

Theory, Application and Explainability of Neural Network Models for Breast Cancer Classification

Orji Chimzurumoke

University of Houston-Downtown Department of Mathematics and Statistics

Abstract

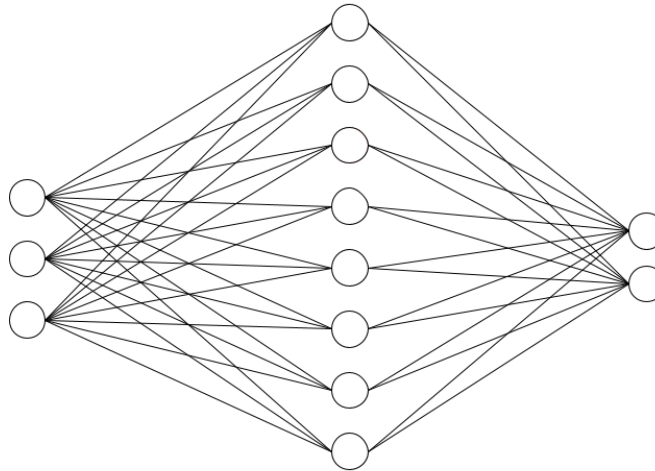
According to literature, breast cancer accounts for about 30% of all new cancer cases in women each year in the United States. As the cancerous tumors grow, cancer cells break off and travel throughout the body, forming new tumors. Some of these tumors are benign, meaning they are formed only in one spot without spreading to surrounding tissues and others are malignant tumors which are cancerous and can spread to nearby tissue. This means that being able to classify one as benign or malignant would be very helpful since early detection of breast cancer can save lives. The main goal of this project goes beyond just classifying the data into these two cancerous tumors. The study investigated and interpreted how attributes affect the classification using Local Interpretable Model-Agnostic Explanation (LIME). To start with, the project provides a general mathematical review of the theory of neural networks (NN), apply the technique (NN) classify the famous UCI machine learning repository breast cancer data. The breast cancer database has different attributes assigned to benign and malignant tumors such as clump thickness, uniformity of cell size, cell shape, marginal adhesion, single epithelial cell size, base nuclei, bland chromatin, normal nucleoli, and mitoses. From the LIME results, we were able to understand the extent to which each of the attributes listed above influence how the tumors are reclassified. This can help us explain how the attributes contribute to the formation of the cancerous tumors and hence, an detection of breast cancer.

Neural Network Fundamentals

Neural Networks mimics what happens in the brain (perceptrons)[3] and mathematically manipulate neurons passed on to other neurons. It could contain more than one layer of neurons.

- Neural Networks usage

The learning algorithm of a neural network can either be supervised or unsupervised. A neural net is said to learn supervised, if the desired output is already known. Some of the application domains of neural networks includes; Recognizing images including handwriting, Enhancing grainy images, Translating from one language to another (NLP) and Advertising.

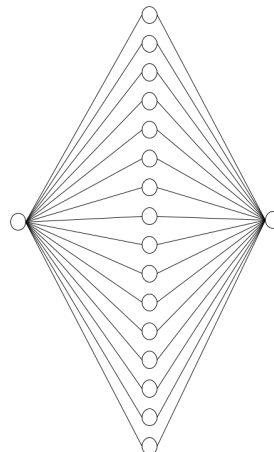


Network Architecture

A typical feed-forward neural network has a left input layer, right output layer and one hidden layer in between. Other networks might have multiple hidden layers. The input layers usually contain numerical quantities and the output layers are entries. Hidden layers manipulate the numerical quantities.

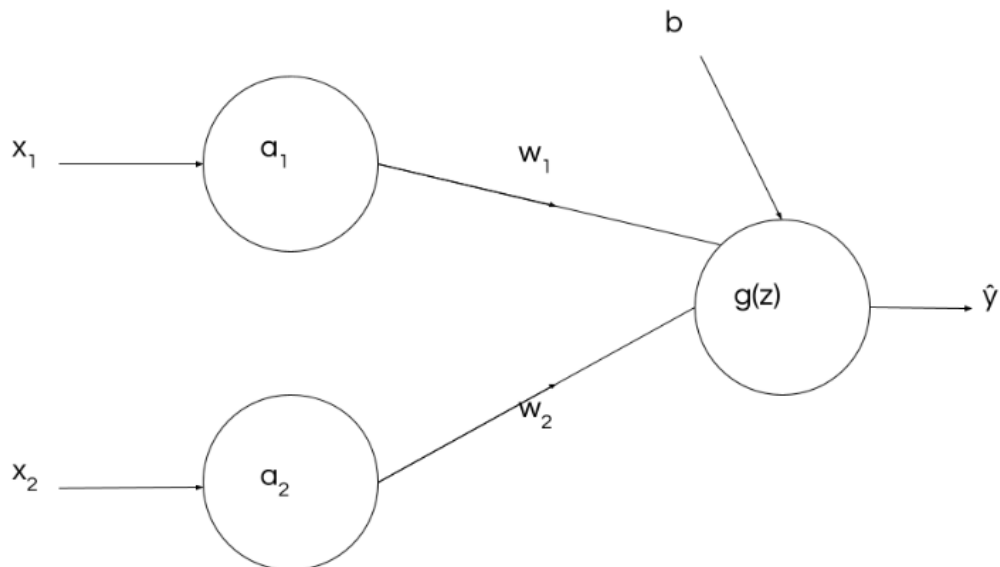
Universal Approximation Theorem

The Universal Approximation Theorem says that, under quite weak conditions, as long as you have enough nodes (in the hidden layer) then you can approximate any continuous function with a single-layer neural network to any degree of accuracy[3]. Let the single input be (x) and a single output be (y).



- The manipulations in the classical neural network follows the steps below.
 - One or two inputs, $x_1 = a_1$ and $x_2 = a_2$
 - A forecast/output \hat{y}

- Weights w_1 and w_2
- Bias b
- Function $g(z) = w_1 \cdot a_1 + w_2 \cdot a_2 + b$

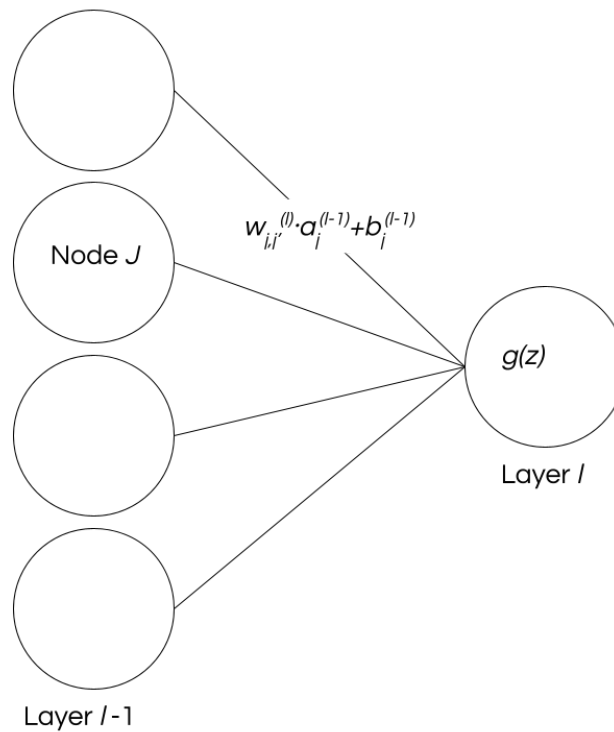


- The details in the classical neural network (Notation)
 - These quantities are then input into the first node. The input layer of the nodes contain values $a_m^{(1)}$

$$a_m^{(1)} = x_m$$
 - Inputs are x
 - Outputs are y . The hat in \hat{y} means that this is the forecast value, as opposed to the actual value for the dependent quantity from the training data.
 - Values in each node are $x_1^{(n)} \rightarrow a_1^{(1)}$
 - The subscript represents the specific node
 - The superscript represents the number of the layer (data point)

Propagation and Activation Functions

Propagation



- In going from one layer to the next the first thing that happens is a linear combination of the values in the nodes

$$\sum_{j=1}^{J_{l-1}} W_{j,j'}^{(l)} \cdot a_j^{(l-1)} + b_{j'}^{(l)}$$

- J_l means the number of nodes in layer l . Expression $z_{j'}^{(l)}$
 - Two layers as $l - 1$ and l
 - Left node labelled l and right node labelled j' (j prime)
$$z^{(l)} = W^{(l)} \cdot a^{(l-1)} + b^{(l)}$$
- Matrix $W^{(l)}$ contains all the multiplicative parameters, i.e. weights and bias

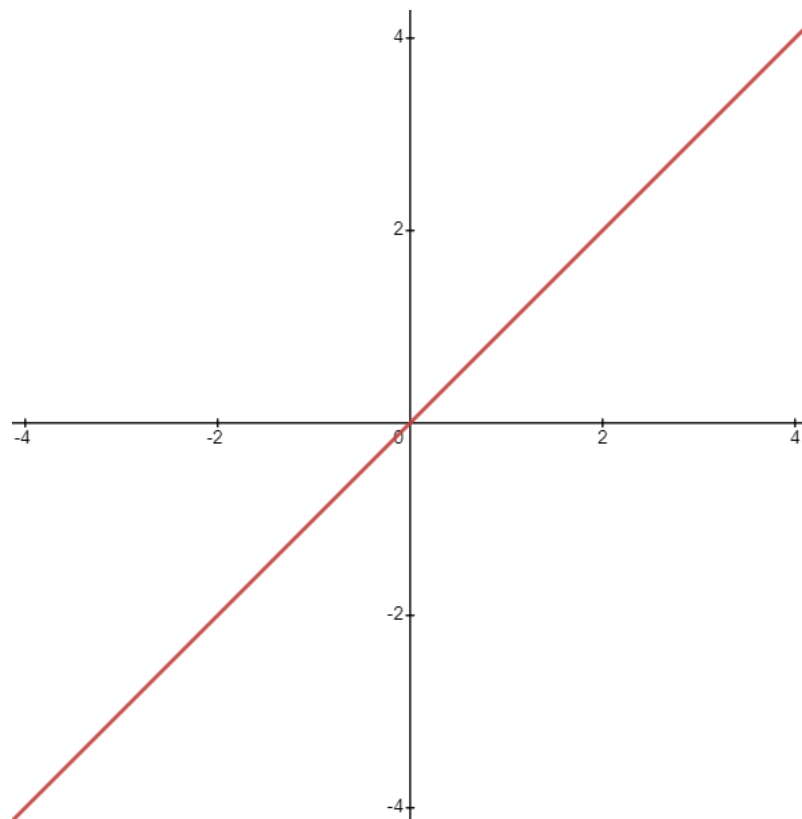
The activation function

An Activation Function decides whether a neuron should be activated or not. This means that it will decide whether the neuron's input to the network is important or not in the process of prediction using simpler mathematical operations.

$$a^{(l)} = g^{(l)}(z^{(l)}) = (W^{(l)} \cdot a^{(l-1)} + b^{(l)})$$

- Function of a vector means taking the function of each entry

Common Activation Functions

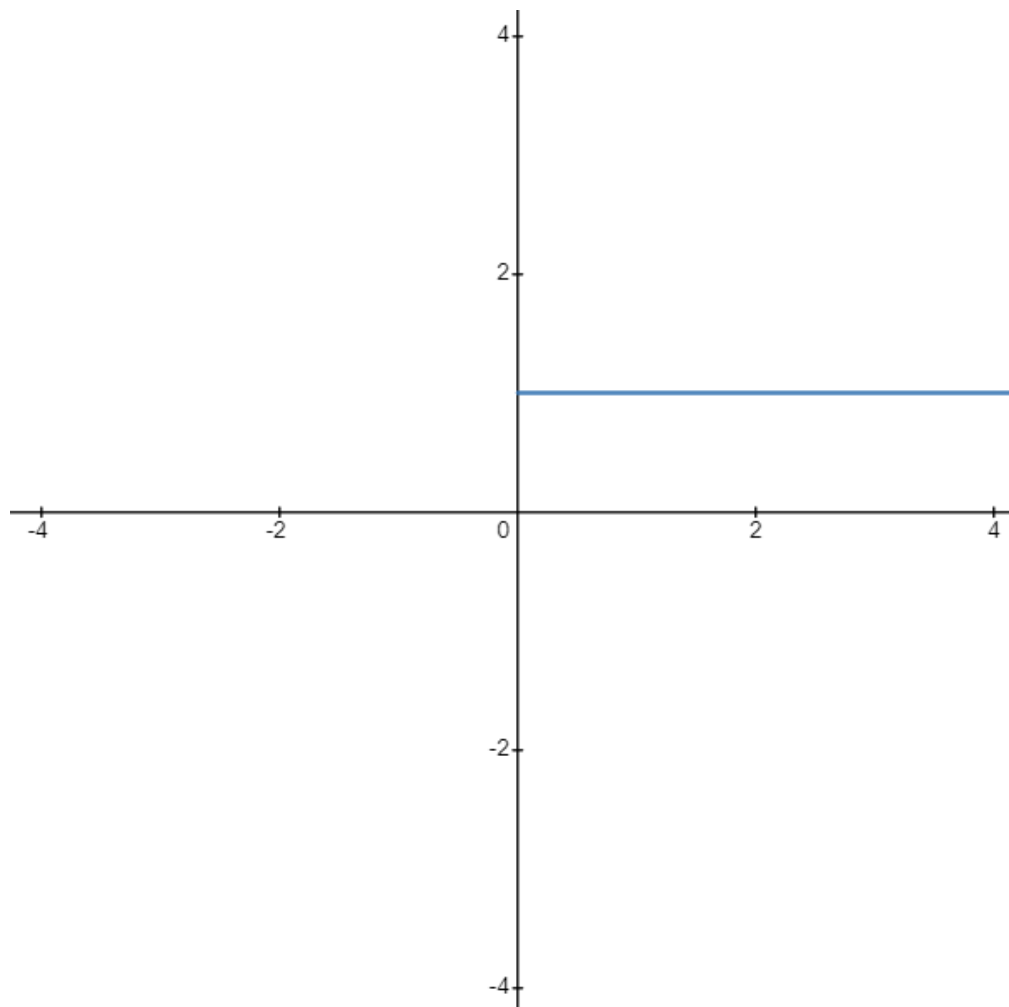


- **Linear function**

$$g(x) = x$$

- Gradient is one everywhere and this can cause problems with gradient descent
- Misses essential non-linear transformation nature of neural networks

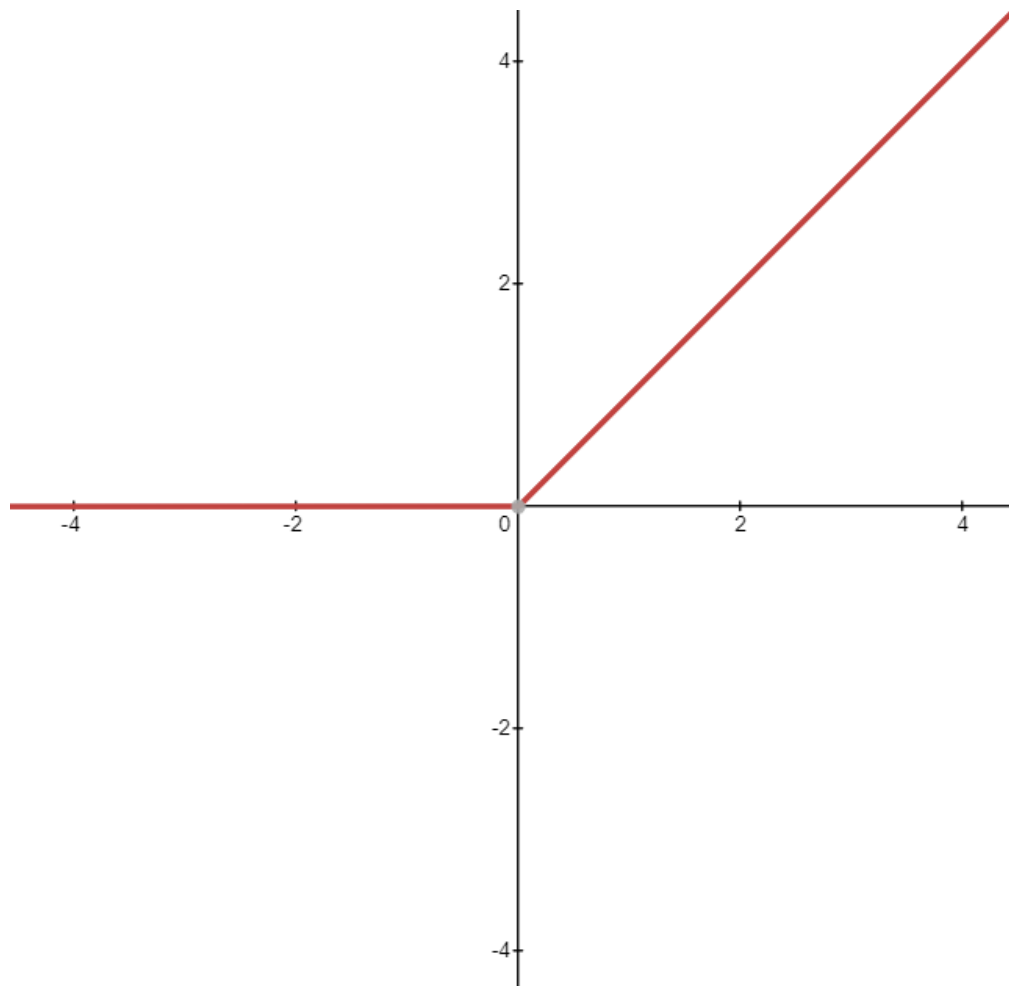
Step function / Hard limit



$$g(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

- The step function behaves like the biological activation function
- The signal either gets through as a fixed quantity, or it dies
- Leaves no room for a probabilistic interpretation of the signal for example.
- Zero gradient everywhere except at a point, where it is infinite, messing up gradient descent

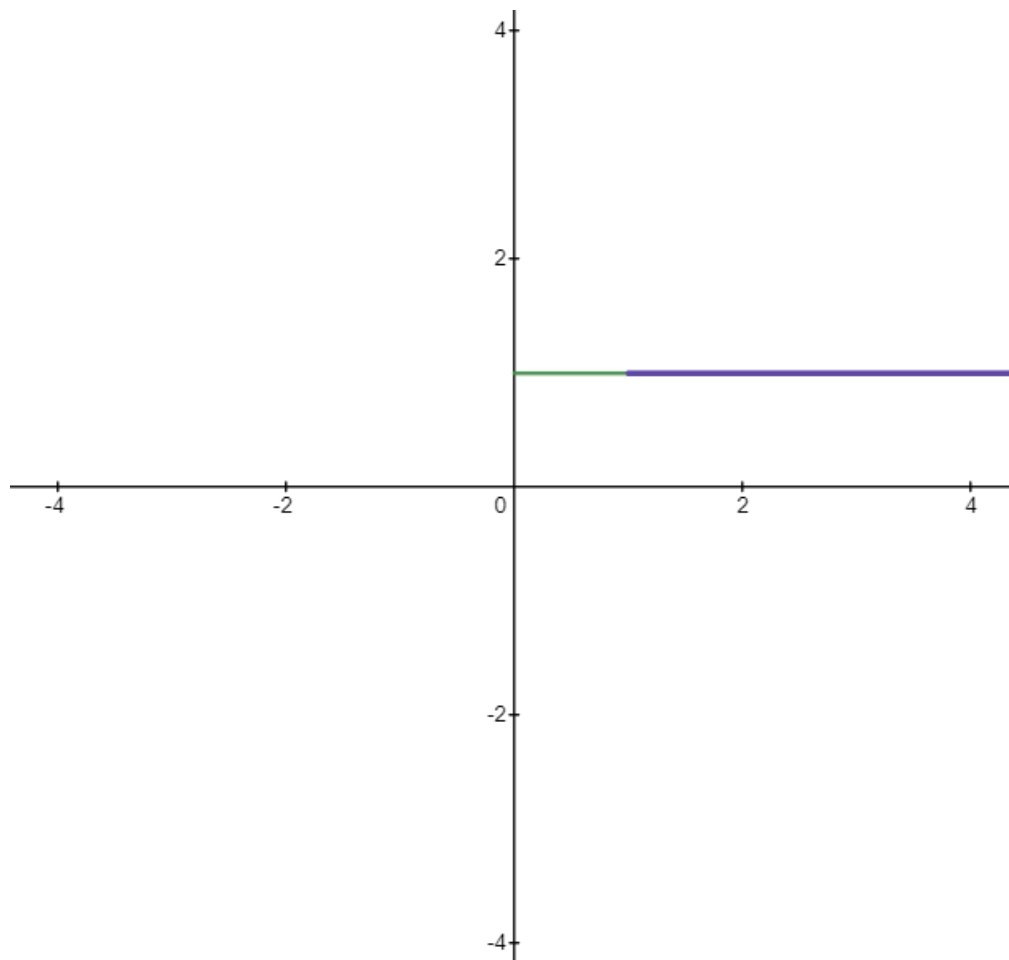
Positive linear / Re-LU function



$$g(x) = \max(0, x)$$

- Re-LU stands for Rectified Linear Units
- One of the most common used activation function
- Sufficiently non-linear even with many interacting nodes
- The signal either passes through untouched or dies completely

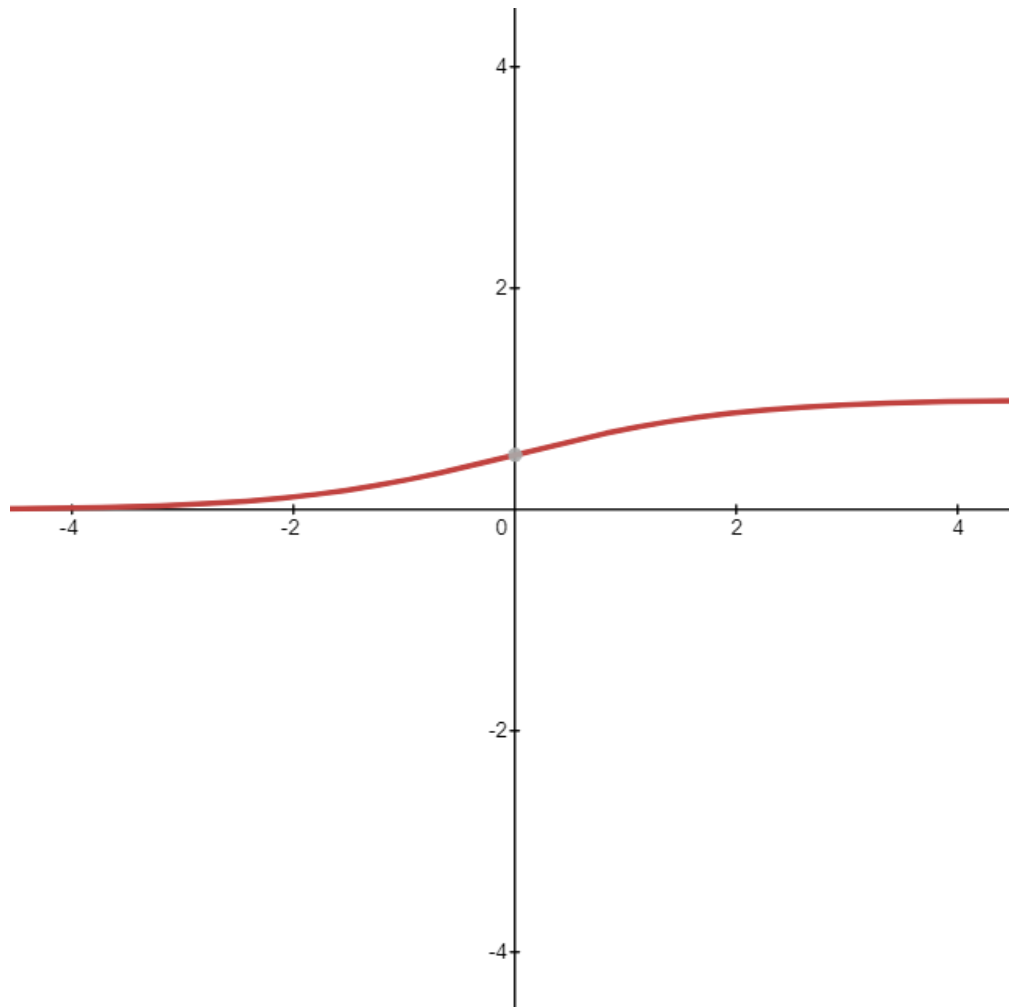
Saturating linear function



$$g(x) = \begin{cases} 0 & x < 0 \\ x & 0 \leq x \leq 1 \\ 1 & x > 1 \end{cases}$$

- Similar to the step function but not as extreme

Sigmoid or logistic function



$$g(x) = \frac{1}{1 + e^{-x}}$$

- Adds up each probability of all nodes to 1
- A gentler version of the step function. A good choice for a classification problem

Soft-max function

$$\frac{e^{z_k}}{\sum_{k=1}^K e^{z_k}}$$

- This activation function is often used in the final, output, layer of a neural network, especially with classification problems
- Gives the probability of each individual output node
- Does not add up each probabilities of all nodes to 1 like the sigmoid function

Hidden layer versus output layer

There is more flexibility with choosing activation functions for hidden layers compared to output layer.

Classification

- Simplest form being binary classification, requiring a single output. With types in the classification, using a vector with dimension the same number of classes would be ideal
 - The cost function is commonly used for such an output
 - Binary yes/no classification

$$J = - \sum_{n=1}^N (y^{(n)} \cdot \ln(\hat{y}_k^{(n)}) \cdot (1 - \ln(\hat{y}_k^{(n)})))$$

- These cost functions could be added a regularization term

$$\frac{\lambda}{2} \|W\|^2$$

Backpropagation

In order to minimize the sensitive cost function, J , along with parameters W and b we need to implement gradient descent

$$\frac{\partial J}{\partial w_{j,j'}^{(l)}}$$

and

$$\frac{\partial J}{\partial b_{j'}^{(l)}}$$

- This method is much more complicated compared to any other machine-learning techniques. This is due to those parameters being embedded within a function of a function etc.
 - A better way of using the chain rule for differentiation is by using quantity

$$\delta_j^{(l)} = \frac{\partial J}{\partial z_j^{(l)}}$$

- z being the linear transformation of the values in the previous layer, but before going into the activation function
- Backpropagation is similar to calculating the error between the y and they \hat{y} in the output layer and assigning that error to the hidden layers
- Error being the difference between the value y and the forecast value \hat{y}
- Using the **chain rule**

$$\delta_j^{(l-1)} = \frac{\partial J}{\partial z_j^{(l-1)}} = \sum_{j'} \frac{\partial J}{\partial z_{j'}^{(l)}} \cdot \frac{\partial z_{j'}^{(l)}}{\partial z_j^{(l-1)}}$$

$$z_{j'}^{(l)} = \sum_j w_{j,j'}^{(l)} \cdot a_j^{(l-1)} + b_{j'}^{(l)} = \sum_j w_{j,j'}^{(l)} \cdot g^{(l-1)} \cdot (z_j^{(l-1)}) + b_{j'}^{(l)}$$

$$\delta_j^{(l-1)} = \frac{dg^{(l-1)}}{dz} \Big|_{z_j^{(l-1)}} \cdot \sum_{j'} \frac{\partial J}{\partial z_{j'}^{(l)}} \cdot w_{j,j'}^{(l)}$$

$$\delta_j^{(l-1)} = \frac{dg^{(l-1)}}{dz} \Big|_{z_j^{(l-1)}} \sum_{j'} \delta_{j'}^{(l)} \cdot w_{j,j'}^{(l)}$$

- This equation demonstrates how to find δ s in a layer if you know the δ s in all layers to the right.

$$\frac{\partial J}{\partial w_{j,j'}^{(l)}} = \frac{\partial J}{\partial z_{j'}^{(l)}} \cdot \frac{\partial z_{j'}^{(l)}}{\partial w_{j,j'}^{(l)}}$$

$$\frac{\partial J}{\partial w_{j,j'}^{(l)}} = \delta_{j'}^{(l)} \cdot a_j^{(l-1)}$$

- The sensitivity of the cost function, J , to the w s can be written in terms of the δ s which in turn are backpropagated from the network layers that are just to the right
- The last hidden layer is different because it feeds into the output. If the cost function is quadratic, for example, then the equation would be

$$\delta_j^{(L)} = \frac{dg^{(L)}}{dz} \mid z_j^{(L)} \cdot (\hat{y}_j - y_j)$$

- If there is a single output then you can drop the j subscripts

The Backpropagation Algorithm

The backpropagation algorithm follows the steps listed below;

- Step 0: Initialize weights and biases
 - Choose whether the weights and biases are picked by random
 - The size of the weights should decrease as the number of nodes increases
- Step 1: Pick one of the data points at random
 - Vector x resides on the left side of the network, to calculate z , a , etc \item Calculate the output \hat{y}
- Step 2: Calculate contribution to the cost function
 - The actual value of the cost function is not necessary to find the weights and biases
 - The actual value of the cost function is useful for monitoring convergence
- Step 3: Starting at the right, calculate all the δ s
 - Using the quadratic cost function in one dimension

$$\delta_j^{(L)} = \frac{dg^{(L)}}{dz} \mid z_j^{(L)} \cdot (\hat{y} - y)$$

- Moving to the left

$$\delta_j^{(L-1)} = \frac{dg^{(L-1)}}{dz} \mid z_j^{(L-1)} \sum_{j'} \delta_{j'}^{(l)} \cdot w_{j,j'}^{(l)}$$

- Step 4: Update the weights and biases using (stochastic) gradient decent (SGD)
 - Optimized the gradient descent during each search once a random weight vector is picked
 - In SGD, the user initializes the weights, and the process updates the weight vector using one data point.

$$\text{New } w_{j,j'}^{(l)} = \text{Old } w_{j,j'}^{(l)} - \beta \frac{\partial J}{\partial w_{j,j'}^{(l)}} = \text{Old } w_{j,j'}^{(l)} - \beta \delta_{j'}^{(l)} \cdot a_j^{(l-1)}$$

$$\text{New } b_{j'}^{(l)} = \text{Old } b_{j'}^{(l)} - \beta \frac{\partial J}{\partial b_{j'}^{(l)}} = \text{Old } b_{j'}^{(l)} - \beta \delta_{j'}^{(l)}$$

- Return to Step 1

Exploratory Data Analysis (EDA)

- Features 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[4]

```
In [193]: 1 import numpy as np
          2 import pandas as pd
          3 import seaborn as sns
          4 import matplotlib.pyplot as plt
          5 %matplotlib inline
          6
          7 import warnings
          8 warnings.filterwarnings('ignore')
```

```
In [72]: 1 df = pd.read_csv("cancer_classification.csv")
```

```
In [73]: 1 %%capture
          2 df.info()
```

- The following real-valued features are computed for each cell nucleus:
 - radius (mean of distances from center to points on the perimeter)
 - texture (standard deviation of gray-scale values)
 - perimeter
 - area
 - smoothness (local variation in radius lengths)
 - compactness (perimeter² / area - 1.0)
 - concavity (severity of concave portions of the contour)
 - concave points (number of concave portions of the contour)
 - symmetry
 - fractal dimension ("coastline approximation" - 1)
- The mean, standard error and "worst" of these features were computed for each image, resulting in 30 features.

In [74]: 1 df.head()

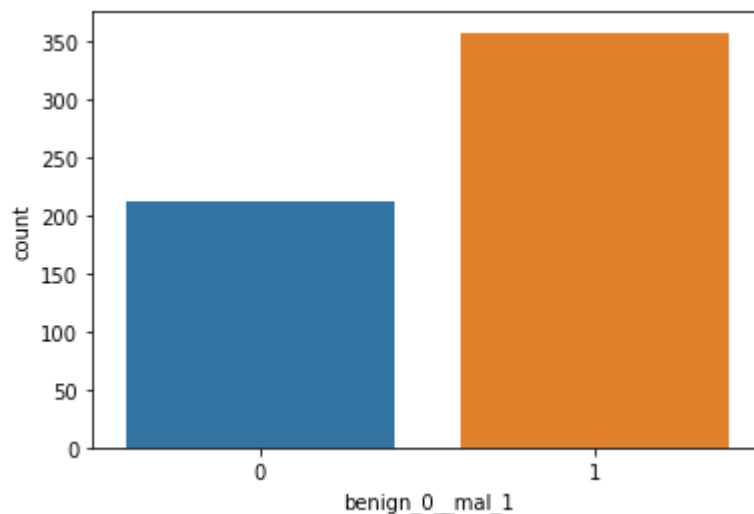
Out[74]:

	mean radius	mean texture	mean perimeter	mean area	mean smoothness	mean compactness	mean concavity	mean concave points	mean symmetry	d
0	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	
1	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	
2	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	
3	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	
4	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	

5 rows × 31 columns

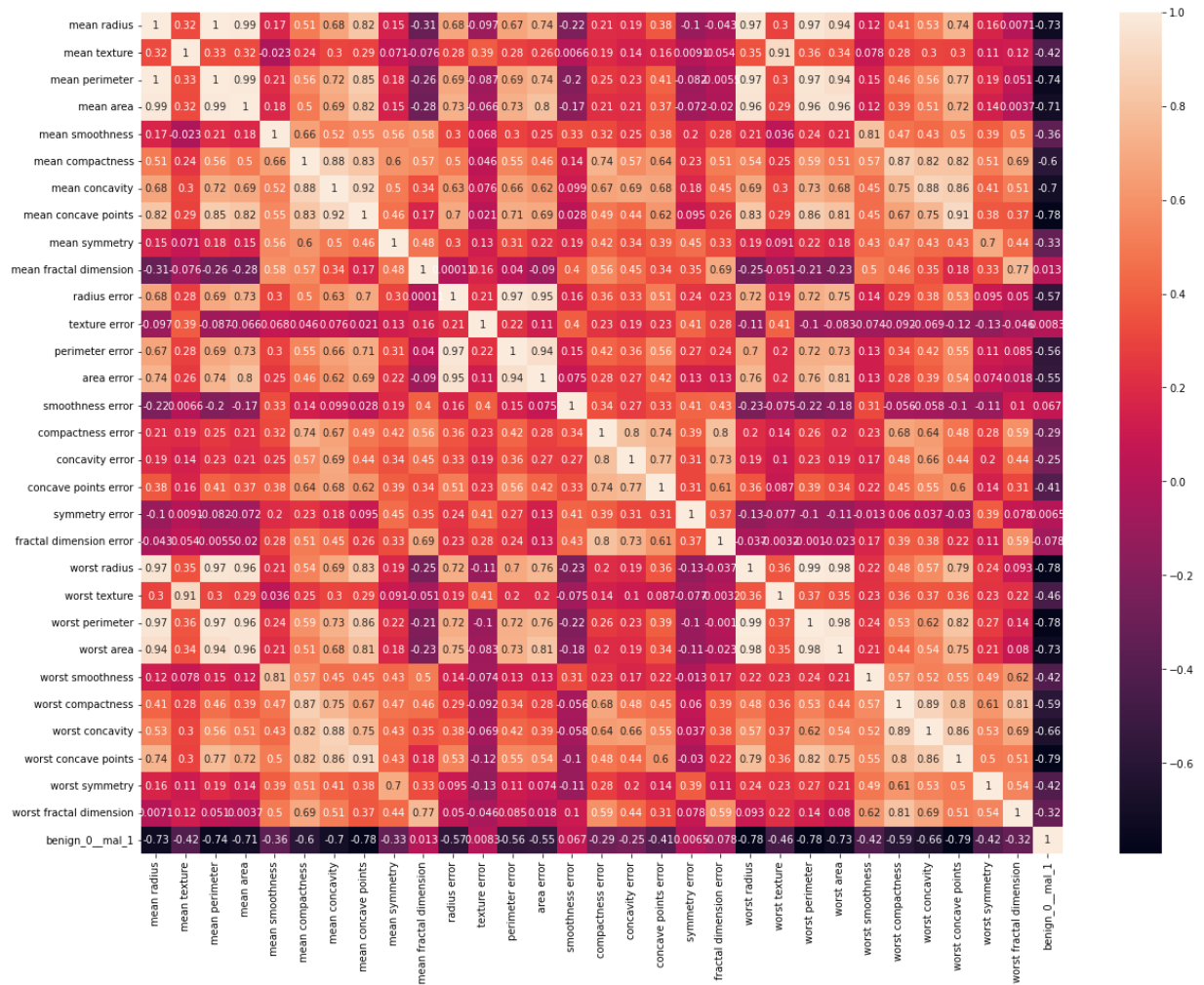
In [75]: 1 %%capture
2 df.shape

In [76]: 1 sns.countplot(x="benign_0__mal_1", data=df);



- Shows how many entries are either benign (0) or malignant (1)
- There is almost double the number of malignant tumors compared to benign tumors in the count plot
- This shows an imbalance in the data which makes the machine learning accuracy metric is biased and untrustworthy

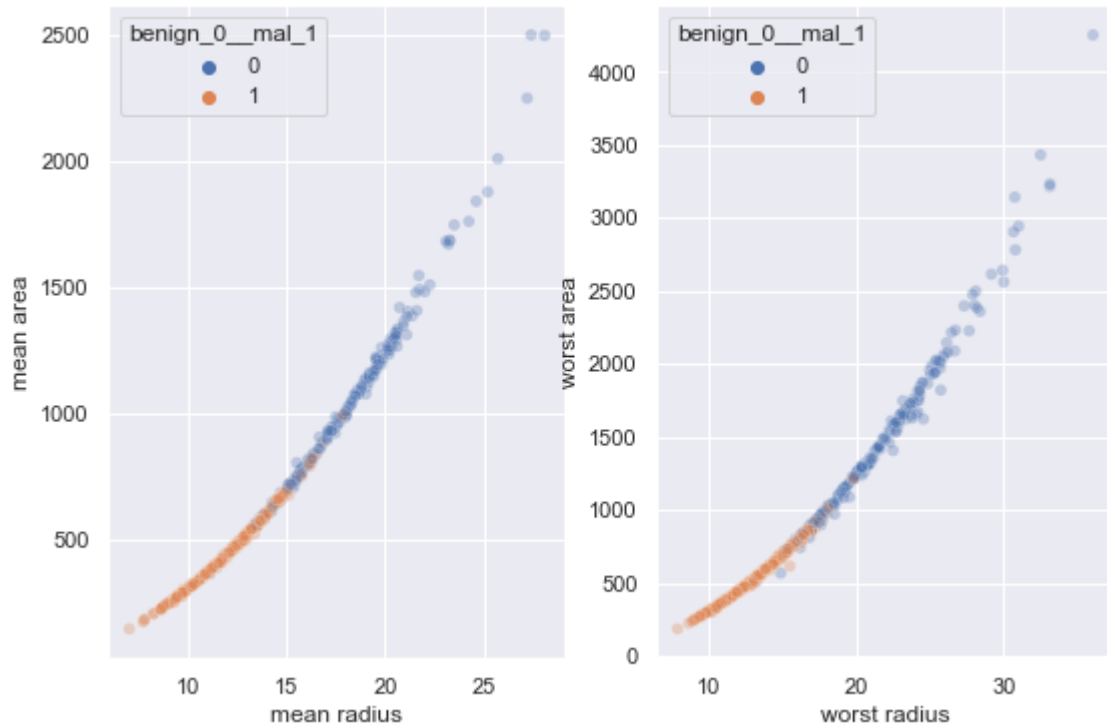
```
In [77]: 1 plt.figure(figsize=(20, 15))
2 sns.heatmap(df.corr(), annot=True);
```



- Shows a heat map of each column correlating with each other
- There is a repeating 4x4 pattern in the entire heat map where perimeter and area have a strong correlation with radius

```
In [223]: 1 fig, axes = plt.subplots(1, 2, figsize=(9, 6))
2          sns.scatterplot(ax=axes[0], data=df, x='mean radius', y='mean area', hue='be
3          sns.scatterplot(ax=axes[1], data=df, x='worst radius', y='worst area', hue='
< >
```

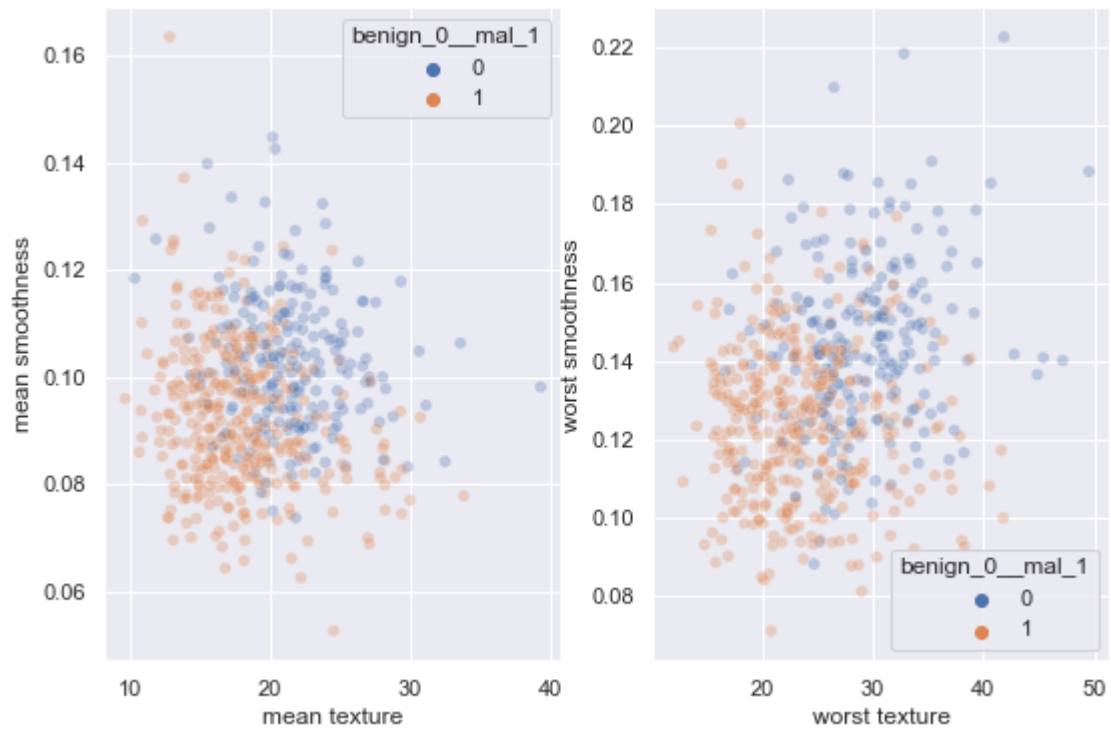
Out[223]: <AxesSubplot:xlabel='worst radius', ylabel='worst area'>



- Compares the area and radius between benign and malignant tumors
- It consistently shows that radius and area have a high similarity whether it is the mean, worst, or error estimate in the pair plot

```
In [222]: 1 fig, axes = plt.subplots(1, 2, figsize=(9, 6))
          2 sns.scatterplot(ax=axes[0], data=df, x='mean texture', y='mean smoothness',
          3 sns.scatterplot(ax=axes[1], data=df, x='worst texture', y='worst smoothness')
```

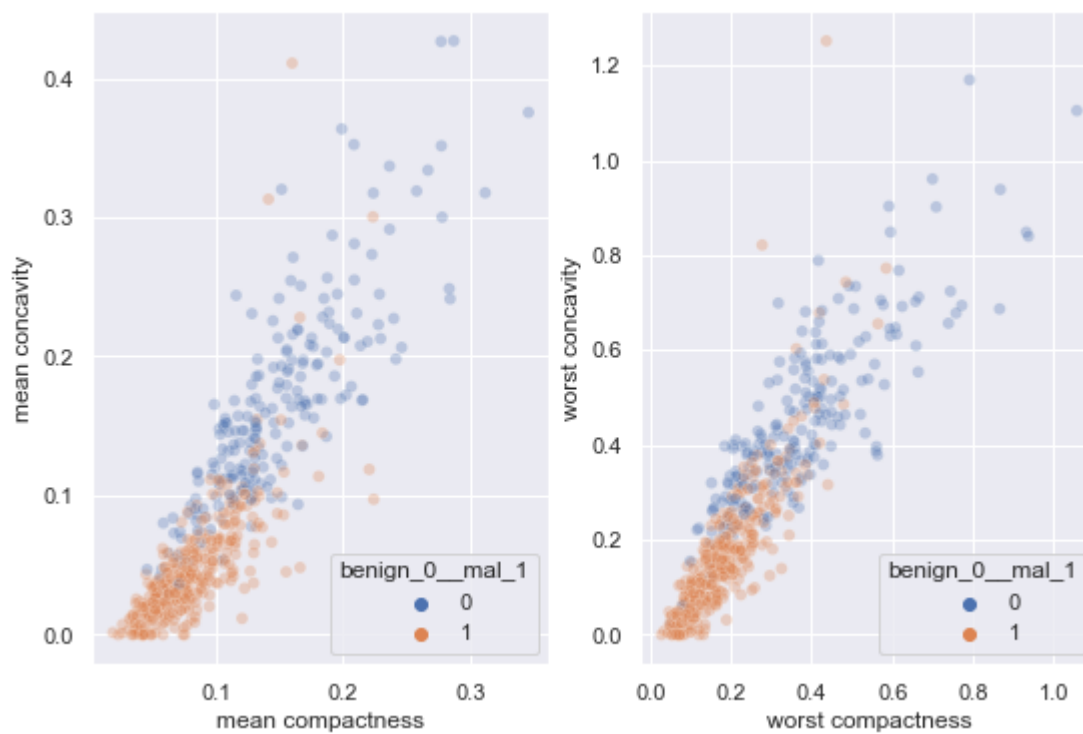
Out[222]: <AxesSubplot:xlabel='worst texture', ylabel='worst smoothness'>



- Compares the smoothness and texture between benign and malignant tumors
- Malignant tumors are shown to be more rugged and rough compared to benign tumors


```
In [221]: figsize=(9, 6))
          a=df, x='mean compactness', y='mean concavity', hue='benign_0__mal_1', alpha=0.3)
          a=df, x='worst compactness', y='worst concavity', hue='benign_0__mal_1', alpha=0.3)
```

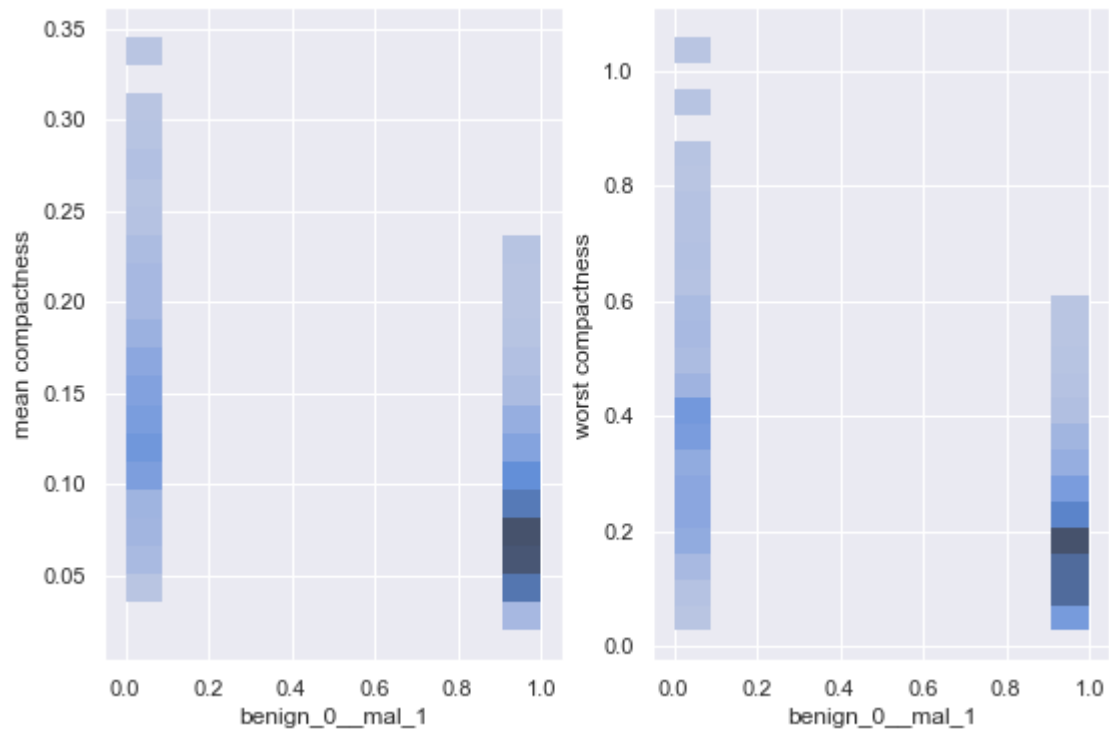
Out[221]: <AxesSubplot:xlabel='worst compactness', ylabel='worst concavity'>



- Compares the concavity and the compactness between benign and malignant tumors
- Benign tumors are shown to be consistently more compact and concave compared to malignant tumors

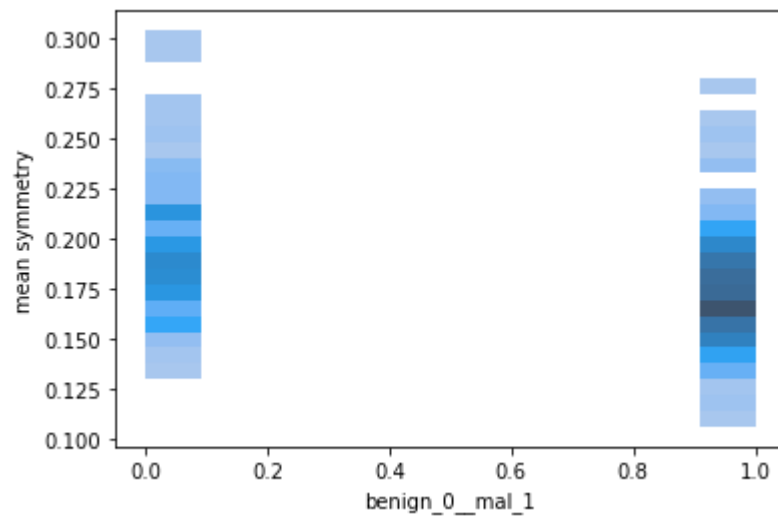
```
In [224]: 1 fig, axes = plt.subplots(1, 2, figsize=(9, 6))
          2 sns.histplot(ax=axes[0], data=df, x='benign_0__mal_1', y='mean compactness')
          3 sns.histplot(ax=axes[1], data=df, x='benign_0__mal_1', y='worst compactness')
```

Out[224]: <AxesSubplot:xlabel='benign_0__mal_1', ylabel='worst compactness'>



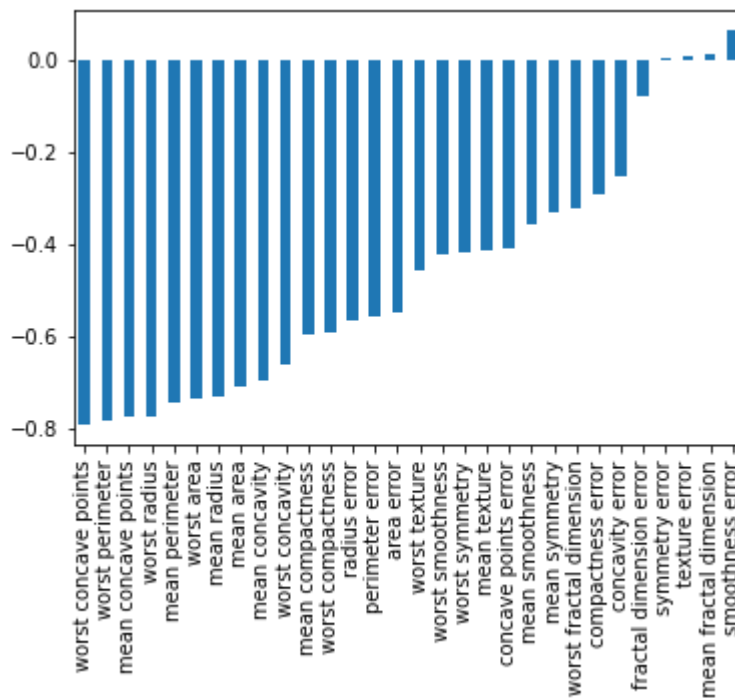
- The reason that the histogram point at around 0.32 - 0.36 is missing is because there are no benign tumors with a mean compactness at that size.

```
In [87]: 1 sns.histplot(data=df, x='benign_0__mal_1', y='mean symmetry');
```



- The reason that the histogram point at around 0.275 - 0.285 is missing is because there are no benign tumors with a mean symmetry at that size.

```
In [91]: 1 df.corr()["benign_0__mal_1"][: -1].sort_values().plot(kind='bar');
```



- This is a bar distribution correlation plot comparing every column with benign and malignant tumors.

Splitting the Data into Train and Test

Importing sklearn learning to split data into train and test

```
In [92]: 1 X = df.drop('benign_0__mal_1', axis=1)
2 y = df['benign_0__mal_1']
```

dropping "benign_0__mal_1" due to using it as the main distinction in the data set

all the other features used except for benign or malignant (mean, worst, error) work together to procure better predictions

```
In [93]: 1 from sklearn.model_selection import train_test_split
```

```
In [94]: 1 X_train, X_test, y_train, y_test = train_test_split(X,y,test_size = 0.25,
2 random_state = 51)
```

```
In [95]: 1 from sklearn.preprocessing import MinMaxScaler
```

```
In [96]: 1 scaler = MinMaxScaler()  
        2 scaler.fit(X_train)
```

Out[96]: MinMaxScaler()

Scaling the training set normalizes the data. This makes sure the number are not as volatile as it is in the original data set.

```
In [97]: 1 X_train = scaler.transform(X_train)  
        2 X_test = scaler.transform(X_test)
```

```
In [98]: 1 from keras.callbacks import TensorBoard
```

```
In [99]: 1 board1 = TensorBoard(log_dir='logs/run1', histogram_freq=1, write_graph=True  
        < [REDACTED] >
```

```
In [100]: 1 board1 = TensorBoard(log_dir='logs/run1', histogram_freq=1, write_graph=True, write_images=True, update_freq='epoch', profile_batch=2, embeddings_freq=1)  
        < [REDACTED] >
```

```
In [101]: 1 board3 = TensorBoard(log_dir='logs/run3', histogram_freq=1, write_graph=True, w  
        < [REDACTED] >
```

```
In [102]: 1 board4 = TensorBoard(log_dir='logs/run4', histogram_freq=1, write_graph=True, w  
        < [REDACTED] >
```

Training and Testing the Model

```
In [103]: 1 import tensorflow as tf  
        2 from tensorflow.keras.models import Sequential  
        3 from tensorflow.keras.layers import Dense, Activation  
        4 from tensorflow.keras.layers import Dropout
```

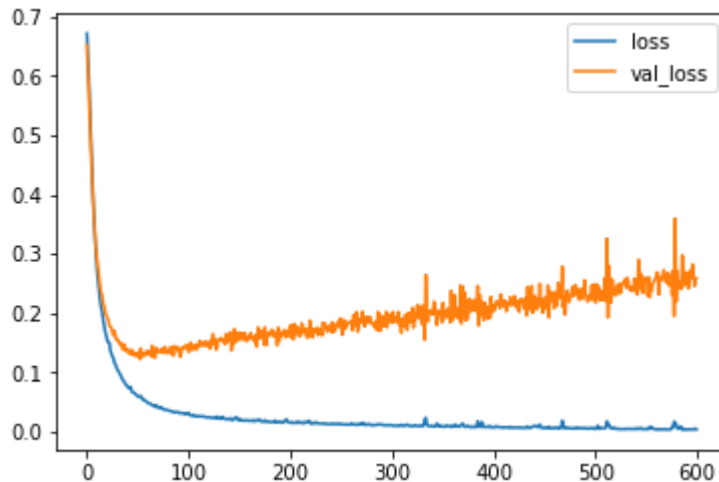
```
In [104]: 1 model1 = Sequential()
```

```
In [105]: 1 model1.add(Dense(units=30, activation='relu'))  
        2 model1.add(Dense(units=15, activation='relu'))  
        3 model1.add(Dense(units=1, activation='sigmoid'))
```

```
In [106]: 1 model1.compile(loss='binary_crossentropy',  
        2 optimizer='adam')
```

```
In [107]: 1 %capture  
        2  
        3 model1.fit(x = X_train, y = y_train, epochs = 600, validation_data = (X_test, y  
        4 )  
        < [REDACTED] >
```

```
In [108]: 1 model_loss1 = pd.DataFrame(model1.history.history)
          2 model_loss1.plot();
```



Since the epoch was set at such a high number (600), there was too much training applied to the model resulting in overfitting. This is when the model already knows too much to really predict anything new.

Adding Early Stopping

```
In [109]: 1 model2 = Sequential()
          2 model2.add(Dense(units=30,activation='relu'))
          3 model2.add(Dense(units=15,activation='relu'))
          4 model2.add(Dense(units=1,activation='sigmoid'))
          5 model2.compile(loss='binary_crossentropy',
          6 optimizer='adam')
```

```
In [110]: 1 from tensorflow.keras.callbacks import EarlyStopping
```

```
In [111]: 1 early_stop = EarlyStopping(monitor='val_loss',
          2 mode='min',
          3 verbose=1,
          4 patience=25)
```

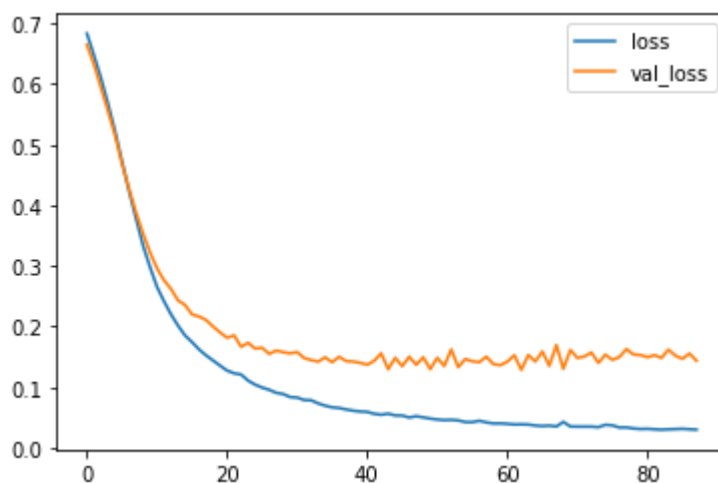
Setting the mode to 'min' in order to check if the training has stopped decreasing over several iterations (epochs) works the best to monitor 'val_loss'. Having the patience set to 25 epochs is a starting point.

```
In [112]: 1 %%capture
          2
          3 model2.fit(x=X_train, y=y_train, epochs=600, validation_data=(X_test, y_test
          4 )
```

WARNING:tensorflow:Callbacks method `on_train_batch_end` is slow compared to the batch time (batch time: 0.0010s vs `on_train_batch_end` time: 0.1969s). Check your callbacks.

Using 600 epochs again to determine the difference between the original model (overfitting) compared to one with early stoppage.

```
In [113]: 1 model_loss = pd.DataFrame(model2.history.history)
          2 model_loss.plot();
```



Even though the model shows that there is still overfitting, it is not as prominent as the original model

Adding in DropOut Layers

Also known as Dilution is another regularization technique for reducing overfitting in Artificial Neural Networks

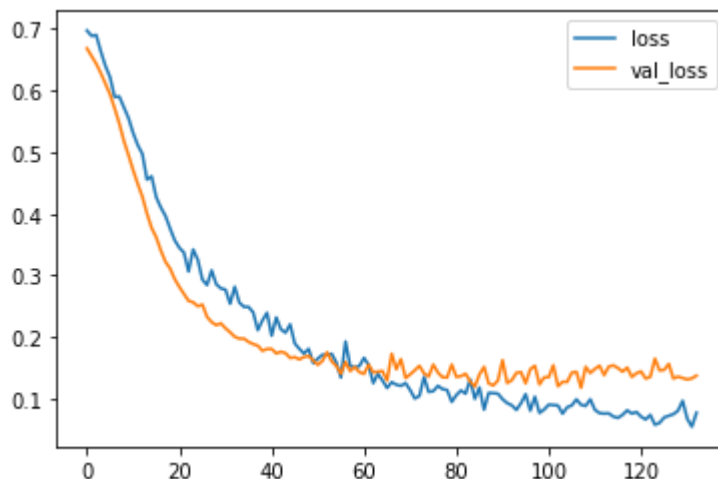
```
In [114]: 1 from tensorflow.keras.layers import Dropout
```

```
In [115]: 1 model3 = Sequential()  
2  
3 model3.add(Dense(units=30,activation='relu'))  
4 model3.add(Dropout(0.5))  
5  
6 model3.add(Dense(units=15,activation='relu'))  
7 model3.add(Dropout(0.5))  
8  
9 model3.add(Dense(units=1,activation='sigmoid'))  
10 model3.compile(loss='binary_crossentropy',  
11               optimizer='adam')
```

Decided to drop half of the input units

```
In [116]: 1 %%capture  
2  
3 model3.fit(x=X_train, y=y_train, epochs=600, validation_data=(X_test, y_test  
4 )
```

```
In [117]: 1 model_loss = pd.DataFrame(model3.history.history)  
2 model_loss.plot();
```



Using a dropout layer along with early stoppage reduces overfitting way more combined

Model Evaluation

```
In [118]: 1 predictions = (model3.predict(X_test)>0.5).astype("int32")
```

```
In [119]: 1 from sklearn.metrics import classification_report  
2 from sklearn.metrics import confusion_matrix
```



```
In [120]: 1 print(classification_report(y_test,predictions))
          2 print(confusion_matrix(y_test,predictions))
```

	precision	recall	f1-score	support
0	1.00	0.90	0.95	49
1	0.95	1.00	0.97	94
accuracy			0.97	143
macro avg	0.97	0.95	0.96	143
weighted avg	0.97	0.97	0.96	143

```
[[44  5]
 [ 0 94]]
```

Since all the models used for evaluation had 600 epochs, the overfitting made it hard for the model to make mistakes such as false positives and false negatives. There is a high rating in both the classification report and confusion matrix due to the overfitting.

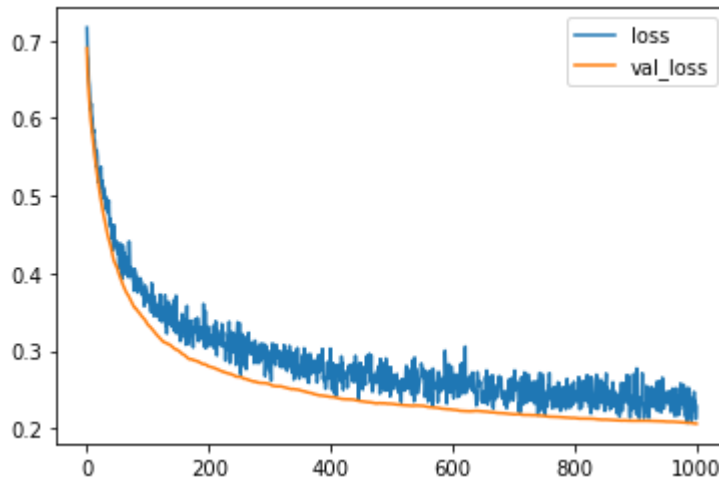
Adding Learning Rate to Current Model

```
In [121]: 1 from keras.optimizers import SGD
```

```
In [122]: 1 model4 = Sequential()
          2
          3 model4.add(Dense(units=30,activation='relu'))
          4 model4.add(Dropout(0.5))
          5
          6 model4.add(Dense(units=15,activation='relu'))
          7 model4.add(Dropout(0.5))
          8
          9 model4.add(Dense(units=1,activation='sigmoid'))
         10 opt = tf.keras.optimizers.SGD(learning_rate=0.005, momentum=0.9, decay=0.01)
         11 model4.compile(loss='binary_crossentropy',
         12                 optimizer=opt)
```

```
In [123]: 1 %%capture
          2
          3 model4.fit(x=X_train,
          4             y=y_train,
          5             epochs=1000,
          6             batch_size=36,
          7             validation_data=(X_test, y_test),
          8             verbose=1,
          9             callbacks=[early_stop,board4]
         10 )
```

```
In [124]: 1 model_loss = pd.DataFrame(model4.history.history)
          2 model_loss.plot();
```



```
In [140]: 1 predictions = (model4.predict(X_test)>0.5).astype("int32")
          2 print(classification_report(y_test,predictions))
          3 print(confusion_matrix(y_test,predictions))
```

	precision	recall	f1-score	support
0	1.00	0.80	0.89	49
1	0.90	1.00	0.95	94
accuracy			0.93	143
macro avg	0.95	0.90	0.92	143
weighted avg	0.94	0.93	0.93	143

```
[[39 10]
 [ 0 94]]
```

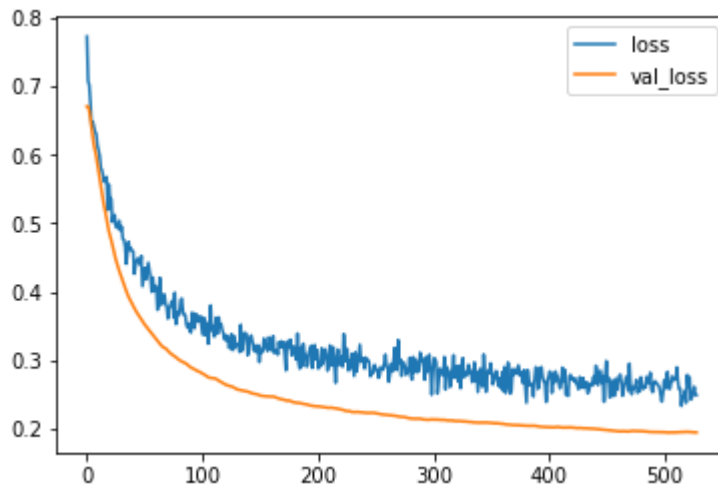
Using Tensorboard to display all Models

```
In [127]: 1 board5 = TensorBoard(log_dir='logs/run5', histogram_freq=1, write_graph=True, w
```

```
In [128]: 1 model5 = Sequential()
2
3 model5.add(Dense(units=30,activation='relu'))
4 model5.add(Dropout(0.5))
5
6 model5.add(Dense(units=15,activation='relu'))
7 model5.add(Dropout(0.5))
8
9 model5.add(Dense(units=1,activation='sigmoid'))
10 opt =tf.keras.optimizers.SGD(learning_rate=0.005, momentum=0.9, decay=0.01)
11 model5.compile(loss='binary_crossentropy',
12                optimizer=opt)
```

```
In [129]: 1 %%capture
2
3 model5.fit(X_train, y_train, epochs=1000, batch_size=36, validation_data=(X_
4 )
5
```

```
In [130]: 1 model_loss = pd.DataFrame(model5.history.history)
2 model_loss.plot();
```



```
In [69]: 1 #import shutil
2 #shutil.rmtree(r'C:\Users\HLixir\Documents\DATA3401\Logs')#clears up old Log
3
```

```
In [68]: 1 pwd
```

```
Out[68]: 'C:\\Users\\HLixir\\Documents\\DATA3401'
```

```
In [67]: 1 %load_ext tensorboard
```

In [70]:

1 %tensorboard --logdir logs

Reusing TensorBoard on port 6006 (pid 13952), started 0:00:09 ago. (Use '!kill 13952' to kill it.)

TensorBoard

INACTIVE

UPLOAD

☐ Show data download links

☐ Ignore outliers in chart scaling

Tooltip sorting method:

default

Smoothing

0

Horizontal Axis

STEP

RELATIVE

WALL

Runs

Write a regex to filter runs

run2\validation

run3\train

run3\validation

run4\train

run4\validation

run5\train

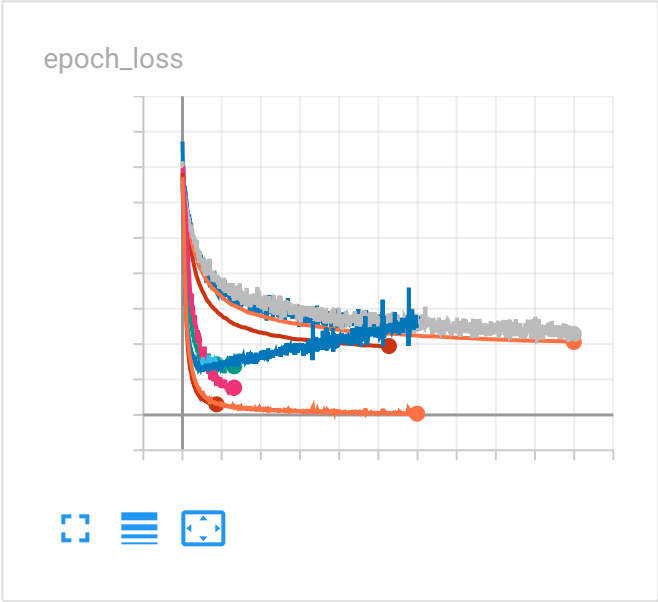
run5\validation

TOGGLE ALL RUNS

logs

epoch_loss

epoch_loss



Using LIME to Interpret the Model

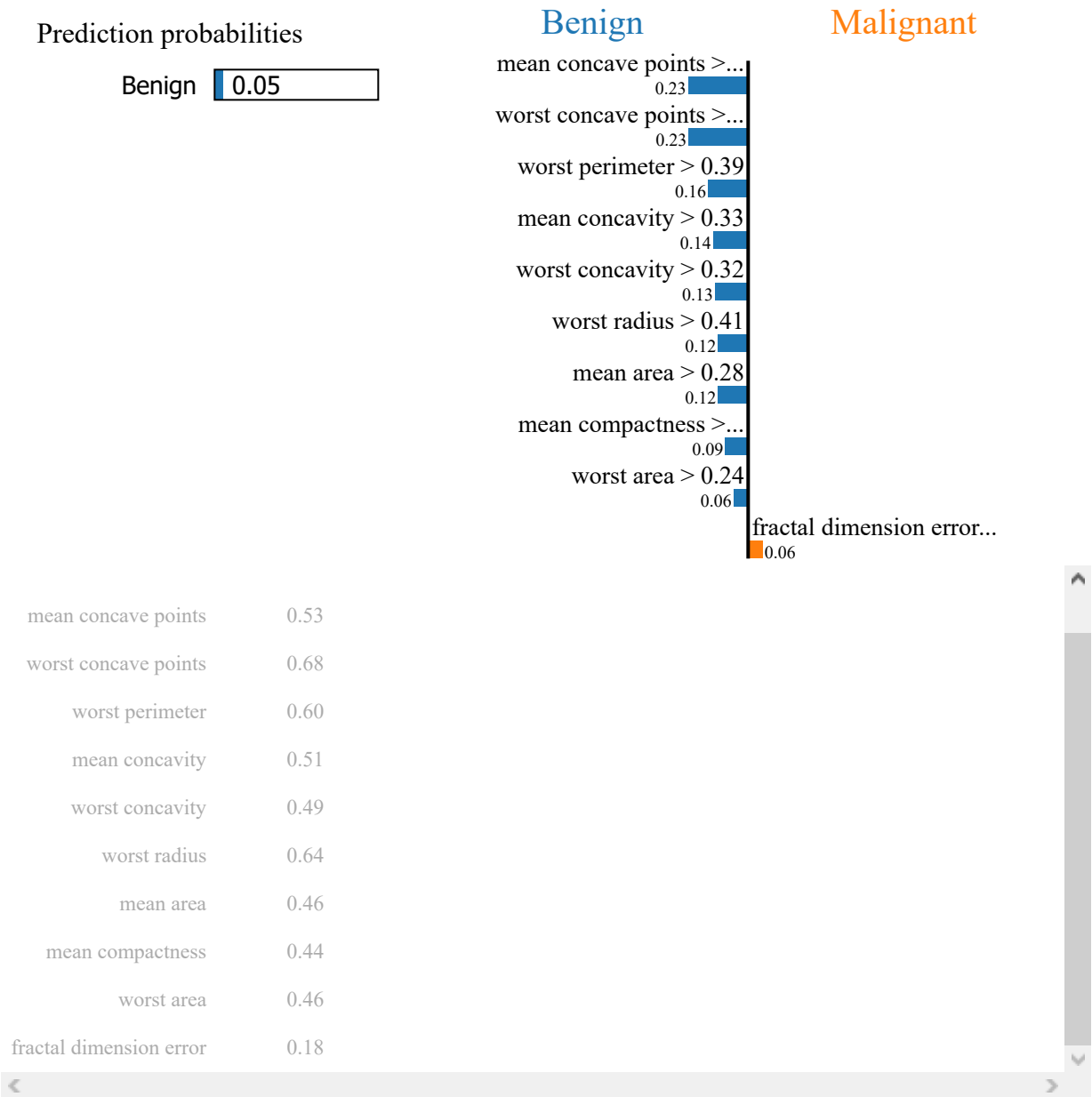
- Local Interpretable Model-agnostic Explanations (LIME)
- LIME can be considered as a "white-box," which locally approximates the behavior of the machine in a neighborhood of input values[5]

```
In [131]: 1 import lime
          2 from lime import lime_tabular
          3 from lime.lime_tabular import LimeTabularExplainer
```

```
In [132]: 1 explainer = LimeTabularExplainer(training_data=np.array(X_train), feature_na
          < |----->
```

In [134]:

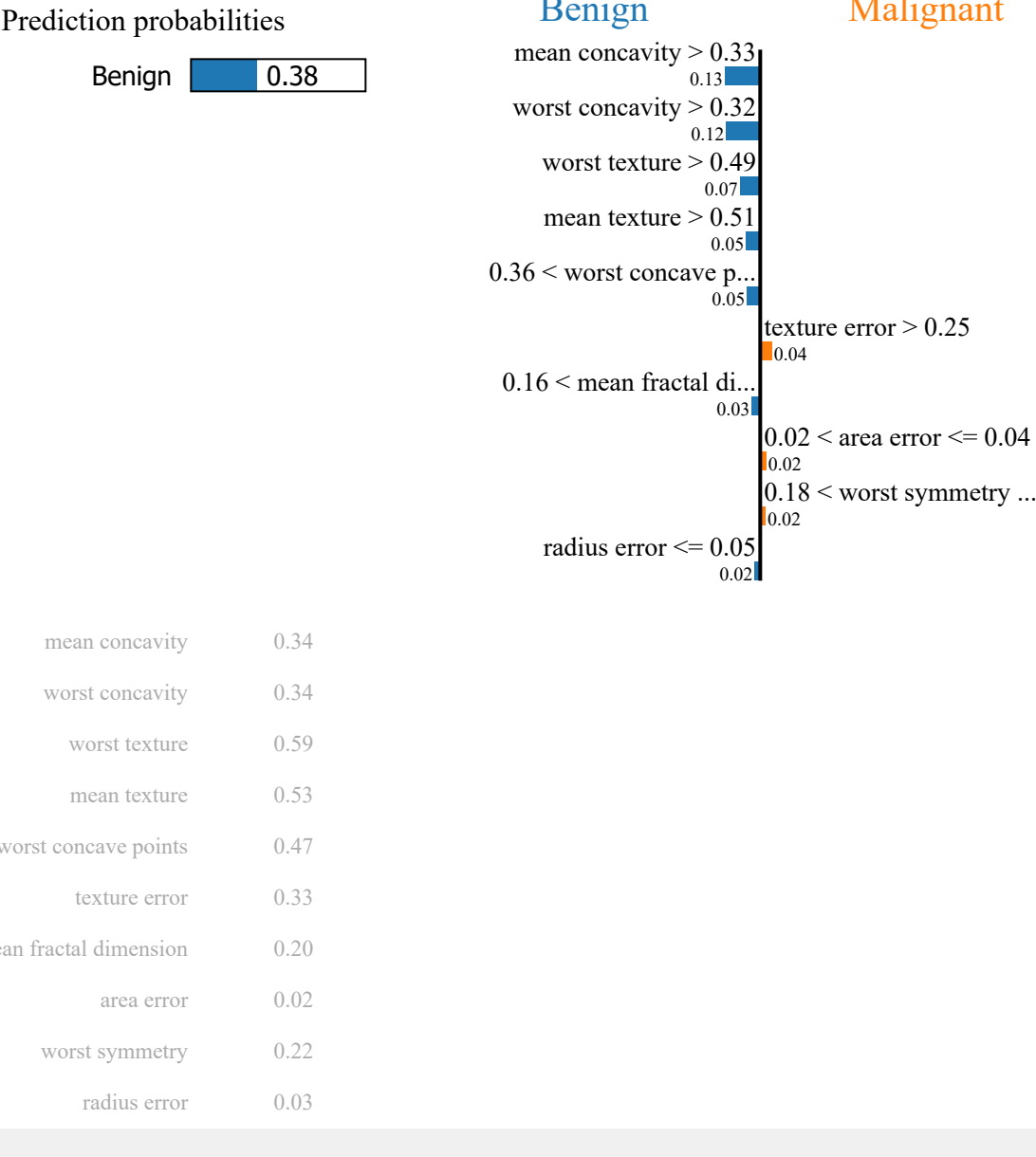
1 exp.show_in_notebook(show_table=True, show_all=False)



In [137]:

1 %%javascript
2 IPython.OutputArea.auto_scroll_threshold = 9999;

```
In [139]: 1 for i in range(2):
2         exp = explainer.explain_instance(data_row = X_test[i+5],
3                                           predict_fn = model15.predict,
4                                           labels=(0,))
5         exp.show_in_notebook(show_table=True, show_all=False)
```

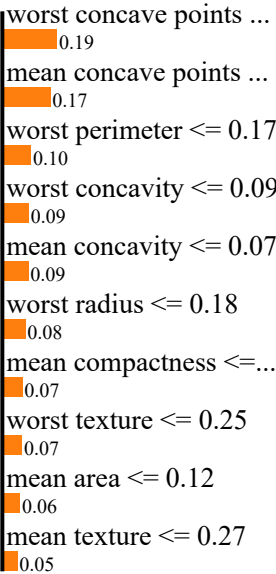


Prediction probabilities

Benign 0.98

Benign

Malignant



Feature	Value
worst concave points	0.18
mean concave points	0.07
worst perimeter	0.09
worst concavity	0.03
mean concavity	0.03
worst radius	0.10
mean compactness	0.08
worst texture	0.24
mean area	0.06

Conclusions

- Explained what artificial neural networks are and its flexibility through multiple activation functions

- Implemented overfitting techniques to reduce model miss-classifications to 9 out of 143 entries and a F-1 score of 90% and 95%
- Neural Networks have shown to help further research in early detection of both benign and malignant tumors.
- Interpretable machine learning (LIME) gives insight to how **worst concave points** is the strongest feature in determining a tumor's state.

References

1. C. E. DeSantis, J. Ma, M. M. Gaudet, L. A. Newman, K. D. Miller, A. Goding Sauer, A. Jemal, and R. L. Siegel, "Breast cancer statistics, 2019," CA: a cancer journal for clinicians, vol. 69, no. 6, pp. 438–451, 2019.
2. A. Raad, A. Kalakech, and M. Ayache, "Breast cancer classification using neural network approach: Mlp and rbf," Ali Mohsen Kabalan, vol. 7, no. 8, p. 105, 2012.
3. P. Wilmott, "Machine learning: an applied mathematics introduction," Machine Learning and the City: Applications in Architecture and Urban Design, pp. 147–171, 2022.
4. K. P. Bennett and O. L. Mangasarian, "Robust linear programming discrimination of two linearly inseparable sets," Optimization methods and software, vol. 1, no. 1, pp. 23–34, 1992.!
5. Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "Why should i trust you?" Explaining the predictions of any classifier." Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. 2016.