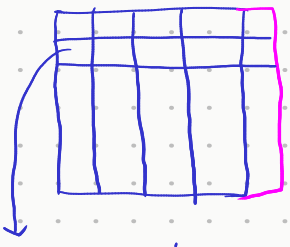
Plotting for Exploratory Data Analysis:-

→ Analyzing data using plotting tools, statistics, linear algebra etc.,

→ Column to be predicted = class label / dependant variable

Data already there = data points / vectors → n dimensional numerical array.

→ columns are called features / input variable / independent variable



1 row = 1 array/vector.

→ 1D vectors = Scalar.

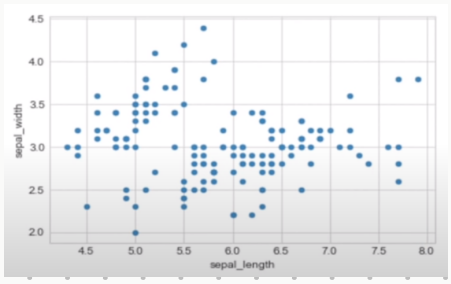
[a], [1], [3.1]

'a' 1 3.1

→ Balanced Dataset → Each class has equal number of datapoints.

→ When reading a plot, always read axis labels & values. It doesn't always start at 0.

→ iris.plot(kind=scatter, x="sepal length", y="petal_width");



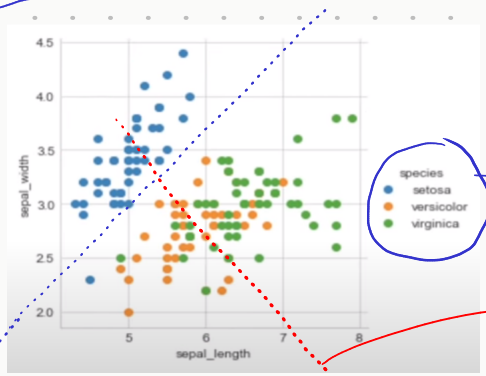
(sns.scatterplot() only plots one plot
sns.FacetGrid() can plot multiple plots.)

→ import seaborn as sns

sns.set_style("whitegrid") → white grid structure

sns.FacetGrid(iris, hue="species", size=4).map(plt.scatter, "sepal_length", "sepal_width").add_legend();

- iris: dataset
- hue="species": by which column should the dataset be colored
- size=4: size
- map(plt.scatter, ...): type of plot
- "sepal_length": x-Axis
- "sepal_width": y-Axis



→ legend.

→ can't separate as too many outliers.

→ line could separate setosa from versicolor & virginica (this is called linear separable)

Observations:- ① S_length & S_width can separate setosa flowers from others.

② Separating versicolor & virginica is harder.

QUICK SEABORN INTRODUCTION:-

→ Simpler way to plot attractive plots.

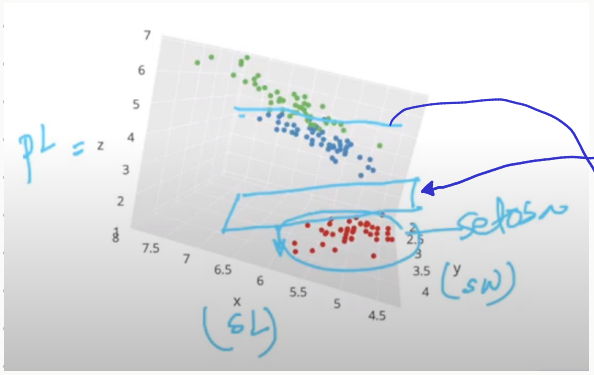
→ high level interface to matplotlib.

→ some features include:

- default aesthetic themes
- custom color palettes
- * Visualizing info from matrices & dataframes.
- * attractive statistical plots.
- * easily displaying distributions

→ Seaborn is a complement, not a substitute of matplotlib

3D Scatter Plot :-



when 3D plot is plotted, we use a plane to separate instead of a line.

Plotting petal length helps notice that plotting a line can seperate them which we were not able to do in a 2D plot.

Drawbacks :- → while being interactive is a pro, it can't be viewed on 2D surfaces like paper or in an ipynb.

→ Since we can't visualize 4D datasets, we use pair plots.

↓

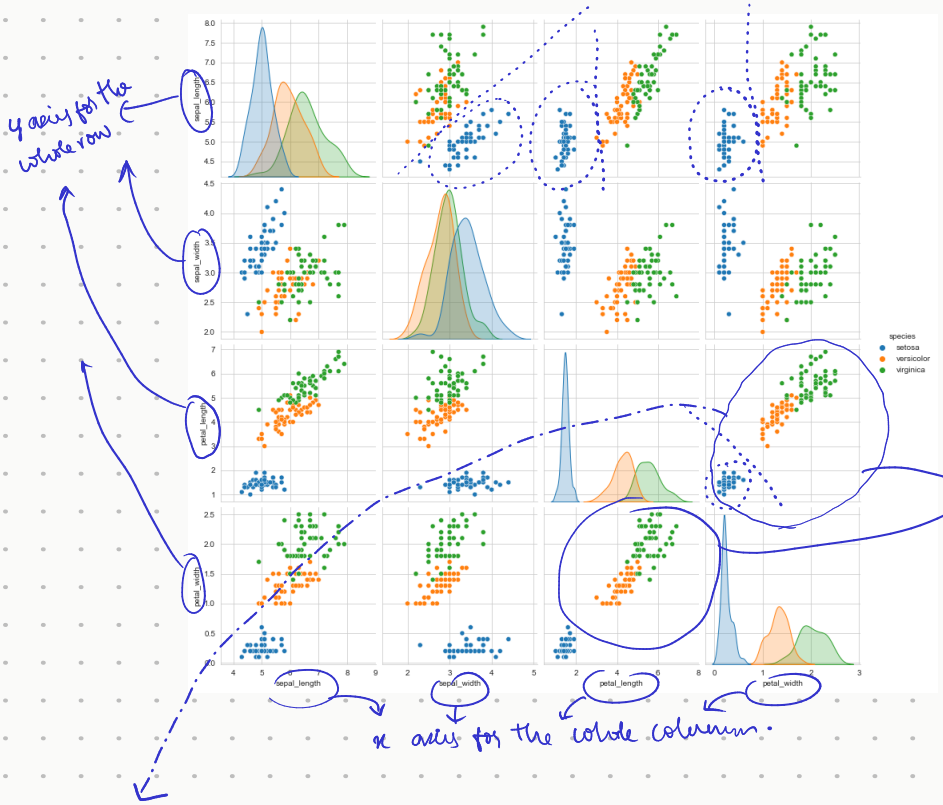
there are 4 variable $\Rightarrow {}^4C_2$ pairs are possible.

↓

6 → {(SL, SW), (SW, PL), (SL, PL), (PW, PL), (SL, PW), (SW, PW)}

Since we can't visualize 4D, we try & visualize these 6.

< sns.pairplot(iris, hue="species", size=3) >



y axis for the whole row

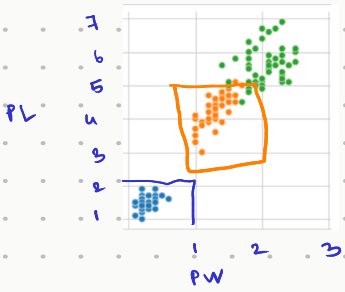
(The diagonal plots are PDFs of each feature)

mirrors. As both are b/w PL, PW.

Similarly there are 3 pairs in total.

So we focus only on 6 plots.

x axis for the whole column.



$\Rightarrow$ if PL ≤ 2 & PW ≤ 1 then flower type = setosa

if (PW ≤ 2 & PW ≥ 1) & (PL ≤ 5 & PL ≥ 2.5) then versicolor

→ There will be some error but that's OK.

Drawback of pairplots :-

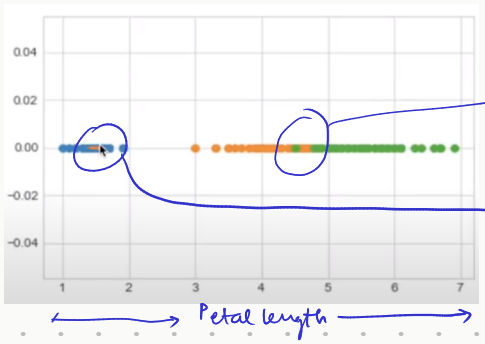
→ When dimensionality is 4, we get ${}^4C_2$ plots

If the dimensionality is n, we will get ${}^nC_2$ plots which is not that useful.

So we use techniques like PCA, t-SNE that will help with this.

In real world, we limit pairplots to only important features & not apply it on all features.

1D Scatter Plots :-



→ harder to read.

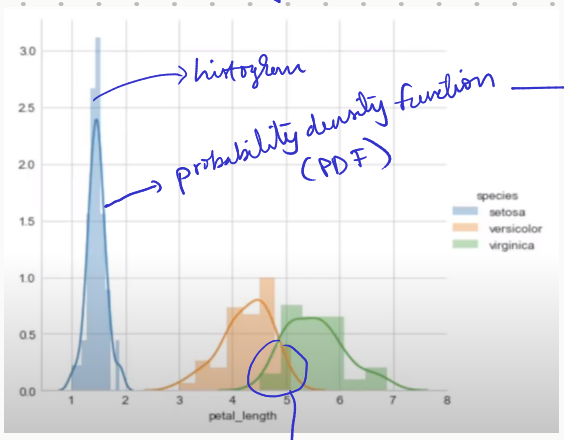
→ green overlaps orange.

→ Number of points here is not clear.

To overcom this, instead of using a scatterplot, we can use a histogram

```
sns.FacetGrid(iris, hue="species", size=5).
    map(sns.distplot, "petal_length").add_legend()
```

distplot aka histogram aka density plot



→ histogram

→ probability density function (PDF) → Smoothed histogram. Smoothed using kernel Density Estimation. (KDE)

→ y-axis: count.

if petal length < 2 then setosa.

if petal length > 2 & petal length < 4.7 then versicolor ≈

if petal length > 4.7 then virginica ≈

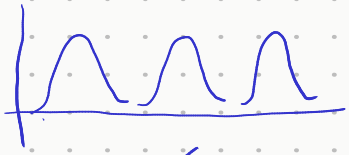
But there will be some

overlapping region

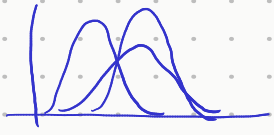
Univariate Analysis :-

→ Analysis of one variable (uni + variate)

→ The farther the distributions are, the better.

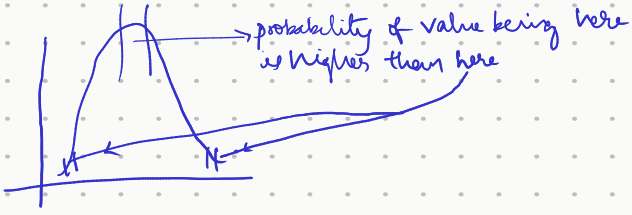


✓



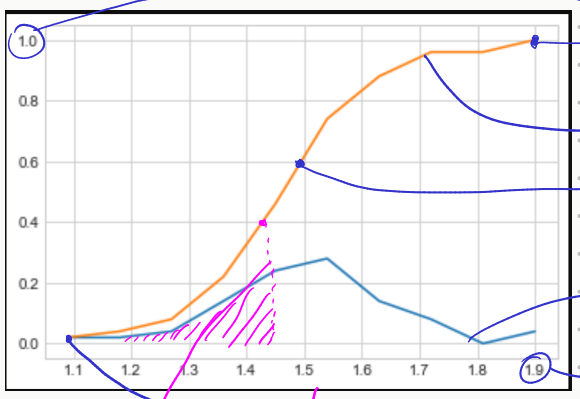
✗

PDF → probability of variable to be in the range of values. Ex:-



Cumulative Data Function :- (CDF)

→ What % of values are ≤ the value at that point.



100 % of the values are ≤ 1.9

CDF

60% of the values are ≤ 1.5

PDF

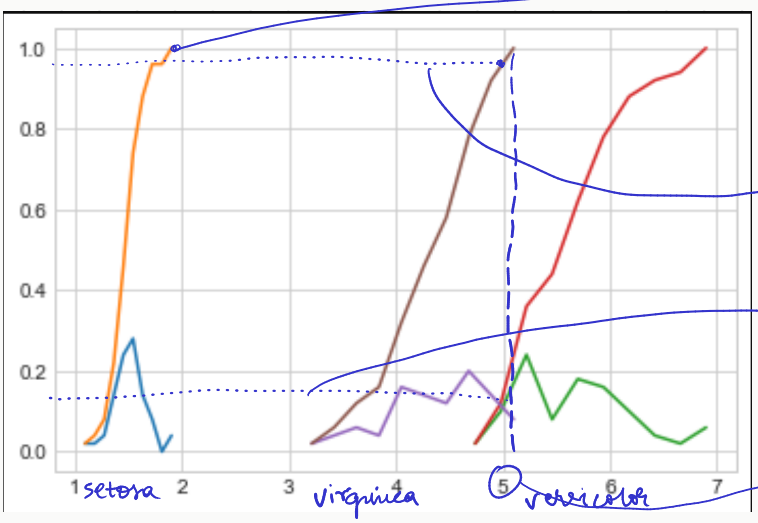
0 % of the values are ≤ 1.1

Differentiation of CDF = PDF

integration of PDF = CDF

The CDF value at this point = sum(values) till that point = cumulative sum.

→ Using CDF, we can make assumptions such as



100 % of setosa flowers petal length ≤ 2.

95% of virginica have petal length ≤ 5

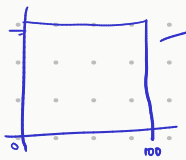
⇒ error % = 5 %

15% of versicolor have petal length ≤ 5

⇒ 85% are ≥ 5.

⇒ error % = 15 %

bin = number of bars in a region in a histogram.



→ 1 bin

→ np.histogram returns counts & bin_edges.

How to compute CDF ? :-

```
counts, bin_edges = np.histogram(iris_setosa['petal_length'], bins=10,
                                 density = True)

pdf = counts/(sum(counts))
print(pdf);
print(bin_edges)

#compute CDF
cdf = np.cumsum(pdf)
plt.plot(bin_edges[1:],pdf)
plt.plot(bin_edges[1:], cdf)
```

→ number of bins

→ if false, it will just return the number of samples in each bin (aka normal hist). But if its true returns PDF

→ cumulative sum.

→ Another way to plot CDF :-

sns.kdeplot(data, cumulative = True)

→ But can't adjust bins in this.

Mean, Variance & Standard deviation :-

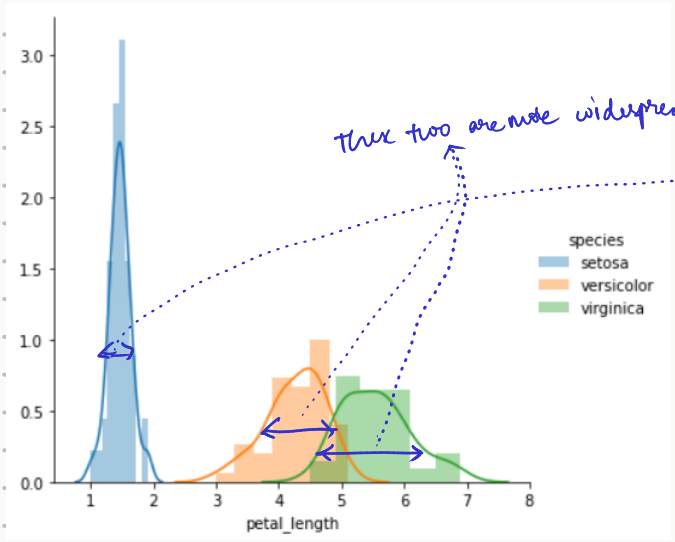
Mean = $\frac{1}{n} \wedge \sum_{i=1}^{n} (x_i)$

Gives an average value. But outliers affect this value. For ex; mean of setosa = 1.464.

Adding one extra value 50 will change the mean to 2.41.

How does mean help ?

Spread = how spreaded is the graph.



These two are more widespread than setosa.

We calculate standard deviation which gives us a range. Useful when we can't plot the data.

Std_dev = $\sqrt{\text{variance}}$ → unit

Variance = $\frac{1}{n} \sum_{i=1}^{n} (\mu_i - x_i)^2$ → unit²

Square because diff could be -ve.

μ_i → mean of dataset

Now we can use this to estimate type of iris

np.mean() → mean

np.std() → Standard deviations.

μ of setosa = 1.4

std_dev of setosa = 0.171

Now if flower is in the range (1.4 − 0.171) to (1.4 + 0.171)

it could be a setosa.

Median :-

→ To overcome mean outlier problem, we use median.

→ Median = sort all elements & pick middle element :

if number of elements = even, mean of middle two elements = median.

→ Median is not corrupted as long as number of corrupted elements ≤ 50% of total elements.

np.median() to get median.