LESSON19 Transformers and Pipelining

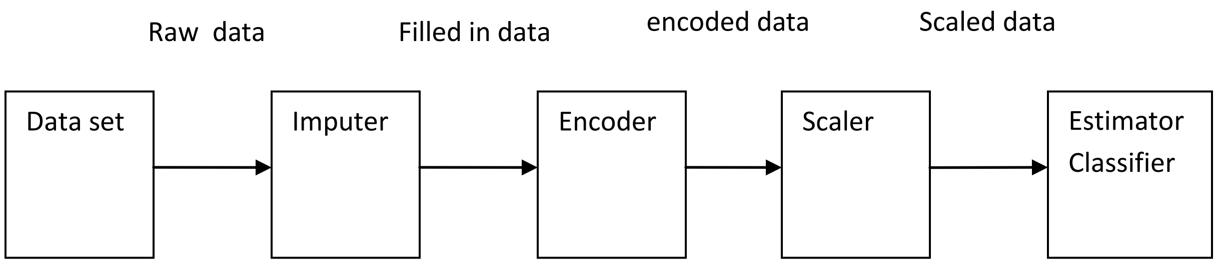
Last update June 1,2021

Machine Language algorithms typically process tabular data. You may want to do preprocessing and post-processing of this data before and after your Machine Language algorithm.

You may need to perform sequence of different transformations fill missing data with an average value, add new data, select good features, remove features, encode categorical data columns, scale data of the raw dataset before applying the final estimator.

A Pipeline gives you a single interface for all 3 steps of transformation filling in missing data, encoding, scaling, and a resulting estimator. It encapsulates transformers and predictors inside it.

A pipeline is a way to chain data processing steps together.



Transformers

A **Transformer** is some class that has a **fit** and **transform** method, or a **fit_transform** method.

Fit and transform methods

Pipelines and/or steps in the pipeline must have these two methods

- "**fit"** to learn on the data and acquire state (e.g.: neural network's neural weights are such state)
- "**transform"** (or "predict") to actually process the data and generate a prediction.

fit_transform method

call this method to chain both (fit and transform):

- "fit_transform" to fit and then transform the data, but in one pass, which allows for potential code optimizations when the two methods must be done one after the other directly.

Fit and transform process

- You must prepare your raw data using data transforms prior to fitting a machine learning model.

- This is required to ensure that you best expose the structure of your predictive modeling problem to the learning algorithms.

- Applying data transforms like scaling or encoding categorical variables is straightforward when all input variables are the same type. It can be challenging when you have a dataset with mixed types and you want to selectively apply data transforms to some, but not all, input features.

Column Transformer

The ColumnTransformer allows you to selectively apply data transforms to different columns in your dataset. It is important to prepare data prior to modeling. This may involve replacing missing values, scaling numerical values, and one hot encoding categorical data.

Data transforms can be performed using the scikit-learn library:

SimpleImputer class can be used to replace missing values
MinMaxScaler class can be used to scale numerical values
OneHotEncoder can be used to encode categorical variables
OrdinalEncoder can be used to encode a categorical column to a numeric column

Sequences of different transforms can also be chained together using the Pipeline, such as imputing missing values, then scaling numerical values.

You may want to impute missing numerical values with a median value, then scale the values and impute missing categorical values using the most frequent value and one hot encode the categories.

Estimators

Estimators are classes that implement both fit() and predict(). You'll find that many of the classifiers and regression models implement both these methods, and as such you can readily test many different models. It is possible to use another transformer as the final estimator (i.e., it doesn't necessarily implement predict(), but definitely implements fit()

Predictors

Predictor - some class that has fit and predict methods, or fit_predict method.

Pipeline

Pipelines apply the transformations and final estimator.

Pipeline gives you a single interface for all 3 steps of transformation supplying missing values, scaling, encoding and a resulting estimator. It encapsulates transformers and predictors inside:

Scikit-learn's pipeline class is a useful tool for encapsulating multiple different transformers alongside an estimator into one object, so that you only have to call your important methods once like fit() and predict(). Example pipeline:

```
# make pipeline
pipeline = Pipeline([
   ('imputer', Imputer()),
       ('encoder', OneHotEncoder()),
       ('encoder', OrdinalEncoder()),
       ('encoder', OneHotEncoder()),
       ('scaler', StandardScaler()),
   ('classifier', SGDClassifier()),
])
```

```
# fit
pipeline.fit(Xtrain).predict(Xtrain)

# predict
predicted = pipeline.predict(Xtest)
```

All steps of the pipeline except last one must be transforms, but the last step can be transformer or predictor.

When you call **pipeline.fit(),** each transformer inside pipeline will be fitted on outputs of previous transformer (First transformer is learned on raw dataset).

Last estimator may be transformer or predictor, you can call **fit_transform()** on pipeline only if your last estimator is transformer (that implements fit_transform, or transform and fit methods separately), you can call **fit_predict()** or **predict()** on pipeline only if your last estimator is predictor. So you just can't call **fit_transform** or transform on pipeline, last step of which is predictor.

PCA

Principal Component Analysis, or **PCA**, is a dimensionality-reduction method that is often **used to** reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set

The steps to perform PCA are the following:

1. Standardize the data.
2. Compute the covariance matrix of the features from the dataset.
3. Perform eigendecompositon on the covariance matrix.
4. Order the eigenvectors in decreasing order based on the magnitude of their corresponding eigenvalues.

Eigendecomposition is the factorization of a matrix into a canonical form, whereby the matrix is represented in terms of its eigenvalues and eigenvectors. Only diagonalizable matrices can be factorized in this way.

Eigenvalues and **eigenvectors** allow us to "reduce" a linear operation to separate, simpler, problems

Principal Component Analysis (**PCA**) is very useful to speed up the computation by reducing the dimensionality of the data. Plus, when you have high dimensionality with high correlated variable of one another, the **PCA can improve** the **accuracy** of classification model.

Principal component analysis (**PCA**) is a technique to bring out strong patterns in a dataset by suppressing variations. It is used to clean data sets to make it easy to explore and analyze. The **algorithm** of **Principal Component Analysis** is based on a few mathematical ideas namely: Variance and Covariance.

Creating a PCA

The main purpose of a PCA is to convert large data sets into smaller data sets. The first thing we need to do is get a large data sets. It would be too much trouble to generate one ourselves. There are many ready made on the internet. We will use a famous one called the iris data set from sklearn.

This data sets consists of 3 different types of irises' (Setosa, Versicolour, and Virginica) petal and sepal length, stored in a 150x4 numpy.ndarray

The rows being the samples and the columns being: Sepal Length, Sepal Width, Petal Length and Petal Width.

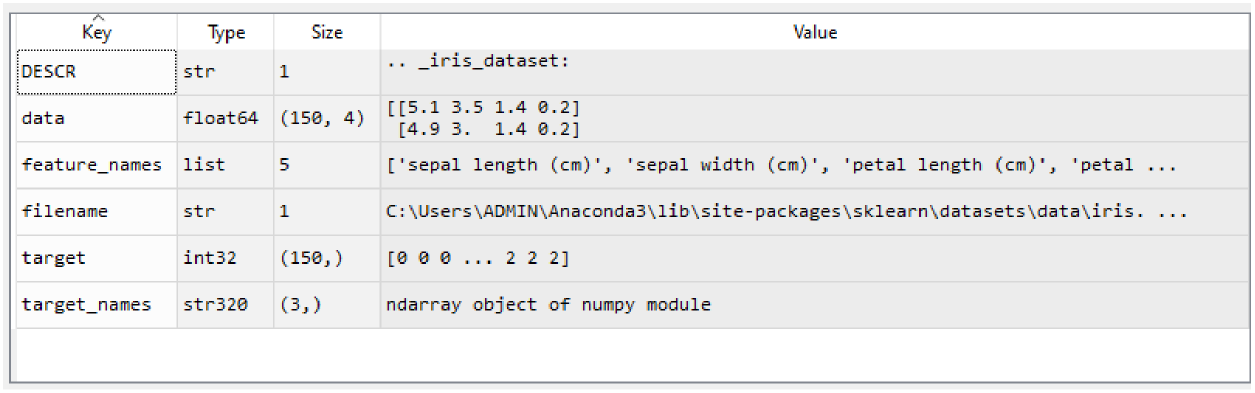
the data set consists of 50 samples from each of three species of *Iris* (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Four features were measured from each sample: the length and the width of the sepals and petals, in centimeters. Based on the combination of these four features, we can distinguish the species from each other.

Petals are modified leaves that surround the reproductive parts of flowers.

Sepal: The outer parts of the **flower** (often green and leaf-like) that enclose a developing bud. The sepal is a defensive organ that encloses and protects the developing reproductive structures. At maturity, the sepal opens when the flower blooms

Iris setosa	*Iris versicolor*	*Iris virginica*

The iris data set is available from sklearn as a dictionary of values.



Key	Type	Size	Value
DESCR	str	1	.. _iris_dataset:
data	float64	(150, 4)	[[5.1 3.5 1.4 0.2] [4.9 3. 1.4 0.2]
feature_names	list	5	['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal ...
filename	str	1	C:\Users\ADMIN\Anaconda3\lib\site-packages\sklearn\datasets\data\iris. ...
target	int32	(150,)	[0 0 0 ... 2 2 2]
target_names	str320	(3,)	ndarray object of numpy module

We first load in the iris data set available from sklearn.

```
# import iris data set
iris = datasets.load_iris()

# get data and target
X = iris.data
y = iris.target
```

We then make a data frame so we can view the data more easier. We add the target flower numbers as the last column.

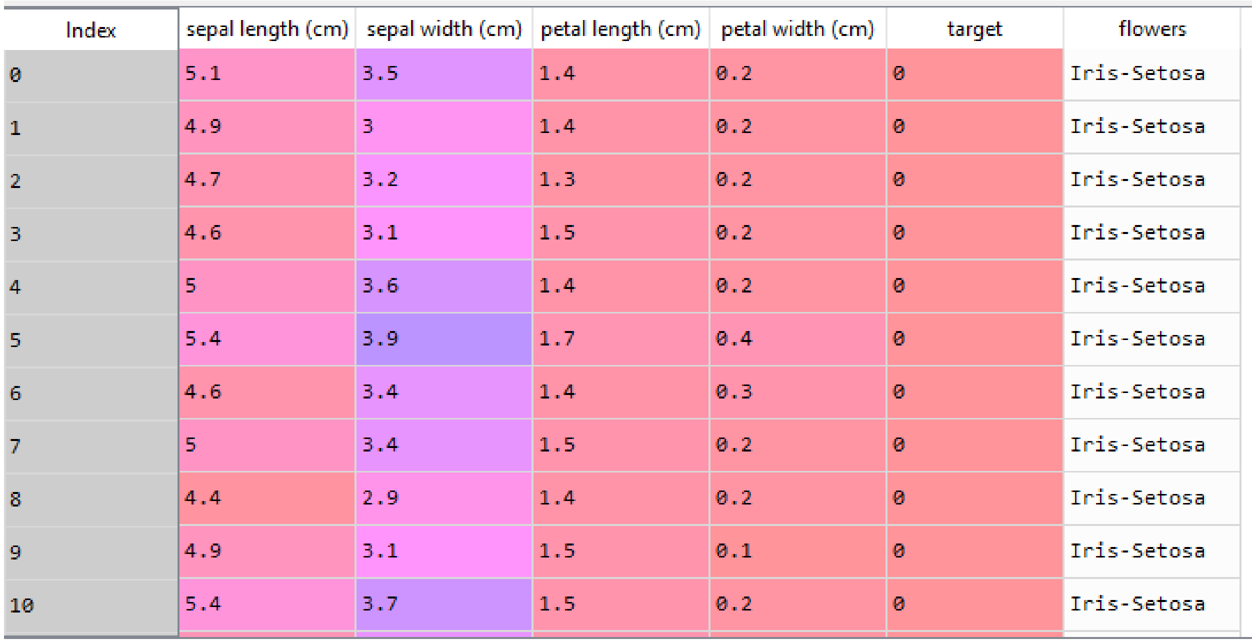
```
# add target to feature names
columns = iris.feature_names + ["target"] columns.append("target")
print(columns)

# make data  frame
df = pd.DataFrame(np.column_stack((X,y)), columns = columns)
```

We also make an extra column in our data set to identify the flower name

```
# map flower names to target numbers
target_flowers = {0:"Iris-Setosa",1:"Iris-Versicolour",2:"Iris-Virginica"}
df["flowers"] = df['target'].map(target_flowers)
```

We then print out the data frame. (We show a snapshot for more clarification)

```
# print data frame
print(df)
```



Index	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target	flowers
0	5.1	3.5	1.4	0.2	0	Iris-Setosa
1	4.9	3	1.4	0.2	0	Iris-Setosa
2	4.7	3.2	1.3	0.2	0	Iris-Setosa
3	4.6	3.1	1.5	0.2	0	Iris-Setosa
4	5	3.6	1.4	0.2	0	Iris-Setosa
5	5.4	3.9	1.7	0.4	0	Iris-Setosa
6	4.6	3.4	1.4	0.3	0	Iris-Setosa
7	5	3.4	1.5	0.2	0	Iris-Setosa
8	4.4	2.9	1.4	0.2	0	Iris-Setosa
9	4.9	3.1	1.5	0.1	0	Iris-Setosa
10	5.4	3.7	1.5	0.2	0	Iris-Setosa

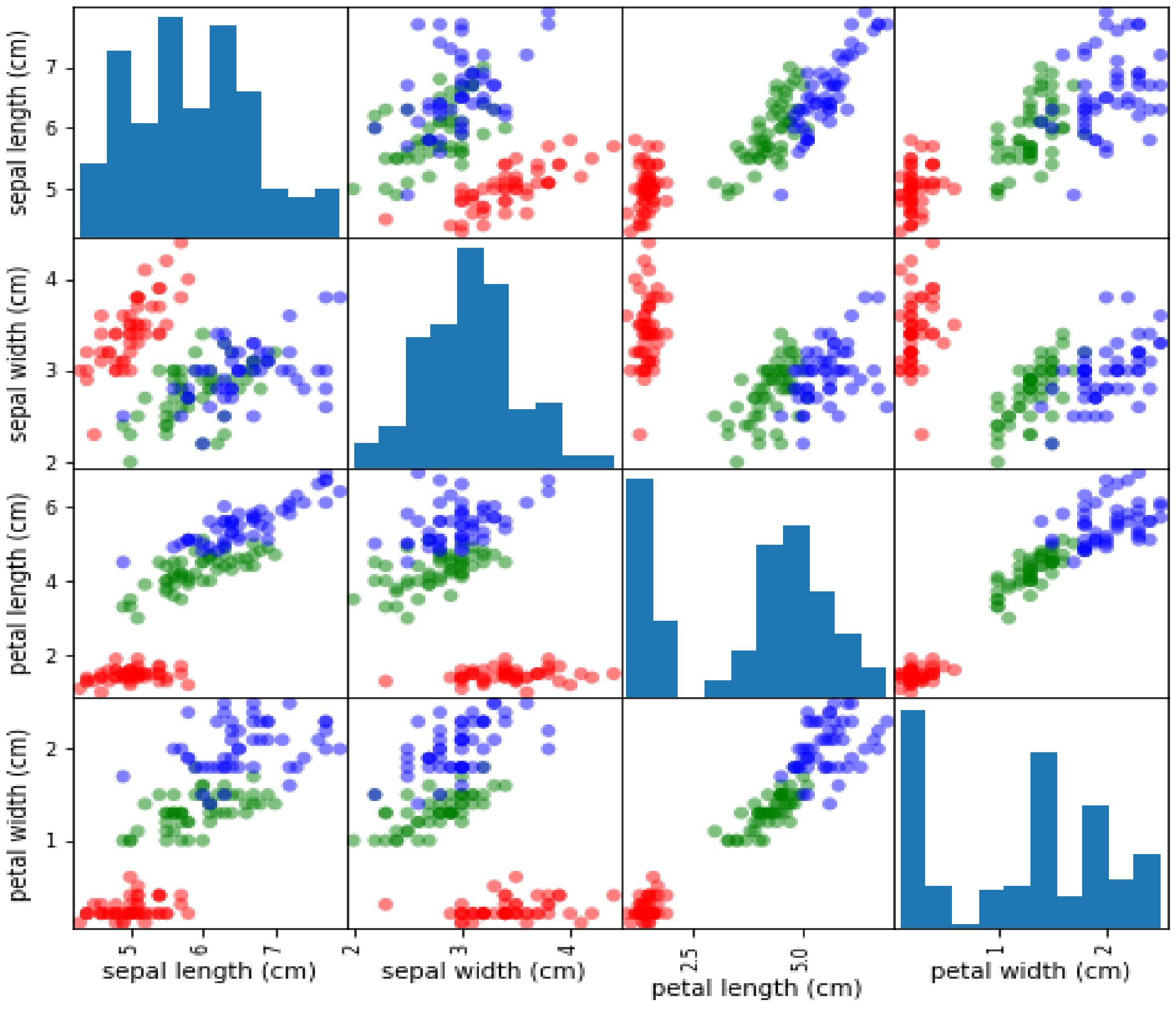
We can also plot a pairplot showing the relationships between sepal and petal lengths and widths

The data set has 4 measurements: sepal width, sepal length, petal_length and petal_width. The data contains measurements of different flowers.

We use the pandas **plotting.scatter_matrix** function to plot the pair plot

We have also used color to represent the different types of flowers.

```
# assign colors to flower types
# 0:Iris-Setosa,1:Iris-Versicolour,2:Iris-Virginica
target_colors = {0:"red",1:"green",2:"blue"}

# plot pair plot
pd.plotting.scatter_matrix(df.iloc[:,0:4], c=df['target'].map(target_colors),figsize=(8,8), marker='o')
plt.show()
```



The next thing we do is to spit our data into training and test sets. We fit with the train data and we test with the test data.

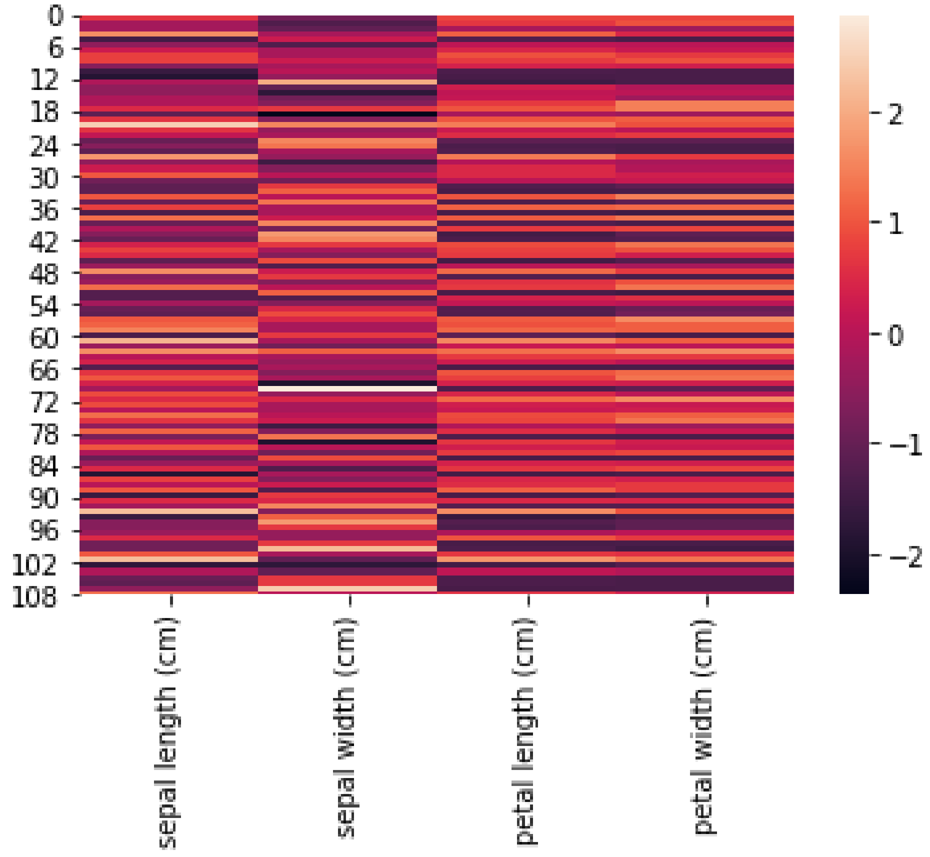
```
# Splitting data into train and testing part
# The 25 % of data is test size of the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)

print("train data size: ",X_train.shape)
```

We then make a **scalar** so we can transform the dat. Scaling transforms the data so that its data distribution will have a mean value 0 and standard deviation of 1. If feature scaling is not done, then a machine learning algorithm tends to weigh greater values, higher and consider smaller values as the lower values, regardless of the unit of the values.

```
# make scalar
scalar = StandardScaler()

# scale train data
scalar.fit(X_train)
scaled_data = scalar.transform(X_train)
```

We then plot a heat map of our scaled data. A heat map is a two-dimensional representation of information using colors to represent values. Heat maps help the user visualize the data set.

```
# put scaled data in a data frame
df_comp = pd.DataFrame(scaled_data, columns = iris.feature_names)

# plotting heatmap
sns.heatmap(df_comp)
plt.show()
```



Next we do the PCA

Principal Component Analysis, or **PCA**, is a dimensionality-reduction method that is often **used to** reduce the dimensionality of large data sets, by transforming a large set of variables into a smaller one that still contains most of the information in the large set. We will use 2 components meaning we will only have 2 rows of data for the 4 columns.

```
# use 2  components
pca = PCA(n_components = 2)
pca.fit(scaled_data)
x_pca = pca.transform(scaled_data)

print("PCA size: ", x_pca.shape)
```

```
PCA size:  (112, 2)
```

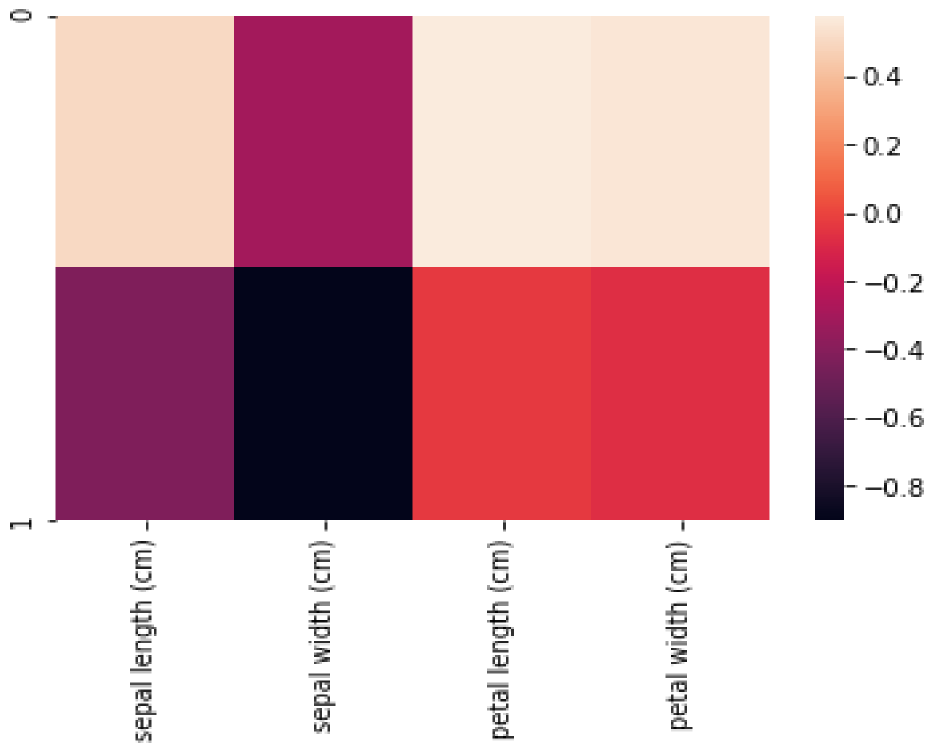
We now plot a heat map of the PCA results. We first make a dataframe to house pca components and the flower names. Note: our PCA data only has two rows.

```
# make data frame of pca components
df_comp = pd.DataFrame(pca.components_, columns = iris.feature_names)

# print pca components
print(pca.components_)
```

```
pca components:
[[ 0.53689105 -0.23408337  0.58145301  0.56468166]
 [ 0.30556685  0.94853642  0.0218382   0.08019139]]
```

```
# plotting heatmap
sns.heatmap(df_comp)
plt.show()
```

Here is the heat map:



We can now apply the classifier to classify the data to their flower classes.
We use the LogisticRegression classifier, to the train data and then print out the accuracy score. We need to set the arguments of the classifier to avoid warnings.

```
# make classifier
classifier = LogisticRegression(multi_class='auto',solver='lbfgs')

# apply classifier
classifier.fit(X_train, y_train)

# print accuracy score
print(accuracy_score(y_test, classifier.predict(X_test)))
```

```
classifier accuracy:  0.9473684210526315
```

Here is the complete program:

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn import svm
from sklearn.metrics import accuracy_score
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.pipeline import Pipeline

# import iris data set
iris = datasets.load_iris()

# get data and target
X = iris.data
y = iris.target
```

```
# add target to feature names
columns = iris.feature_names + ["target"] columns.append("target")
print(columns)

# make data  frame
df = pd.DataFrame(np.column_stack((X,y)), columns = columns)

# map flower names to target numbers
target_flowers = {0:"Iris-Setosa",1:"Iris-Versicolour",2:"Iris-Virginica"}
df["flowers"] = df['target'].map(target_flowers)

# print data frame
print(df)

# assign colors to flower types
# 0:Iris-Setosa,1:Iris-Versicolour,2:Iris-Virginica
target_colors = {0:"red",1:"green",2:"blue"}

# plot pair plot
pd.plotting.scatter_matrix(df.iloc[:,0:4], c=df['target'].map(target_colors),figsize=(8,8),
marker='o')
plt.show()

# Splitting data into train and testing part
# The 25 % of data is test size of the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)

print("train data size: ",X_train.shape)

# make scaler
scalar = StandardScaler()

# scale train data
scalar.fit(X_train)
scaled_data = scalar.transform(X_train)

# put scaled data in a data frame
df_comp = pd.DataFrame(scaled_data, columns = iris.feature_names)

# plotting heatmap
sns.heatmap(df_comp)
plt.show()
```

```
# use 2  components
pca = PCA(n_components = 2)
pca.fit(scaled_data)
x_pca = pca.transform(scaled_data)

print("PCA size: ", x_pca.shape)

# make data frame of pca components
df_comp = pd.DataFrame(pca.components_, columns = iris.feature_names)

# print pca components
print(pca.components_)

# plotting heatmap
sns.heatmap(df_comp)
plt.show()

# make classifier
classifier = LogisticRegression(multi_class='auto',solver='lbfgs')

# apply classifier
classifier.fit(X_train, y_train)

# print accuracy score
print(accuracy_score(y_test, classifier.predict(X_test)))
```

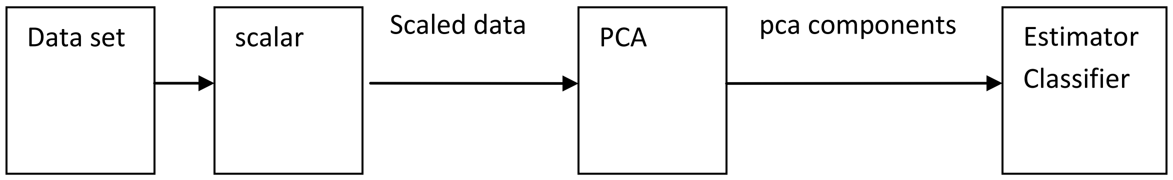
to do

Try these different classifiers and compare accuracy scores.

```
classifier = DecisionTreeClassifier()
classifier = svm.SVC(gamma='auto')
```

using a pipeline

A Pipeline is a list of transforms with a final estimator. The PipeLine sequentially applies a list of transforms and a final estimator. A pipeline let us to do scaling, PCA and classifying all in one statement. The output of one transform is sent to the other transform until the estimator classifier is reached.



Here is the pipeline code:

```
from sklearn.pipeline import Pipeline

print("using pipeline")
```

We first make the pipeline and give it the PCA, scalar and classifier as a tuple in a list. Each tuple has a identification name and the transformer or estimator classifier object like this:

[(‘name’,transformer),(‘name2’,tranformer2),(‘name3’, estimator)]

Here we make the pipeline using the PCA, Standard Scaler and LogisticRegression estimator classifier.

```
# make pipeline
pipe = Pipeline([('pca', PCA(n_components = 2)), ('std', StandardScaler()),
            ('logistic', LogisticRegression(multi_class='auto',solver='lbfgs'))])
```

we then fit the data using the pipe using the X_train and y_train data from the previous section.

```
# fitting the data in the pipe
pipe.fit(X_train, y_train)
```

Lastly we calculate the fit accuracy score using the test data.

```
# score data
score = accuracy_score(y_test, pipe.predict(X_test))
print("pipeline accuracy: ", score)
```

```
pipeline accuracy:  0.9736842105263158
```

Here is the complete pipeline code part:

```
from sklearn.pipeline import Pipeline

print("using pipeline")

# make pipeline
pipe = Pipeline([('pca', PCA(n_components = 2)), ('std', StandardScaler()),
            ('logistic', LogisticRegression(multi_class='auto',solver='lbfgs'))])

# fitting the data in the pipe
pipe.fit(X_train, y_train)

# score data
score = accuracy_score(y_test, pipe.predict(X_test))
print("pipeline accuracy: ", score)
```

To do:

try these different classifiers:

```
classifier = DecisionTreeClassifier()
classifier = svm.SVC(gamma='auto')
```

COLUMN TRANSFORMER

The **ColumnTransformer** is a class in the scikit-learn Python machine learning library that allows you to selectively apply data transforms.

Each transformer is a three-element tuple that defines the identification name of the transformer, the transform to apply, and the column indices or names to apply it to. For example:

(Name, Object, Columns)

('cat', OneHotEncoder(handle_unknown='ignore'), [0, 1])

Where 'cat' is name of the transformer and OneHotEncoder() is the transformer object and [0,1] are the column indices to apply the transformer object to.

('cat', OneHotEncoder(handle_unknown='ignore'), ['country','education'])

We use handle_unknown='ignore' because sometimes a value if the test set do not appear in the train/test set

Where 'cat' is name of the transformer and OneHotEncoder() is the transformer object and ['country','education'] are the column names to apply the transformer object to.

Available transformers

Transformer	**Use**
SimpleImputer(strategy='median')	replace missing values
OneHotEncoder(handle_unknown='ignore')	used to encode categorical variables.
OrdinalEncoder()	Convert string data to numeric codeing
PCA(n_components = 2)	Convert large data sets into smaller data sets.
MinMaxScaler()	scale numerical values
StandardScaler()	moving the mean and scaling to unit variance

We can make a Column Transformer like this

```
# define column transformer
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])],
remainder='passthrough'))
```

or

```
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(),
[‘column1’, ‘column2’])],
remainder='passthrough'))
```

remainder = ‘passthrough’ means to pass columns to the output as is without changing them. If we do not specify   remainder = ‘passthrough’ then only the specified columns are specified to the output.

Once the transformer is defined, it can be used to transform a dataset using the **fit_transform** function

```
transformed_data = transformer.fit_transform(X_data)
```

A ColumnTransformer can also be used in a Pipeline to selectively prepare the columns of your dataset before fitting a model estimator classified on the transformed data. Here are the steps in pseudo code syntax.

```
# define model estimator classifier
classifier = LogisticRegression()

# define column transformer
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])])
```

or

```
transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(),
[‘column1’, ‘column2’])])
```

```
# define pipeline
pipeline = Pipeline(steps=[('transformer', transformer), (‘classifier’,classifier)])

# fit the pipeline on the transformed data
pipeline.fit(X_train,y_train)

# make predictions
y_pred =  pipeline.predict(X_test)
```

Available Estimator Classifiers

Estimator Classifier
DecisionTreeClassifier()
svm.SVC(gamma='auto')
LogisticRegression(multi_class='auto',solver='lbfgs')
SGDClassifier(random_state=42, max_iter=30, tol=1e-3)

Example using Column Transformer and Pipeline

We can use the data set from the scaling and decoding lesson where we predict range of salaries as High , Medium and Low given name, age, gender,country ,education and salary. Note: we drop the name since it is irreverent.

Salaries.csv

```
         name  age  gender country    education   salary   range
0   tom smith   34    Male  Canada   University  75000.0    High
1   bob jones   36    Male     USA  High School  25000.0     Low
2  dave smith   24    Male  Europe      College      NaN  Medium
3  bill smith   28  Female   India  High School  25000.0  Medium
4   jay smith   23    Male   China  High School  25000.0     Low
5   sue smith   44  Female     USA   University  55000.0    High
6   sid jones   42    Male   China  High School  25000.0  Medium
7  mary smith   34    Male   India      College      NaN  Medium
8  dodo smith   34    Male  Europe      College  20000.0     Low
9   sid smith   24    Male  Europe      College  60000.0    High
```

Things we have to do:

(1) load in data set
(2) drop name column
(3) remove NAN from salary and replace with an average value column using
    SimpleImputer(strategy='median') transformer
(4) one hot encode education column using
OneHotEncoder(**handle_unknown='ignore'**),
(5) one hot encode country column using
OneHotEncoder(**handle_unknown='ignore'**),
(6) encode gender column using OrdinalEncoder()
(7) put all transformers in a ColumnTransformer
(8) make scaler just to print out scaled data
(9) split data into test and train sets
(10) make classifier estimator model
(11) make pipeline with transformer and classifier model
(12) fit the data using pipeline
(13) predict data using pipeline mode
(14) calculate accuracy score

Here is the program:

```
import pandas as pd;
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline
from sklearn.svm import SVR
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import OrdinalEncoder
from sklearn.preprocessing import OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import SGDClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn import svm

# read in salaries.csv file
df = pd.read_csv("salaries.csv")

# Printing data frame
print(df.head())

# we need to drop name column
df = df.drop("name",axis='columns')
print(df.head())


# make column transformer
transformer =
ColumnTransformer(transformers=[('cat',OneHotEncoder(handle_unknown='ignore'),
                                  ['country','education']),
                                  ('ordinal',OrdinalEncoder(),['gender']),
                                  ('inputer',SimpleImputer(strategy='median'),['salary'])],
                                remainder='passthrough')

# obtain transformed data forviwing
transformed_data = transformer.fit_transform(df.iloc[:,0:5]).astype(int)

# print transformed data
print(transformed_data)

# scale X data
scaler = StandardScaler()
scaler.fit(transformed_data.astype(float))
X_scaled = scaler.transform(transformed_data.astype(float))

# print scaled data
print("\nscaled x")
print(X_scaled)

# spilr data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:,0:5], df.iloc[:,5], test_size=0.5,
random_state=42)
```

```
# define estimator classifier  model
classifier = SGDClassifier(random_state=42, max_iter=30, tol=1e-3)
classifier=DecisionTreeClassifier()
#classifier = LogisticRegression(multi_class='auto',solver='lbfgs')
#classifier = svm.SVC(gamma='auto')
# define transform
#transformer = ColumnTransformer(transformers=[('cat', OneHotEncoder(), [0, 1])])

# make pipeline using transforer, scalar and classifier
# we scale the data aftr the data has been transformed
pipeline = Pipeline(steps=[('transformer', transformer),
('scaler',StandardScaler()),('classifier',classifier)])

# fit the pipeline on the train data
pipeline.fit(X_train,y_train)

# make predictions with test data
y_pred = pipeline.predict(X_test)

# print out prediction and expected test values
print(f"{'y_pred':10} {'y_test':10}")

for i in range(len(y_pred)):
   print(f'{y_pred[i]:10} {y_test.values[i]:10}')

# calculate accuracy score
score = accuracy_score(y_test, y_pred)

# print accuracy score
print("pipeline accuracy: ", score)
```

Here is the output:

```
salaries.csv
         name  age  gender country    education   salary   range
0   tom smith   34    Male  Canada   University  75000.0    High
1   bob jones   36    Male     USA  High School  25000.0     Low
2  dave smith   24    Male  Europe      College      NaN  Medium
3  bill smith   28  Female   India  High School  25000.0  Medium
4   jay smith   23    Male   China  High School  25000.0     Low
salaries without name
   age  gender country    education   salary   range
0   34    Male  Canada   University  75000.0    High
1   36    Male     USA  High School  25000.0     Low
2   24    Male  Europe      College      NaN  Medium
3   28  Female   India  High School  25000.0  Medium
4   23    Male   China  High School  25000.0     Low
```

```
transformed data
[[    1     0     0     0     0     0     0     1     1 75000    34]
 [    0     0     0     0     1     0     1     0     1 25000    36]
 [    0     0     1     0     0     1     0     0     1 27000    24]
 [    0     0     0     1     0     0     1     0     0 25000    28]
 [    0     1     0     0     0     0     1     0     1 25000    23]
 [    0     0     0     0     1     0     0     1     0 55000    44]
 [    0     1     0     0     0     0     1     0     1 25000    42]
 [    0     0     1     0     0     1     0     0     0 27000    34]
 [    0     0     1     0     0     1     0     0     1 20000    34]
 [    0     0     1     0     0     1     0     0     1 60000    24]
 [    1     0     0     0     0     0     0     1     1 65000    34]
 [    0     0     0     0     1     0     1     0     1 35000    36]
 [    0     0     1     0     0     1     0     0     1 27000    24]
 [    0     0     0     1     0     0     1     0     0 15000    28]
 [    0     1     0     0     0     0     1     0     1 27000    23]
 [    0     0     0     0     1     0     0     1     0 45000    44]
 [    0     1     0     0     0     0     1     0     1 28000    42]
 [    0     0     0     1     0     1     0     0     1 27000    34]
 [    0     0     1     0     0     1     0     0     1 24000    34]]

scaled x
[[ 2.91547595 -0.51639778 -0.67936622 -0.4330127  -0.51639778 -0.76376262
  -0.85280287  1.93649167  0.5976143   2.45835847  0.18155787]
 [-0.34299717 -0.51639778 -0.67936622 -0.4330127   1.93649167 -0.76376262
   1.17260394 -0.51639778  0.5976143  -0.58257974  0.46902449]
 [-0.34299717 -0.51639778  1.47196014 -0.4330127  -0.51639778  1.30930734
  -0.85280287 -0.51639778  0.5976143  -0.46094221 -1.25577526]
 [-0.34299717 -0.51639778 -0.67936622  2.30940108 -0.51639778 -0.76376262
   1.17260394 -0.51639778 -1.67332005 -0.58257974 -0.68084201]
 [-0.34299717  1.93649167 -0.67936622 -0.4330127  -0.51639778 -0.76376262
   1.17260394 -0.51639778  0.5976143  -0.58257974 -1.39950857]
 [-0.34299717 -0.51639778 -0.67936622 -0.4330127   1.93649167 -0.76376262
  -0.85280287  1.93649167 -1.67332005  1.24198319  1.618891  ]
 [-0.34299717  1.93649167 -0.67936622 -0.4330127  -0.51639778 -0.76376262
   1.17260394 -0.51639778  0.5976143  -0.58257974  1.33142437]
 [-0.34299717 -0.51639778  1.47196014 -0.4330127  -0.51639778  1.30930734
  -0.85280287 -0.51639778 -1.67332005 -0.46094221  0.18155787]
 [-0.34299717 -0.51639778  1.47196014 -0.4330127  -0.51639778  1.30930734
  -0.85280287 -0.51639778  0.5976143  -0.88667356  0.18155787]
 [-0.34299717 -0.51639778  1.47196014 -0.4330127  -0.51639778  1.30930734
  -0.85280287 -0.51639778  0.5976143   1.54607701 -1.25577526]
 [ 2.91547595 -0.51639778 -0.67936622 -0.4330127  -0.51639778 -0.76376262
  -0.85280287  1.93649167  0.5976143   1.85017083  0.18155787]
 [-0.34299717 -0.51639778 -0.67936622 -0.4330127   1.93649167 -0.76376262
   1.17260394 -0.51639778  0.5976143   0.0256079   0.46902449]
 [-0.34299717 -0.51639778  1.47196014 -0.4330127  -0.51639778  1.30930734
  -0.85280287 -0.51639778  0.5976143  -0.46094221 -1.25577526]
 [-0.34299717 -0.51639778 -0.67936622  2.30940108 -0.51639778 -0.76376262
   1.17260394 -0.51639778 -1.67332005 -1.19076738 -0.68084201]
 [-0.34299717  1.93649167 -0.67936622 -0.4330127  -0.51639778 -0.76376262
   1.17260394 -0.51639778  0.5976143  -0.46094221 -1.39950857]
 [-0.34299717 -0.51639778 -0.67936622 -0.4330127   1.93649167 -0.76376262
  -0.85280287  1.93649167 -1.67332005  0.63379554  1.618891  ]
 [-0.34299717  1.93649167 -0.67936622 -0.4330127  -0.51639778 -0.76376262
```

```
   1.17260394 -0.51639778  0.5976143  -0.40012345  1.33142437]
 [-0.34299717 -0.51639778 -0.67936622  2.30940108 -0.51639778  1.30930734
  -0.85280287 -0.51639778  0.5976143  -0.46094221  0.18155787]
 [-0.34299717 -0.51639778  1.47196014 -0.4330127  -0.51639778  1.30930734
  -0.85280287 -0.51639778  0.5976143  -0.64339851  0.18155787]]

y_pred     y_test
High       High
High       High
Medium     Low
Medium     Low
Low        Low
Medium     Medium
Medium     Medium
Low        Medium
High       High
Medium     Medium
pipeline accuracy:  0.7
```

Our Accuracy is very low because we only have 20 entries.

Todo:

Try different classifiers and make a chart of calculated accuracy scores.

DecisionTreeClassifier()
svm.SVC(gamma='auto')
LogisticRegression(multi_class='auto',solver='lbfgs')
SGDClassifier(random_state=42, max_iter=30, tol=1e-3)

Transformers and Pipeline Homework

Simple weather prediction

Take the weather prediction program from the Scaling and Encoding Lesson where you have predicted weather accordingly to temperature, atmospheric pressure, wind, humidity, precipitation, and cloudiness. Use a Column Transformer with a One-Hot Encode transformer to encode wind and Cloudiness and a Scaler. Use a classifier mode of your choice that gives you good results. Split weather data into train and test sets.
 Feed the column transformer and classifier into a pipeline. Use the pipeline to predict weather using the test data. Print out the accuracy score.

temperature	**atmospheric _pressure**	**wind**	**humidity**	**Precipitation**	**cloudiness**	**weather**
40	45	Very	67	56	sparse	raining
45	45	None	78	90	heavy	snowing
30	66	slightly	67	67	few	cold
80	77	breezy	78	78	many	hot
45	88	moderate	86	99	light	cloudy
45	56	None	78	66	many	windy

END