*) MP neuron notebook.

1. loading dataset from sklearn. → import sklearn.datasets.

↓

breast_cancer = sklearn.datasets.

load_breast_cancer().

↓

utility object ← data, labels, feature vector

↓

X = breast_cancer.data

y = breast_cancer.target

↓

numpy array

↓

can be converted to dataframe for better analysis

↓

breast_cancer.feature_names

↓

breast_cancer.target_names.

↓

pd.DataFrame(X, columns = breast_cancer.feature_names)

↓

df.describe() → gives basic statistical information regarding the dataset per feature

↓

df['class'].value_counts()

↓

gives a freq of the different values in the column of the dataframe.

df.groupby('class').mean()

2. Splitting data using train test split from sklearn

↓

from sklearn.model_selection import train_test_split

↓

Xtrain, Xtest, y_train, y_test = train_test_split(X, Y)

↓

basic setting

↓

default test size → 0.2

test_size ← can be passed as argument

2. spar
3. evaluate $\hat{A}^{\pi}(s,a)$ single sam
to policy gradien $\hat{A}^{\pi}(s,a)$

*) maintaining class ratios of in the main dataset to the ones in train & test set for proper representation of data.

↓

(stratify = y) → as argument to the train-test-split function

*) to maintain the same partition over the dataset, random state can be used.

↓

x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.1, stratify = y, random_state = 1)

3. Binarisation of i/p. (i/p to MP neuron are binary, but feature values for breast-cancer are real)

↓

i) plotting the dataset to better analyse.

↓

import matplotlib and seaborn.

↓

plt.plot(x_train, '*')

plt.show()

feature represented by a diff color, with samples existing over the x-axis

↓ transpose plot

plt.plot(x_train.T, '*') ← transpose

plt.show()

plt.xticks(rotation = 'vertical')

↓

better visualisation of the labels on x-axis

feature on x axis and datapoints on y axis

gives the spread of the data.

ii) to binarise: threshold value, above which 1, below which 0. The value can be μ, or median.

and we ... parallel ...

•) binarise a column using map function over the column

↓

```
col = x_train['mean_area'].map(lambda
      x : 0 if x<1000 else 1)
```

•) This can be done using the cut fn. in pandas module. → splits value in certain no of bins

```
x_btrain = x_train.apply(pd.cut,
                         bins = 2, labels = [0,1])
```

with given labels

remain a dataframe

↓

convert to np. → df.values

4. MP neuron → only 1 parameter b.

↓

sample inference for some value of b → the o/p of the model

↓

picking random index to work/check on the dataset,

↓

(range as input) ← randint from random

↓

```
i = randint(0, x_btrain.shape[0])
```

```
aggregation = np.sum(x_btrain[i,:])
```

row, column

↓

comparing with b. and accordingly o/p.

2. spou...
3. evaluate $\hat{A}^{\pi}(s,a)$
4. evaluate policy gradie... single sam... $\hat{A}^{\pi}(s,a)$

to iterate over 2 diff. vectors at the same time, we use the zip function

↓

```
for x, y in zip(xb_train,
                y_train):
```

•) running over the training set

↓

iterate over the training set

↓

store predictions as well

↓

calculate accuracy

↓

baseline here would be that of selecting one class & predicting that continously, i.e. 62% for malignant.

↓

searching for b (optimising loss) can be done over all values of b (by applying a basic search)

↓

The max accuracy is of around 37ish (if b≠0).

↓

this is because the binarisation of data is not done properly. The feature values of malignant (label=0) are more than that of benign (label=1), whereas the binarisation puts 1 for values more than mean and 0 for less than mean, as 1 is ~~as~~ a contributer to benign, ∴ it should represent lesser value than mean.

↓

On rerun, accuracy hits 82.45%

rightway ← of binarisation plays a imp. role in accurate inference

•) accuracy calculation using sklearn.metrics.

↓

```
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_pred, labels)
```

5) templatise MCP model as class → way to proceed ahead for all concepts moving forward

```
class MPNeuron:

    def __init__(self):
        self.b = None

    def model(self, x):
        return sum(x) >= self.b

    def predict(self, X):
        Y = []
        for x in X:
            result = self.model(x)
            Y.append(result)
        return np.array(Y)

    def fit(self, X, Y):
        accuracy = {}
        for b in range(X.shape[1]+1):
            self.b = b
            Y_pred = self.predict(X)
            accuracy[b] = accuracy_score(Y_pred, Y)
        best_b = max(accuracy, key =
                     accuracy.get)
        self.b = best_b
        print('optimal b')
        print('max accuracy')
```

•) make an instance of the class
•) run on binarised data.

3. Evalu...

1. ...valuate policy gradie... $\hat{A}^{\pi}(s,a)$

*) Perceptron class

↓

\mbox

$y = 1$, if $\sum_i w_i x_i >= b$ → latex representation of the model for perceptron.

$y = 0$, \mbox{otherwise}

↓

class Perceptron

def __init__(self):

self.w = None → array

self.b = None → scalar

↙

n+1 parameters

↓

real valued i/ps

def model(self, x):

return np.dot(self.w, x) >= self.b

def predict(self, x)

Y_pred = []

for x in X:

result = self.model(x)

Y_pred.append(result)

return np.array(Y_pred)

→ create a plot for accuracies as well, along with weights if needed.

epochs = 20 → default value.

def fit(self, X, Y):

self.b = 0

self.w = np.ones(X.shape[1]) → loop for epochs.

for x, y in zip(X, Y):

result = self.model(x)

if result == 0 and y == 1:

self.w = self.w + x

self.b = self.b + 1

elif result == 1 and y == 0:

self.w = self.w - x

self.b = self.b - 1

•) Try doing a plot learned weights

•) hyper parameter → a parameter on which the performance depends indirectly (like epochs, learning rate, momentum, etc.) Tuning them is important to extract best performance. → grid search

•) checkpointing → store the weights & bias corresponding to the increase in accuracy. So that sudden change in accuracy due to some change in weights is not lost.

↓

and then put those checkpt. values to the weights and bias of the model.

another hyperparameter

↑

(•) learning rate :→ slowing down the changes in the parameters of the model to reduce unstability of the training phase.

(•) plt.ylim([0,1]) → specifying ranges for axis

(•) animation of the weight plot

↓

embedding matplotlib animations in Jupyter notebooks.

↓

refer article.

3. evaluate $\hat{A}^{\pi}(s,a)$

4. evaluate policy gradie

r sing

$\hat{A}^{\pi}(s,a)$

*) one-hot encoding : done for categorical data, for example sim type can be of 5 different types. assigning them numerical magnitude might introduce magnitude based bias, which is actually not present in the nature of the feature. Therefore the feature representation is converted to a set of vector representation where each possible category gets its own vector and wherever the value / persists, there the sample has value corresponding to that value representation.

↓

for example simtype which can take 5 different values or categories gets converted to 5 vectors, one for each category and 0 or 1 indicating if that value / category is true for the sample

↓

More on lines with an indicatorfunction.