ex2_sol

May 2, 2018

1 Exercise 2 - Introduction to (Deep) Neural Networks

This exercise uses some images and information from https://github.com/YaleATLAS/CERNDeepLearningTuto

2 Table of Contents

3 1 Introduction to Keras

* Modular, powerful and intuitive Deep Learning Python library built on
and and

> Developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.
>
> https://keras.io

- Minimalist, user-friendly interface
- Extremely well documented, lots of working examples
- Very shallow learning curve $\rightarrow$ by far one of the best tools for both beginners and experts
- Open-source, developed and maintained by a community of contributors, and publicly hosted on GitHub
- Extensible: possibility to customize layers

From the Keras website:

4 2 Breast cancer dataset

4.1 Loading the dataset

4.1.1 Task 1: For this exercise we want to use the breast cancer dataset from sci-kit learn. Prepare the dataset in the following way:

- Load the dataset (`load_breast_cancer`), inspect it and create a pandas `DataFrame`.
- How many example and how many features do we have? What are the names of the classes? How many examples of each class do we have?
- Plot the mean radius and the mean smoothness of the training data in a 2D scatter plot for the two classes

```
In [1]: from sklearn.datasets import load_breast_cancer

        breast_cancer = load_breast_cancer()
        print(breast_cancer['DESCR'])
```

```
Breast Cancer Wisconsin (Diagnostic) Database
=============================================

Notes
-----
Data Set Characteristics:
    :Number of Instances: 569

    :Number of Attributes: 30 numeric, predictive attributes and the class

    :Attribute Information:
        - 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 (perimeter^2 / 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" or largest (mean of the three
        largest values) of these features were computed for each image,
        resulting in 30 features.  For instance, field 3 is Mean Radius, field
        13 is Radius SE, field 23 is Worst Radius.

        - class:
                - WDBC-Malignant
                - WDBC-Benign
    :Summary Statistics:

    ===================================== ====== ======
                                           Min    Max
    ===================================== ====== ======
    radius (mean):                        6.981  28.11
    texture (mean):                       9.71   39.28
    perimeter (mean):                     43.79  188.5
    area (mean):                          143.5  2501.0
    smoothness (mean):                    0.053  0.163
    compactness (mean):                   0.019  0.345
    concavity (mean):                     0.0    0.427
    concave points (mean):                0.0    0.201
    symmetry (mean):                      0.106  0.304
    fractal dimension (mean):             0.05   0.097
    radius (standard error):              0.112  2.873
    texture (standard error):             0.36   4.885
    perimeter (standard error):           0.757  21.98
    area (standard error):                6.802  542.2
    smoothness (standard error):          0.002  0.031
    compactness (standard error):         0.002  0.135
    concavity (standard error):           0.0    0.396
    concave points (standard error):      0.0    0.053
    symmetry (standard error):            0.008  0.079
    fractal dimension (standard error):   0.001  0.03
    radius (worst):                       7.93   36.04
    texture (worst):                      12.02  49.54
    perimeter (worst):                    50.41  251.2
    area (worst):                         185.2  4254.0
    smoothness (worst):                   0.071  0.223
    compactness (worst):                  0.027  1.058
    concavity (worst):                    0.0    1.252
    concave points (worst):               0.0    0.291
    symmetry (worst):                     0.156  0.664
    fractal dimension (worst):            0.055  0.208
    ===================================== ====== ======

    :Missing Attribute Values: None

    :Class Distribution: 212 - Malignant, 357 - Benign

    :Creator:  Dr. William H. Wolberg, W. Nick Street, Olvi L. Mangasarian

    :Donor: Nick Street

    :Date: November, 1995

This is a copy of UCI ML Breast Cancer Wisconsin (Diagnostic) datasets.
```

```
https://goo.gl/U2Uwz2

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.

Separating plane described above was obtained using
Multisurface Method-Tree (MSM-T) [K. P. Bennett, "Decision Tree
Construction Via Linear Programming." Proceedings of the 4th
Midwest Artificial Intelligence and Cognitive Science Society,
pp. 97-101, 1992], a classification method which uses linear
programming to construct a decision tree.  Relevant features
were selected using an exhaustive search in the space of 1-4
features and 1-3 separating planes.

The actual linear program used to obtain the separating plane
in the 3-dimensional space is that described in:
[K. P. Bennett and O. L. Mangasarian: "Robust Linear
Programming Discrimination of Two Linearly Inseparable Sets",
Optimization Methods and Software 1, 1992, 23-34].

This database is also available through the UW CS ftp server:

ftp ftp.cs.wisc.edu
cd math-prog/cpo-dataset/machine-learn/WDBC/

References
----------
   - W.N. Street, W.H. Wolberg and O.L. Mangasarian. Nuclear feature extraction
     for breast tumor diagnosis. IS&T/SPIE 1993 International Symposium on
     Electronic Imaging: Science and Technology, volume 1905, pages 861-870,
     San Jose, CA, 1993.
   - O.L. Mangasarian, W.N. Street and W.H. Wolberg. Breast cancer diagnosis and
     prognosis via linear programming. Operations Research, 43(4), pages 570-577,
     July-August 1995.
   - W.H. Wolberg, W.N. Street, and O.L. Mangasarian. Machine learning techniques
     to diagnose breast cancer from fine-needle aspirates. Cancer Letters 77 (1994)
     163-171.
```

```
In [2]: import pandas as pd
        df = pd.DataFrame(breast_cancer.data, columns=breast_cancer.feature_names)
        df.head(10)
```

```
Out[2]:    mean radius  mean texture  mean perimeter  mean area  mean smoothness  \
        0        17.99         10.38          122.80     1001.0          0.11840
        1        20.57         17.77          132.90     1326.0          0.08474
```

```
2      19.69        21.25          130.00    1203.0          0.10960
3      11.42        20.38           77.58     386.1          0.14250
4      20.29        14.34          135.10    1297.0          0.10030
5      12.45        15.70           82.57     477.1          0.12780
6      18.25        19.98          119.60    1040.0          0.09463
7      13.71        20.83           90.20     577.9          0.11890
8      13.00        21.82           87.50     519.8          0.12730
9      12.46        24.04           83.97     475.9          0.11860

   mean compactness  mean concavity  mean concave points  mean symmetry  \
0           0.27760         0.30010              0.14710         0.2419
1           0.07864         0.08690              0.07017         0.1812
2           0.15990         0.19740              0.12790         0.2069
3           0.28390         0.24140              0.10520         0.2597
4           0.13280         0.19800              0.10430         0.1809
5           0.17000         0.15780              0.08089         0.2087
6           0.10900         0.11270              0.07400         0.1794
7           0.16450         0.09366              0.05985         0.2196
8           0.19320         0.18590              0.09353         0.2350
9           0.23960         0.22730              0.08543         0.2030

   mean fractal dimension           ...             worst radius  \
0                 0.07871           ...                    25.38
1                 0.05667           ...                    24.99
2                 0.05999           ...                    23.57
3                 0.09744           ...                    14.91
4                 0.05883           ...                    22.54
5                 0.07613           ...                    15.47
6                 0.05742           ...                    22.88
7                 0.07451           ...                    17.06
8                 0.07389           ...                    15.49
9                 0.08243           ...                    15.09

   worst texture  worst perimeter  worst area  worst smoothness  \
0          17.33           184.60      2019.0            0.1622
1          23.41           158.80      1956.0            0.1238
2          25.53           152.50      1709.0            0.1444
3          26.50            98.87       567.7            0.2098
4          16.67           152.20      1575.0            0.1374
5          23.75           103.40       741.6            0.1791
6          27.66           153.20      1606.0            0.1442
7          28.14           110.60       897.0            0.1654
8          30.73           106.20       739.3            0.1703
9          40.68            97.65       711.4            0.1853

   worst compactness  worst concavity  worst concave points  worst symmetry  \
0             0.6656           0.7119                0.2654          0.4601
1             0.1866           0.2416                0.1860          0.2750
```

```
2          0.4245          0.4504                0.2430          0.3613
3          0.8663          0.6869                0.2575          0.6638
4          0.2050          0.4000                0.1625          0.2364
5          0.5249          0.5355                0.1741          0.3985
6          0.2576          0.3784                0.1932          0.3063
7          0.3682          0.2678                0.1556          0.3196
8          0.5401          0.5390                0.2060          0.4378
9          1.0580          1.1050                0.2210          0.4366

   worst fractal dimension
0                  0.11890
1                  0.08902
2                  0.08758
3                  0.17300
4                  0.07678
5                  0.12440
6                  0.08368
7                  0.11510
8                  0.10720
9                  0.20750

[10 rows x 30 columns]
```

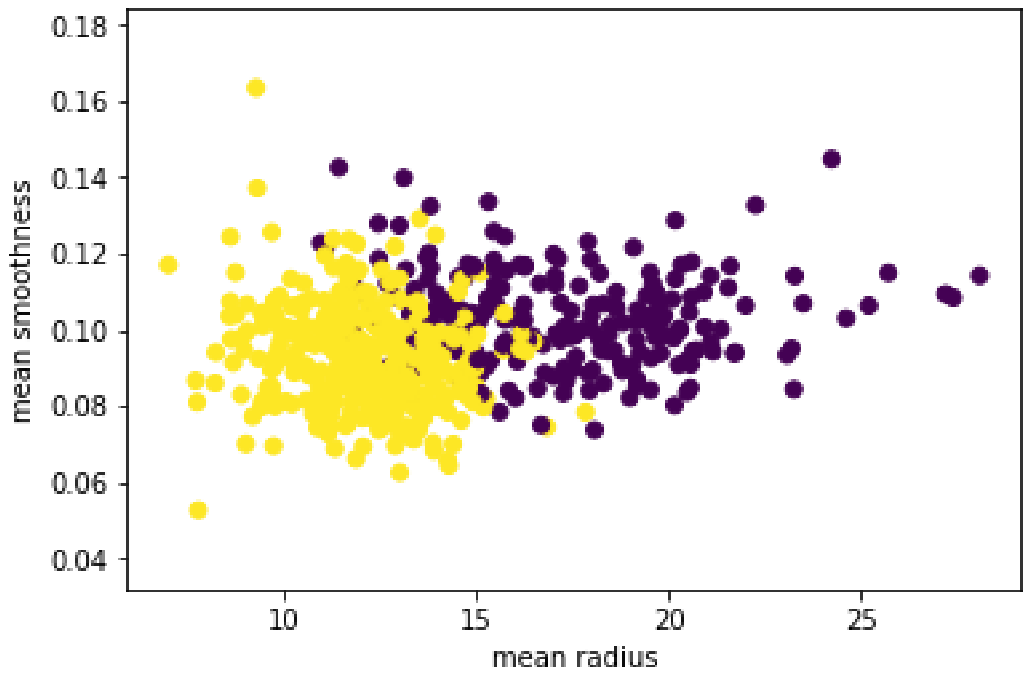
```
In [3]: import numpy as np
        print('Example number, feature number:', df.shape)
        print('Class Names:',list(breast_cancer.target_names))
        print('Class proportions:', np.bincount(breast_cancer.target))
```

```
('Example number, feature number:', (569, 30))
('Class Names:', ['malignant', 'benign'])
('Class proportions:', array([212, 357]))
```

4.2 Plotting the dataset

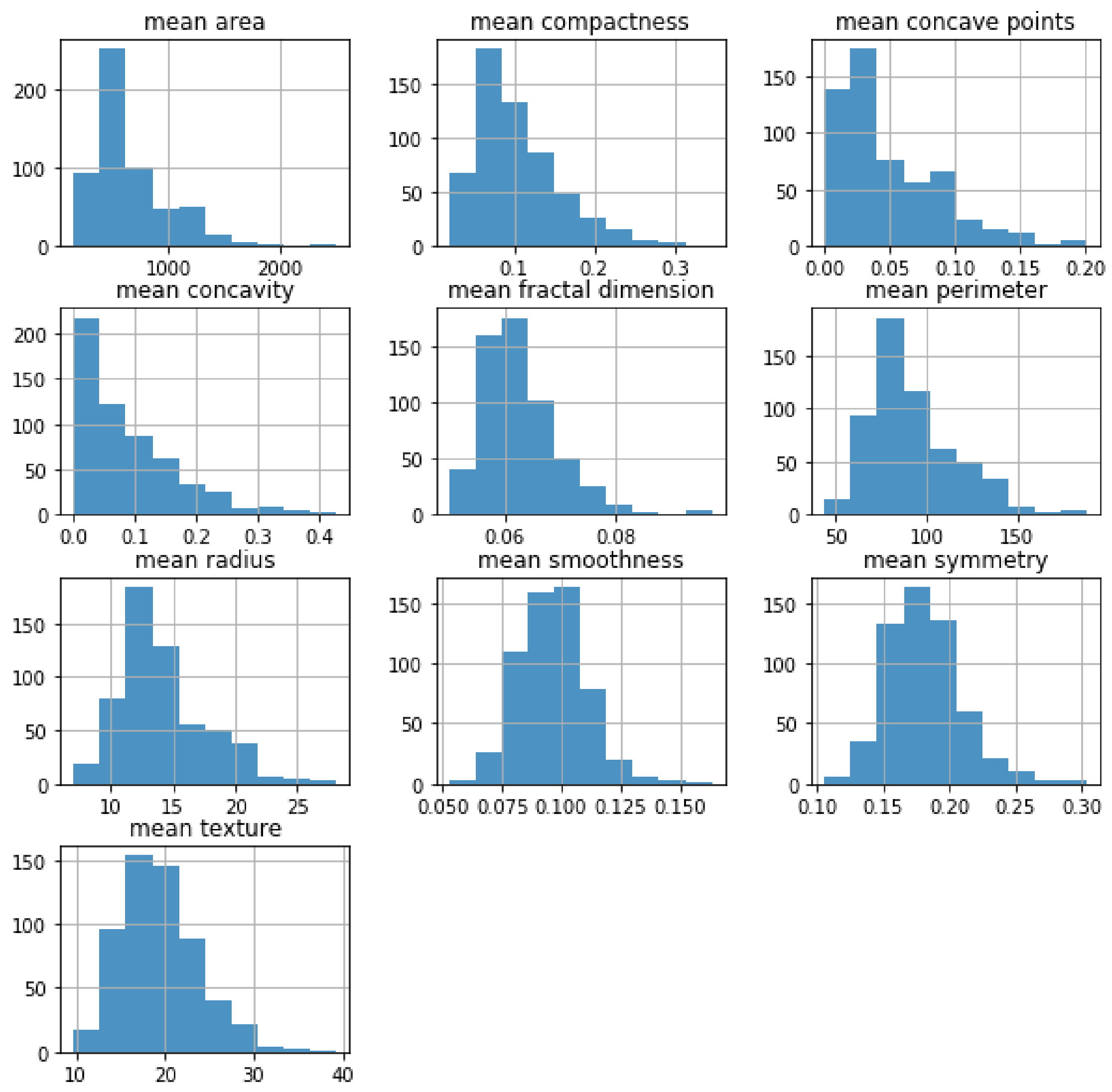
```
In [4]: %matplotlib inline
        import matplotlib.pyplot as plt
        plt.scatter(df['mean radius'], df['mean smoothness'], c=breast_cancer.target)
        plt.xlabel('mean radius')
        plt.ylabel('mean smoothness')
```

```
Out[4]: Text(0,0.5,u'mean smoothness')
```



Pandas has also some nice built-in plotting features, for instance you can plot the histograms of the features:

```
In [5]: df[breast_cancer.feature_names[0:10]].hist(alpha=0.8, figsize=(10, 10))
        plt.show()
```



What if we are interested in the different shape between the two classes? We could simply add the target to the DataFrame:

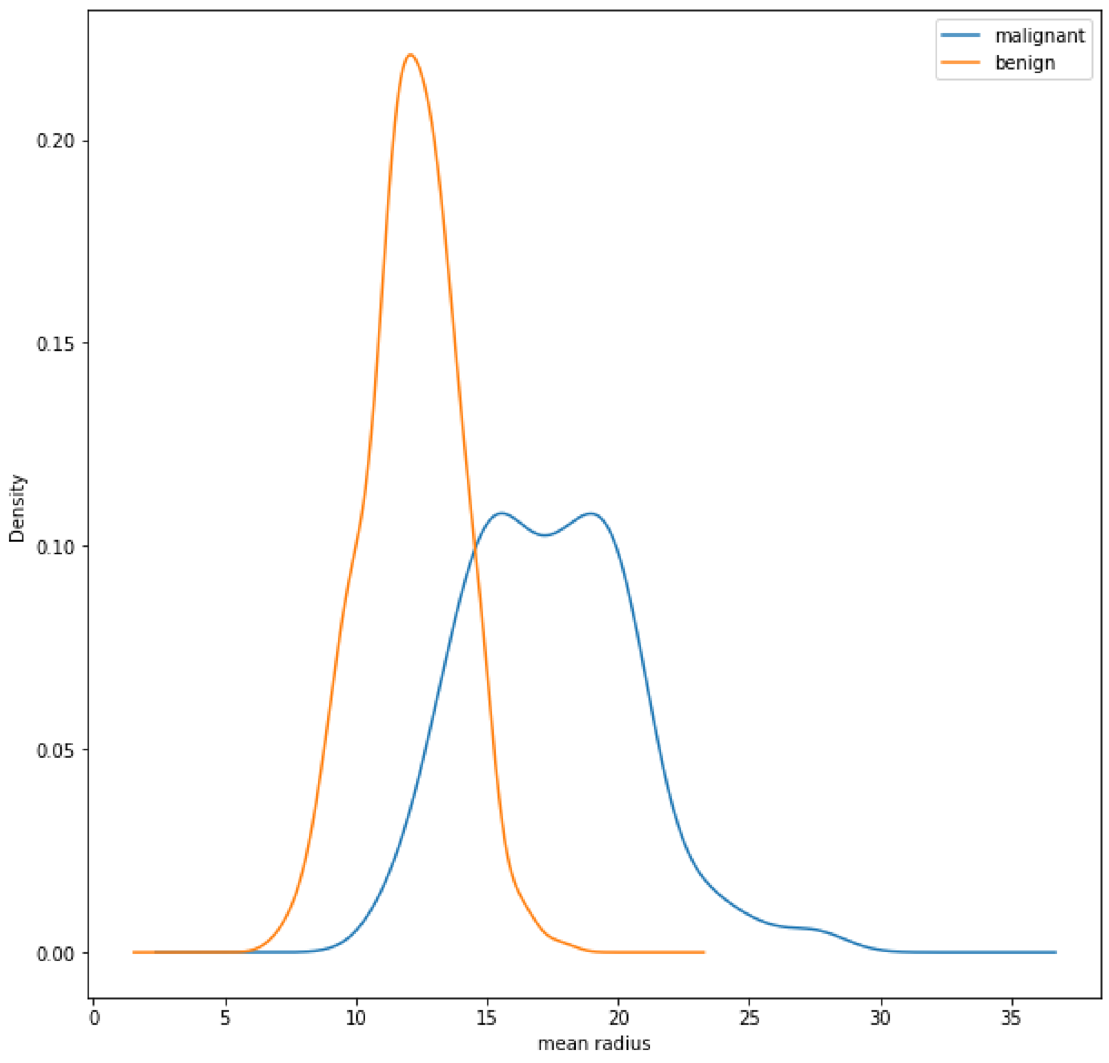
```
In [6]: df = df.assign(target=breast_cancer.target)
        df.keys()
```

```
Out[6]: Index([u'mean radius', u'mean texture', u'mean perimeter', u'mean area',
               u'mean smoothness', u'mean compactness', u'mean concavity',
               u'mean concave points', u'mean symmetry', u'mean fractal dimension',
               u'radius error', u'texture error', u'perimeter error', u'area error',
               u'smoothness error', u'compactness error', u'concavity error',
               u'concave points error', u'symmetry error', u'fractal dimension error',
               u'worst radius', u'worst texture', u'worst perimeter', u'worst area',
               u'worst smoothness', u'worst compactness', u'worst concavity',
               u'worst concave points', u'worst symmetry', u'worst fractal dimension',
               u'target'],
              dtype='object')
```

And then use the useful groupby function and plot a kernel density estimate (kde) plot:

```
In [7]: df.groupby("target")["mean radius"].plot(kind='kde', figsize=(10, 10))
        plt.legend(['malignant', 'benign'], loc='upper right')
        plt.xlabel('mean radius')
```

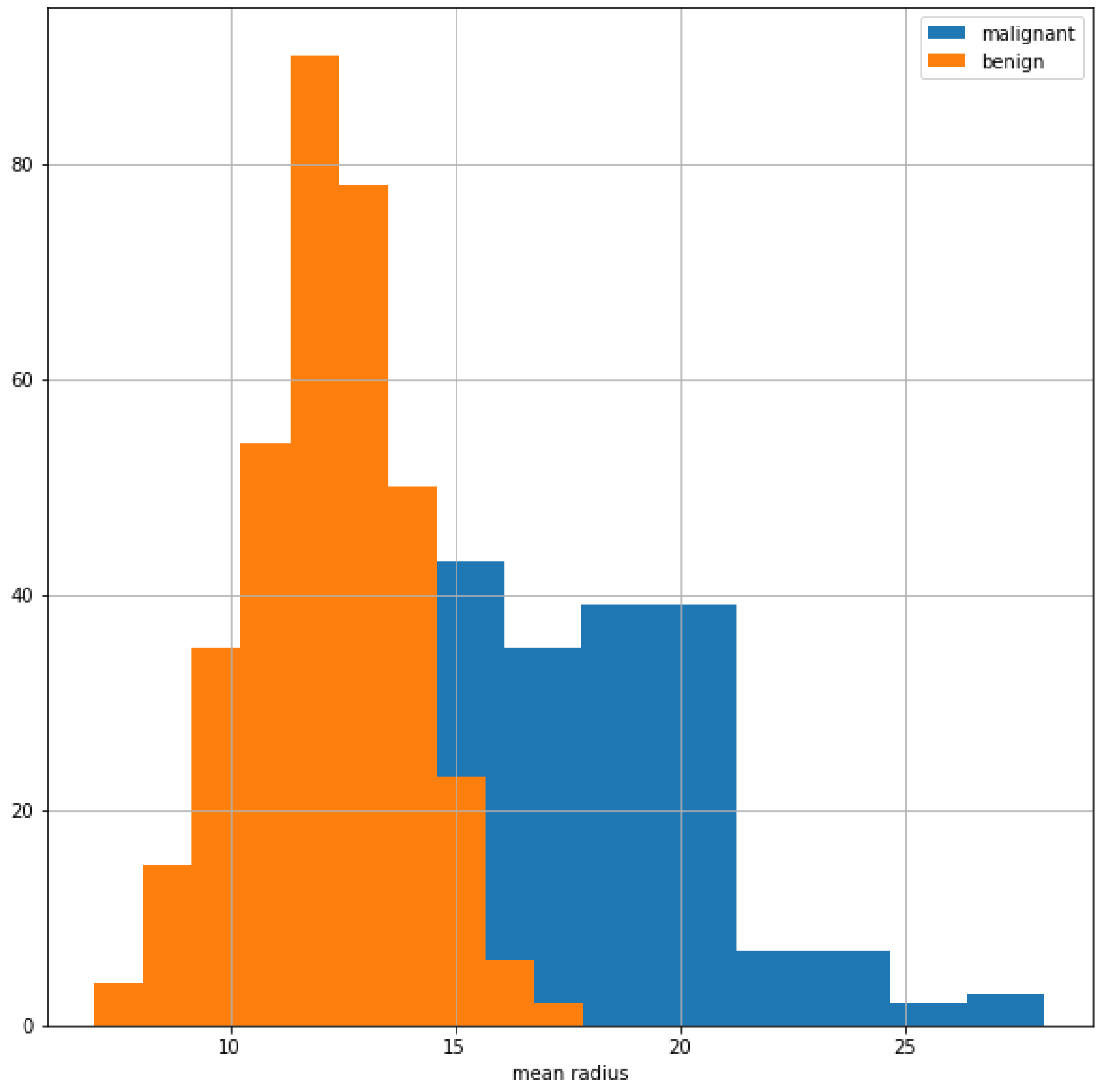
```
Out[7]: Text(0.5,0,u'mean radius')
```



Similarly, we could also plot the histogram:

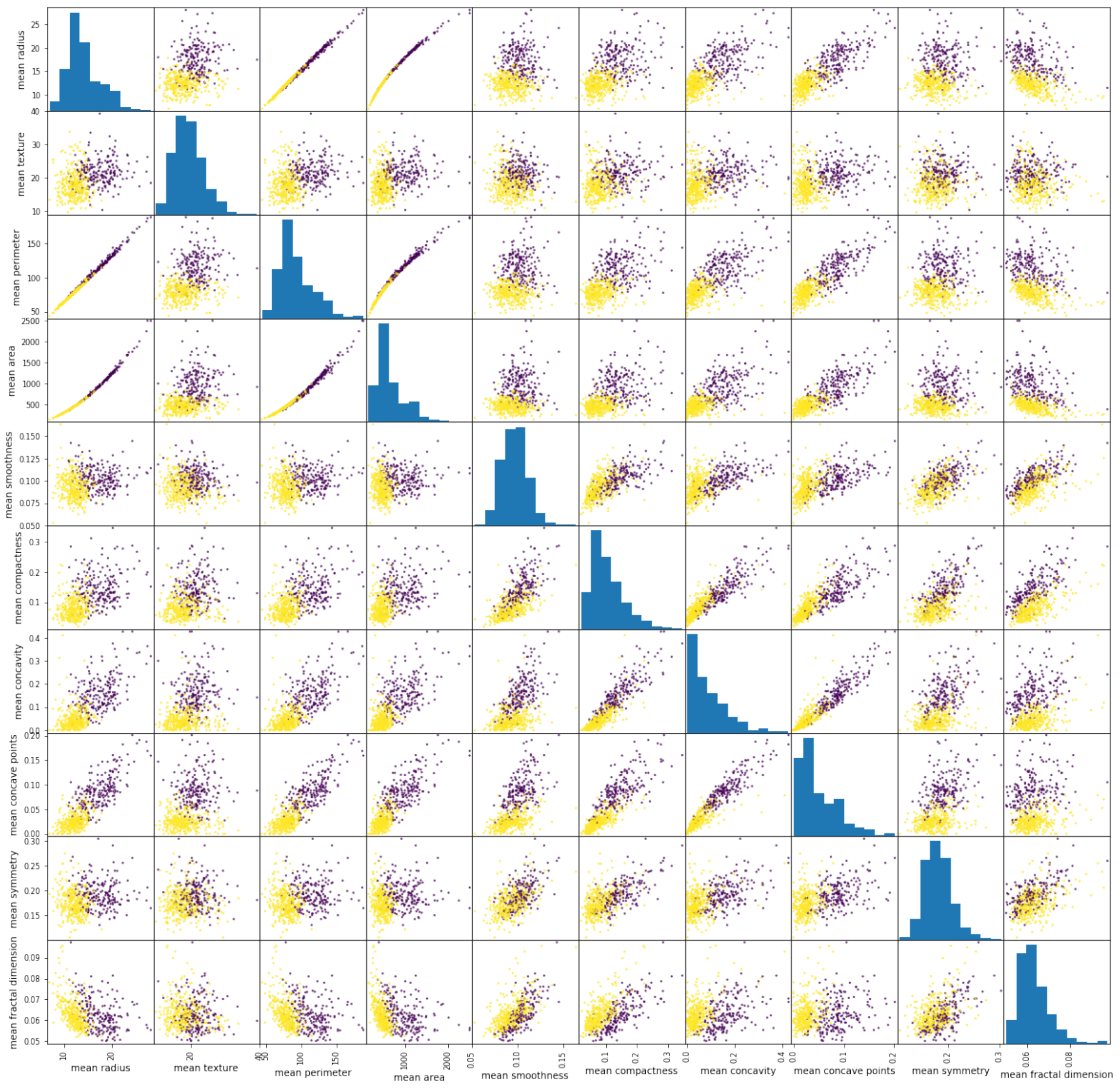
```
In [8]: df.groupby("target")["mean radius"].hist(fill=False, figsize=(10, 10))
        plt.legend(['malignant', 'benign'], loc='upper right')
        plt.xlabel('mean radius')
```

```
Out[8]: Text(0.5,0,u'mean radius')
```



From a DataFrame you can even plot the full scatter plot matrix:

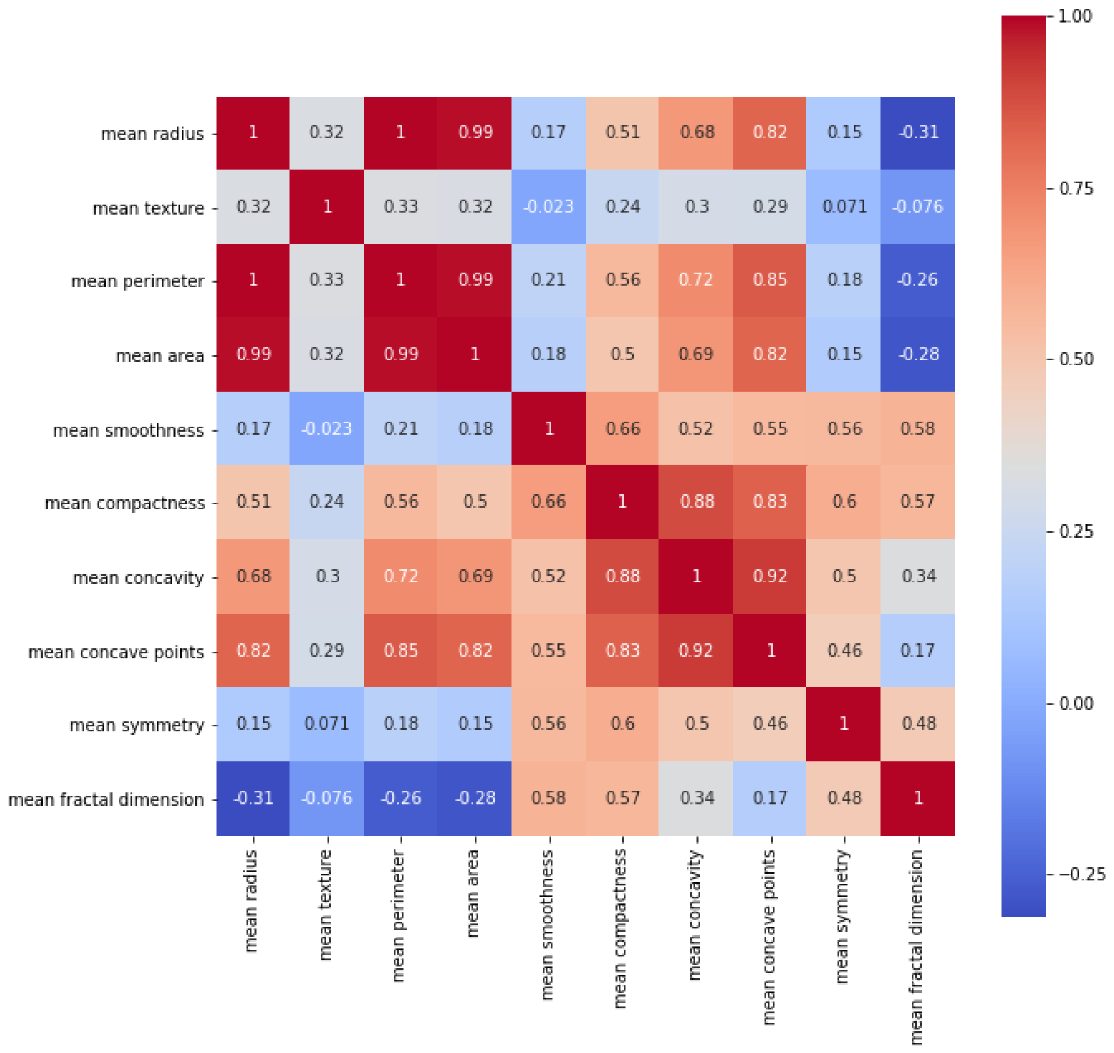
```
In [9]: from pandas.plotting import scatter_matrix
        scatter_matrix(df[breast_cancer.feature_names[0:10]], c=breast_cancer.target, alpha=0.8,
        plt.show()
```



Some of the input features seem highly correlated, so it usually makes sense to quantify their correlation to the other features. We will now use seaborn: statistical data visualization to obtain the (linear) correlations between the input features.

https://seaborn.pydata.org/

```
In [10]: import seaborn as sns
         plt.figure(figsize=(10,10))
         sns.heatmap(df[breast_cancer.feature_names[0:10]].corr(), annot=True, square=True, cmap
         plt.show()
```



4.3 Preparing the dataset

Just like scikit-learn, Keras, takes as inputs the following objects: *

Design matrix X

an `ndarray` of dimensions `[nb_examples, nb_features]` containing the distributions to be used as inputs to the model. Each row is an object to classify, each column corresponds to a specific variable. *

Target vector y

an `array` of dimensions `[nb_examples]` containing the truth labels indicating the class each object belongs to (for classification), or the continuous target values (for regression). *

Weight vector w

(optional) an `array` of dimensions `[nb_examples]` containing the weights to be assigned to each example

The indices of these objects must map to the same examples.

4.3.1 Task 2: Create design matrix and target vector for the first 10 features. Split the data into 70% training data and 30% testing data

```
In [11]: X = df[breast_cancer.feature_names[0:10]].as_matrix()
         y = breast_cancer.target
         print('Class proportions:', np.bincount(y))

('Class proportions:', array([212, 357]))


In [12]: type(X)

Out[12]: numpy.ndarray

In [13]: X.shape

Out[13]: (569, 10)

In [14]: from sklearn.model_selection import train_test_split
         X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=4

         print('Class proportions:', np.bincount(y_train))

('Class proportions:', array([148, 250]))
```

It is common practice to scale the inputs to neural nets such that they have approximately similar ranges. Without this step, you might end up with variables whose values span very different orders of magnitude. This will create problems in the NN convergence due to very wild fluctuations in the magnitude of the internal weights. To take care of the scaling, we use the `sklearn` `StandardScaler`:

```
In [15]: from sklearn.preprocessing import StandardScaler

In [16]: scaler = StandardScaler()
         X_train = scaler.fit_transform(X_train)
         X_test = scaler.transform(X_test)
```

5 3 Training a dense neural network

5.1 Neural network model

5.1.1 Dense layer structure

- Densely connected layer, where all inputs are connected to all outputs
- Linear transformation of the input vector $x \in \mathbb{R}^n$, which can be expressed using the $n \times m$ matrix $W \in \mathbb{R}^{n \times m}$ as:

  $u = Wx + b$

  where $b \in \mathbb{R}^m$ is the bias unit

- All entries in both W and b are trainable
- In Keras: `keras.layers.Dense( units, activation=None, use_bias=True, kernel_initializer='glorot_ bias_initializer='zeros', kernel_regularizer=None, bias_regularizer=None, activity_regularizer=None, kernel_constraint=None, bias_constraint=None )`
- `input_dim` (or `input_shape`) are necessary arguments for the 1st layer of the net:

"'python # as first layer in a sequential model: model = Sequential() model.add(Dense(32, input_shape=(16,))) # now the model will take as input arrays of shape (*, 16) # and output arrays of shape (*, 32)

6 after the first layer, you don't need to specify

7 the size of the input anymore:

model.add(Dense(32)) "'

7.0.1 Activation functions

- Mathematical way of quantifying the activation state of a node $\rightarrow$ whether it's firing or not
- Non-linear activation functions are the key to Deep Learning
- Allow NNs to learn complex, non-linear transformations of the inputs
- Some popular choices: Available activations (https://keras.io/activations/#available-activations):
- **softmax**, **elu**, **selu**, **softplus**, **softsign**, **relu**, **tanh**, **sigmoid**, **hard_sigmoid**, **linear**

Advanced Activation (https://keras.io/layers/advanced-activations/): * **LeakyReLU**, **PReLU**

7.0.2 Loss functions

- Mathematical way of quantifying how much y deviates from y
- Dictates how strongly we penalize certain types of mistakes
- Cost of inaccurately classifying an event
- Many loss functions available in kears (https://keras.io/losses/)

7.1 Build a simple neural network

```
In [17]: from keras.models import Sequential

         model = Sequential()
```

```
Using TensorFlow backend.
```

```
---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

RuntimeError: module compiled against API version 0xc but this version of numpy is 0xb



---------------------------------------------------------------------------

RuntimeError                              Traceback (most recent call last)

RuntimeError: module compiled against API version 0xc but this version of numpy is 0xb
```

```
In [18]: from keras.layers import Dense
```

```
In [19]: model.add(Dense(units=11, activation='relu', input_dim=10))
         model.add(Dense(units=1, activation='sigmoid'))
```

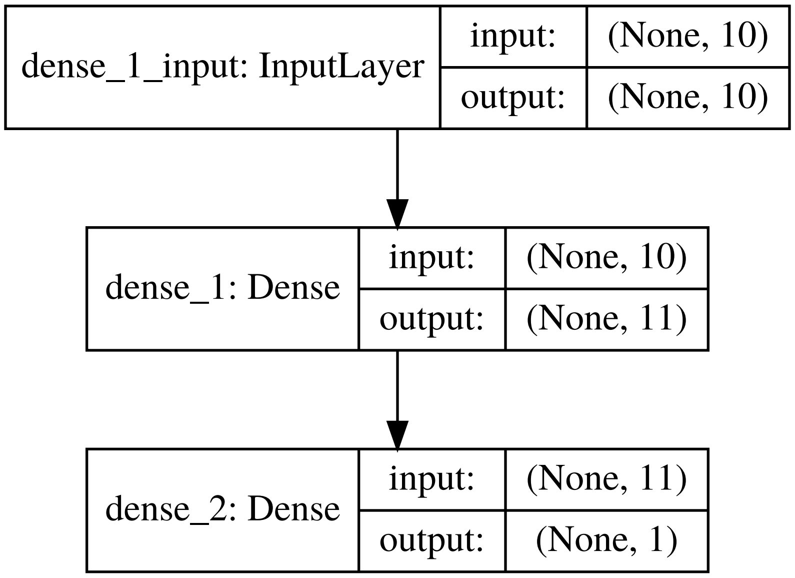
```
In [20]: model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 11)                121
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 12
=================================================================
Total params: 133
Trainable params: 133
Non-trainable params: 0
_________________________________________________________________
```

Let's visualize our net:

```
In [21]: from IPython.display import SVG
         from keras.utils.vis_utils import model_to_dot
         import matplotlib.image as mpimg
         SVG(model_to_dot(model, show_shapes=True).create(prog='dot', format='svg'))
```

```
Out[21]:
```



OK, that is a rather simple model, but let's define a loss function, an optimizer, a performance metric and compile it:

```
In [22]: model.compile(loss='binary_crossentropy',
                       optimizer='sgd',
                       metrics=['accuracy'])
```

7.1.1 Training

In order to train the model, we pass the training data to the fit function. However, part of the training data will be used as validation data, which is used during the training to evaluate the training process.

```
In [23]: # x_train and y_train are Numpy arrays --just like in the Scikit-Learn API.
         history = model.fit(X_train, y_train, validation_split=0.3, epochs=100, batch_size=8)
```

```
Train on 278 samples, validate on 120 samples
Epoch 1/100
278/278 [==============================] - 0s 892us/step - loss: 0.6721 - acc: 0.5935 - val_loss
Epoch 2/100
278/278 [==============================] - 0s 229us/step - loss: 0.5088 - acc: 0.8345 - val_loss
Epoch 3/100
278/278 [==============================] - 0s 226us/step - loss: 0.4219 - acc: 0.8885 - val_loss
Epoch 4/100
278/278 [==============================] - 0s 260us/step - loss: 0.3680 - acc: 0.9245 - val_loss
Epoch 5/100
278/278 [==============================] - 0s 225us/step - loss: 0.3304 - acc: 0.9317 - val_loss
Epoch 6/100
278/278 [==============================] - 0s 225us/step - loss: 0.3022 - acc: 0.9353 - val_loss
```

```
Epoch 7/100
278/278 [==============================] - 0s 221us/step - loss: 0.2806 - acc: 0.9353 - val_loss
Epoch 8/100
278/278 [==============================] - 0s 252us/step - loss: 0.2633 - acc: 0.9353 - val_loss
Epoch 9/100
278/278 [==============================] - 0s 209us/step - loss: 0.2494 - acc: 0.9353 - val_loss
Epoch 10/100
278/278 [==============================] - 0s 217us/step - loss: 0.2380 - acc: 0.9388 - val_loss
Epoch 11/100
278/278 [==============================] - 0s 251us/step - loss: 0.2286 - acc: 0.9388 - val_loss
Epoch 12/100
278/278 [==============================] - 0s 249us/step - loss: 0.2203 - acc: 0.9388 - val_loss
Epoch 13/100
278/278 [==============================] - 0s 214us/step - loss: 0.2133 - acc: 0.9424 - val_loss
Epoch 14/100
278/278 [==============================] - 0s 251us/step - loss: 0.2070 - acc: 0.9424 - val_loss
Epoch 15/100
278/278 [==============================] - 0s 222us/step - loss: 0.2016 - acc: 0.9424 - val_loss
Epoch 16/100
278/278 [==============================] - 0s 239us/step - loss: 0.1965 - acc: 0.9424 - val_loss
Epoch 17/100
278/278 [==============================] - 0s 226us/step - loss: 0.1921 - acc: 0.9424 - val_loss
Epoch 18/100
278/278 [==============================] - 0s 238us/step - loss: 0.1882 - acc: 0.9424 - val_loss
Epoch 19/100
278/278 [==============================] - 0s 240us/step - loss: 0.1847 - acc: 0.9460 - val_loss
Epoch 20/100
278/278 [==============================] - 0s 256us/step - loss: 0.1814 - acc: 0.9460 - val_loss
Epoch 21/100
278/278 [==============================] - 0s 245us/step - loss: 0.1785 - acc: 0.9460 - val_loss
Epoch 22/100
278/278 [==============================] - 0s 243us/step - loss: 0.1758 - acc: 0.9496 - val_loss
Epoch 23/100
278/278 [==============================] - 0s 222us/step - loss: 0.1734 - acc: 0.9496 - val_loss
Epoch 24/100
278/278 [==============================] - 0s 213us/step - loss: 0.1711 - acc: 0.9496 - val_loss
Epoch 25/100
278/278 [==============================] - 0s 240us/step - loss: 0.1692 - acc: 0.9496 - val_loss
Epoch 26/100
278/278 [==============================] - 0s 221us/step - loss: 0.1672 - acc: 0.9496 - val_loss
Epoch 27/100
278/278 [==============================] - 0s 207us/step - loss: 0.1654 - acc: 0.9496 - val_loss
Epoch 28/100
278/278 [==============================] - 0s 232us/step - loss: 0.1637 - acc: 0.9532 - val_loss
Epoch 29/100
278/278 [==============================] - 0s 228us/step - loss: 0.1620 - acc: 0.9532 - val_loss
Epoch 30/100
278/278 [==============================] - 0s 236us/step - loss: 0.1606 - acc: 0.9532 - val_loss
```

```
Epoch 31/100
278/278 [==============================] - 0s 206us/step - loss: 0.1591 - acc: 0.9532 - val_loss
Epoch 32/100
278/278 [==============================] - 0s 217us/step - loss: 0.1578 - acc: 0.9568 - val_loss
Epoch 33/100
278/278 [==============================] - 0s 252us/step - loss: 0.1564 - acc: 0.9568 - val_loss
Epoch 34/100
278/278 [==============================] - 0s 262us/step - loss: 0.1550 - acc: 0.9568 - val_loss
Epoch 35/100
278/278 [==============================] - 0s 253us/step - loss: 0.1538 - acc: 0.9568 - val_loss
Epoch 36/100
278/278 [==============================] - 0s 244us/step - loss: 0.1526 - acc: 0.9568 - val_loss
Epoch 37/100
278/278 [==============================] - 0s 256us/step - loss: 0.1515 - acc: 0.9568 - val_loss
Epoch 38/100
278/278 [==============================] - 0s 238us/step - loss: 0.1504 - acc: 0.9568 - val_loss
Epoch 39/100
278/278 [==============================] - 0s 253us/step - loss: 0.1494 - acc: 0.9568 - val_loss
Epoch 40/100
278/278 [==============================] - 0s 224us/step - loss: 0.1484 - acc: 0.9604 - val_loss
Epoch 41/100
278/278 [==============================] - 0s 220us/step - loss: 0.1474 - acc: 0.9604 - val_loss
Epoch 42/100
278/278 [==============================] - 0s 212us/step - loss: 0.1466 - acc: 0.9604 - val_loss
Epoch 43/100
278/278 [==============================] - 0s 205us/step - loss: 0.1457 - acc: 0.9604 - val_loss
Epoch 44/100
278/278 [==============================] - 0s 229us/step - loss: 0.1449 - acc: 0.9604 - val_loss
Epoch 45/100
278/278 [==============================] - 0s 261us/step - loss: 0.1441 - acc: 0.9604 - val_loss
Epoch 46/100
278/278 [==============================] - 0s 224us/step - loss: 0.1434 - acc: 0.9604 - val_loss
Epoch 47/100
278/278 [==============================] - 0s 230us/step - loss: 0.1426 - acc: 0.9604 - val_loss
Epoch 48/100
278/278 [==============================] - 0s 231us/step - loss: 0.1418 - acc: 0.9604 - val_loss
Epoch 49/100
278/278 [==============================] - 0s 227us/step - loss: 0.1412 - acc: 0.9604 - val_loss
Epoch 50/100
278/278 [==============================] - 0s 224us/step - loss: 0.1404 - acc: 0.9604 - val_loss
Epoch 51/100
278/278 [==============================] - 0s 224us/step - loss: 0.1398 - acc: 0.9604 - val_loss
Epoch 52/100
278/278 [==============================] - 0s 228us/step - loss: 0.1392 - acc: 0.9640 - val_loss
Epoch 53/100
278/278 [==============================] - 0s 232us/step - loss: 0.1385 - acc: 0.9604 - val_loss
Epoch 54/100
278/278 [==============================] - 0s 226us/step - loss: 0.1380 - acc: 0.9640 - val_loss
```

```
Epoch 55/100
278/278 [==============================] - 0s 226us/step - loss: 0.1373 - acc: 0.9640 - val_loss
Epoch 56/100
278/278 [==============================] - 0s 220us/step - loss: 0.1367 - acc: 0.9640 - val_loss
Epoch 57/100
278/278 [==============================] - 0s 227us/step - loss: 0.1362 - acc: 0.9640 - val_loss
Epoch 58/100
278/278 [==============================] - 0s 258us/step - loss: 0.1356 - acc: 0.9640 - val_loss
Epoch 59/100
278/278 [==============================] - 0s 259us/step - loss: 0.1352 - acc: 0.9676 - val_loss
Epoch 60/100
278/278 [==============================] - 0s 234us/step - loss: 0.1347 - acc: 0.9640 - val_loss
Epoch 61/100
278/278 [==============================] - 0s 249us/step - loss: 0.1342 - acc: 0.9640 - val_loss
Epoch 62/100
278/278 [==============================] - 0s 228us/step - loss: 0.1336 - acc: 0.9676 - val_loss
Epoch 63/100
278/278 [==============================] - 0s 207us/step - loss: 0.1331 - acc: 0.9676 - val_loss
Epoch 64/100
278/278 [==============================] - 0s 221us/step - loss: 0.1326 - acc: 0.9676 - val_loss
Epoch 65/100
278/278 [==============================] - 0s 219us/step - loss: 0.1323 - acc: 0.9676 - val_loss
Epoch 66/100
278/278 [==============================] - 0s 204us/step - loss: 0.1317 - acc: 0.9676 - val_loss
Epoch 67/100
278/278 [==============================] - 0s 200us/step - loss: 0.1312 - acc: 0.9676 - val_loss
Epoch 68/100
278/278 [==============================] - 0s 213us/step - loss: 0.1309 - acc: 0.9676 - val_loss
Epoch 69/100
278/278 [==============================] - 0s 206us/step - loss: 0.1305 - acc: 0.9676 - val_loss
Epoch 70/100
278/278 [==============================] - 0s 226us/step - loss: 0.1300 - acc: 0.9676 - val_loss
Epoch 71/100
278/278 [==============================] - 0s 211us/step - loss: 0.1296 - acc: 0.9676 - val_loss
Epoch 72/100
278/278 [==============================] - 0s 232us/step - loss: 0.1294 - acc: 0.9676 - val_loss
Epoch 73/100
278/278 [==============================] - 0s 213us/step - loss: 0.1290 - acc: 0.9676 - val_loss
Epoch 74/100
278/278 [==============================] - 0s 224us/step - loss: 0.1285 - acc: 0.9676 - val_loss
Epoch 75/100
278/278 [==============================] - 0s 221us/step - loss: 0.1282 - acc: 0.9676 - val_loss
Epoch 76/100
278/278 [==============================] - 0s 221us/step - loss: 0.1279 - acc: 0.9676 - val_loss
Epoch 77/100
278/278 [==============================] - 0s 207us/step - loss: 0.1274 - acc: 0.9676 - val_loss
Epoch 78/100
278/278 [==============================] - 0s 227us/step - loss: 0.1271 - acc: 0.9676 - val_loss
```

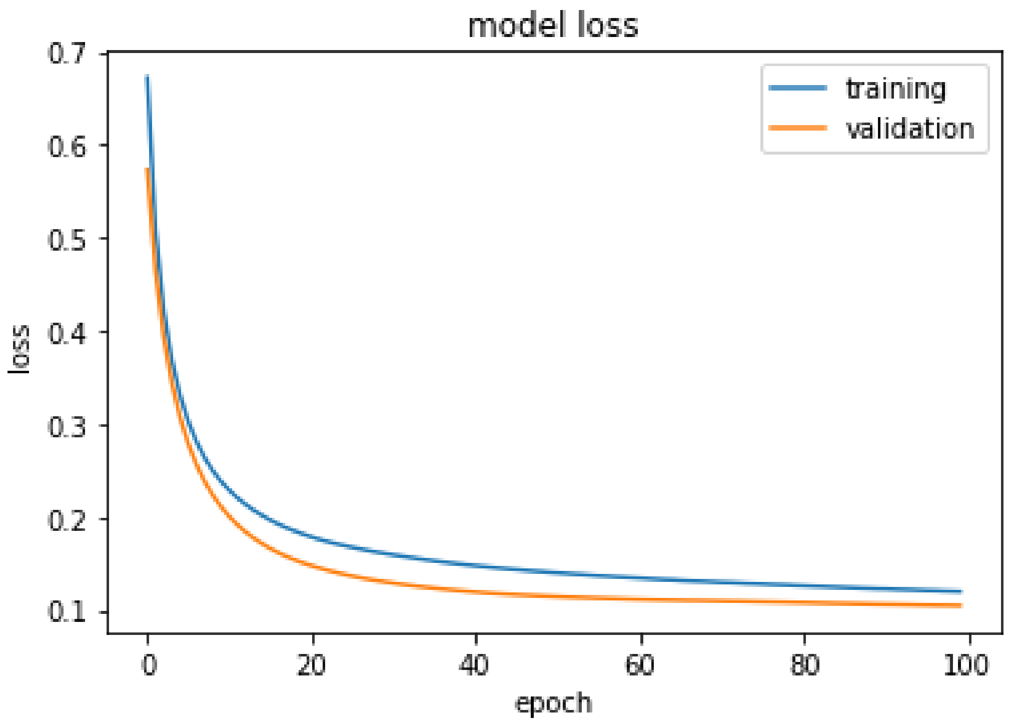
```
Epoch 79/100
278/278 [==============================] - 0s 228us/step - loss: 0.1268 - acc: 0.9676 - val_loss
Epoch 80/100
278/278 [==============================] - 0s 206us/step - loss: 0.1263 - acc: 0.9676 - val_loss
Epoch 81/100
278/278 [==============================] - 0s 211us/step - loss: 0.1260 - acc: 0.9676 - val_loss
Epoch 82/100
278/278 [==============================] - 0s 229us/step - loss: 0.1256 - acc: 0.9676 - val_loss
Epoch 83/100
278/278 [==============================] - 0s 214us/step - loss: 0.1253 - acc: 0.9676 - val_loss
Epoch 84/100
278/278 [==============================] - 0s 198us/step - loss: 0.1248 - acc: 0.9676 - val_loss
Epoch 85/100
278/278 [==============================] - 0s 210us/step - loss: 0.1245 - acc: 0.9676 - val_loss
Epoch 86/100
278/278 [==============================] - 0s 204us/step - loss: 0.1241 - acc: 0.9676 - val_loss
Epoch 87/100
278/278 [==============================] - 0s 213us/step - loss: 0.1238 - acc: 0.9676 - val_loss
Epoch 88/100
278/278 [==============================] - 0s 220us/step - loss: 0.1234 - acc: 0.9676 - val_loss
Epoch 89/100
278/278 [==============================] - 0s 205us/step - loss: 0.1231 - acc: 0.9676 - val_loss
Epoch 90/100
278/278 [==============================] - 0s 208us/step - loss: 0.1228 - acc: 0.9676 - val_loss
Epoch 91/100
278/278 [==============================] - 0s 212us/step - loss: 0.1226 - acc: 0.9676 - val_loss
Epoch 92/100
278/278 [==============================] - 0s 204us/step - loss: 0.1222 - acc: 0.9676 - val_loss
Epoch 93/100
278/278 [==============================] - 0s 257us/step - loss: 0.1219 - acc: 0.9676 - val_loss
Epoch 94/100
278/278 [==============================] - 0s 205us/step - loss: 0.1216 - acc: 0.9676 - val_loss
Epoch 95/100
278/278 [==============================] - 0s 206us/step - loss: 0.1213 - acc: 0.9676 - val_loss
Epoch 96/100
278/278 [==============================] - 0s 222us/step - loss: 0.1209 - acc: 0.9676 - val_loss
Epoch 97/100
278/278 [==============================] - 0s 216us/step - loss: 0.1207 - acc: 0.9676 - val_loss
Epoch 98/100
278/278 [==============================] - 0s 246us/step - loss: 0.1203 - acc: 0.9676 - val_loss
Epoch 99/100
278/278 [==============================] - 0s 224us/step - loss: 0.1201 - acc: 0.9676 - val_loss
Epoch 100/100
278/278 [==============================] - 0s 207us/step - loss: 0.1198 - acc: 0.9676 - val_loss
```

During the training process we have saved the loss and the accuracy of the training and validation data:

```
In [24]: print(history.history.keys())

['acc', 'loss', 'val_acc', 'val_loss']
```

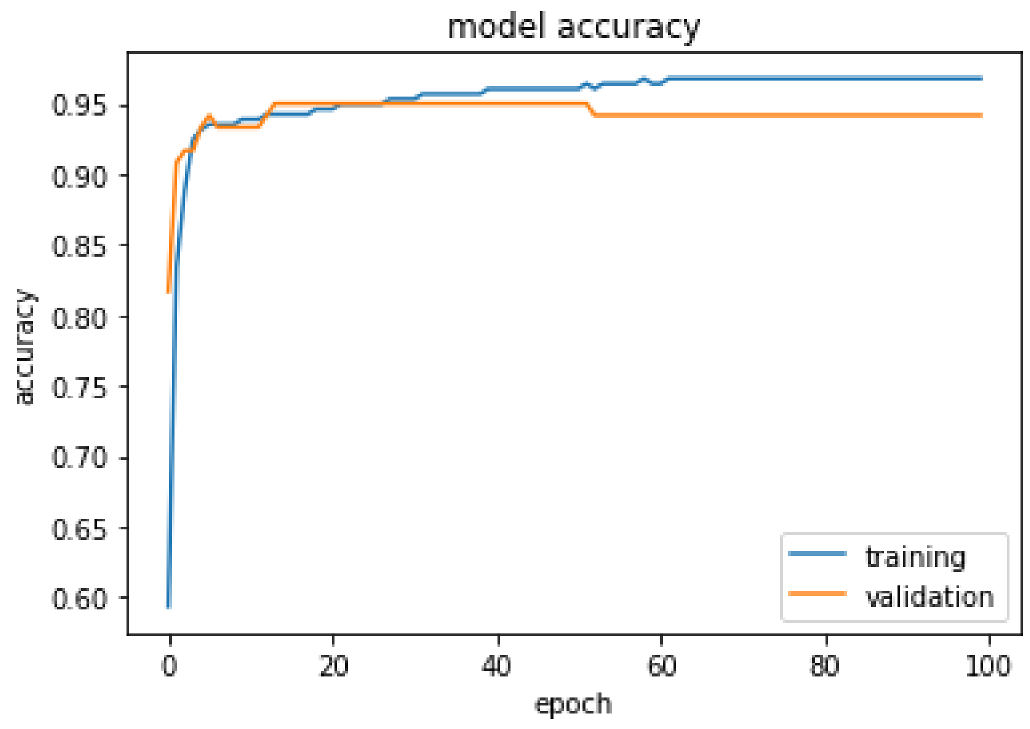
We can now plot the loss evolution over the training epochs for the training and validation dataset:

```
In [25]: # summarize history for loss
         plt.plot(history.history['loss'])
         plt.plot(history.history['val_loss'])
         plt.title('model loss')
         plt.ylabel('loss')
         plt.xlabel('epoch')
         plt.legend(['training', 'validation'], loc='upper right')
         plt.show()
```



Similarly, we can plot the accuracy

```
In [26]: # summarize history for accuracy
         plt.plot(history.history['acc'])
         plt.plot(history.history['val_acc'])
         plt.title('model accuracy')
         plt.ylabel('accuracy')
         plt.xlabel('epoch')
         plt.legend(['training', 'validation'], loc='lower right')
         plt.show()
```



7.1.2 Evaluation

Let's evaluate the loss and accuracy on our test data:

```
In [27]: loss_and_metrics = model.evaluate(X_test, y_test, batch_size=8)
         print loss_and_metrics

171/171 [==============================] - 0s 99us/step
[0.16818849265313984, 0.935672514794165558]
```

Let's predict classes for our test data:

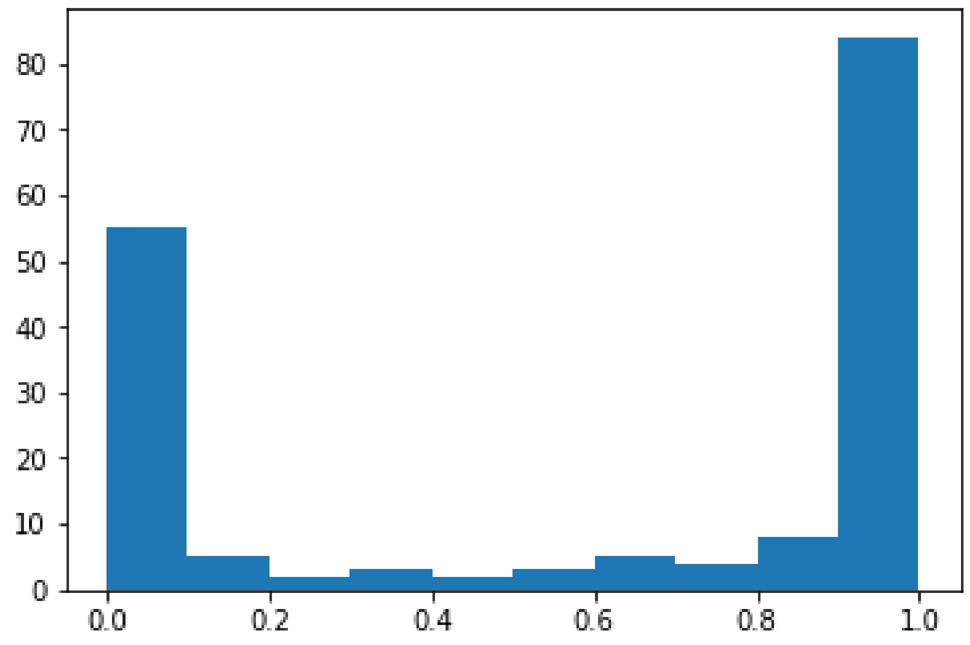
```
In [28]: print 'Testing...'
         y_pred = model.predict(X_test, verbose = True, batch_size=8)

Testing...
171/171 [==============================] - 0s 112us/step


In [29]: # predictions
         y_pred
```

```
Out[29]: array([[  2.46911887e-02],
                [  6.41295671e-01],
                [  9.16472256e-01],
                [  3.35365476e-05],
                [  1.07638262e-01],
                [  4.39718086e-03],
                [  9.99759257e-01],
                [  1.58661097e-01],
                [  9.68111098e-01],
                [  4.81340479e-09],
                [  9.99706089e-01],
                [  9.91387069e-01],
                [  1.16191812e-01],
                [  9.10929143e-01],
                [  9.95714128e-01],
                [  8.39511573e-01],
                [  5.38001861e-03],
                [  3.64673324e-02],
                [  6.35645390e-01],
                [  9.88984287e-01],
                [  9.31416512e-01],
                [  9.73866880e-01],
                [  9.72761154e-01],
                [  1.99083192e-03],
                [  4.01586201e-03],
                [  9.21817064e-01],
                [  9.96057987e-01],
                [  9.90339577e-01],
                [  9.00548825e-04],
                [  1.30316173e-03],
                [  9.06694591e-01],
                [  8.41965899e-03],
                [  7.86374390e-01],
                [  9.91148591e-01],
                [  2.13338208e-05],
                [  9.49174106e-01],
                [  6.65358901e-01],
                [  9.28168952e-01],
                [  9.86651659e-01],
                [  9.57263708e-01],
                [  5.45050859e-01],
                [  8.55147932e-03],
                [  9.98883665e-01],
                [  9.99613345e-01],
                [  9.79291141e-01],
                [  3.40676755e-01],
                [  1.75773516e-01],
                [  9.93146956e-01],
                [  9.95028555e-01],
                [  2.43489019e-04],
                [  9.95117784e-01],
                [  9.93210733e-01],
                [  1.29828128e-04],
                [  2.01991497e-04],
                [  9.95893002e-01],
                [  9.87600446e-01],
                [  1.13517940e-02],
                [  9.90155697e-01],
                [  9.38355029e-01],
                [  7.45531917e-01],
                [  2.37063738e-03],
                [  5.27730491e-03],
                [  8.73368979e-03],
                [  9.99973297e-01],
                [  9.99664426e-01],
                [  9.81994331e-01],
                [  7.49466777e-01],
                [  6.11107183e-08],
                [  9.99742687e-01],
                [  9.90418017e-01],
                [  2.45343131e-07],
                [  4.59963791e-02],
                [  1.44211715e-03],
                [  1.01750129e-05],
                [  8.20312202e-01],
                [  9.99532700e-01],
                [  5.76508284e-01],
                [  8.96664739e-01],
                [  7.80302857e-04],
                [  9.93407965e-01],
                [  9.98713493e-01],
                [  8.94496500e-01],
                [  4.27441997e-03],
                [  9.91489649e-01],
                [  1.64419514e-04],
                [  9.98361170e-01],
                [  9.78278160e-01],
                [  2.99317180e-03],
                [  1.54567685e-03],
                [  2.97812303e-03],
                [  3.28693422e-03],
                [  9.93894875e-01],
                [  9.39137280e-01],
                [  9.98142242e-01],
                [  9.69641387e-01],
                [  9.24268186e-01],
                [  8.45296457e-02],
                [  9.89694715e-01],
                [  4.81933624e-01],
                [  9.99922872e-01],
                [  5.60967147e-01],
                [  9.99429643e-01],
                [  9.27431345e-01],
                [  3.62866074e-02],
                [  8.72557461e-01],
                [  9.89689946e-01],
                [  9.01740313e-01],
                [  9.97870445e-01],
                [  1.30972369e-02],
                [  9.90675330e-01],
                [  2.33159307e-02],
                [  9.30954874e-01],
                [  1.03828922e-01],
                [  9.66733456e-01],
                [  9.00904953e-01],
                [  1.92238819e-02],
                [  3.55284601e-01],
                [  4.68693614e-01],
                [  9.99207556e-01],
                [  3.06845432e-05],
                [  9.77330923e-01],
                [  9.87964869e-01],
                [  1.34032845e-04],
                [  9.95950937e-01],
                [  6.65461421e-01],
                [  9.51046944e-01],
                [  8.55252743e-01],
                [  7.42556062e-03],
                [  7.10113525e-01],
                [  9.56171930e-01],
                [  8.88599336e-01],
                [  9.98733461e-01],
                [  3.78717817e-02],
                [  9.98944223e-01],
                [  2.27548718e-03],
                [  9.99981165e-01],
                [  7.33781653e-03],
                [  2.21329838e-01],
                [  4.70891632e-02],
                [  8.92862678e-01],
                [  9.89287496e-01],
                [  9.83670235e-01],
                [  3.45047726e-03],
                [  1.80506846e-03],
                [  2.31159447e-09],
                [  3.03969719e-03],
                [  9.79367527e-04],
                [  9.55257297e-01],
                [  2.19901547e-01],
                [  9.99822795e-01],
                [  9.98957872e-01],
                [  9.99074578e-01],
                [  1.63144396e-08],
                [  9.96969521e-01],
                [  6.24797354e-03],
                [  9.91940916e-01],
                [  9.99762237e-01],
                [  3.28491330e-02],
                [  9.88190889e-01],
                [  9.59087729e-01],
                [  1.18922731e-02],
                [  4.16963324e-02],
                [  6.14972889e-01],
                [  9.92373168e-01],
                [  9.12556774e-04],
                [  9.97959375e-01],
                [  9.85522151e-01],
                [  9.99719322e-01],
                [  3.16778123e-01],
                [  9.39125597e-01],
                [  9.95830357e-01]], dtype=float32)
```

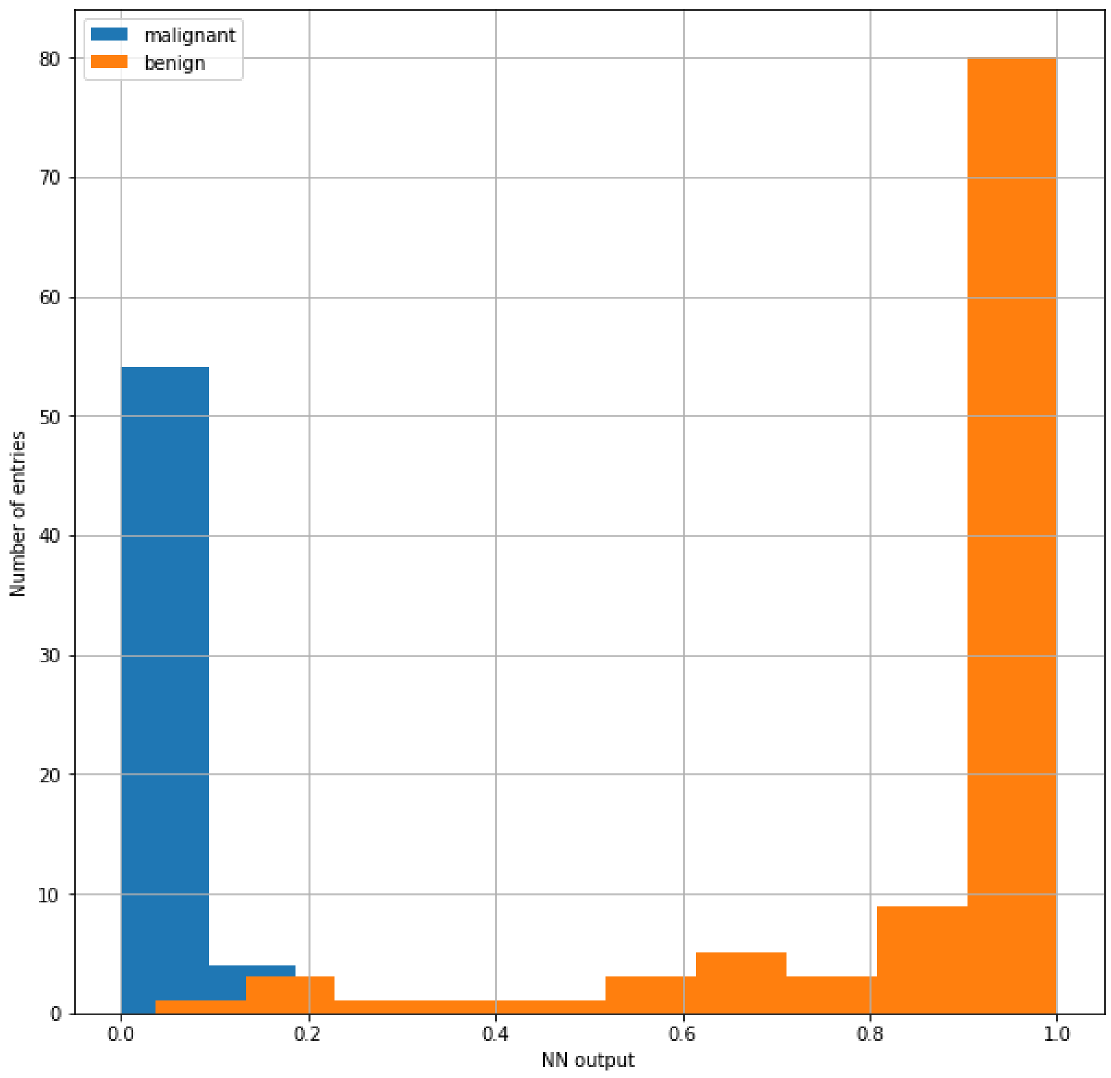
7.1.3 Task 3: Plot the output prediction for malignant and benign breast cancer showing the separation between these two classes.

```
In [30]: plt.hist(y_pred)
```

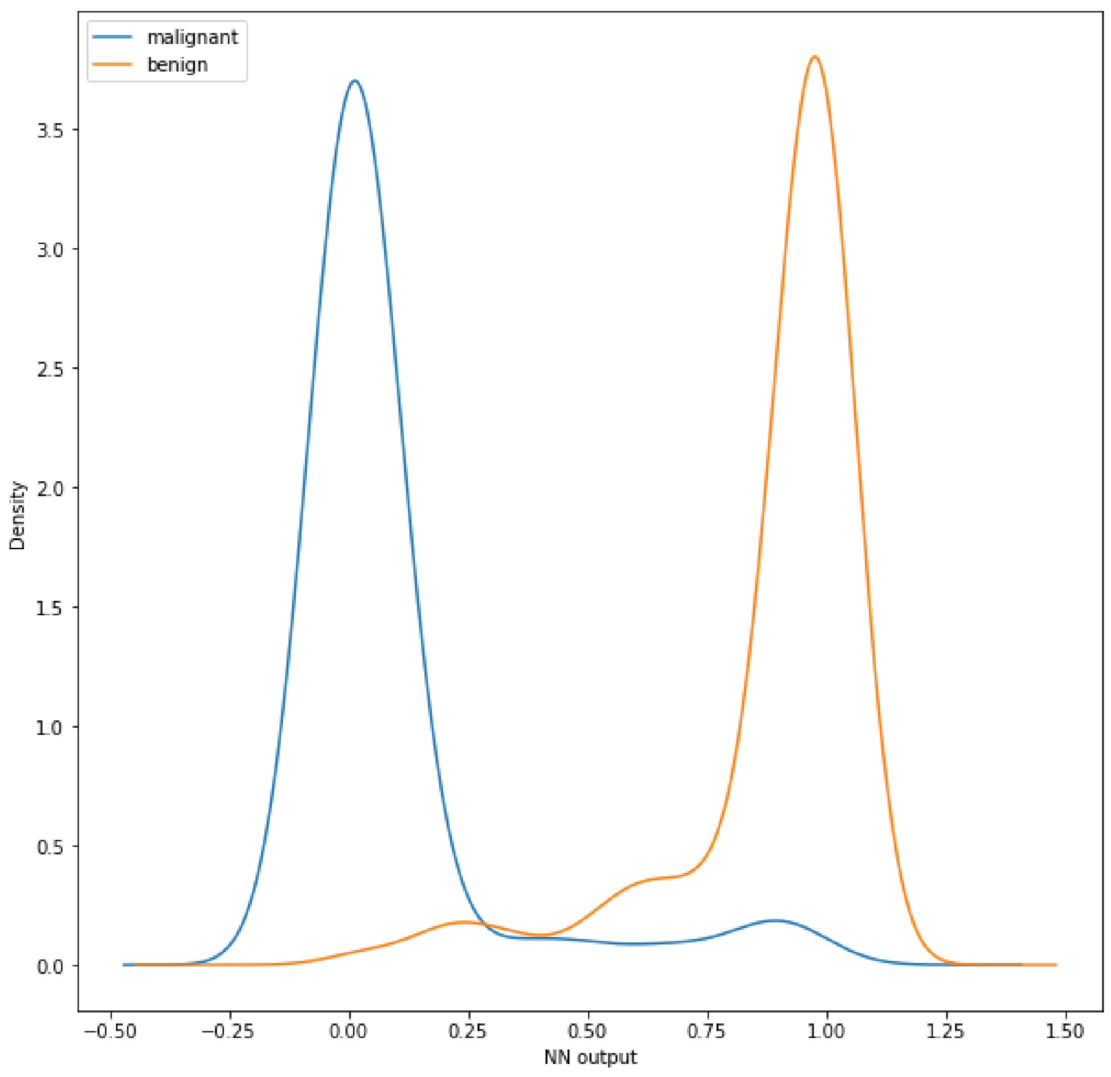
```
Out[30]: (array([ 55.,   5.,   2.,   3.,   2.,   3.,   5.,   4.,   8.,  84.]),
          array([  2.31159447e-09,   9.99981186e-02,   1.99996235e-01,
                   2.99994351e-01,   3.99992467e-01,   4.99990584e-01,
                   5.99988700e-01,   6.99986816e-01,   7.99984932e-01,
                   8.99983049e-01,   9.99981165e-01]),
          <a list of 10 Patch objects>)
```



```
In [31]: nn_out = pd.DataFrame()
         nn_out = nn_out.assign(prediction = y_pred.ravel())
         nn_out = nn_out.assign(target = y_test)
         nn_out.groupby("target")["prediction"].hist(normed=1,figsize=(10, 10))
         plt.legend(['malignant', 'benign'], loc='upper left')
         plt.xlabel('NN output')
         plt.ylabel('Number of entries')

Out[31]: Text(0,0.5,u'Number of entries')
```



```
In [32]: nn_out = pd.DataFrame()
         nn_out = nn_out.assign(prediction = y_pred.ravel())
         nn_out = nn_out.assign(target = y_test)
         nn_out.groupby("target")["prediction"].plot(kind='kde', figsize=(10, 10))
         plt.legend(['malignant', 'benign'], loc='upper left')
         plt.xlabel('NN output')
         plt.ylabel('Density')

Out[32]: Text(0,0.5,u'Density')
```



How do we decide now to which class the test example needs to assigned based on our prediction? Intuitively, we could simply convert our predictions into classes by using a threshold of 0.5:

```
In [33]: y_cls=np.where(y_pred > 0.5, 1, 0)
         print y_cls

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

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

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

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

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

```
In [34]: y_cls = model.predict_classes(X_test, batch_size=1)
         print y_cls
```

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

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

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

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

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

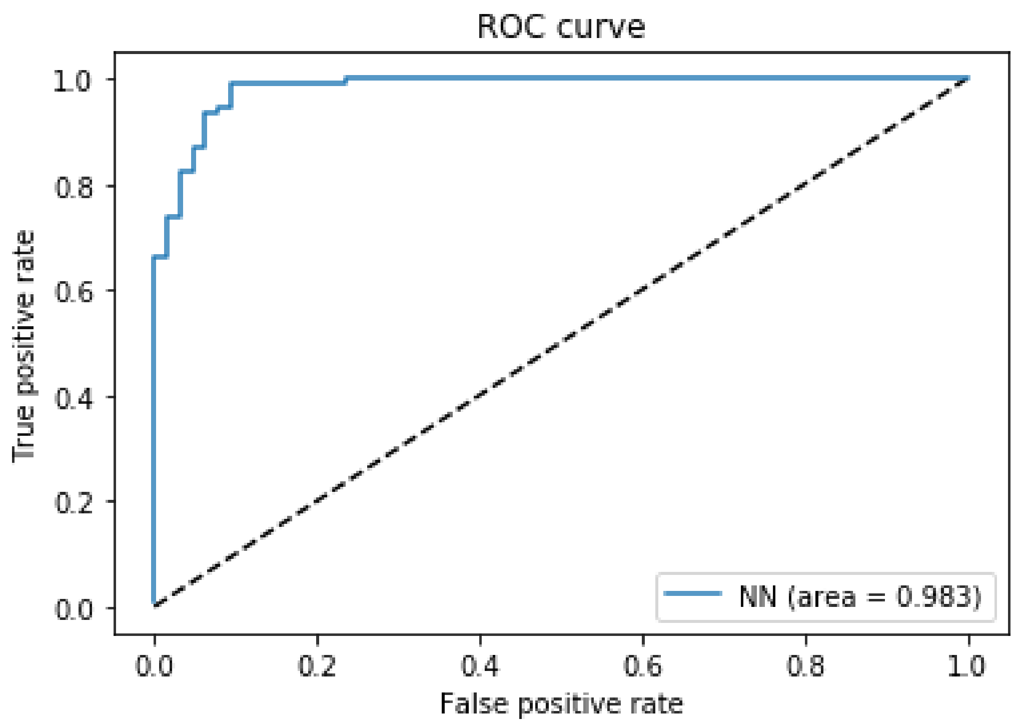
7.1.4 Task 4: Use the scikit learn metrics to evaluate the model

```
In [35]: from sklearn.metrics import accuracy_score, precision_score, recall_score, classificati
         print('Accuracy: %.2f' % accuracy_score(y_test, y_cls))
         print("Precision: %.2f" % precision_score(y_test, y_cls, average='weighted'))
         print("Recall: %.2f" % recall_score(y_test, y_cls, average='weighted'))
         print 'Classification Report:\n', classification_report(y_test, y_cls)
```

```
Accuracy: 0.94
Precision: 0.94
Recall: 0.94
Classification Report:
             precision    recall  f1-score   support

          0       0.90      0.94      0.92        64
          1       0.96      0.93      0.95       107

avg / total       0.94      0.94      0.94       171
```

Now, let's use scikit learn also to plot the ROC curve and calculate the AUC:

```
In [36]: from sklearn.metrics import roc_curve, auc
         fpr, tpr, thresholds = roc_curve(y_test, y_pred.ravel())
         auc = auc(fpr, tpr)
         plt.figure(1)
         plt.plot([0, 1], [0, 1], 'k--')
         plt.plot(fpr, tpr, label='NN (area = {:.3f})'.format(auc))
         plt.xlabel('False positive rate')
         plt.ylabel('True positive rate')
         plt.title('ROC curve')
         plt.legend(loc='best')
         plt.show()
```



7.2 Task 5 (Bonus): Change the neural network model and study the impact on the performance

- Make the neural network wider
- Make the neural network deeper
- Change the activation function of the hidden nodes
- Change the activation function of the output node
- Change the loss function, which ones are allowed?
- Which neural network gives the best performance?