Linear Discriminant Analysis (LDA)

● Advantages: It is better interpretable than PCA. It is a comparatively simple algorithm that is also computationally efficient. It can handle multicollinearity. It can even work with big datasets (such that have more features than the training samples)

● Basics: LDA is used primarily for dimensionality reduction like PCA but also can be used for classification. Unlike PCA, LDA increases interpretability and minimizes loss of information. The goal of LDA is to project a dataset onto a lower-dimensional space with good class-separability in order to avoid overfitting ("curse of dimensionality") and also to reduce computational costs. It works by calculating the directions ("linear discriminants") that will represent the axes that maximize the separation between multiple classes. Like PCA it lowers the dimensions of the features and separates the features into different classes.

● Disadvantages: It assumes that the data has a Gaussian distribution It assumes that the covariance matrices of the different classes are equal It assumes that the data is linearly separable It may not perform well in high-dimensional feature spaces. It doesn't work well with outliers

```
# libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import GridSearchCV


import warnings
warnings.filterwarnings("ignore")
```

```
# read dataset from URL
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
cls = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'Class']
dataset = pd.read_csv(url, names=cls)
dataset.head()
```

	sepal-length	sepal-width	petal-length	petal-width	Class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Exploratory Data Analysis (EDA)

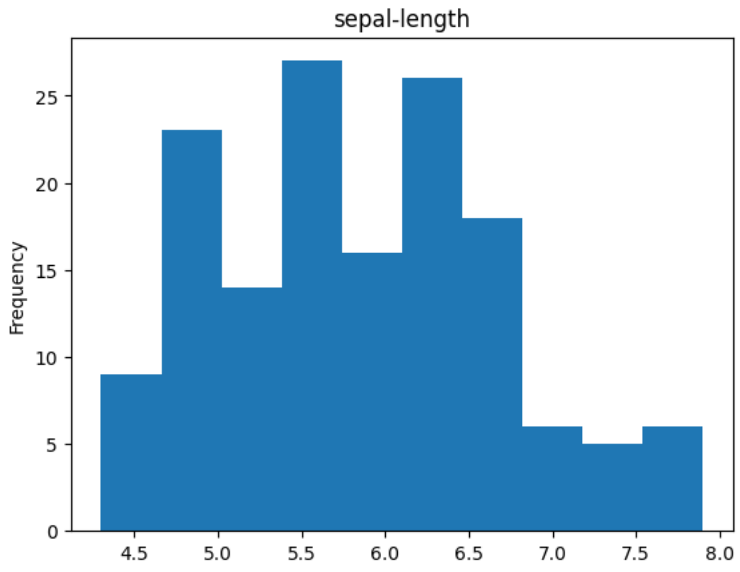
```
dataset.shape
```

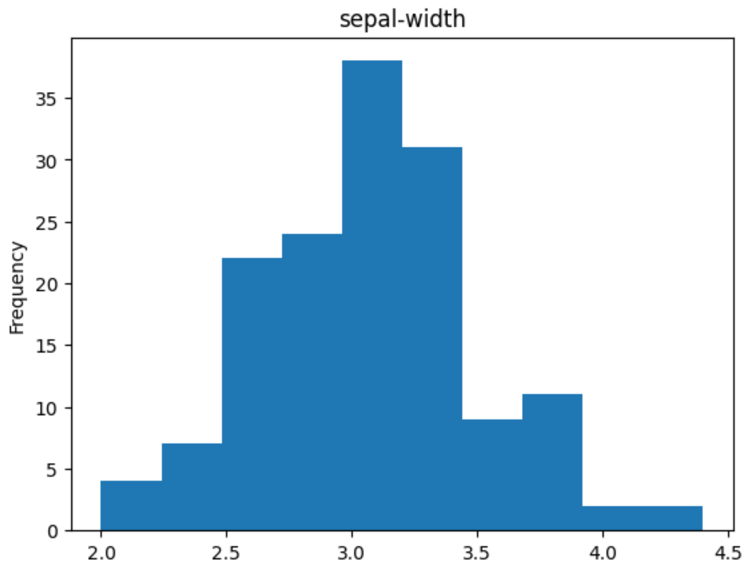
```
(150, 5)
```

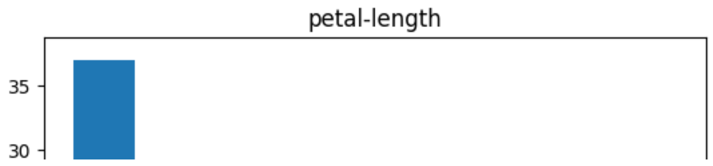
```
dataset.describe()
```

	sepal-length	sepal-width	petal-length	petal-width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000
max	7.900000	4.400000	6.900000	2.500000

```
for col in dataset.columns[:-1]:
    plt.title(col)
    dataset[col].plot.hist() #plotting the histogram with Pandas
    plt.show();
```


sepal-length
Frequency
0
5
10
15
20
25
4.5
5.0
5.5
6.0
6.5
7.0
7.5
8.0


sepal-width
Frequency
0
5
10
15
20
25
30
35
2.0
2.5
3.0
3.5
4.0
4.5


petal-length
30
35

We can see that not all the data has a normal distribution

```
# divide the dataset into class and target variable
X = dataset.iloc[:, 0:4].values
y = dataset.iloc[:, 4].values

# Preprocess the dataset and divide into train and test
sc = StandardScaler()
X = sc.fit_transform(X)
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = 0.2,
                                                    random_state = 1)
```

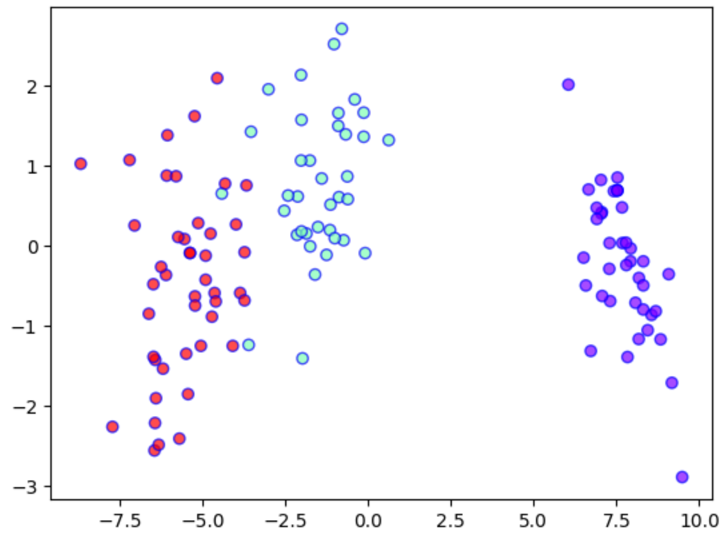
```
# Define the parameter values that should be searched
solver_options = ['svd', 'lsqr', 'eigen']
shrinkage_options = [None, 'auto', 0.1, 0.5, 0.9]

# parameter grid
param_grid = dict(solver=solver_options, shrinkage=shrinkage_options)

# Instantiate the grid
grid = GridSearchCV(LinearDiscriminantAnalysis(), param_grid, cv=10, scoring='accuracy')

# Fit the grid with data
grid.fit(X, y)

# View the complete results
#print(grid.cv_results_)

# Examine the best model
#print(grid.best_score_)
print(grid.best_params_)
```

```
    {'shrinkage': None, 'solver': 'svd'}
```

As we can see after tuning the hyperparameters we find wich are the best ones. In this case the best ones are the default ones and this is why they are set to be default, because in most of the cases these is the best choice.

```
# apply Linear Discriminant Analysis
lda = LinearDiscriminantAnalysis(n_components=2,
                                 solver = 'svd',
                                 shrinkage = None
                                 )

X_train = lda.fit_transform(X_train, y_train)
X_test = lda.transform(X_test)

# plot the scatterplot
plt.scatter(
    X_train[:,0],
    X_train[:,1],
    c=y_train,
    cmap='rainbow',
    alpha=0.7,
    edgecolors='b'
)

# classify using random forest classifier
classifier = RandomForestClassifier(max_depth=2, random_state=1)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)

# print the accuracy and confusion matrix
print('Accuracy : ' + str(accuracy_score(y_test, y_pred)))
conf_m = confusion_matrix(y_test, y_pred)
print(conf_m)
```

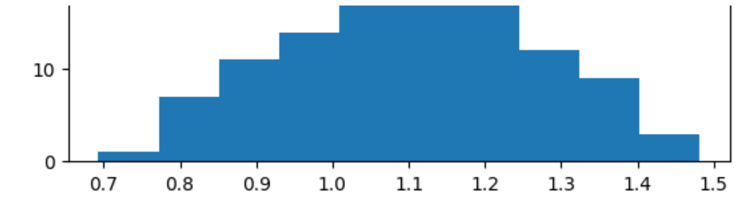
```
Accuracy : 0.8666666666666667
[[11  0  0]
 [ 4  9  0]
 [ 0  0  6]]
```

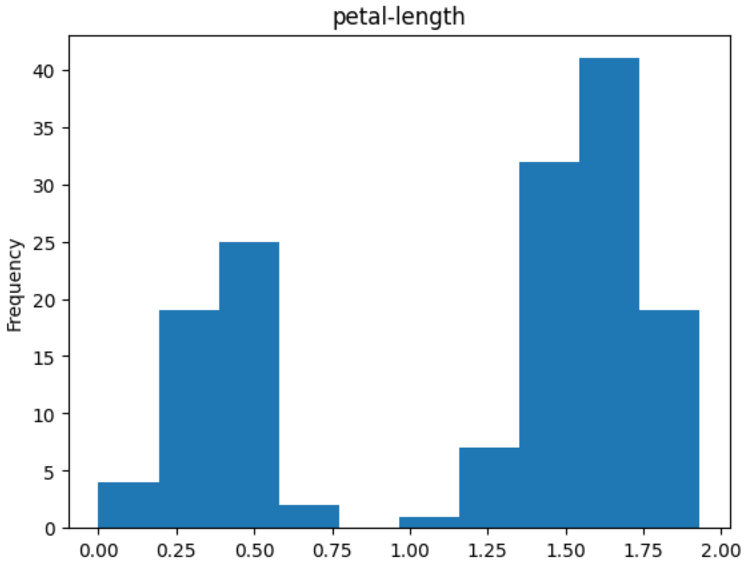


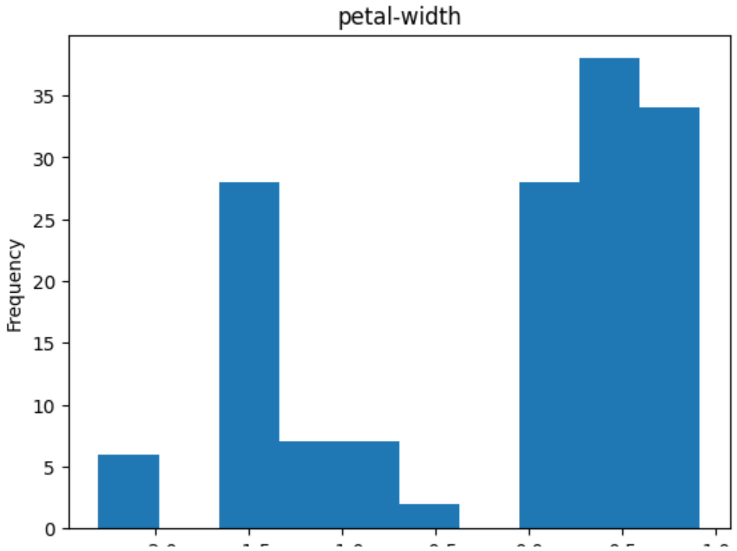
Now let's try a lognorm transformation to see what is going to happen to the results?

```
# divide the dataset into class and target variable and apply LogNorm transformation
X = np.log(dataset.iloc[:, 0:4].values)
y = dataset.iloc[:, 4].values

# Preprocess the dataset and divide into train and test
sc = StandardScaler()
X = sc.fit_transform(X)
le = LabelEncoder()
y = le.fit_transform(y)
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size = 0.2,
                                                    random_state = 1)
```

```
# 'sepal-length', 'sepal-width', 'petal-length', 'petal-width'

loged_dataset = dataset
loged_dataset['sepal-length'] = np.log(dataset['sepal-length'])
loged_dataset['sepal-width'] = np.log(dataset['sepal-width'])
loged_dataset['petal-length'] = np.log(dataset['petal-length'])
loged_dataset['petal-width'] = np.log(dataset['petal-width'])

for col in loged_dataset.columns[:-1]:
    plt.title(col)
    loged_dataset[col].plot.hist() #plotting the histogram with Pandas
    plt.show();
```


10
0
0.7
0.8
0.9
1.0
1.1
1.2
1.3
1.4
1.5


petal-length
Frequency
0
5
10
15
20
25
30
35
40
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


petal-width
Frequency
0
5
10
15
20
25
30
35

Well, still does NOT look like normal distribution. This is maybe because the dataset is not big enough for the Central Limit Theorem to be in full power. Over all a log norm transformation makes the data to be closer to a Normal Distribution, which is important for LDA.

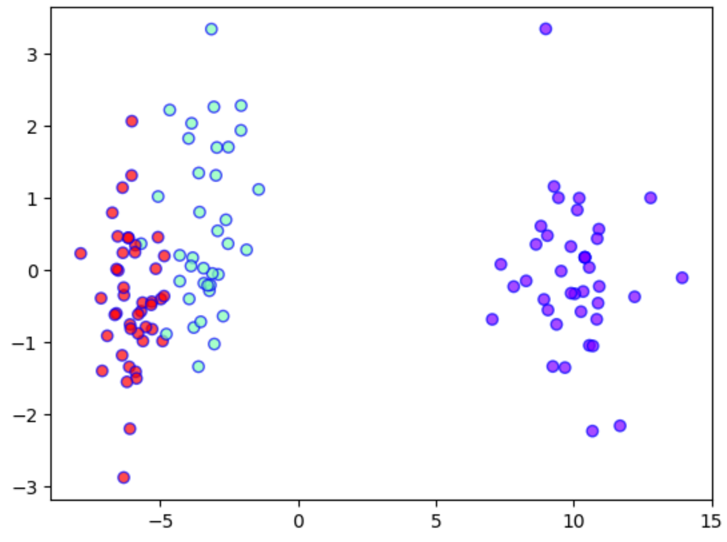
```
# let's see is there any difference
# Define the parameter values that should be searched
solver_options = ['svd', 'lsqr', 'eigen']
shrinkage_options = [None, 'auto', 0.1, 0.5, 0.9]

# parameter grid
param_grid = dict(solver=solver_options, shrinkage=shrinkage_options)

# Instantiate the grid
grid = GridSearchCV(LinearDiscriminantAnalysis(), param_grid, cv=10, scoring='accuracy')

# Fit the grid with data
grid.fit(X, y)

# View the complete results
#print(grid.cv_results_)

# Examine the best model
#print(grid.best_score_)
print(grid.best_params_)
```

```
    {'shrinkage': None, 'solver': 'svd'}
```

Still the best hyperparameters' values are the same, so no difference here

```
# apply Linear Discriminant Analysis
lda = LinearDiscriminantAnalysis(n_components=2,
                                 solver = 'svd',
                                 shrinkage = None
                                 )

X_train = lda.fit_transform(X_train, y_train)
X_test = lda.transform(X_test)

# plot the scatterplot
plt.scatter(
    X_train[:,0],
    X_train[:,1],
    c=y_train,
    cmap='rainbow',
    alpha=0.7,
    edgecolors='b'
)

# classify using random forest classifier
classifier = RandomForestClassifier(max_depth=2, random_state=1)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)

# print the accuracy and confusion matrix
print('Accuracy : ' + str(accuracy_score(y_test, y_pred)))
conf_m = confusion_matrix(y_test, y_pred)
print(conf_m)
```

```
Accuracy : 1.0
[[11  0  0]
 [ 0 13  0]
 [ 0  0  6]]
```



Now the accuracy is perfect. So we proved that LDA works better when the data is normaly distributed

Support Vector Machine (SVM)

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.metrics import classification_report, confusion_matrix
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.model_selection import GridSearchCV
```

Support Vector Machines

Advantages:

- Effective in high-dimensional spaces.
- Memory-efficient as it uses a subset of training points (support vectors) for decision function.
- Versatile as it supports different kernel functions for non-linear decision boundaries.
- Resistant to overfitting, especially in high-dimensional spaces.

Basics:

Support Vector Machine is a supervised machine learning algorithm used for classification and regression tasks. It aims to find a hyperplane in an N-dimensional space (N being the number of features) that distinctly classifies data points into different classes.

Computation/Illustration/Visualization:

The study under SVM model is to decide if the bank note is authentic or not. We import the bank note data from UCI database, explore if it's balanced data, the statistics of the input data including variance, skewness, kurtosis, entropy and class (binary 0 and 1), split 20/80 to test/training data, find the optimal parameters via Grid Search.

```
data_link = "https://archive.ics.uci.edu/ml/machine-learning-databases/00267/data_banknote_authentication.txt"
col_names = ["variance", "skewness", "kurtosis", "entropy", "class"]

bankdata = pd.read_csv(data_link, names=col_names, sep=",", header=None)
bankdata.head()
```

	variance	skewness	kurtosis	entropy	class
0	3.62160	8.6661	-2.8073	-0.44699	0
1	4.54590	8.1674	-2.4586	-1.46210	0
2	3.86600	-2.6383	1.9242	0.10645	0
3	3.45660	9.5228	-4.0112	-3.59440	0
4	0.32924	-4.4552	4.5718	-0.98880	0

```
bankdata['class'].unique()
```

```
array([0, 1])
```

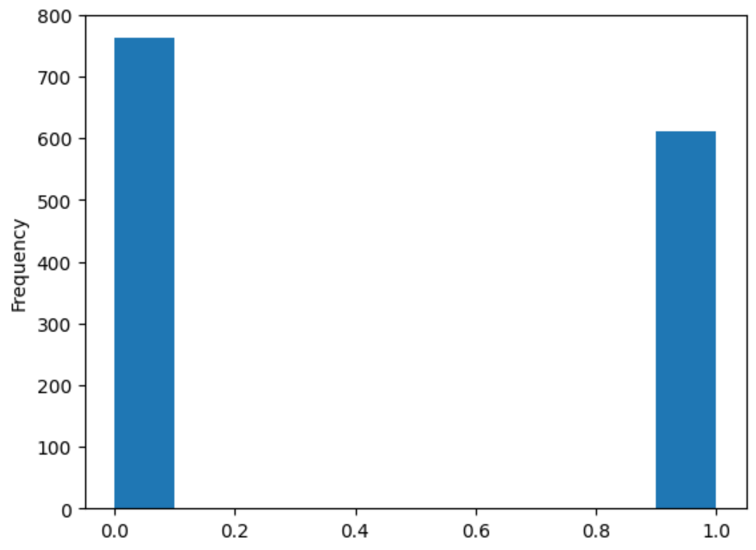
```
bankdata.shape
```

```
(1372, 5)
```

```
bankdata['class'].value_counts(normalize = False)
```

```
0    762
1    610
Name: class, dtype: int64
```

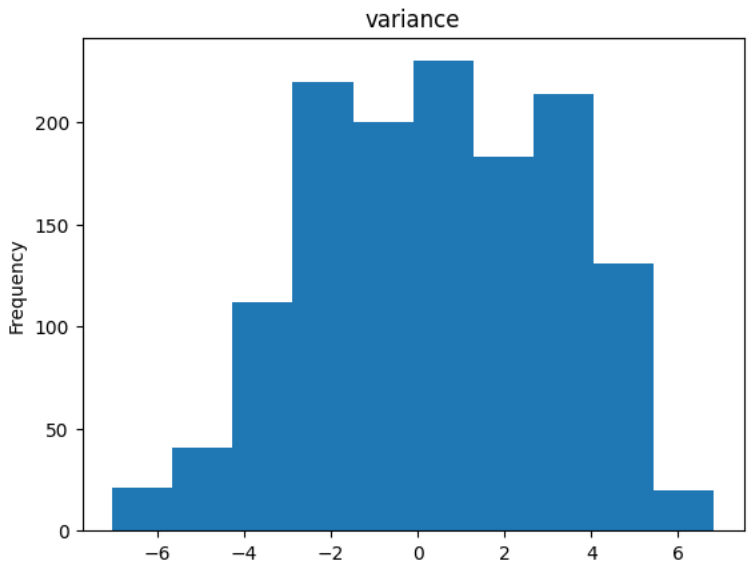
```
bankdata['class'].plot.hist();
```

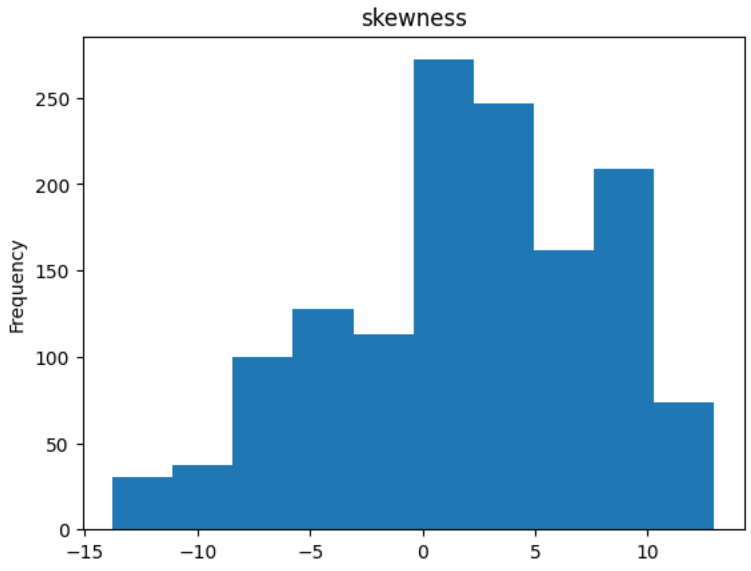


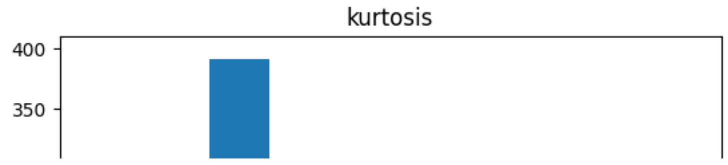
```
bankdata.describe().T
```

	count	mean	std	min	25%	50%	75%	max
variance	1372.0	0.433735	2.842763	-7.0421	-1.773000	0.49618	2.821475	6.8248
skewness	1372.0	1.922353	5.869047	-13.7731	-1.708200	2.31965	6.814625	12.9516
kurtosis	1372.0	1.397627	4.310030	-5.2861	-1.574975	0.61663	3.179250	17.9274
entropy	1372.0	-1.191657	2.101013	-8.5482	-2.413450	-0.58665	0.394810	2.4495
class	1372.0	0.444606	0.497103	0.0000	0.000000	0.00000	1.000000	1.0000

```
for col in bankdata.columns[:-1]:
    plt.title(col)
    bankdata[col].plot.hist() #plotting the histogram with Pandas
    plt.show();
```

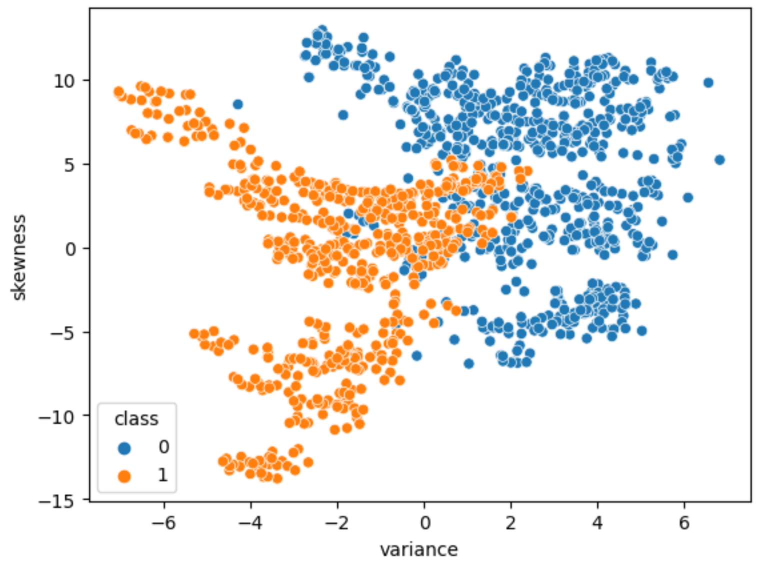

variance
Frequency
0
50
100
150
200
−6
−4
−2
0
2
4
6


skewness
Frequency
0
50
100
150
200
250
−15
−10
−5
0
5
10

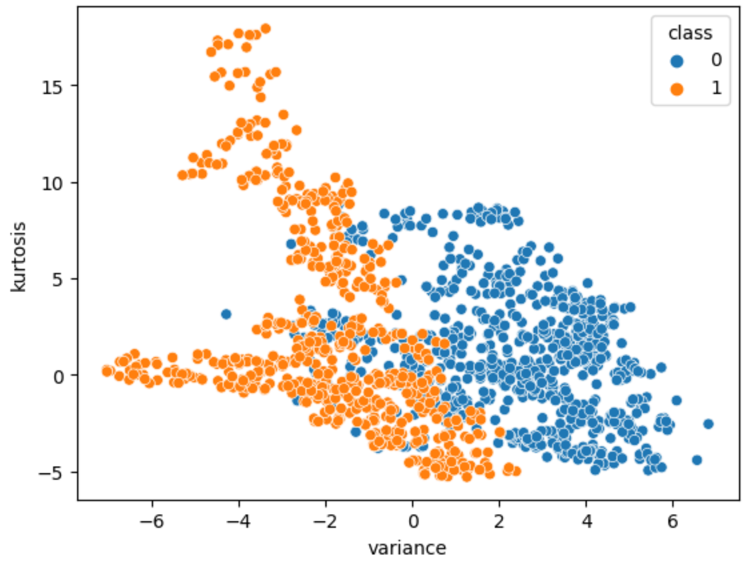

kurtosis
400
350

```
for feature_1 in bankdata.columns[:-1]:
    for feature_2 in bankdata.columns[:-1]:
        if feature_1 != feature_2: # test if the features are different
            print(feature_1, feature_2) # prints features names
            sns.scatterplot(x=feature_1, y=feature_2, data=bankdata, hue='class') # plots each feature points with its color depending
            plt.show();
```

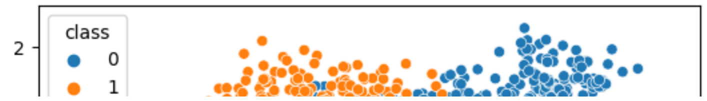
```
variance skewness
```



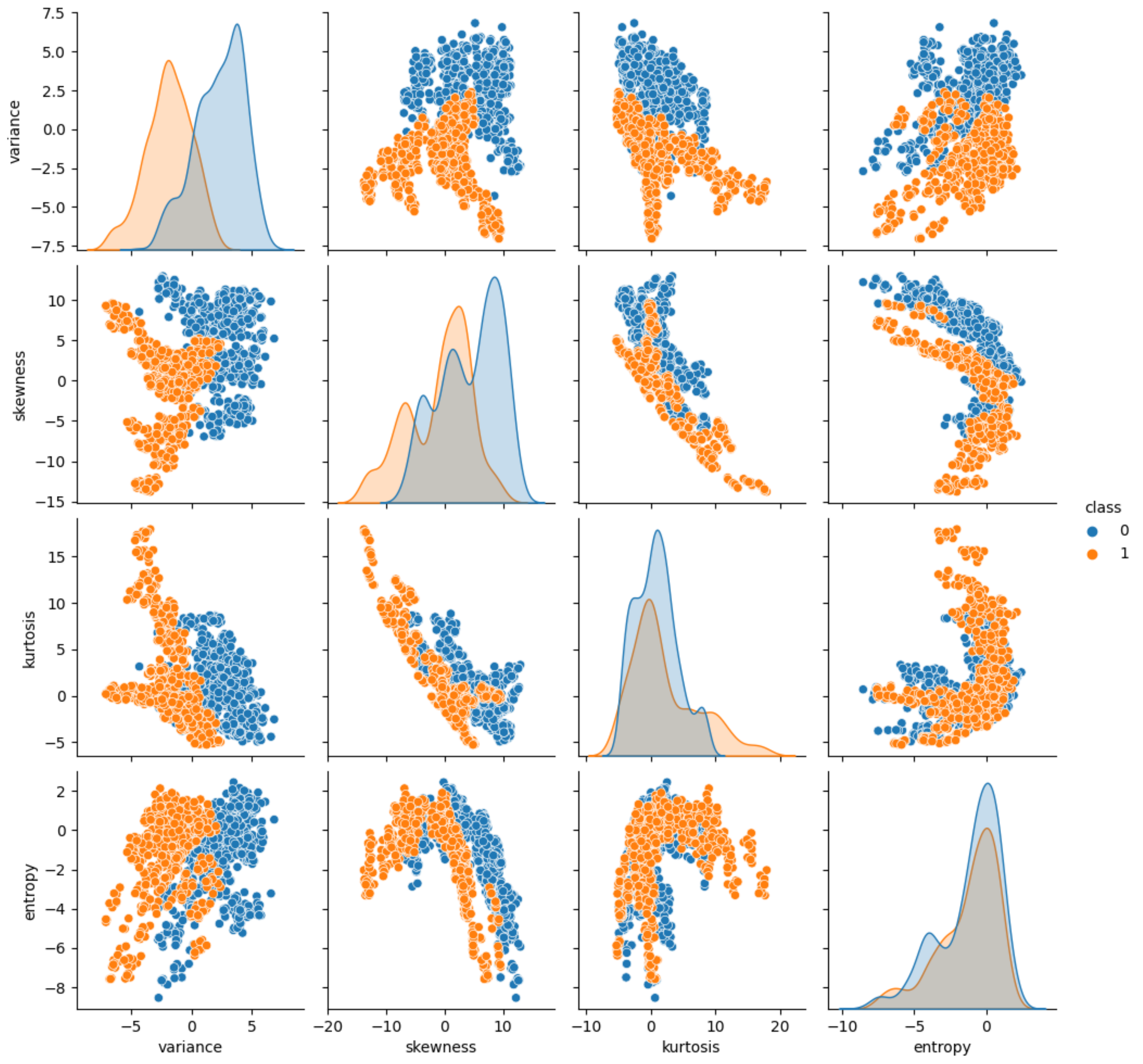
```
variance kurtosis
```



```
variance entropy
```



```
sns.pairplot(bankdata, hue='class');
```



From looking at the plots above, it's likely to have non-linear kernel, instead of linear one. To confirm our hypothesis, we apply grid search and found the following results:

```
The best model was: SVC(C=0.0001, gamma=10, kernel='poly')
The best parameter values were: {'C': 0.0001, 'gamma': 10, 'kernel': 'poly'}
The best f1-score was: 0.9989528795811518
```

To conclude, according to the results from grid search and the confusion matrix of different kernels, the best model doesn't have a linear kernel, but a nonlinear one, polynomial.

```
y = bankdata['class']
X = bankdata.drop('class', axis=1) # axis=1 means dropping from the column axis

SEED = 42

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20, random_state = SEED)
```

```
xtrain_samples = X_train.shape[0]
xtest_samples = X_test.shape[0]

print(f'There are {xtrain_samples} samples for training and {xtest_samples} samples for testing.')
```

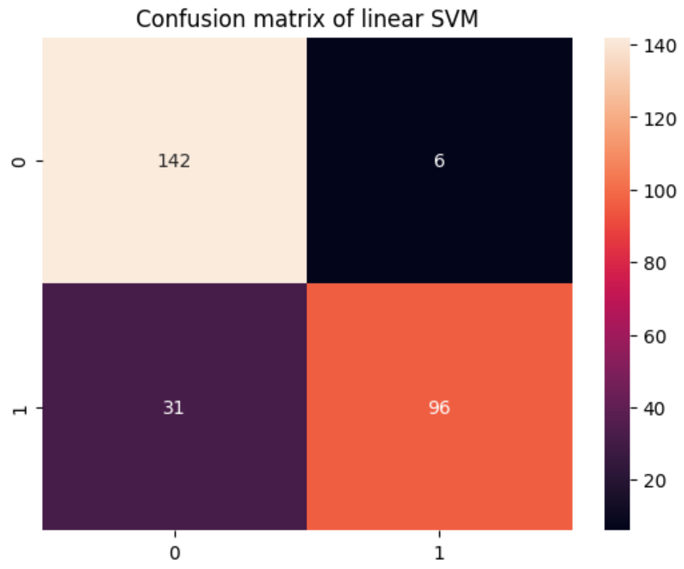
```
     There are 1097 samples for training and 275 samples for testing.
```

```
# calculate gamma
# svc._gamma
number_of_features =  X_train.shape[1]
features_variance = X_train.values.var()
gamma = 1/(number_of_features * features_variance)
print('gamma:', gamma)
```

```
     gamma: 0.013924748072859962
```

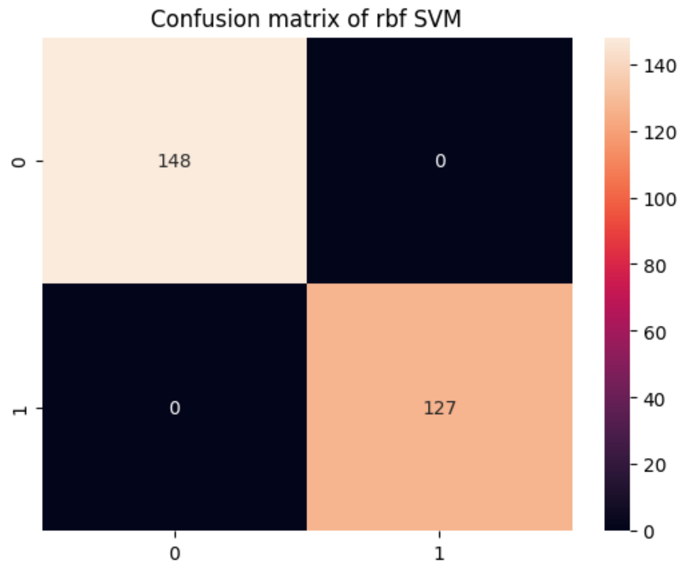
```
# linear kernel

svc = SVC(C=0.0001, gamma=10, kernel='linear')

svc.fit(X_train, y_train)

y_pred = svc.predict(X_test)

cm = confusion_matrix(y_test,y_pred)
sns.heatmap(cm, annot=True, fmt='d').set_title('Confusion matrix of linear SVM') # fmt='d' formats the numbers as digits, which means i

print(classification_report(y_test,y_pred))
```

```
                   precision    recall  f1-score   support

              0       0.82      0.96      0.88       148
              1       0.94      0.76      0.84       127

       accuracy                           0.87       275
      macro avg       0.88      0.86      0.86       275
   weighted avg       0.88      0.87      0.86       275
```



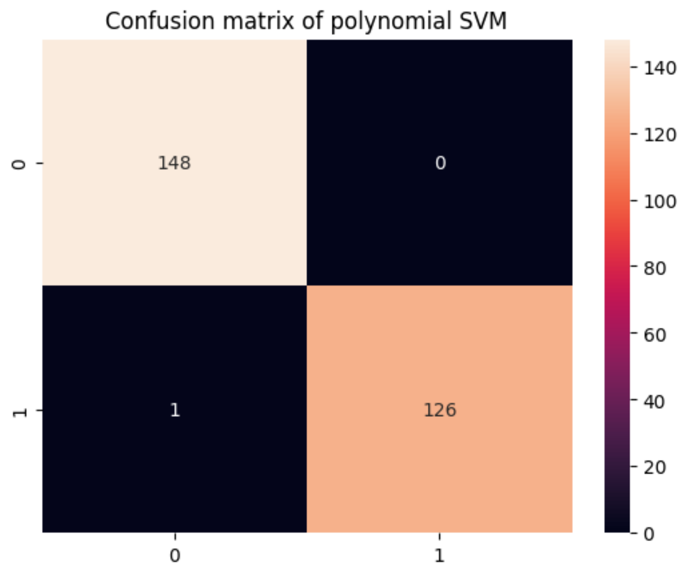
```
# rbf f1 score: 0.9979166666666666
svc = SVC(C=1, gamma=1, kernel='rbf')

svc.fit(X_train, y_train)

y_pred = svc.predict(X_test)

cm = confusion_matrix(y_test,y_pred)
sns.heatmap(cm, annot=True, fmt='d').set_title('Confusion matrix of rbf SVM') # fmt='d' formats the numbers as digits, which means inte

print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       148
           1       1.00      1.00      1.00       127

    accuracy                           1.00       275
   macro avg       1.00      1.00      1.00       275
weighted avg       1.00      1.00      1.00       275
```



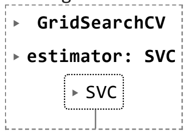
```
# polynomial f1 score 0.9989528795811518
svc = SVC(C=0.0001, gamma=10, kernel='poly')

svc.fit(X_train, y_train)

y_pred = svc.predict(X_test)

cm = confusion_matrix(y_test,y_pred)
sns.heatmap(cm, annot=True, fmt='d').set_title('Confusion matrix of polynomial SVM') # fmt='d' formats the numbers as digits, which mea

print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      1.00       148
           1       1.00      0.99      1.00       127

    accuracy                           1.00       275
   macro avg       1.00      1.00      1.00       275
weighted avg       1.00      1.00      1.00       275
```



```
parameters_dictionary = {'kernel':['rbf', 'sigmoid', 'linear', 'poly'],
                         'C':[0.0001, 1, 10],
                         'gamma':[1, 10, 100]}
svc = SVC()

grid_search = GridSearchCV(svc,
                           parameters_dictionary,
                           scoring = 'f1',
                           return_train_score=True,
                           cv = 5,
                           verbose = 1) # Displays how many combinations of parameters and folds we'll have, for more information as tl
grid_search.fit(X_train, y_train)
```

```
    Fitting 5 folds for each of 36 candidates, totalling 180 fits
```



```
best_model = grid_search.best_estimator_
best_parameters = grid_search.best_params_
best_f1 = grid_search.best_score_

print('The best model was:', best_model)
print('The best parameter values were:', best_parameters)
print('The best f1-score was:', best_f1)
```

```
    The best model was: SVC(C=0.0001, gamma=10, kernel='poly')
    The best parameter values were: {'C': 0.0001, 'gamma': 10, 'kernel': 'poly'}
    The best f1-score was: 0.9989528795811518
```

```
# to see if there is any overfitting issue
gs_mean_test_scores = grid_search.cv_results_['mean_test_score']
gs_mean_train_scores = grid_search.cv_results_['mean_train_score']

print("The mean test f1-scores were:", gs_mean_test_scores)
print("The mean train f1-scores were:", gs_mean_train_scores)
```

```
    The mean test f1-scores were: [0.         0.         0.78017291 0.98571413 0.         0.
     0.78017291 0.99895288 0.         0.         0.78017291 0.99895288
     0.99791667 0.53456647 0.98865407 0.99895288 0.76553515 0.53395457
     0.98865407 0.99895288 0.040291   0.54523168 0.98865407 0.99895288
```