizer' with all of the features in the Neural Network, the model gives a prediction accuracy of 96% and a cross-validation score of 96% for the test data set. This model performs reasonably well.