1. **Grid Search**

from sklearn.model\_selection import GridSearchCV

from sklearn.ensemble import RandomForestClassifier

# Define the model

model = RandomForestClassifier()

# Specify the hyperparameters to tune

param\_grid = {

    'n\_estimators': [10, 50, 100],

    'max\_depth': [10, 20, None]

}

# Perform grid search

grid\_search = GridSearchCV(estimator=model, param\_grid=param\_grid, cv=5)

grid\_search.fit(X\_train, y\_train)

# Get the best parameters

print(grid\_search.best\_params\_)

1. **PIPELINES**

**B4 PIPELINE**

from sklearn.preprocessing import StandardScaler

from sklearn.neighbors import KNeighborsClassifier

from sklearn.model\_selection import train\_test\_split, GridSearchCV

from sklearn.ensemble import RandomForestClassifier

from sklearn.pipeline import Pipeline

# Split the predictor and target variables

y = df['quality']

X = df.drop(columns='quality')

# Split into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.25,random\_state=42)

# Instantiate StandardScaler

scaler = StandardScaler()

# Transform the training and test sets

scaled\_data\_train = scaler.fit\_transform(X\_train)

scaled\_data\_test = scaler.transform(X\_test)

# Convert into a DataFrame(Conver scaled data to datafrmae)

scaled\_df\_train = pd.DataFrame(scaled\_data\_train, columns=X\_train.columns)

scaled\_df\_train.head()

# Instantiate KNeighborsClassifier

clf = KNeighborsClassifier()

# Fit the classifier

clf.fit(scaled\_data\_train,y\_train)

# Print the accuracy on test set

clf.score(scaled\_data\_test,y\_test)

**AFTER PIPELINE**

# Split into training and test sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=0.25,random\_state=42)

# Build a pipeline with StandardScaler and KNeighborsClassifier

scaled\_pipeline\_1 = Pipeline([('mms',StandardScaler()),

                              ('knn',KNeighborsClassifier())

                              ])

# Fit the training data to pipeline

scaled\_pipeline\_1.fit(scaled\_data\_train,y\_train)

# Print the accuracy on test set

scaled\_pipeline\_1.score(scaled\_data\_test,y\_test)



**PIPELINE 2**

# Create a pipeline

pipeline = Pipeline([

    ('scaler', StandardScaler()),  # Step 1: Standardize the features

    ('classifier', RandomForestClassifier())  # Step 2: Train a Random Forest model

])

# Train the pipeline

pipeline.fit(X\_train, y\_train)

# Make predictions

y\_pred = pipeline.predict(X\_test)

# Evaluate the model

accuracy = (y\_pred == y\_test).mean()

print(f'Accuracy: {accuracy}')

1. **PIPELINE 3 –Best to use processing and modelling pipelines differently**

#Now define those steps for our pipeline

processor = Pipeline(steps=[

    ('imputer',SimpleImputer(strategy='median')),

    ('scaler',StandardScaler())

])

#we go ahead and test this on our X\_train from before, to make sure we get the same result

processor.fit\_transform(X\_train).shape

# Now add the model - using another pipeline

pipe = Pipeline(steps=[

    ('preprocessor',processor),

    ('linreg',LinearRegression())

])

pipe.fit(X\_train,y\_train)

#Grab predictions out on our train and test sets, to evaluate

train\_preds = pipe.predict(X\_train)

test\_preds =pipe.predict(X\_test)

#print out R2-Score and Root Mean squared Error for our train and test data

print(f'Train set r2-score:{r2\_score(y\_train,train\_preds)}')

print(f'Train set RMSE: {mean\_squared\_error(y\_train,train\_preds,squared=False)}')

print('\*'\*20)

print(f'test set r2-score:{r2\_score(y\_test,test\_preds)}')

print(f'test set RMSE: {mean\_squared\_error(y\_test,test\_preds,squared=False)}')

#Import our cross validation

#Note that cross\_val\_score and cross\_val\_predict are variants of this function

from sklearn.model\_selection import cross\_validate

#Time to cross val!

#Pass in our pipeline and training dta ,set cv=5 and return the train score

#because we're using cross-validatte instead of cross\_val\_score , can set scoring  metrics

results = cross\_validate(pipe,X\_train,y\_train,cv=5,

                         return\_train\_score=True,

                         scoring=['r2','neg\_root\_mean\_squared\_error']

                         )

results

#Let's look at the avreage, plus a meaure of variance for train and test

print(f'Average Train Set r2-Score: {results["train\_r2"].mean()} +/- {results["train\_r2"].std()}')

print(f'Average Train Set RMSE: {results["train\_neg\_root\_mean\_squared\_error"].mean()\*-1} +/- {results["train\_neg\_root\_mean\_squared\_error"].std()}')

print('\*'\*20)

#Let's look at the avreage, plus a meaure of variance for train and test

print(f'Average test Set r2-Score: {results["test\_r2"].mean()} +/- {results["test\_r2"].std()}')

print(f'Average test Set RMSE: {results["test\_neg\_root\_mean\_squared\_error"].mean()\*-1} +/- {results["test\_neg\_root\_mean\_squared\_error"].std()}')

1. **GRID SEARCH WITH PIPELINE**

# Build a pipeline with StandardScaler and RandomForestClassifier

scaled\_pipeline\_2 = Pipeline([

    ('ss',StandardScaler()),

    ('RF',RandomForestClassifier())

])

# Define the grid

grid = [{'RF\_\_max\_depth': [4, 5, 6],

         'RF\_\_min\_samples\_split': [2, 5, 10],

         'RF\_\_min\_samples\_leaf': [1, 3, 5]}]

# Define a grid search

gridsearch = GridSearchCV(estimator=scaled\_pipeline\_2,

                          param\_grid=grid,

                          scoring='accuracy',

                          cv=5

                          )

# Fit the training data

gridsearch.fit(X\_train,y\_train)

# Print the accuracy on test set

gridsearch.score(X\_test,y\_test)

**GRID SEARCH WITH PIPELINE 2**

from sklearn.model\_selection import GridSearchCV

# Define parameter grid

param\_grid = {

    'classifier\_\_n\_estimators': [50, 100],  # 'classifier' is the RandomForest model

    'classifier\_\_max\_depth': [10, 20]

}

# Perform grid search on the pipeline

grid\_search = GridSearchCV(pipeline, param\_grid, cv=5)

grid\_search.fit(X\_train, y\_train)

# Get best parameters

print("Best parameters:", grid\_search.best\_params\_)

1. **GRID SEARCH CV SCORES 1**

# Mean training score

dt\_gs\_training\_score = dt\_grid\_search.cv\_results\_['mean\_train\_score'].mean()

dt\_gs\_training\_score

# # Mean test score

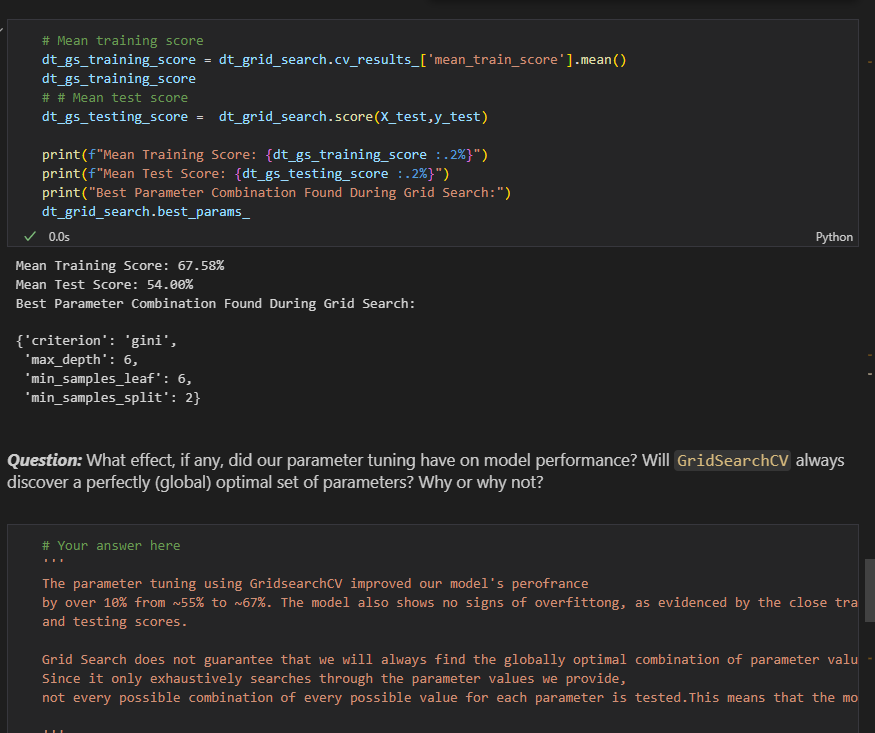
dt\_gs\_testing\_score =  dt\_grid\_search.score(X\_test,y\_test)

print(f"Mean Training Score: {dt\_gs\_training\_score :.2%}")

print(f"Mean Test Score: {dt\_gs\_testing\_score :.2%}")

print("Best Parameter Combination Found During Grid Search:")

dt\_grid\_search.best\_params\_



1. **GRID SEARCH CV SCORES 2**

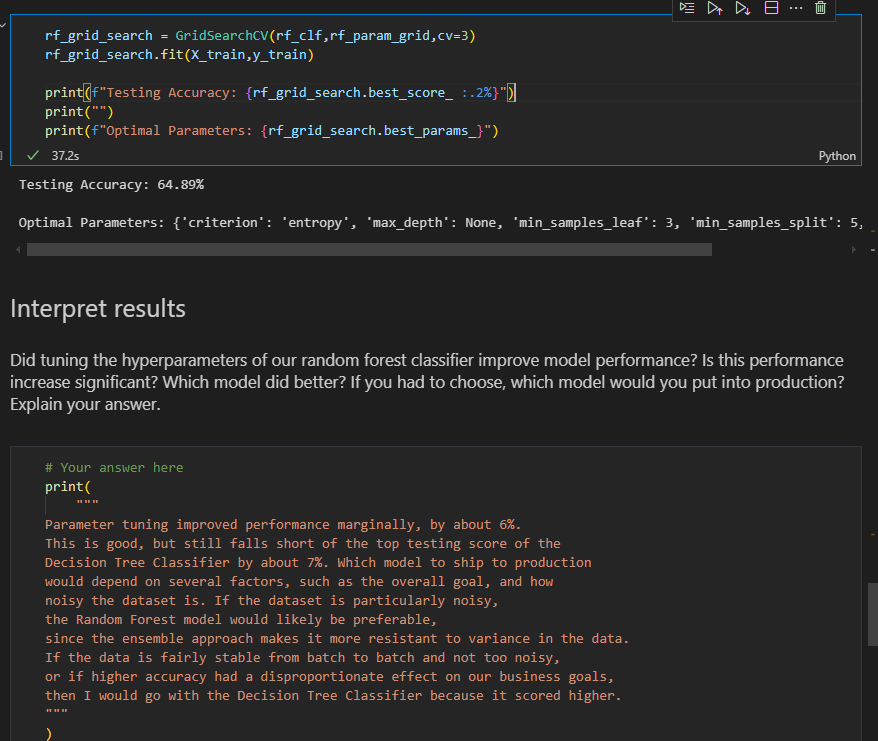
rf\_grid\_search = GridSearchCV(rf\_clf,rf\_param\_grid,cv=3)

rf\_grid\_search.fit(X\_train,y\_train)

print(f"Testing Accuracy: {rf\_grid\_search.best\_score\_ :.2%}")

print("")

print(f"Optimal Parameters: {rf\_grid\_search.best\_params\_}")



1. **Modify a pipeline(changing scaler and modelling)**

# Make Pipelines for stardadization and modeling

pipe = Pipeline([

    ('std',StandardScaler()),

    ('model',LogisticRegression())

])

# switching to Decision tree

from sklearn.tree import DecisionTreeClassifier

pipe.set\_params(model=DecisionTreeClassifier())

pipe.set\_params(std=MinMaxScaler())

1. **Example 2**

# Pipeline

pipe3 = Pipeline([

    ('pre-pro',transformer3),

    ('model',LogisticRegression())

])

Changing from logistic regression to Random forest

from sklearn.ensemble import RandomForestClassifier

pipe3.set\_params(model=RandomForestClassifier())

pipe3.fit(X\_train,y\_train)

y\_pred = pipe3.predict(X\_test)

print(classification\_report(y\_true=y\_test,y\_pred=y\_pred))

1. **Function transformer-** Let's say for the sake of example that we wanted to add a new feature called `number\_odd`, which is `1` when the value of `number` is odd and `0` when the value of `number` is even.

from sklearn.preprocessing import FunctionTransformer

def is\_odd(data):

    """

    Helper function that returns 1 if odd, 0 if even

    """

    return data % 2

# Instantiate transformer

func\_transformer = FunctionTransformer(is\_odd)

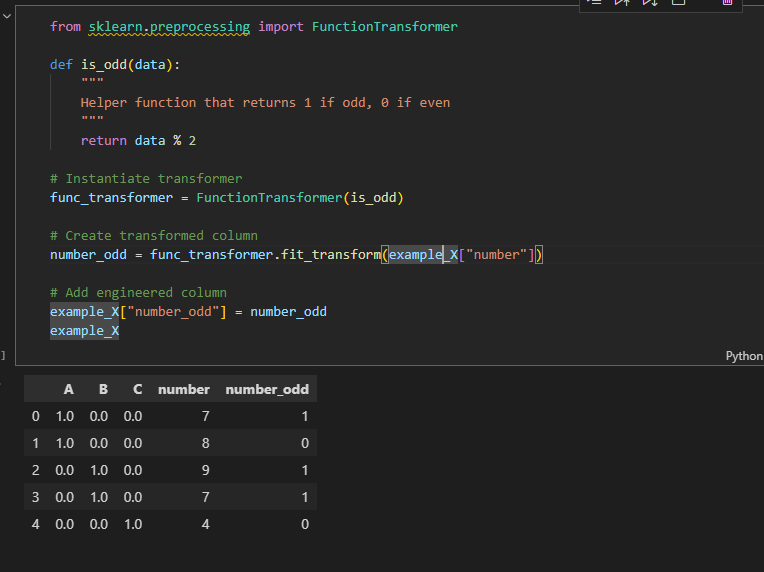
# Create transformed column

number\_odd = func\_transformer.fit\_transform(example\_X["number"])

# Add engineered column

example\_X["number\_odd"] = number\_odd

example\_X



1. **Scaling**

from sklearn.preprocessing import StandardScaler

# Instantiate transformer

scaler = StandardScaler()

# Create transformed dataset

data\_scaled = scaler.fit\_transform(example\_X)

# Replace dataset with transformed one

example\_X = pd.DataFrame(

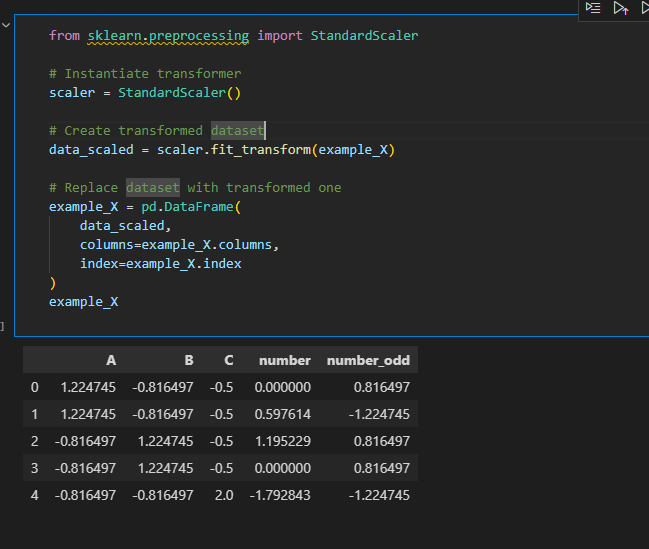
    data\_scaled,

    columns=example\_X.columns,

    index=example\_X.index

)

example\_X



1. **Preprocessing without pipeline(**combine ohe and function transform above)

def preprocess\_data\_without\_pipeline(X):

    transformers = []

    ### Encoding categorical data ###

    # Make a transformer

    ohe = OneHotEncoder(categories="auto", handle\_unknown="ignore", sparse=False)

    # Create transformed dataframe

    category\_encoded = ohe.fit\_transform(X[["category"]])

    category\_encoded = pd.DataFrame(

        category\_encoded,

        columns=ohe.categories\_[0],

        index=X.index

    )

    transformers.append(ohe)

    # Replace categorical data with encoded data

    X.drop("category", axis=1, inplace=True)

    X = pd.concat([category\_encoded, X], axis=1)

    ### Feature engineering ###

    def is\_odd(data):

        """

        Helper function that returns 1 if odd, 0 if even

        """

        return data % 2

    # Instantiate transformer

    func\_transformer = FunctionTransformer(is\_odd)

    # Create transformed column

    number\_odd = func\_transformer.fit\_transform(X["number"])

    transformers.append(func\_transformer)

    # Add engineered column

    X["number\_odd"] = number\_odd

    ### Scaling ###

    # Instantiate transformer

    scaler = StandardScaler()

    # Create transformed dataset

    data\_scaled = scaler.fit\_transform(X)

    transformers.append(scaler)

    # Replace dataset with transformed one

    X = pd.DataFrame(

        data\_scaled,

        columns=X.columns,

        index=X.index

    )

    return X, transformers

# Reset value of example\_X

example\_X = example\_data.drop("target", axis=1)

# Test out our function

result, transformers = preprocess\_data\_without\_pipeline(example\_X)

result

1. **Column Transformer 1**

from sklearn.compose import ColumnTransformer

from sklearn.pipeline import Pipeline

# Reset value of example\_X

example\_X = example\_data.drop("target", axis=1)

# Create a column transformer

col\_transformer = ColumnTransformer(transformers=[

    ("ohe", OneHotEncoder(categories="auto", handle\_unknown="ignore"),

["category"])

], remainder="passthrough")

# Create a pipeline containing the single column transformer

pipe = Pipeline(steps=[

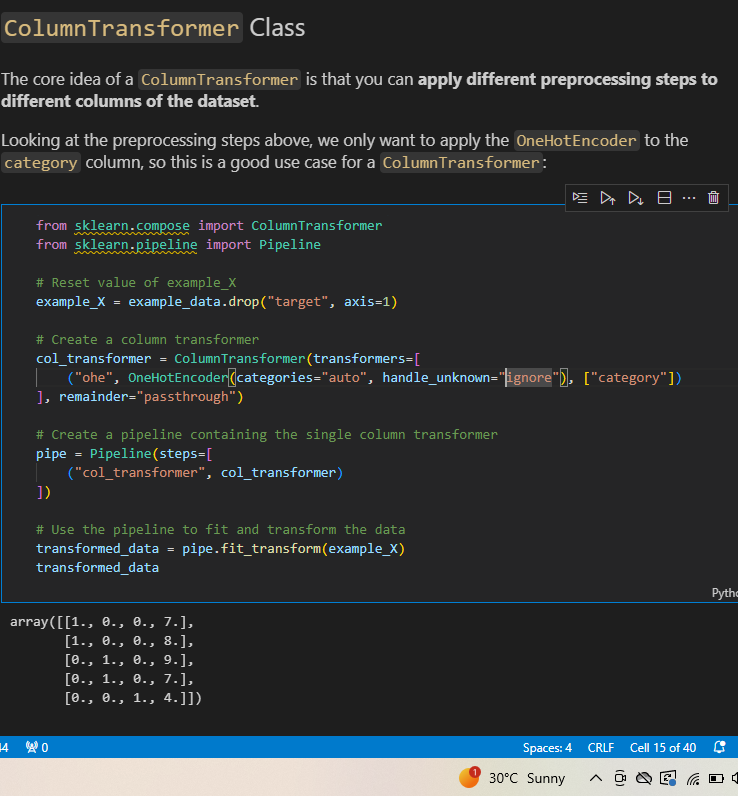
    ("col\_transformer", col\_transformer)

])

# Use the pipeline to fit and transform the data

transformed\_data = pipe.fit\_transform(example\_X)

transformed\_data



import numpy as np

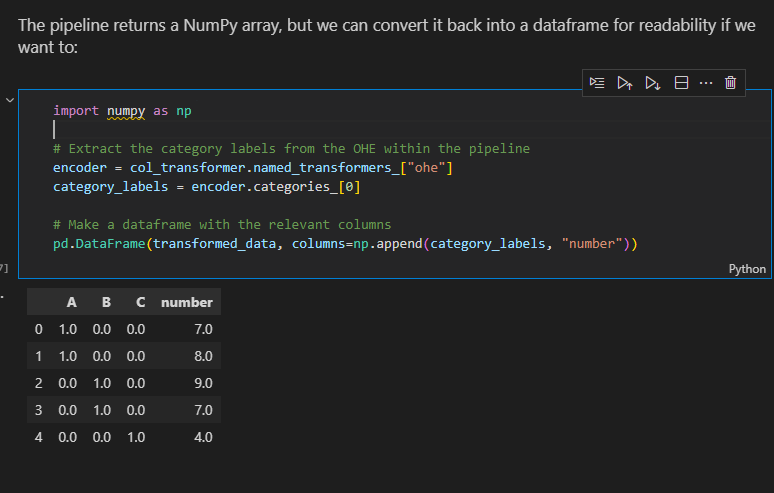
# Extract the category labels from the OHE within the pipeline

encoder = col\_transformer.named\_transformers\_["ohe"]

category\_labels = encoder.categories\_[0]

# Make a dataframe with the relevant columns

pd.DataFrame(transformed\_data, columns=np.append(category\_labels, "number"))



**Column Transfomer 2(becomes messy for many processing)**

#Creating tranformer

transfomer = ColumnTransformer([

    ('ohe',OneHotEncoder(),['Sex']),

    ('impute',SimpleImputer(strategy='mean'),['Age','Fare']),

    ('std',MinMaxScaler(),['Age','Fare'])

])

# Pipeline

pipe = Pipeline([

    ('pre-pro',transfomer),

    ('model',LogisticRegression())

])

# re assign X and y

X=orginal\_df.drop("Survived",axis=1)

y=orginal\_df["Survived"]

# train\_test\_split

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=20,random\_state=42)

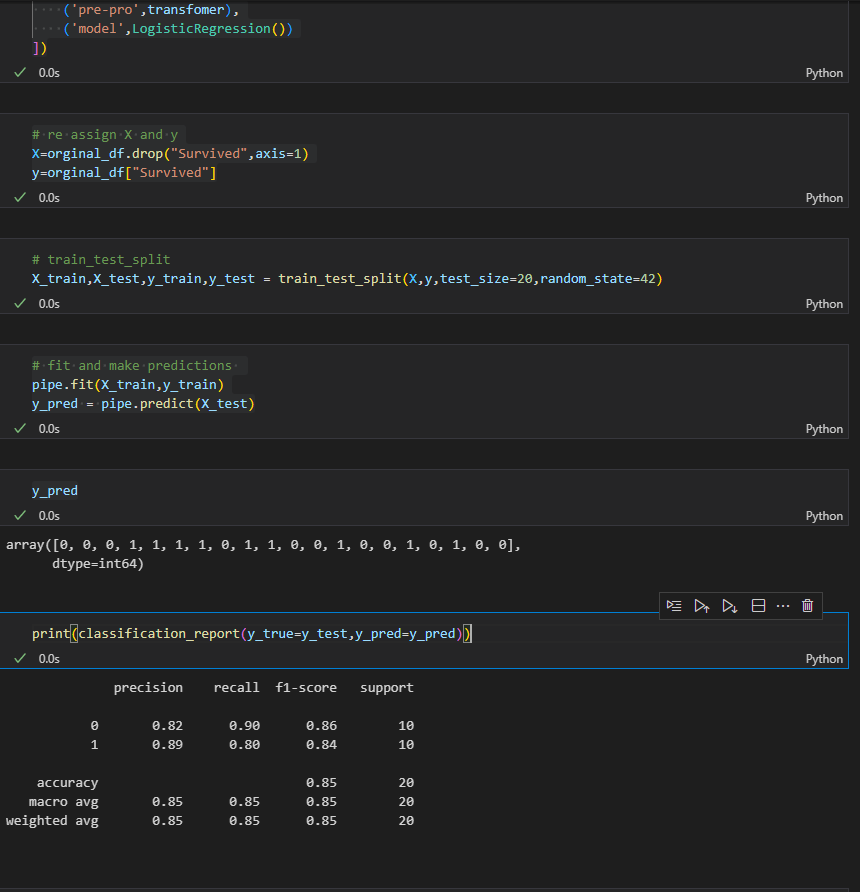
# fit and make predictions

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

y\_pred

print(classification\_report(y\_true=y\_test,y\_pred=y\_pred))



1. **Column Transfomer 3(best)**

from sklearn.compose import ColumnTransformer

# cat\_ord=['grades']

transformer3 = ColumnTransformer([

    ('ohe\_sex',Pipeline([

        ('impute\_mode',SimpleImputer(strategy='most\_frequent')),

        ('ohe',OneHotEncoder())

    ]),['Sex']),

    ('pre\_age',Pipeline([

        ('impute\_mean',SimpleImputer(strategy='mean')),

        ('scaler',MinMaxScaler())

    ]),['Age'])

    # ('ordinal',Pipeline([

    # ]),cat\_ord)

])

# Pipeline

pipe3 = Pipeline([

    ('pre-pro',transformer3),

    ('model',LogisticRegression())

])

# re assign X and y

X=orginal\_df.drop("Survived",axis=1)

y=orginal\_df["Survived"]

# train\_test\_split

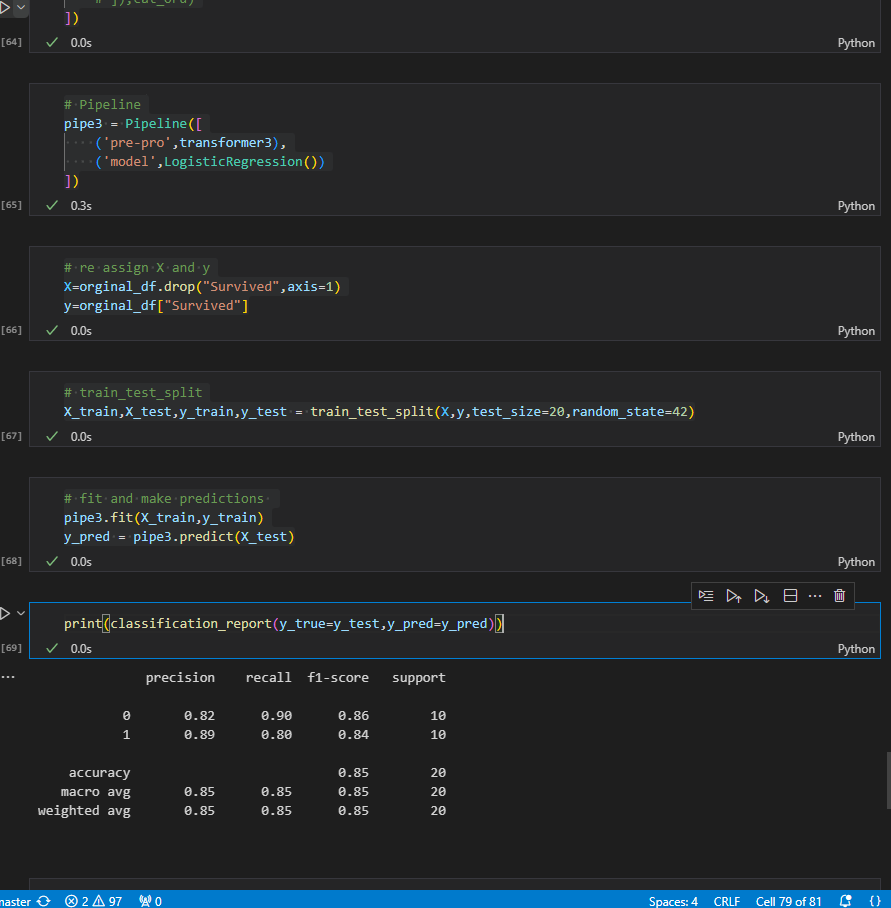
X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=20,random\_state=42)

# fit and make predictions

pipe3.fit(X\_train,y\_train)

y\_pred = pipe3.predict(X\_test)

print(classification\_report(y\_true=y\_test,y\_pred=y\_pred))



**SET PARAMS(SET\_PARAMS)- Change from Logistic Regression to Random Forest**

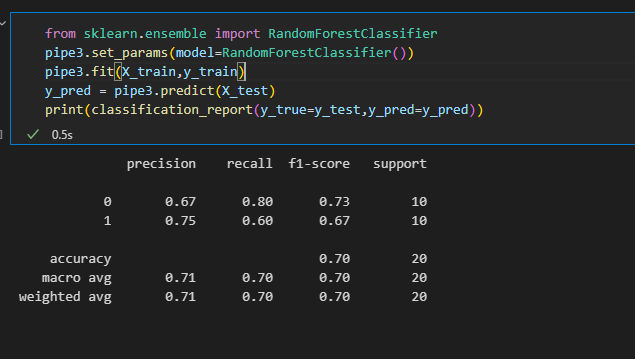
from sklearn.ensemble import RandomForestClassifier

pipe3.set\_params(model=RandomForestClassifier())

pipe3.fit(X\_train,y\_train)

y\_pred = pipe3.predict(X\_test)

print(classification\_report(y\_true=y\_test,y\_pred=y\_pred))



1. **Feature Union**

# Create a ColumnTransformer for feature engineering

feature\_eng = ColumnTransformer(transformers=[

    ("add\_number\_odd", FunctionTransformer(is\_odd), ["number"])

], remainder="drop")

# Create a ColumnTransformer to encode categorical data

# and keep numeric data as-is

original\_features\_encoded = ColumnTransformer(transformers=[

    ("ohe", OneHotEncoder(categories="auto", handle\_unknown="ignore"), ["category"])

], remainder="passthrough")

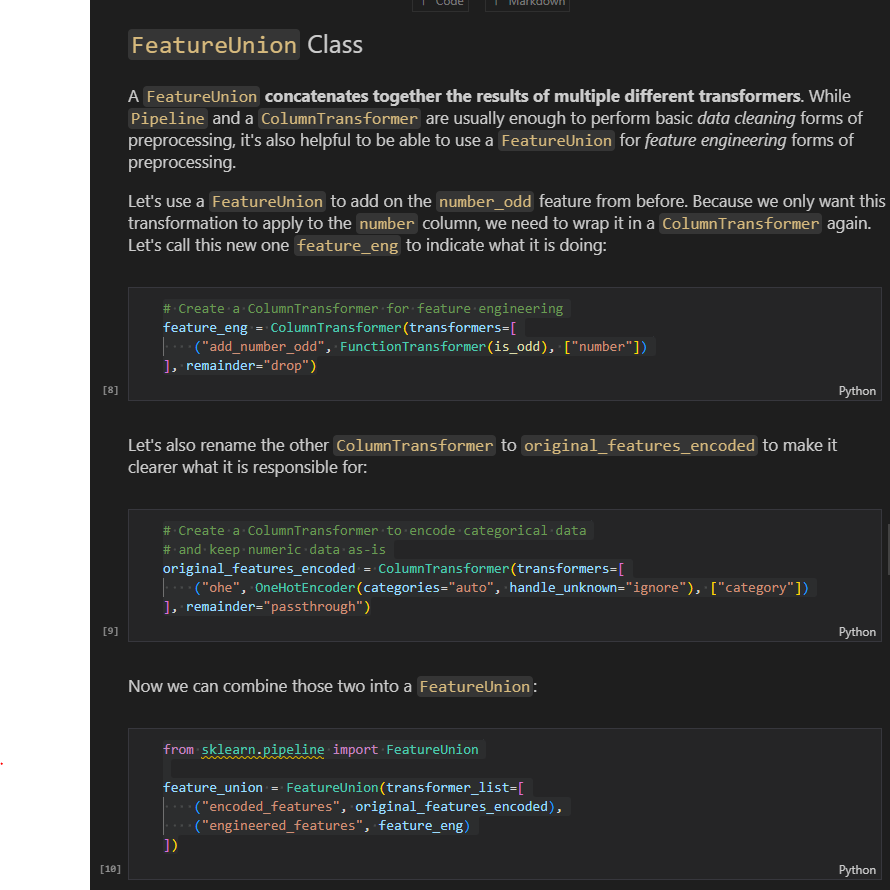
from sklearn.pipeline import FeatureUnion

feature\_union = FeatureUnion(transformer\_list=[

    ("encoded\_features", original\_features\_encoded),

    ("engineered\_features", feature\_eng)

])



**.**

# Create a pipeline containing union of encoded

# original features and engineered features

pipe = Pipeline(steps=[

    ("feature\_union", feature\_union)

])

# Use the pipeline to fit and transform the data

transformed\_data = pipe.fit\_transform(example\_X)

transformed\_data

# Extract the category labels from the OHE within the pipeline

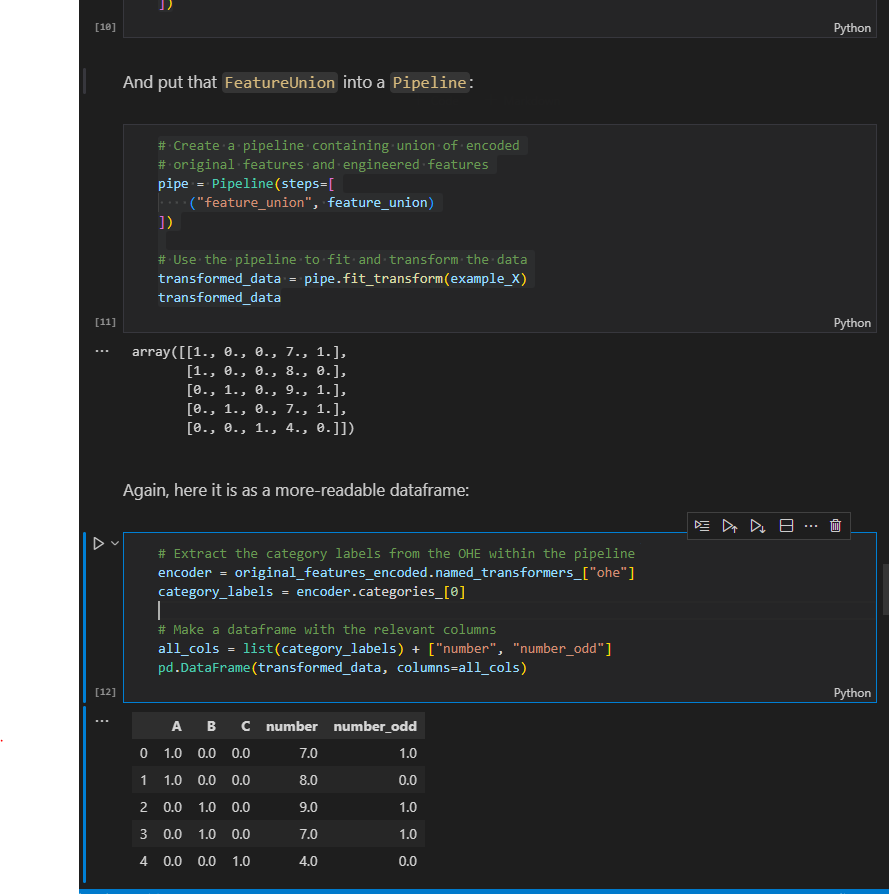
encoder = original\_features\_encoded.named\_transformers\_["ohe"]

category\_labels = encoder.categories\_[0]

# Make a dataframe with the relevant columns

all\_cols = list(category\_labels) + ["number", "number\_odd"]

pd.DataFrame(transformed\_data, columns=all\_cols)



**If we want to add an estimator (model) as the last step we can do like this**

from sklearn.linear\_model import LogisticRegression

# Create a pipeline containing union of encoded

# original features and engineered features, then

# all features scaled, then feed into a model

pipe = Pipeline(steps=[

    ("feature\_union", feature\_union),

    ("scale", StandardScaler()),

    ("model", LogisticRegression())

])

# Use the pipeline to fit the model and score it

pipe.fit(example\_X, example\_y)

pipe.score(example\_X, example\_y)

**COMPLETE REFACTORED PIPELINE EXAMPLE**

def preprocess\_data\_with\_pipeline(X):

    ### Encoding categorical data ###

    original\_features\_encoded = ColumnTransformer(transformers=[

        ("ohe", OneHotEncoder(categories="auto", handle\_unknown="ignore"), ["category"])

    ], remainder="passthrough")

    ### Feature engineering ###

    def is\_odd(data):

        """

        Helper function that returns 1 if odd, 0 if even

        """

        return data % 2

    feature\_eng = ColumnTransformer(transformers=[

        ("add\_number\_odd", FunctionTransformer(is\_odd), ["number"])

    ], remainder="drop")

    ### Combine encoded and engineered features ###

    feature\_union = FeatureUnion(transformer\_list=[

        ("encoded\_features", original\_features\_encoded),

        ("engineered\_features", feature\_eng)

    ])

    ### Pipeline (including scaling) ###

    pipe = Pipeline(steps=[

        ("feature\_union", feature\_union),

        ("scale", StandardScaler())

    ])

    transformed\_data = pipe.fit\_transform(X)

    ### Re-apply labels (optional step for readability) ###

    encoder = original\_features\_encoded.named\_transformers\_["ohe"]

    category\_labels = encoder.categories\_[0]

    all\_cols = list(category\_labels) + ["number", "number\_odd"]

    return pd.DataFrame(transformed\_data, columns=all\_cols, index=X.index), pipe

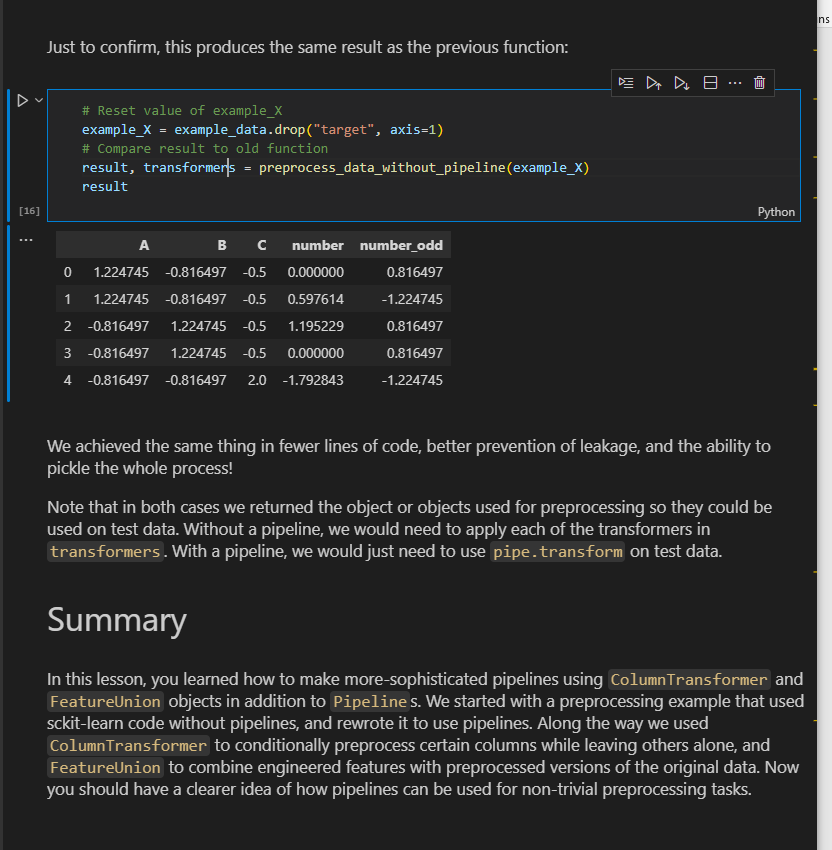
# Reset value of example\_X

example\_X = example\_data.drop("target", axis=1)

# Test out our new function

result, pipe = preprocess\_data\_with\_pipeline(example\_X)

result



1. **Using simple imputer and scaling**

# Instantiate our imputer

imputer =  SimpleImputer(strategy='median')

#Fit our imputer on the training data

imputer.fit(X\_train)

#create no-null versions of our train and test data

X\_train\_no\_nulls = imputer.transform(X\_train)

X\_test\_no\_nulls = imputer.transform(X\_test)

#instantiate our scaler

scaler = StandardScaler()

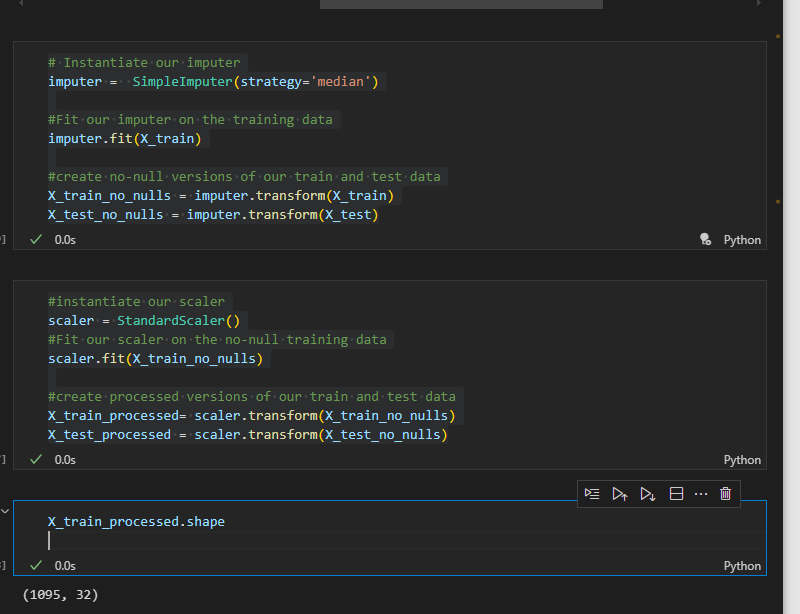
#Fit our scaler on the no-null training data

scaler.fit(X\_train\_no\_nulls)

#create processed versions of our train and test data

X\_train\_processed= scaler.transform(X\_train\_no\_nulls)

X\_test\_processed = scaler.transform(X\_test\_no\_nulls)



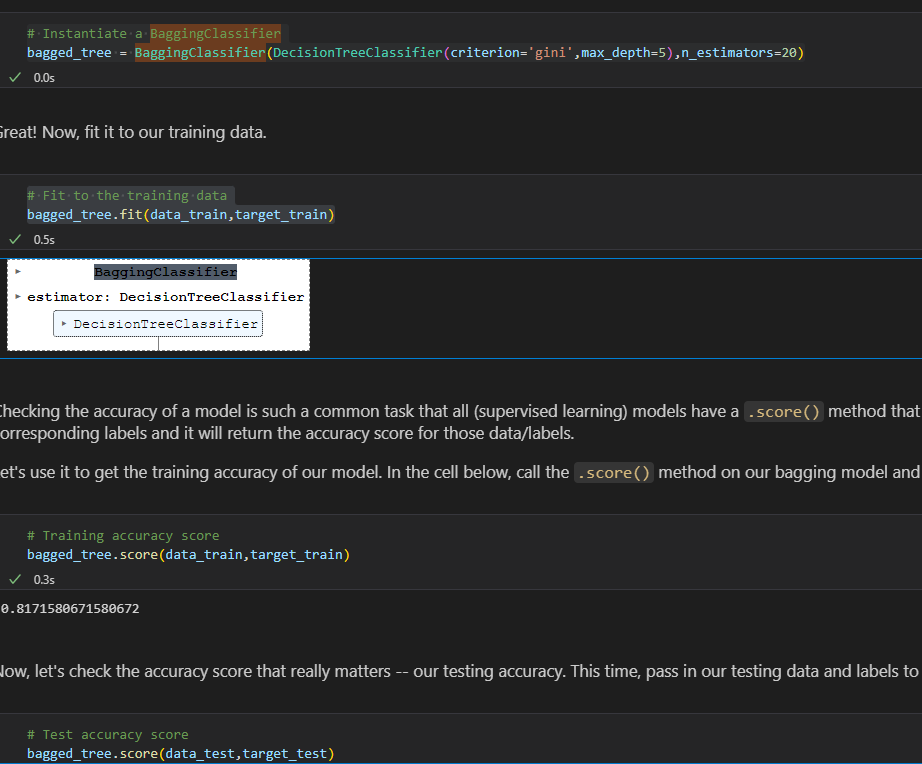
1. **Bagging**

# Instantiate a BaggingClassifier

bagged\_tree = BaggingClassifier(DecisionTreeClassifier(criterion='gini',max\_depth=5),n\_estimators=20)

# Fit to the training data

bagged\_tree.fit(data\_train,target\_train)



1. **Random Forest**

forest = RandomForestClassifier(n\_estimators=100,max\_depth=5)

forest.fit(data\_train,target\_train)

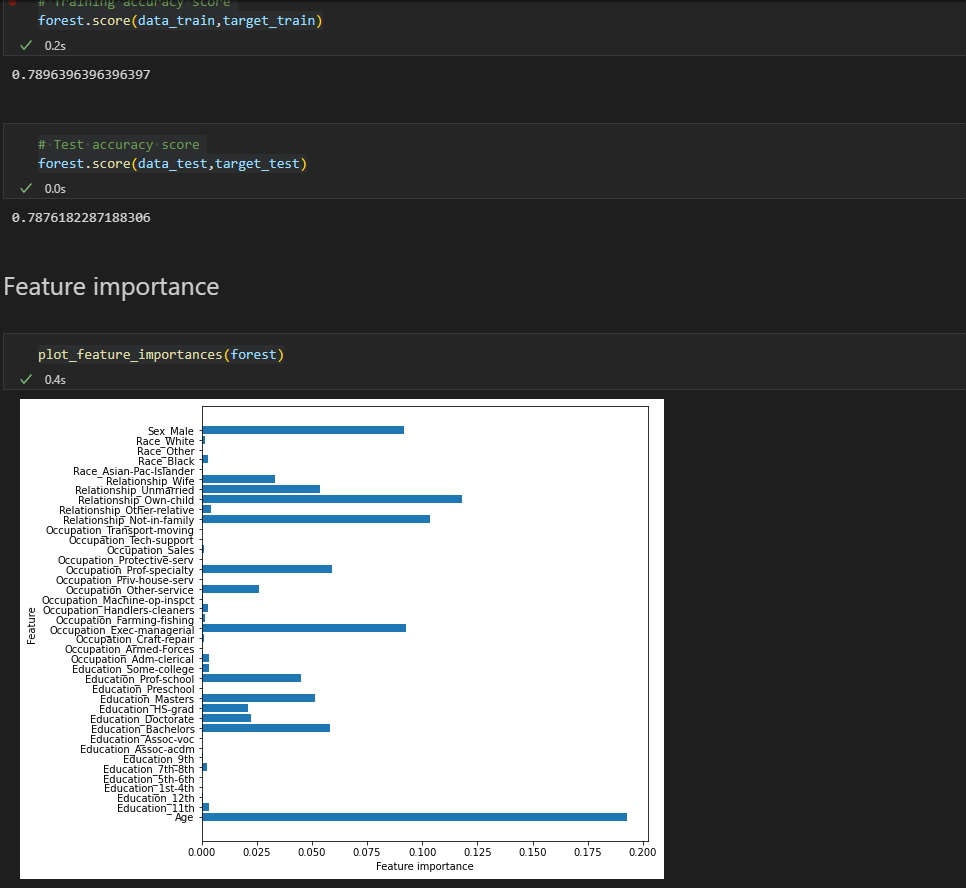
# Training accuracy score

forest.score(data\_train,target\_train)

# Test accuracy score

forest.score(data\_test,target\_test)

plot\_feature\_importances(forest)



**Trees in the forest**

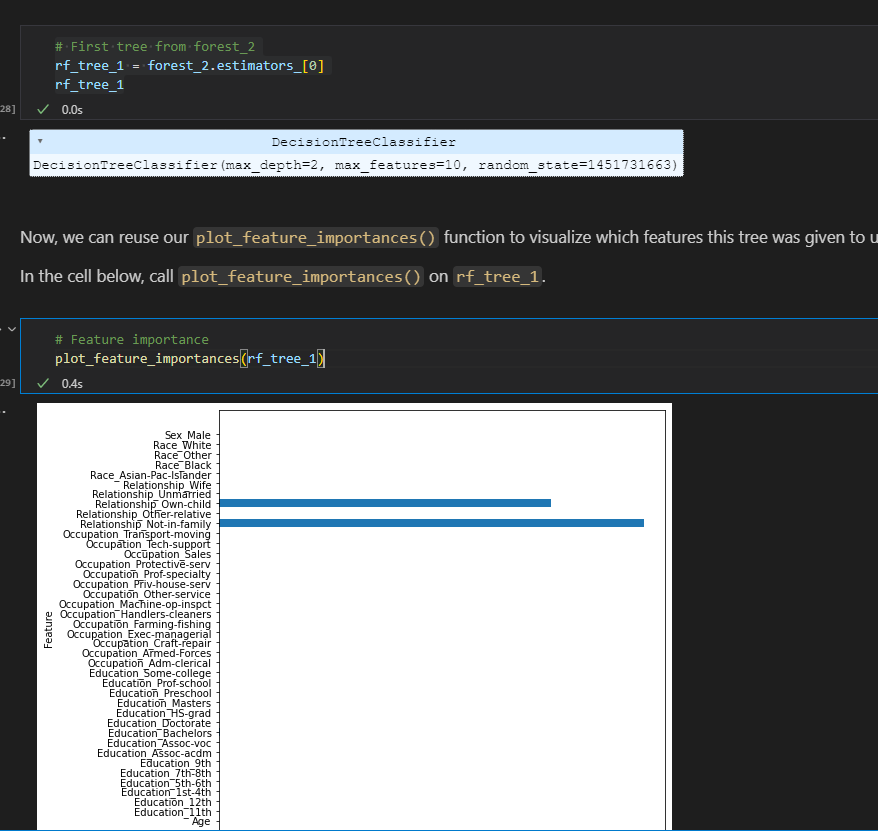
# First tree from forest\_2

rf\_tree\_1 = forest\_2.estimators\_[0]

rf\_tree\_1

# Feature importance

plot\_feature\_importances(rf\_tree\_1)



**Second Tree**

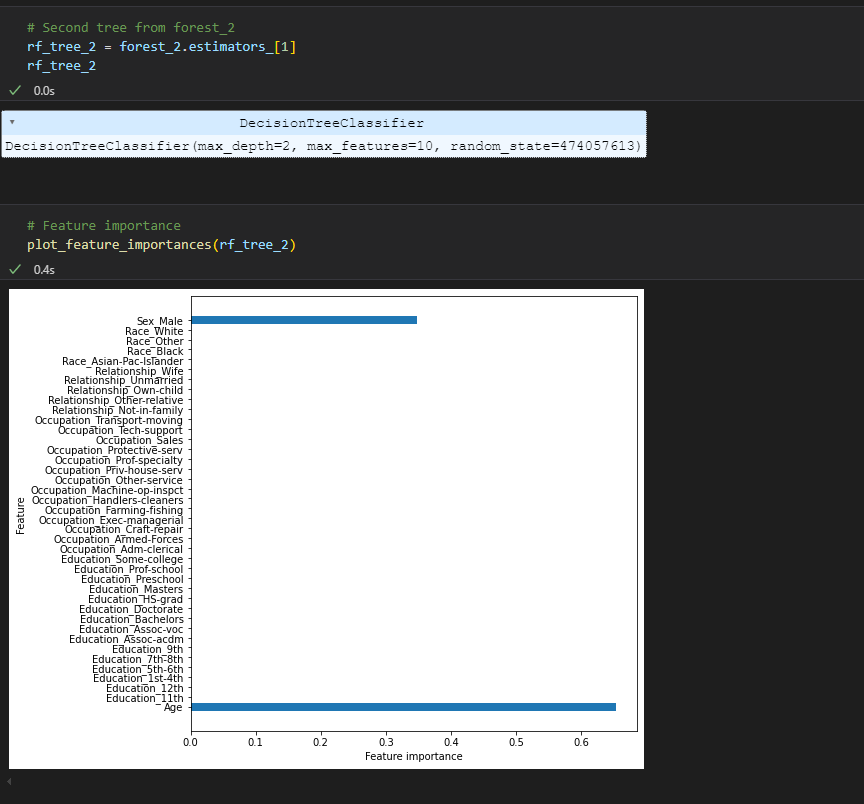
# Second tree from forest\_2

rf\_tree\_2 = forest\_2.estimators\_[1]

rf\_tree\_2

# Feature importance

plot\_feature\_importances(rf\_tree\_2)



1. **XG BOOST**

Requires that classification categories be integers that count up from 0, not starting at 3. Therefore you should instantiate a `LabelEncoder`and convert both `y\_train` and `y\_test` into arrays containing label encoded values (i.e. integers that count up from 0).

# Instantiate the encoder

encoder = LabelEncoder()

# Fit and transform the training data

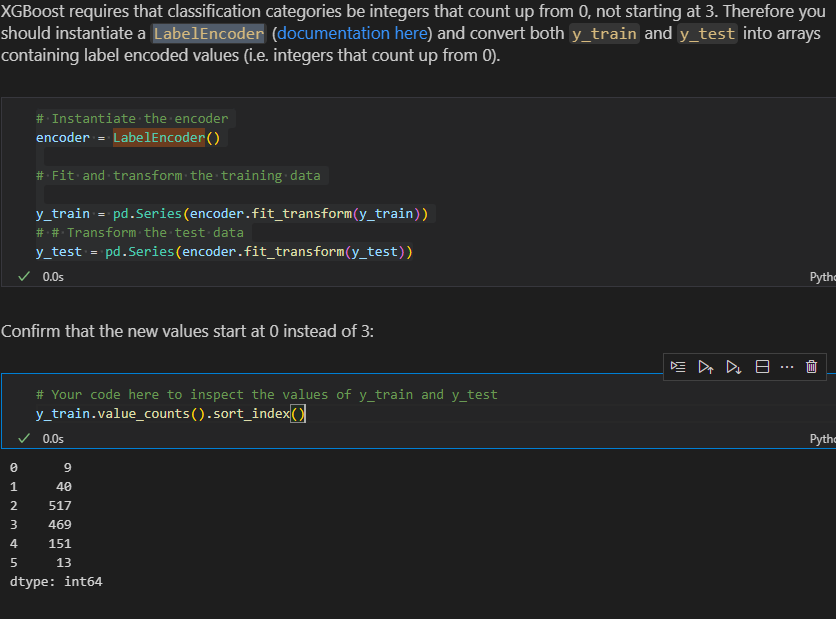
y\_train = pd.Series(encoder.fit\_transform(y\_train))

# # Transform the test data

y\_test = pd.Series(encoder.fit\_transform(y\_test))

# Your code here to inspect the values of y\_train and y\_test

y\_train.value\_counts().sort\_index()



**Build XGBOOST Model**

from xgboost import XGBClassifier

# Instantiate XGBClassifier

clf = XGBClassifier()

# Fit XGBClassifier

clf.fit(X\_train,y\_train)

# Predict on training and test sets

training\_preds = clf.predict(X\_train)

test\_preds = clf.predict(X\_test)

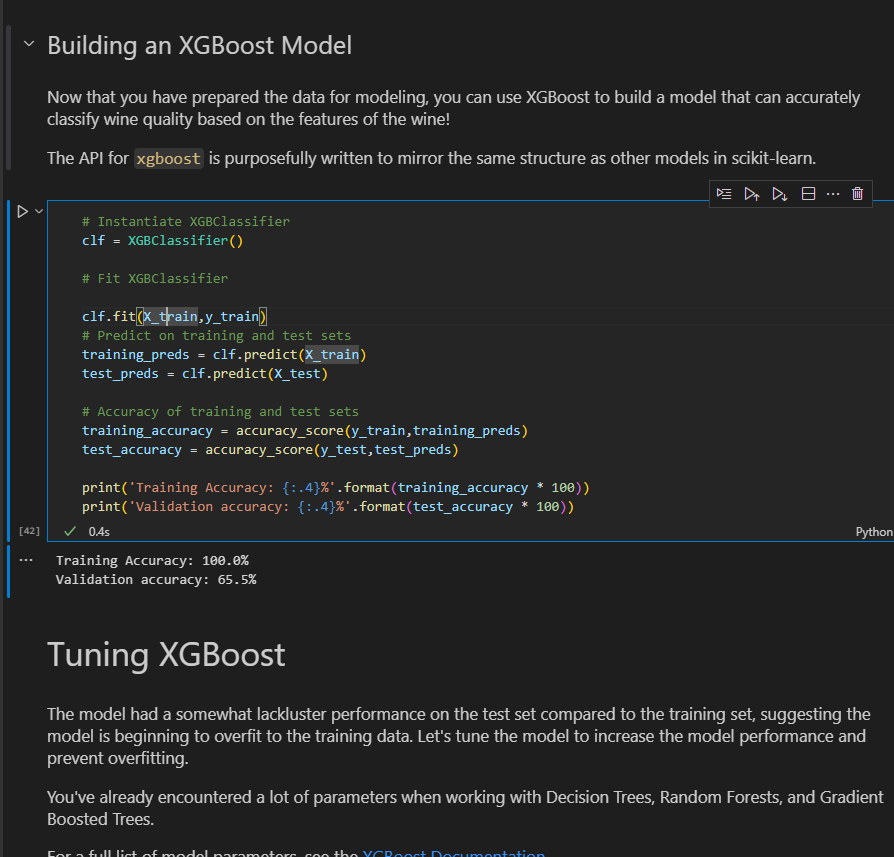
# Accuracy of training and test sets

training\_accuracy = accuracy\_score(y\_train,training\_preds)

test\_accuracy = accuracy\_score(y\_test,test\_preds)

print('Training Accuracy: {:.4}%'.format(training\_accuracy \* 100))

print('Validation accuracy: {:.4}%'.format(test\_accuracy \* 100))



**TUNING XGBOOST**

param\_grid = {

    'learning\_rate': [0.1, 0.2],

    'max\_depth': [6],

    'min\_child\_weight': [1, 2],

    'subsample': [0.5, 0.7],

    'n\_estimators': [100],

}

grid\_clf = GridSearchCV(clf,param\_grid,cv=None,n\_jobs=1)

grid\_clf.fit(X\_train, y\_train)

best\_parameters = grid\_clf.best\_params\_

print('Grid Search found the following optimal parameters: ')

for param\_name in sorted(best\_parameters.keys()):

    print('%s: %r' % (param\_name, best\_parameters[param\_name]))

training\_preds = grid\_clf.predict(X\_train)

test\_preds = grid\_clf.predict(X\_test)

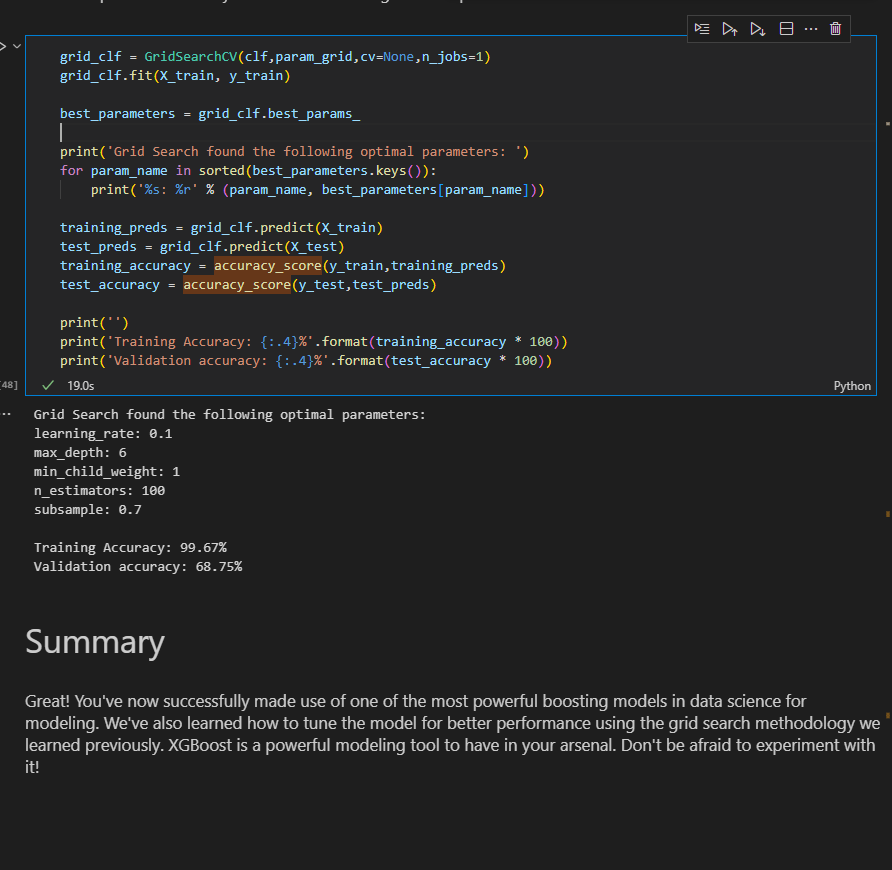
training\_accuracy = accuracy\_score(y\_train,training\_preds)

test\_accuracy = accuracy\_score(y\_test,test\_preds)

print('')

print('Training Accuracy: {:.4}%'.format(training\_accuracy \* 100))

print('Validation accuracy: {:.4}%'.format(test\_accuracy \* 100))



1. **PROJECT ON PIPELINES AND ENSEMBLE METHODS**

X = df.drop("Target",axis=1)

y = df["Target"]

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,train\_size=0.2,random\_state=42)

cat\_cols = ["Education","Occupation","Relationship","Race","Sex"]

num\_cols = ["Age"]

#create a tranformer

transfomer = ColumnTransformer([

    ('cat',Pipeline([

        ('imputer',SimpleImputer(strategy='most\_frequent')),

        ('ohe',OneHotEncoder())

    ]),cat\_cols),

    ('num',Pipeline([

        ('imputer',SimpleImputer(strategy='mean')),

        ('scaler',MinMaxScaler())

    ]),num\_cols)

])

# base estimator for ensemble methods

base= DecisionTreeClassifier(class\_weight='balanced')

# create a pipeline

pipe = Pipeline([

    ('pre\_pro',transfomer),

    ('model',DecisionTreeClassifier(class\_weight='balanced'))

])

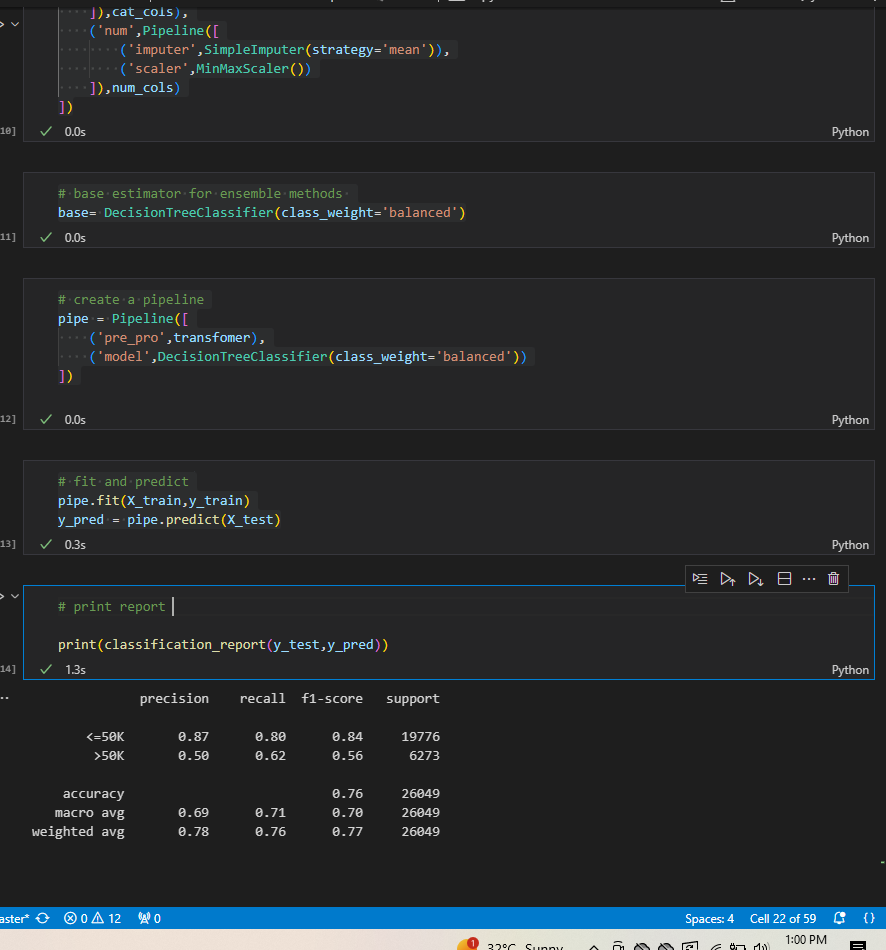
# fit and predict

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

# print report

print(classification\_report(y\_test,y\_pred))



**Change to Bagging model(Using set params)**

# set up a BaggingClassifier

pipe.set\_params(model=BaggingClassifier(base\_estimator=base))

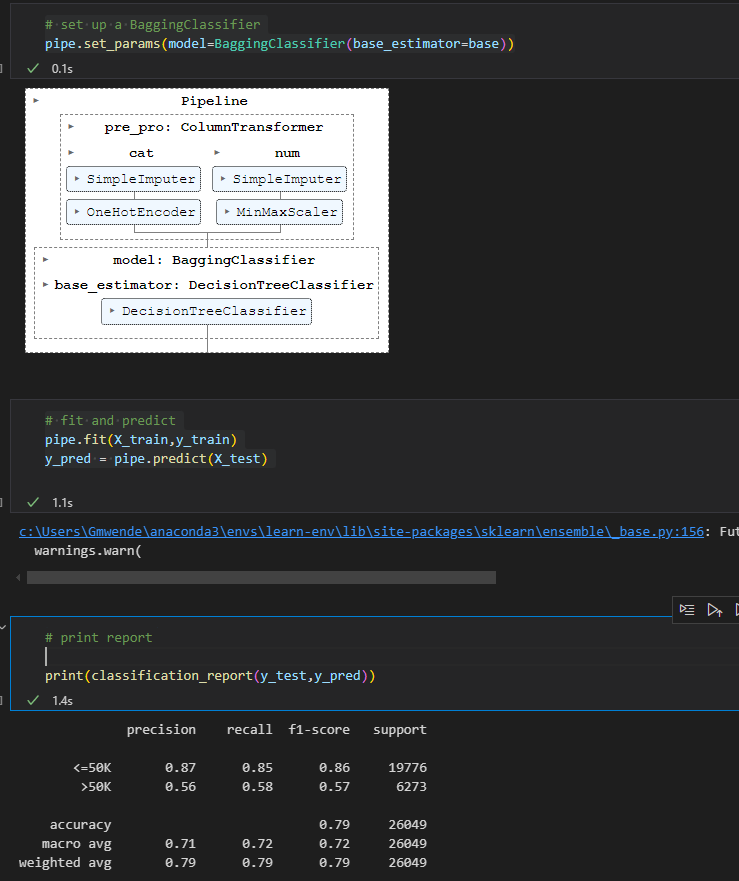
# fit and predict

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

# print report

print(classification\_report(y\_test,y\_pred))



**CHANGE TO RANDOM FOREST**

# set up Random Forest Classifier

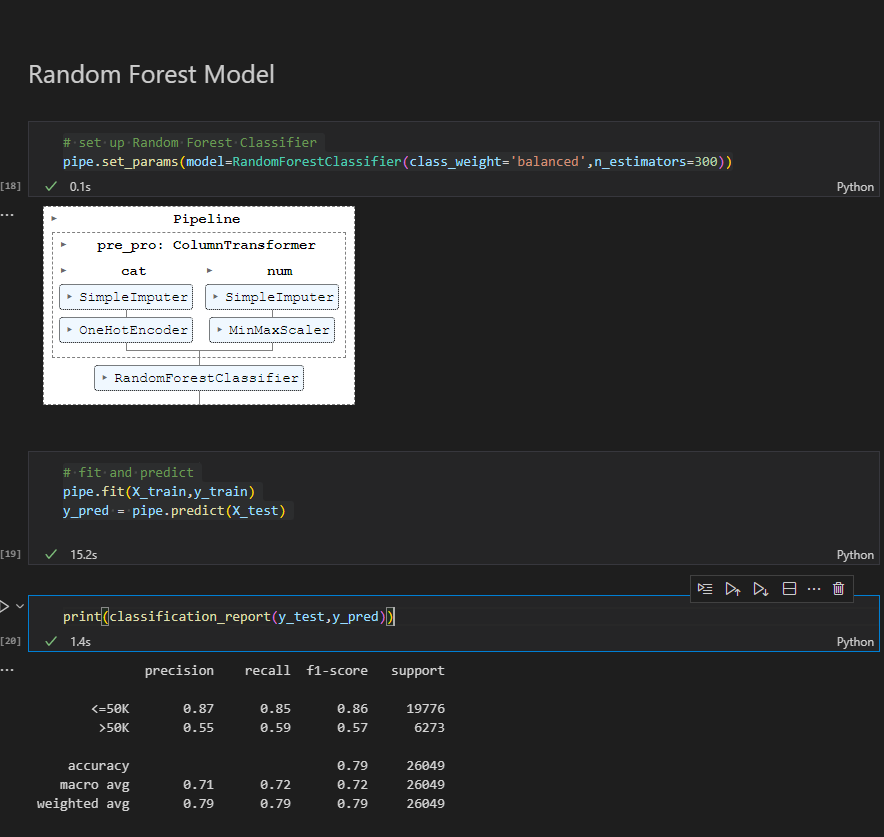
pipe.set\_params(model=RandomForestClassifier(class\_weight='balanced',n\_estimators=300))

# fit and predict

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

print(classification\_report(y\_test,y\_pred))



**CHANGE TO ADABOOST**

# set up adaboost classifier

# pipe.set\_params(model=AdaBoostClassifier(n\_estimators=base))

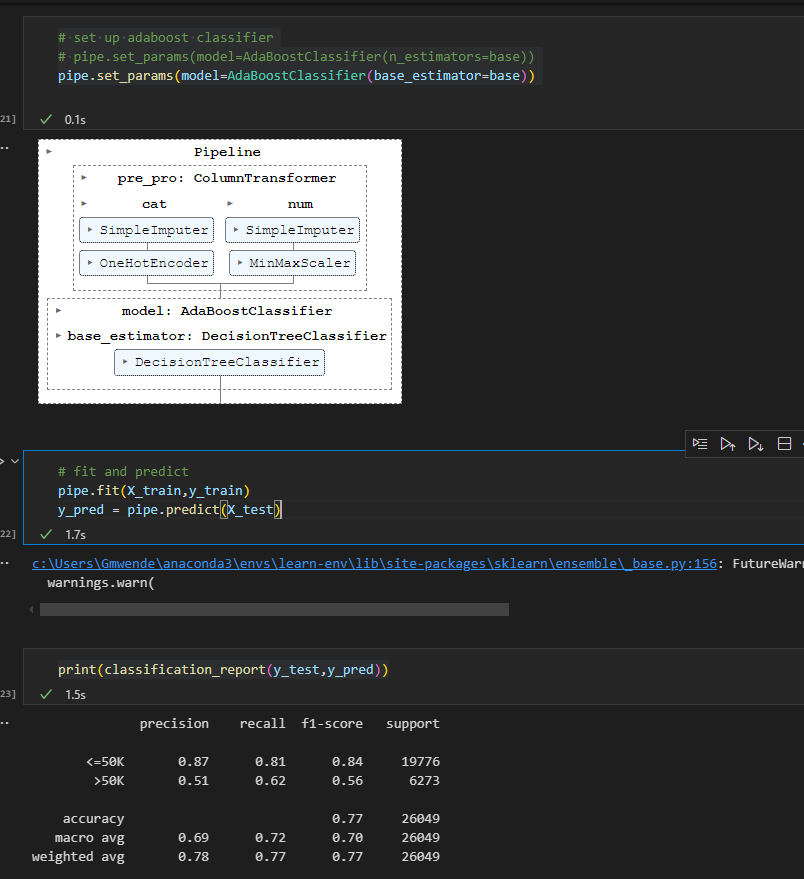
pipe.set\_params(model=AdaBoostClassifier(base\_estimator=base))

# fit and predict

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

print(classification\_report(y\_test,y\_pred))



**Change to Gradient Boosting**

# Gradient Bossting model

# set up adaboost classifier

pipe.set\_params(model=GradientBoostingClassifier())

# Gradient Bossting model

# set up adaboost classifier

pipe.set\_params(model=GradientBoostingClassifier())

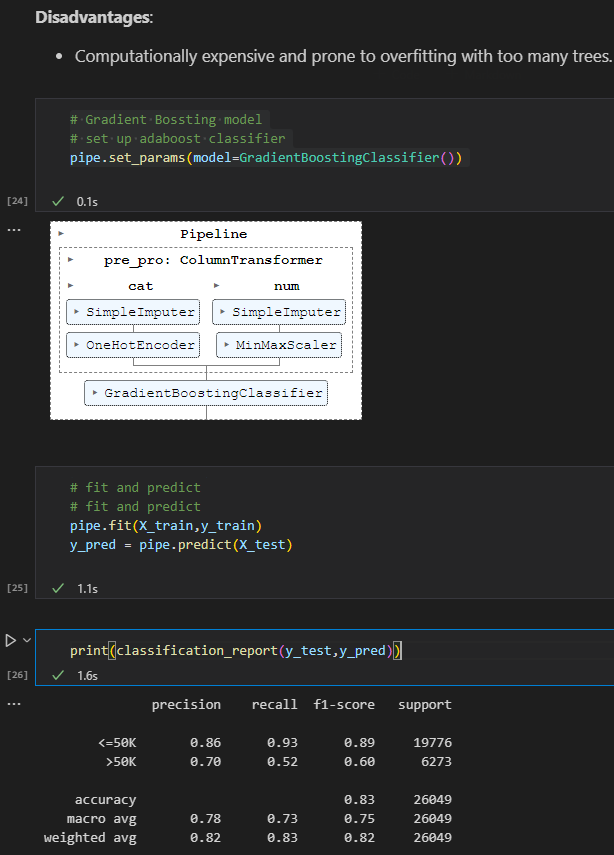
# fit and predict

# fit and predict

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

print(classification\_report(y\_test,y\_pred))



**Extreme Gradient Boosting**

# xg boost

from xgboost import XGBClassifier

# set up model

pipe.set\_params(model=XGBClassifier())

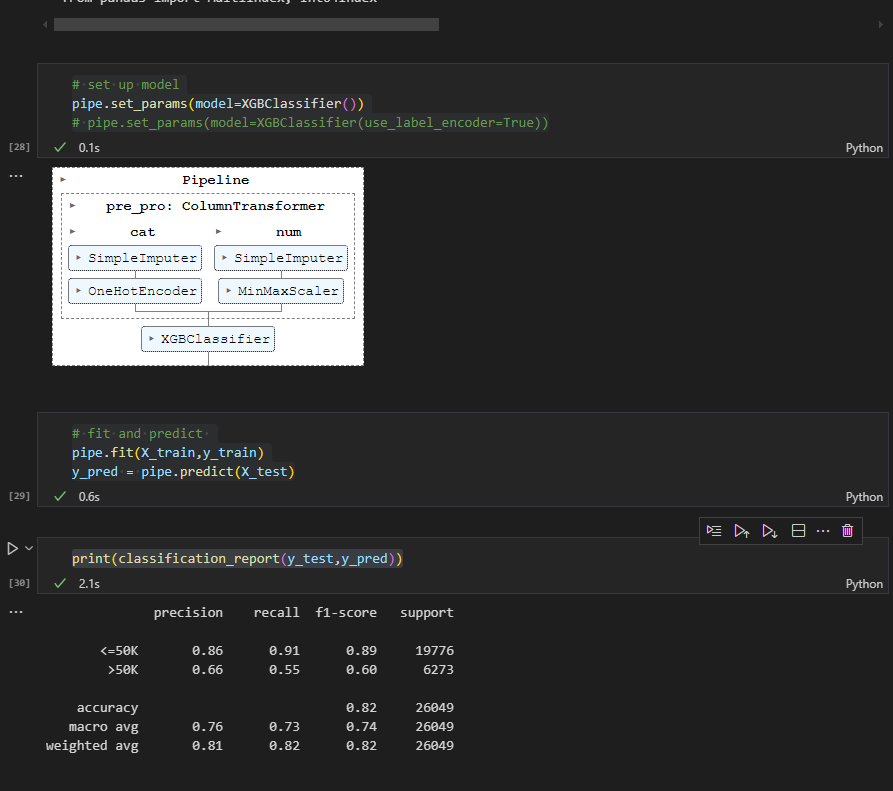
# pipe.set\_params(model=XGBClassifier(use\_label\_encoder=True))

# fit and predict

pipe.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

print(classification\_report(y\_test,y\_pred))



**TUNING USING GRID SEARCH CV**

param\_grid = {

    'model\_\_booster': ['gbtree', 'dart'],  # Type of boosting model ('gbtree' or 'dart')

    'model\_\_learning\_rate': [0.01, 0.1, 0.2],  # Step size in each boosting round

    'model\_\_n\_estimators': [50, 100, 200],  # Number of trees (boosting rounds)

    'model\_\_max\_depth': [3, 6, 10],  # Maximum depth of each tree

    'model\_\_min\_child\_weight': [1, 3, 5],  # Minimum sum of instance weight in a child node

    'model\_\_subsample': [0.7, 0.8, 1.0],  # Fraction of data to sample for each tree

    'model\_\_colsample\_bytree': [0.7, 0.8, 1.0],  # Fraction of features to sample for each tree

    'model\_\_gamma': [0, 0.1, 0.2]  # Minimum loss reduction required to make a split

}

pipe.set\_params(model = XGBClassifier() )

# perform grid search for the above model

grid = GridSearchCV(estimator=pipe,param\_grid=param\_grid)

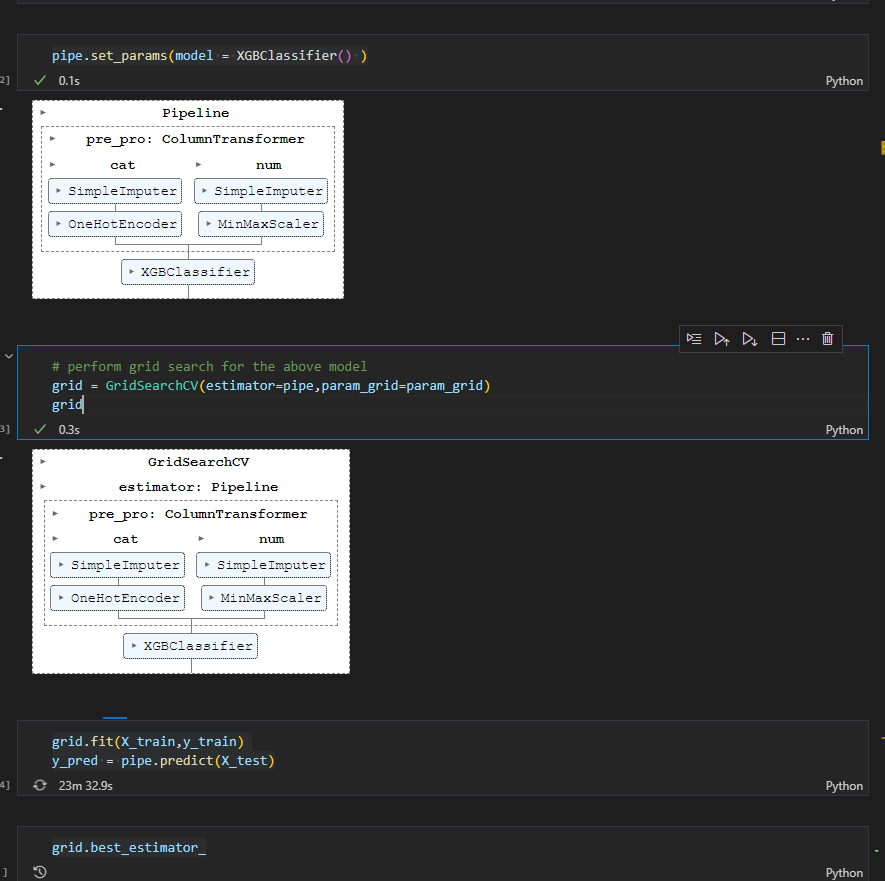
grid

grid.fit(X\_train,y\_train)

y\_pred = pipe.predict(X\_test)

grid.best\_estimator\_

print(classification\_report(y\_test,y\_pred))



1. **Access Jokes Dataset on surprise library**

Workaround is to download and unzip http://eigentaste.berkeley.edu/dataset/archive/jester\_dataset\_2.zip and save the .dat files to ~/.surprise\_data/jester/

i.e in ur working directory create folder surprise\_data then inside surprise\_data create folder jester and paste the unzipped files(.dat files) to jester folder and rerun the code

1. .fit not working in kmeans clustering( **AttributeError: 'NoneType' object has no attribute 'split'**)

for below code

k\_means = KMeans(n\_clusters=6)

k\_means.fit(X)

predicted\_clusters = k\_means.predict(X)

Had to install threadpoolctl by running below code and then restart the kernel

!pip install threadpoolctl==3.1.0

1. **PCA**

# Import PCA

from sklearn.decomposition import PCA

# Instantiate PCA

pca = PCA(n\_components=2)

# Fit PCA

principalComponents = pca.fit\_transform(X)

# Create a new dataset from principal components

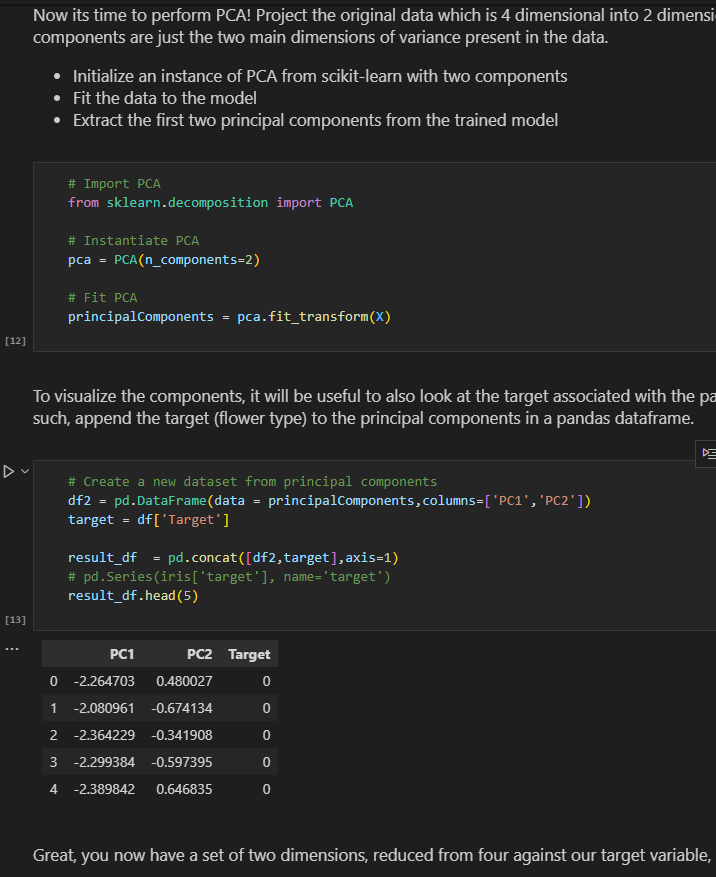
df2 = pd.DataFrame(data = principalComponents,columns=['PC1','PC2'])

target = df['Target']

result\_df  = pd.concat([df2,target],axis=1)

# pd.Series(iris['target'], name='target')

result\_df.head(5)



**Visualizing Principal Components** -Using the target data we can visualize the principal components according to the class distribution

# Principal Components scatter plot

# Your code here

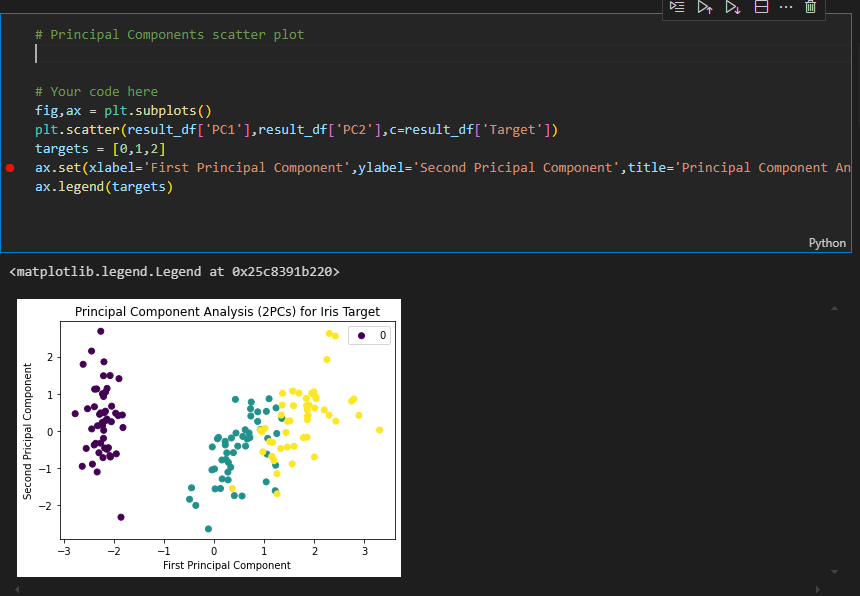
fig,ax = plt.subplots()

plt.scatter(result\_df['PC1'],result\_df['PC2'],c=result\_df['Target'])

targets = [0,1,2]

ax.set(xlabel='First Principal Component',ylabel='Second Pricipal Component',title='Principal Component Analysis (2PCs) for Iris Target' )

ax.legend(targets)



**Another Method**

#Teachers code

targets = [0,1,2]

colors = ['r','g','b']

fig, ax = plt.subplots()

for target, color in zip(targets, colors):

    indicesToKeep = iris['target'] == target

    ax.scatter(result\_df.loc[indicesToKeep,'PC1'],

               result\_df.loc[indicesToKeep,'PC2'],

               c=color,

               s=50

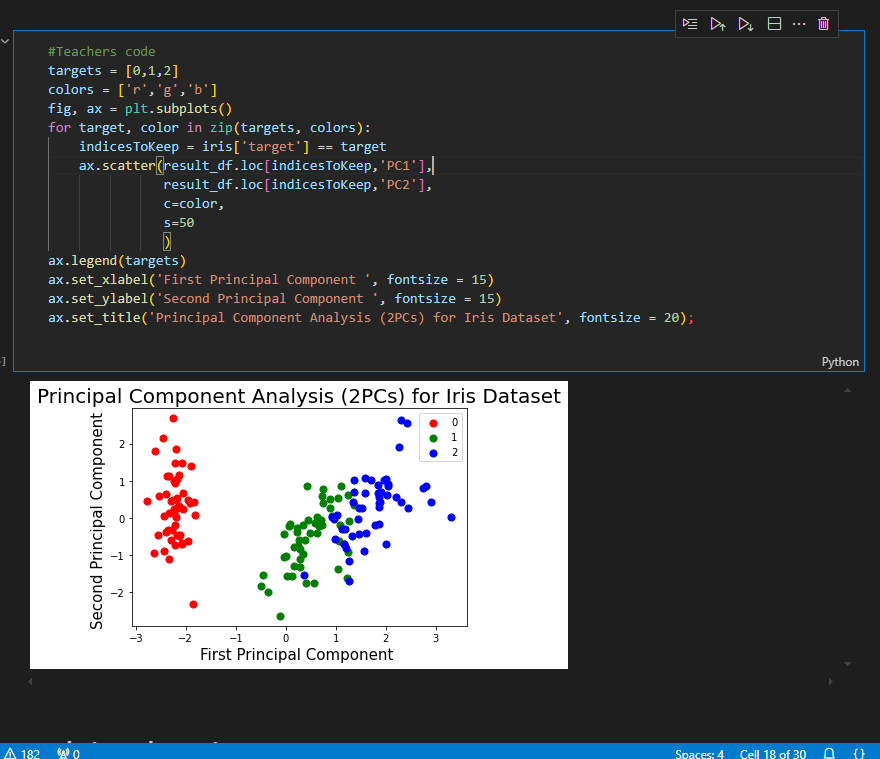
               )

ax.legend(targets)

ax.set\_xlabel('First Principal Component ', fontsize = 15)

ax.set\_ylabel('Second Principal Component ', fontsize = 15)

ax.set\_title('Principal Component Analysis (2PCs) for Iris Dataset', fontsize = 20);



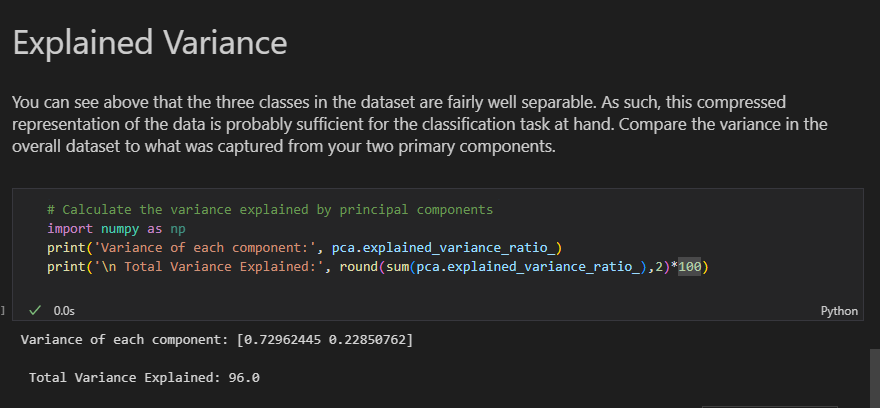
**Variance in PCA**

# Calculate the variance explained by principal components

import numpy as np

print('Variance of each component:', pca.explained\_variance\_ratio\_)

print('\n Total Variance Explained:', round(sum(pca.explained\_variance\_ratio\_),2))



**Run Classifier on PCA Data**

X = result\_df[['PC1', 'PC2']]

y = result\_df['Target']

start = time.time()

X\_train, X\_test, Y\_train, Y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=9)

model = KNeighborsClassifier()

model.fit(X\_train.values, Y\_train.values)

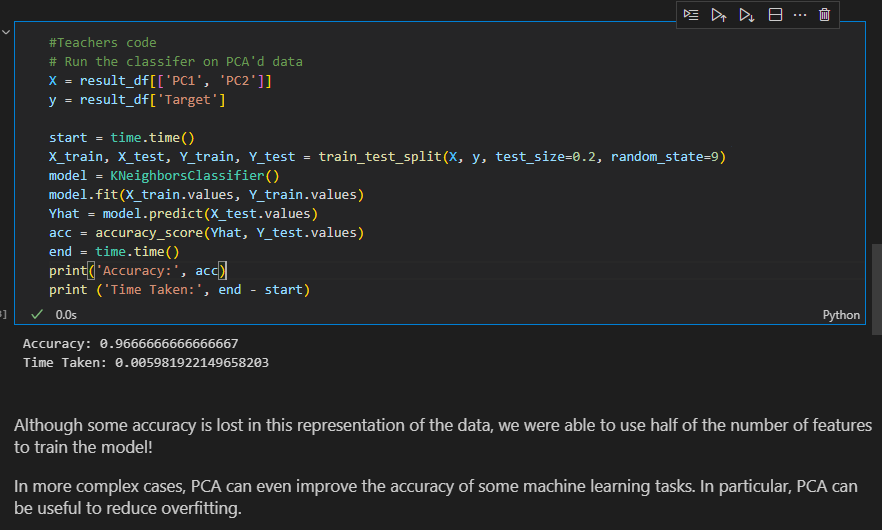
Yhat = model.predict(X\_test.values)

acc = accuracy\_score(Yhat, Y\_test.values)

end = time.time()

print('Accuracy:', acc)

print ('Time Taken:', end - start)



**Integrating PCA with Pipelines**

**a)BaseModel**

y = data['target']

X = data.loc[:,'feat\_1':'feat\_93']

# Your code here

# Load and split the data

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.4,random\_state=42)

# Your code here

# Construct some pipelines

pipe = Pipeline([

    ('pca',PCA(n\_components=27,random\_state=123)),  # PCA step

     ('model',LogisticRegression(random\_state=123)) # Logistic Regression step

])

# Your code here

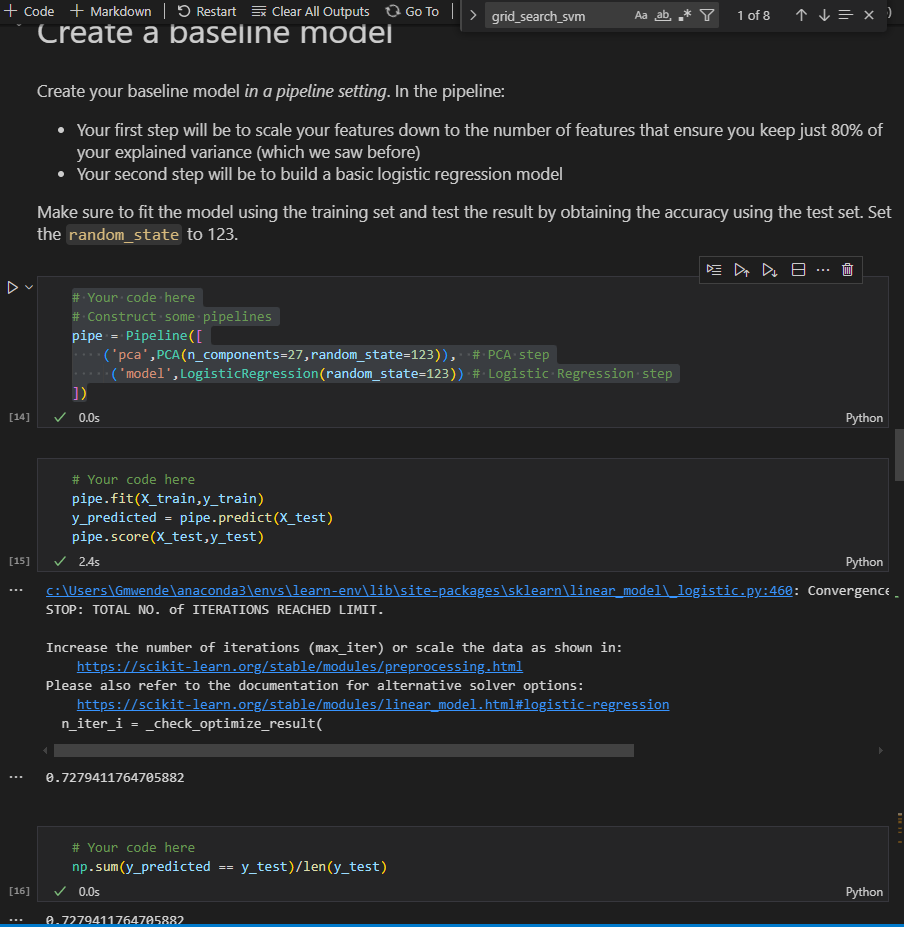
pipe.fit(X\_train,y\_train)

y\_predicted = pipe.predict(X\_test)

pipe.score(X\_test,y\_test)

# Your code here

np.sum(y\_predicted == y\_test)/len(y\_test)



**b) Create a pipeline consisting of a linear SVM, a simple decision tree, and a simple random forest classifier**

# Your code here

# ⏰ This cell may take several minutes to run

pipe\_decision= Pipeline([

    ('pca',PCA(n\_components=27,random\_state=123)),  # PCA step

     ('model\_decision',DecisionTreeClassifier(random\_state=123)) # Decision Tree step

])

pipe\_rf = Pipeline([

    ('pca',PCA(n\_components=27,random\_state=123)),  # PCA step

     ('model\_random',RandomForestClassifier(random\_state=123)) # Random Forest step

])

pipe\_svm = Pipeline([

    ('pca',PCA(n\_components=27,random\_state=123)),  # PCA step

     ('model\_svm',svm.SVC(random\_state=123)) # SVM step

])

# Loop to fit each of the three pipelines

pipelines = [pipe\_decision,pipe\_rf,pipe\_svm]

pipeline\_name=['Decision Tree','Random Forest','Support Vector Machine']

for pipe in pipelines:

    print(pipe)

    pipe.fit(X\_train,y\_train)

    # scores = pipe.score(X\_test,y\_test)

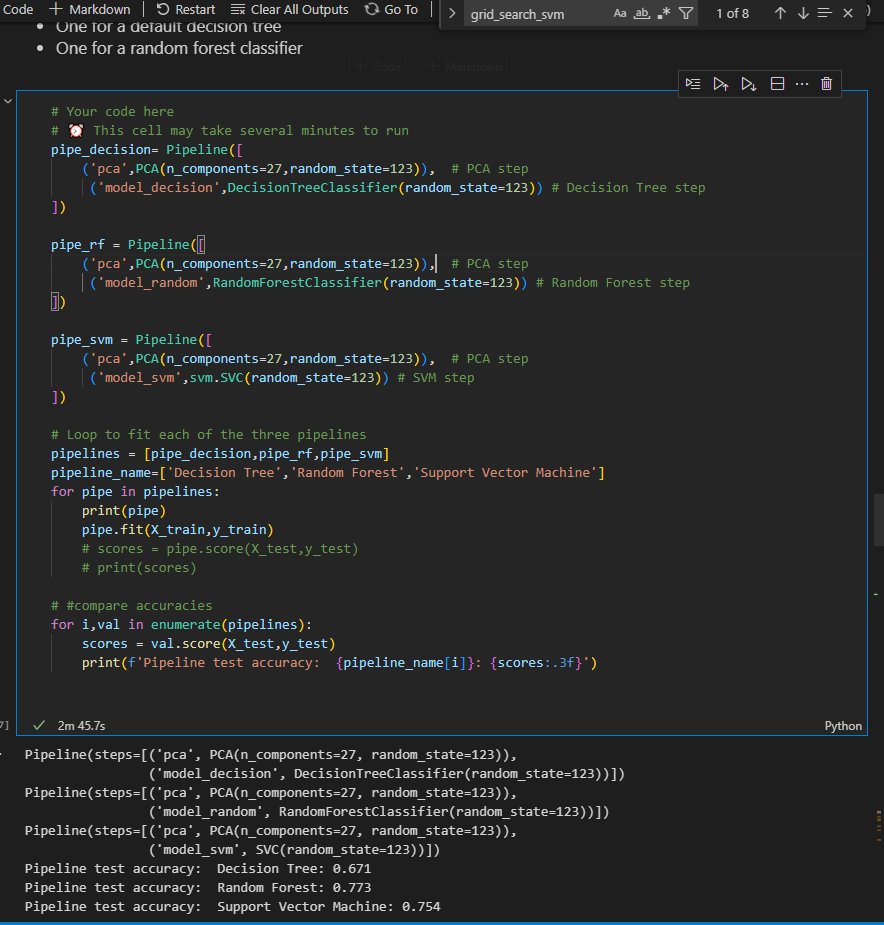
    # print(scores)

# #compare accuracies

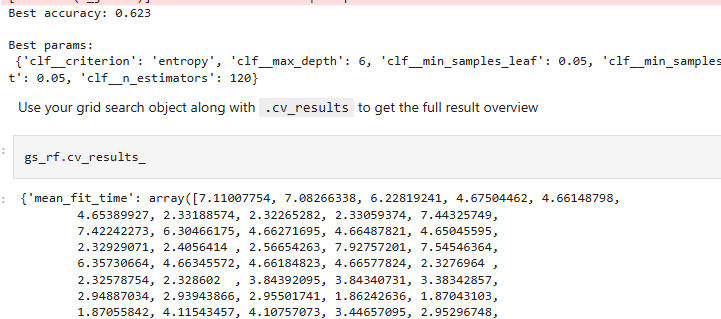
for i,val in enumerate(pipelines):

    scores = val.score(X\_test,y\_test)

    print(f'Pipeline test accuracy:  {pipeline\_name[i]}: {scores:.3f}')



1. **Pipeline with Grid Search**
2. # Your code here
3. # imports
4. pipe\_rf = Pipeline([
5. ('pca',PCA(n\_components=27,random\_state=123)),  # PCA step
6. ('model\_random',RandomForestClassifier(random\_state=123)) # Random Forest step
7. ])
8. param\_grid = [
9. {'model\_random\_\_n\_estimators': [120],
10. 'model\_random\_\_criterion': ['entropy', 'gini'],
11. 'model\_random\_\_max\_depth': [4, 5, 6],
12. 'model\_random\_\_min\_samples\_leaf':[0.05 ,0.1, 0.2],
13. 'model\_random\_\_min\_samples\_split':[0.05 ,0.1, 0.2]
14. }
15. ]
16. grid\_search\_rf = GridSearchCV(estimator=pipe\_rf,param\_grid=param\_grid,scoring='accuracy',
17. cv=3, verbose=2, return\_train\_score = True)
18. grid\_search\_rf.fit(X\_train,y\_train)
19. grid\_search\_rf.score(X\_test,y\_test)
20. # Your code here
21. # Best accuracy
22. print('Best accuracy: %.3f' % grid\_search\_rf.best\_score\_)
23. # Best params
24. print('\nBest params:\n', grid\_search\_rf.best\_params\_)
25. grid\_search\_rf.cv\_results\_



1. **Market Segmentation/Clustering**

channels = raw\_df['Channel']

# df = raw\_df.drop(columns=['Channel','Region'])

df = raw\_df.drop(['Channel','Region'],axis=1)

df.head()

**Clustering Analysis Using Kmeans**

from sklearn.cluster import KMeans

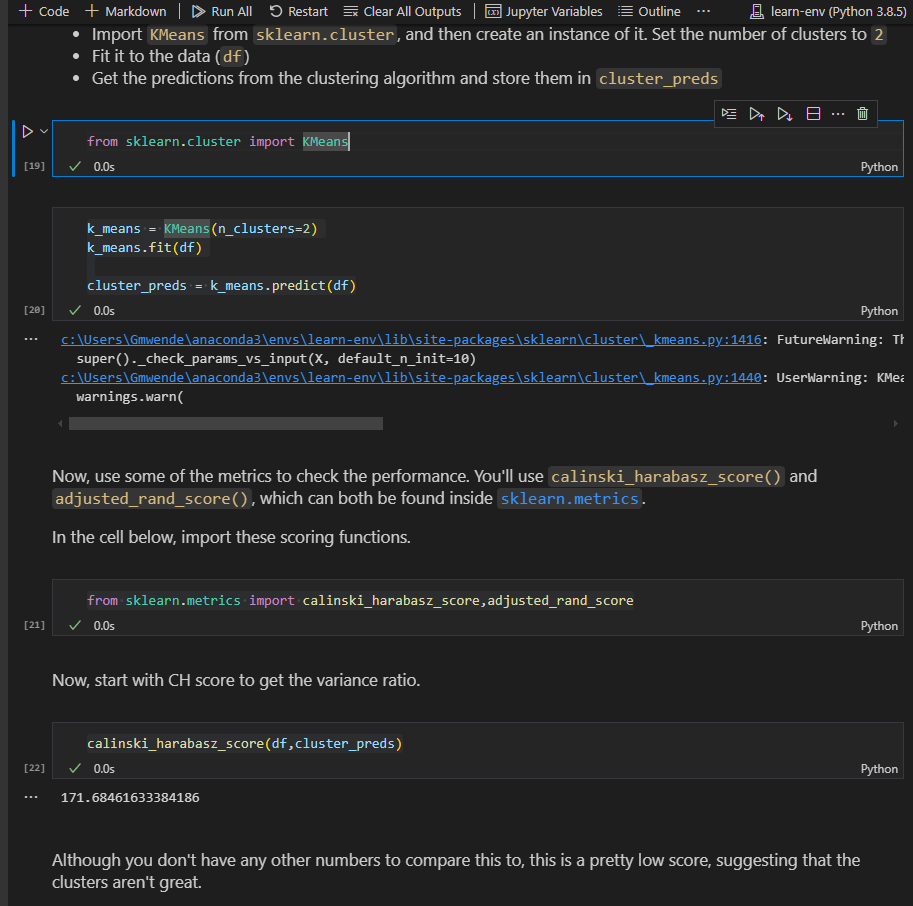
k\_means = KMeans(n\_clusters=2)

k\_means.fit(df)

cluster\_preds = k\_means.predict(df)

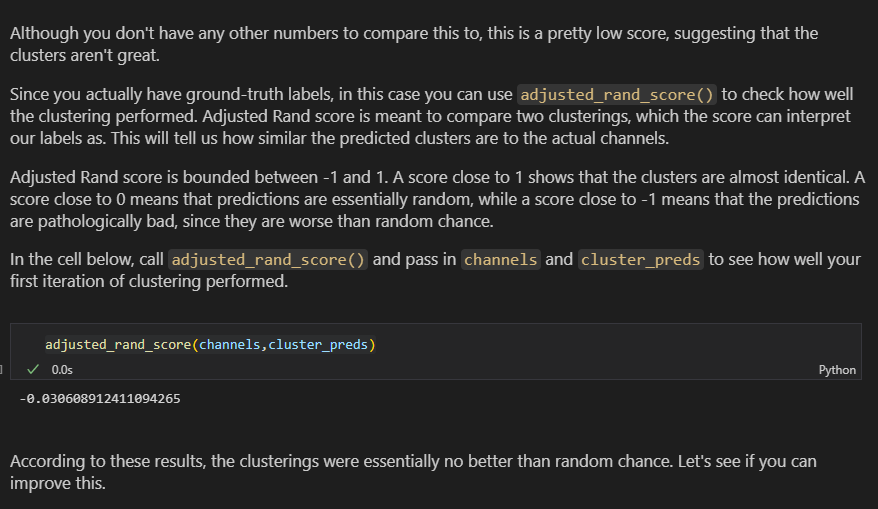
from sklearn.metrics import calinski\_harabasz\_score,adjusted\_rand\_score

calinski\_harabasz\_score(df,cluster\_preds)



**adjusted\_rand\_score**

adjusted\_rand\_score(channels,cluster\_preds)



**Kmeans After Scaling**

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

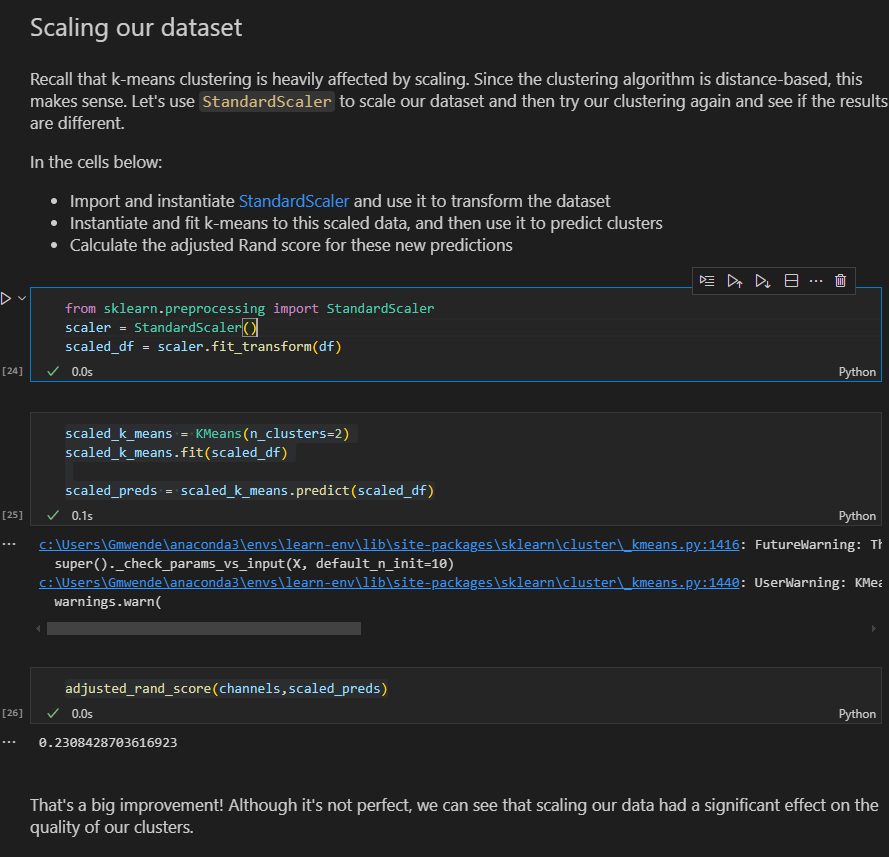
scaled\_df = scaler.fit\_transform(df)

scaled\_k\_means = KMeans(n\_clusters=2)

scaled\_k\_means.fit(scaled\_df)

scaled\_preds = scaled\_k\_means.predict(scaled\_df)

adjusted\_rand\_score(channels,scaled\_preds)



**Incorporating pca**

from sklearn.decomposition import PCA

pca=  PCA(n\_components=4)

pca\_df = pca.fit\_transform(scaled\_df)

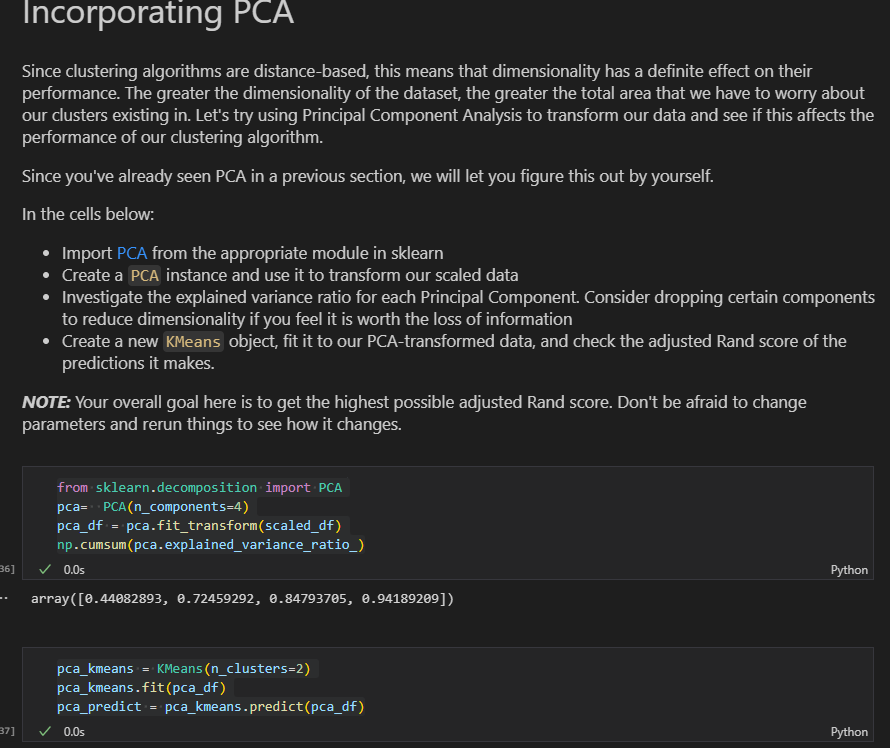
np.cumsum(pca.explained\_variance\_ratio\_)

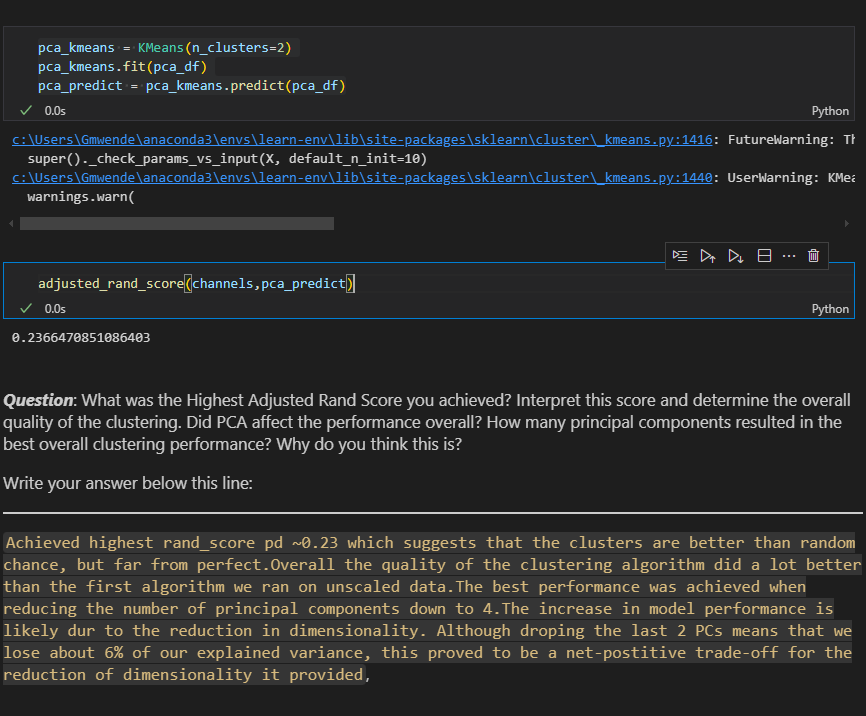
pca\_kmeans = KMeans(n\_clusters=2)

pca\_kmeans.fit(pca\_df)

pca\_predict = pca\_kmeans.predict(pca\_df)

adjusted\_rand\_score(channels,pca\_predict)





1. **Hierarchical Agglomerative Clustering**

from sklearn.cluster import AgglomerativeClustering

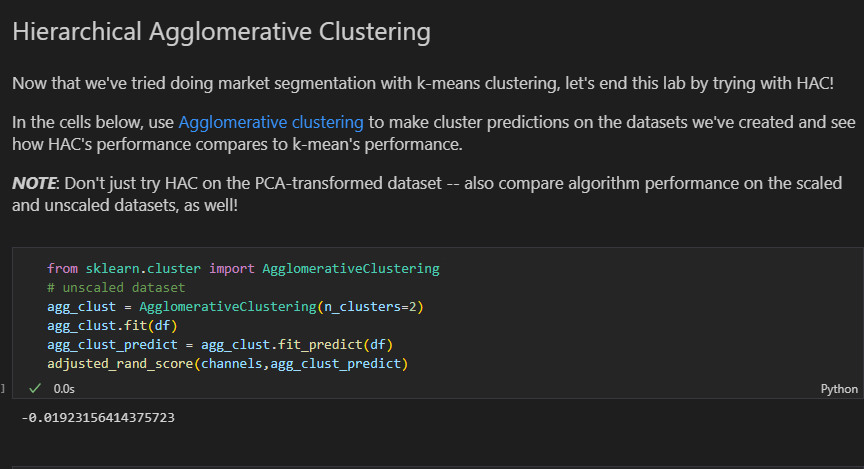
# unscaled dataset

agg\_clust = AgglomerativeClustering(n\_clusters=2)

agg\_clust.fit(df)

agg\_clust\_predict = agg\_clust.fit\_predict(df)

adjusted\_rand\_score(channels,agg\_clust\_predict)



**#Scaled dataset and after pca**

#scaled dataset

agg\_clust\_sc = AgglomerativeClustering()

agg\_clust\_sc.fit(scaled\_df)

agg\_clust\_sc\_predict = agg\_clust\_sc.fit\_predict(scaled\_df)

adjusted\_rand\_score(channels,agg\_clust\_sc\_predict)

#after pca

pca = PCA(n\_components=4)

pc\_df\_agg = pca.fit\_transform(scaled\_df)

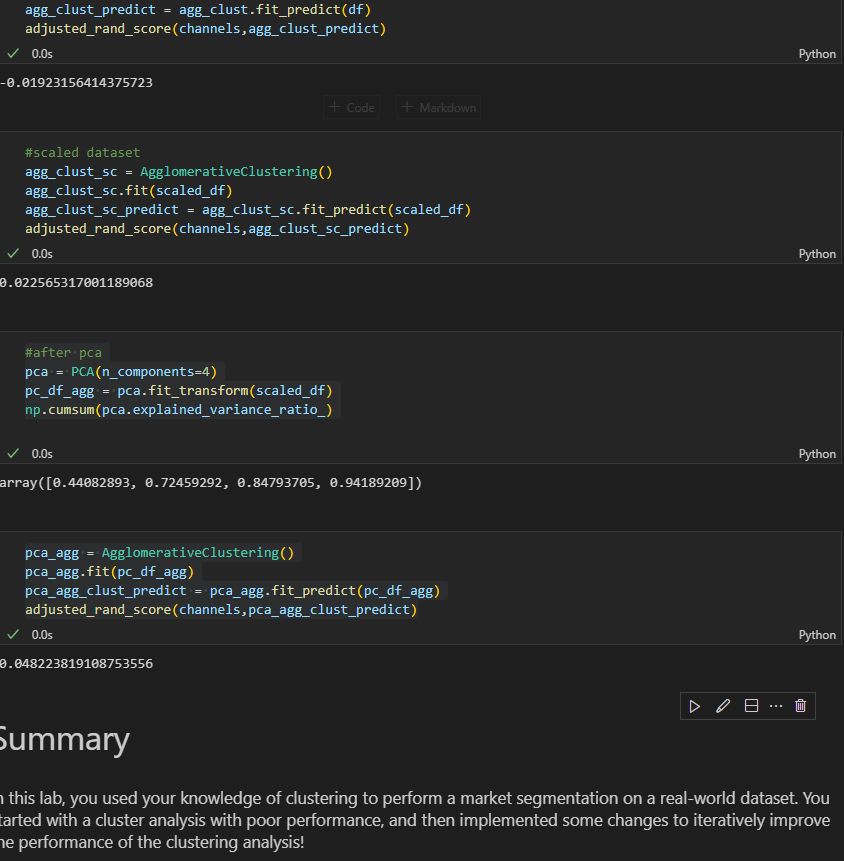
np.cumsum(pca.explained\_variance\_ratio\_)

pca\_agg = AgglomerativeClustering()

pca\_agg.fit(pc\_df\_agg)

pca\_agg\_clust\_predict = pca\_agg.fit\_predict(pc\_df\_agg)

adjusted\_rand\_score(channels,pca\_agg\_clust\_predict)



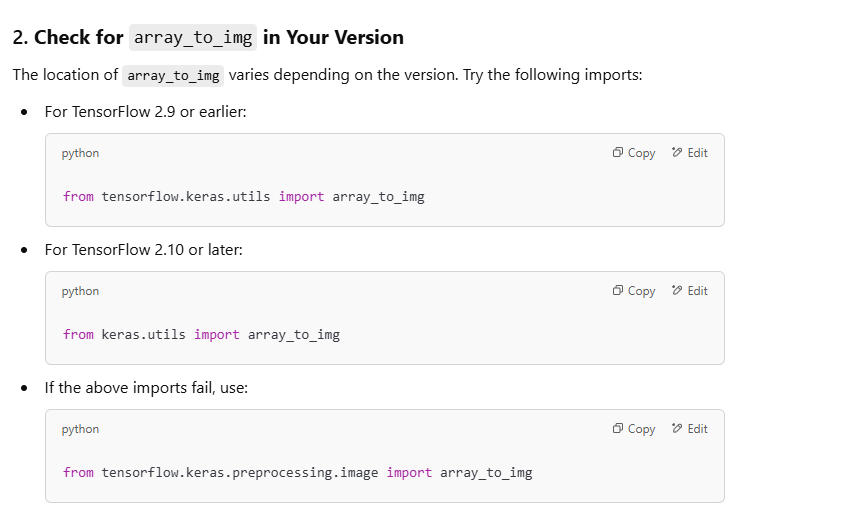
1. **Tensorflow error**

**ImportError**: cannot import name 'array\_to\_img' from 'tensorflow.keras.utils'

from tensorflow.keras.utils import array\_to\_img

from keras.utils import array\_to\_img

from tensorflow.keras.preprocessing.image import array\_to\_img



1. **Deep Learning**

**Create model and add layers**

model\_3 = Sequential()

model\_3.add(Dense(10,activation='tanh',input\_shape=(30,))) #30 columns

model\_3.add(Dense(5,activation='tanh'))

model\_3.add(Dense(1,activation='sigmoid')) #binary and softmax for multiclass

1. **Model Images**

model =  Sequential()

model.add(Input(shape=(28,28,1)))

#hidden

model.add(Conv2D(32,(3,3),activation='relu'))

model.add(Conv2D(64,(3,3),activation='relu'))

model.add(Flatten())

model.add(Dense(32,activation='relu'))

model.add(Dense(16,activation='relu')) #improves model perfomance by removing negative function

model.add(Dense(10,activation='softmax'))

1. **Compiling the model**

model\_3.compile(

    optimizer='sgd',

    loss = 'binary\_crossentropy',

    metrics=['acc'] #metrics = ['precision','accuracy']

)

1. **Train the model**

results\_3 = model\_3.fit(

    df,

    labels,

    epochs = 25,

    batch\_size=1,

    validation\_split = 0.2

)

**Or**

model.fit(training\_data,

         epochs=20,

         batch\_size=32,

         validation\_data=validation)

**Or**

# Train the model

normalized\_input\_model.fit(X\_train,

                          y\_train,

                          batch\_size = 32,

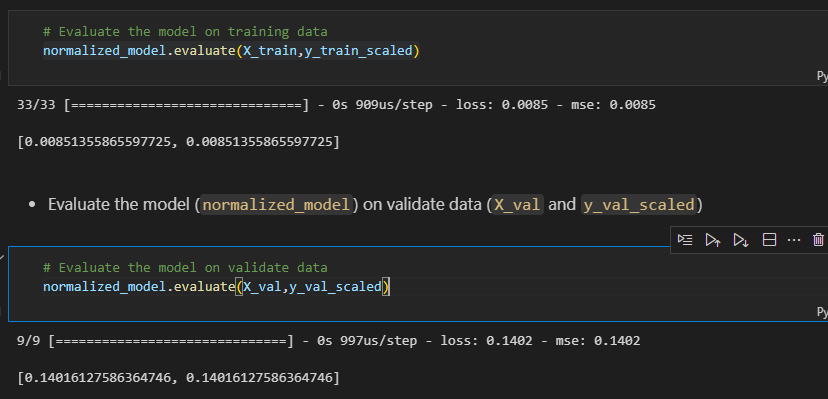
                          epochs =150,

                          validation\_data = (X\_val,y\_val)

                          )

1. **Evaluate Model**

normalized\_model.evaluate(X\_train,y\_train\_scaled)



1. **If we had normalized data we have to take it back to original scale**

# Generate predictions on validate data

y\_val\_pred\_scaled = normalized\_model.predict(X\_val)

# Transform the predictions back to original scale

y\_val\_pred = ss\_y.inverse\_transform(y\_val\_pred\_scaled)

# RMSE of validate data

np.sqrt(mean\_squared\_error(y\_val,y\_val\_pred))

1. **Train test split**
2. # Split data into training and test sets
3. X\_train, X\_test, y\_train, y\_test = train\_test\_split(X,y,test\_size=1500,random\_state=42)
4. **Data for validation**

it is good practice to set aside a validation set, which is then used during hyperparameter tuning. Afterwards, when you have decided upon a final model, the test set can then be used to determine an unbiased perforance of the model.

1. **Get history, loss and accuracy**

history\_dict = history.history

loss\_values = history\_dict['loss']

epochs = range(1,len(loss\_values)+1)

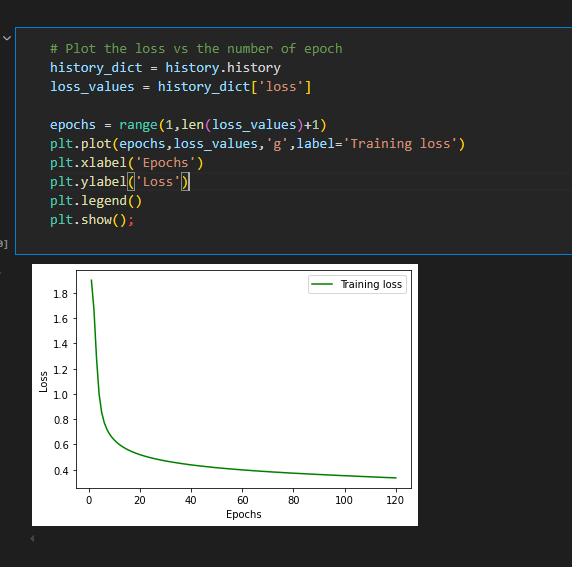
plt.plot(epochs,loss\_values,'g',label='Training loss')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

1. plt.show();



1. **Accuracy**

# Plot the training accuracy vs the number of epochs

acc\_values = history\_dict['acc']

plt.plot(epochs,acc\_values,'r',label='Training acc')

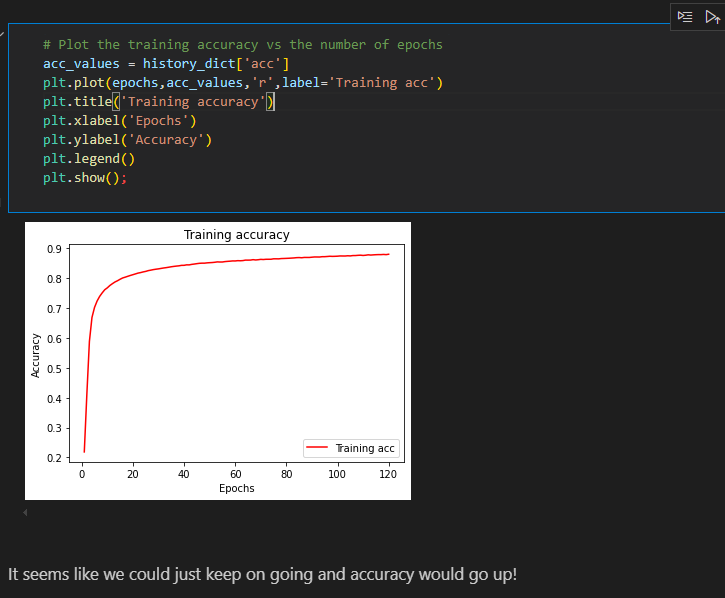
plt.title('Training accuracy')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

plt.show();



1. # Split the data into training and validation sets
2. X\_train\_final, X\_val, y\_train\_final, y\_val = train\_test\_split(X\_train, y\_train, test\_size=1000, random\_state=42)