1. **INTERPRETING ONE-HOT ENCODED COEFFICIENTS**

That is a much more manageable number of coefficients. Let's go through and interpret these:

\* The **\*\*reference category\*\*** for `origin` is `1` (US) and for `make` is `amc` (American Motor Company)

\* `const`, `weight`, and `model year` are all still statistically significant

  \* When all other predictors are 0, the MPG would be about -18.3

  \* For each increase of 1 lb in weight, we see an associated decrease of about 0.006 in MPG

  \* For each year newer the vehicle is, we see an associated increase of about 0.75 in MPG

\* `origin\_2` and `origin\_3` are not statistically significant any more

  \* While this might seem surprising, our data understanding can explain it. The `origin` feature and the `make` feature are really providing the same information, except that `make` is more granular. Every `make` category (except for `other`) corresponds to exactly one `origin` category. Therefore it probably does not make sense to include both `origin` and `make` in the same model

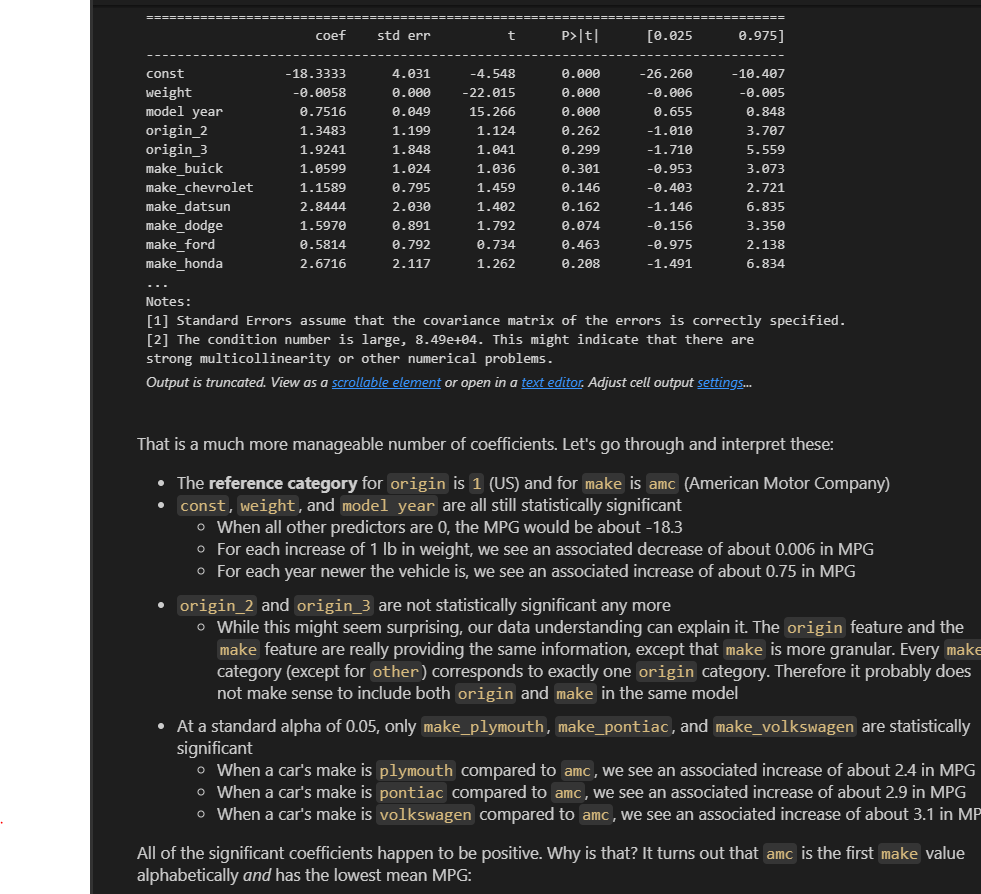
\* At a standard alpha of 0.05, only `make\_plymouth`, `make\_pontiac`, and `make\_volkswagen` are statistically significant

  \* When a car's make is `plymouth` compared to `amc`, we see an associated increase of about 2.4 in MPG

  \* When a car's make is `pontiac` compared to `amc`, we see an associated increase of about 2.9 in MPG

  \* When a car's make is `volkswagen` compared to `amc`, we see an associated increase of about 3.1 in MPG

All of the significant coefficients happen to be positive. Why is that? It turns out that `amc` is the first `make` value alphabetically *\_and\_* has the lowest mean MPG:



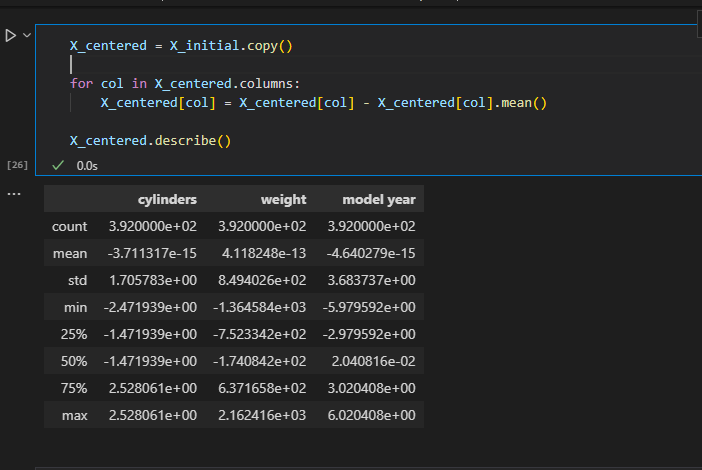
1. **CENTERING**

X\_centered = X\_initial.copy()

for col in X\_centered.columns:

    X\_centered[col] = X\_centered[col] - X\_centered[col].mean()

X\_centered.describe()



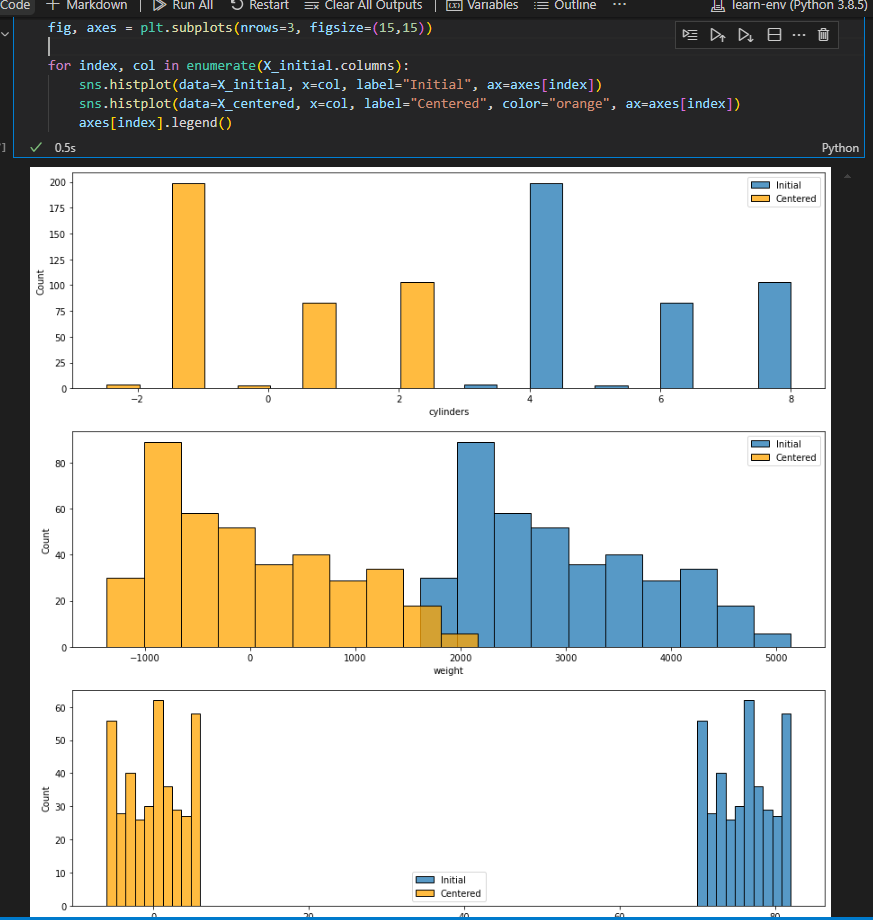
fig, axes = plt.subplots(nrows=3, figsize=(15,15))

for index, col in enumerate(X\_initial.columns):

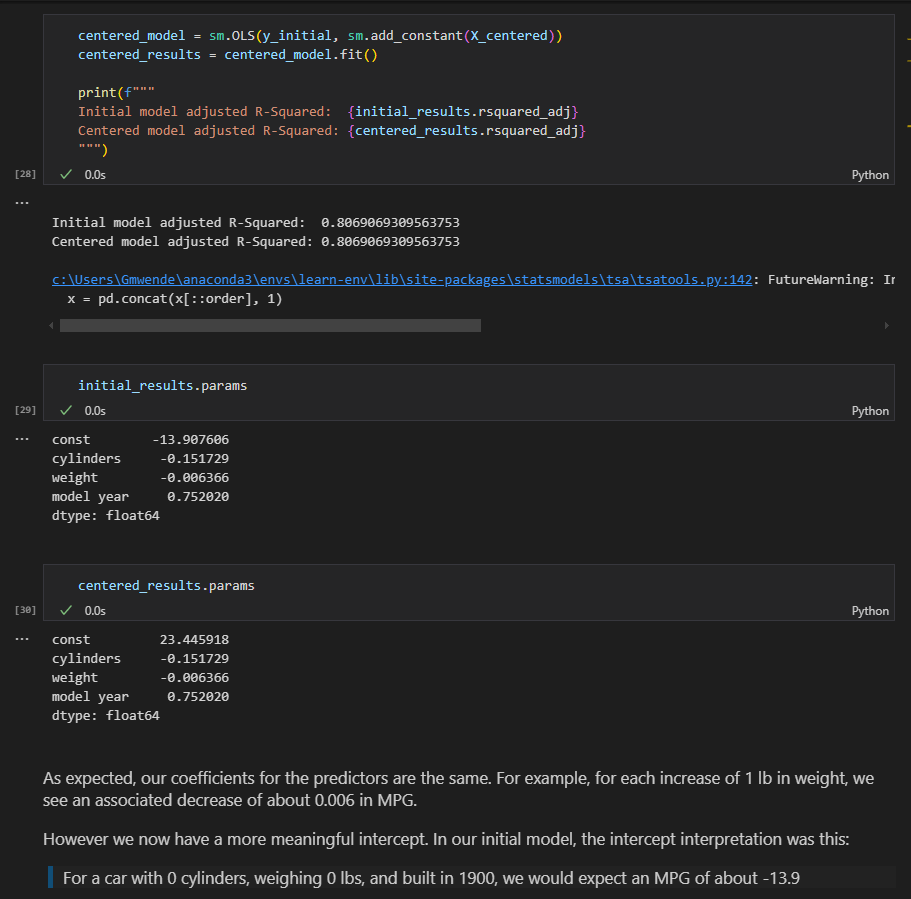
    sns.histplot(data=X\_initial, x=col, label="Initial", ax=axes[index])

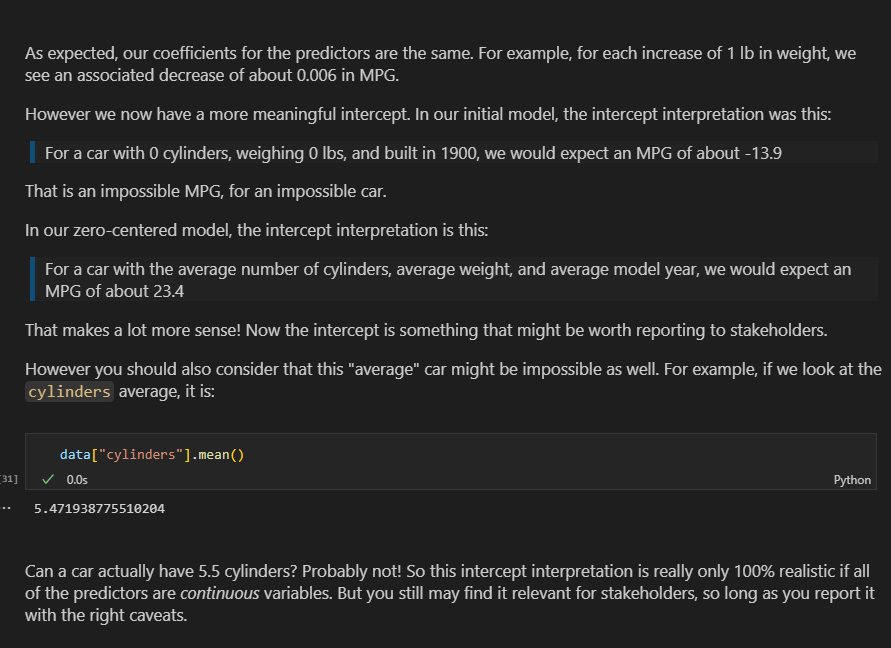
    sns.histplot(data=X\_centered, x=col, label="Centered", color="orange", ax=axes[index])

    axes[index].legend()

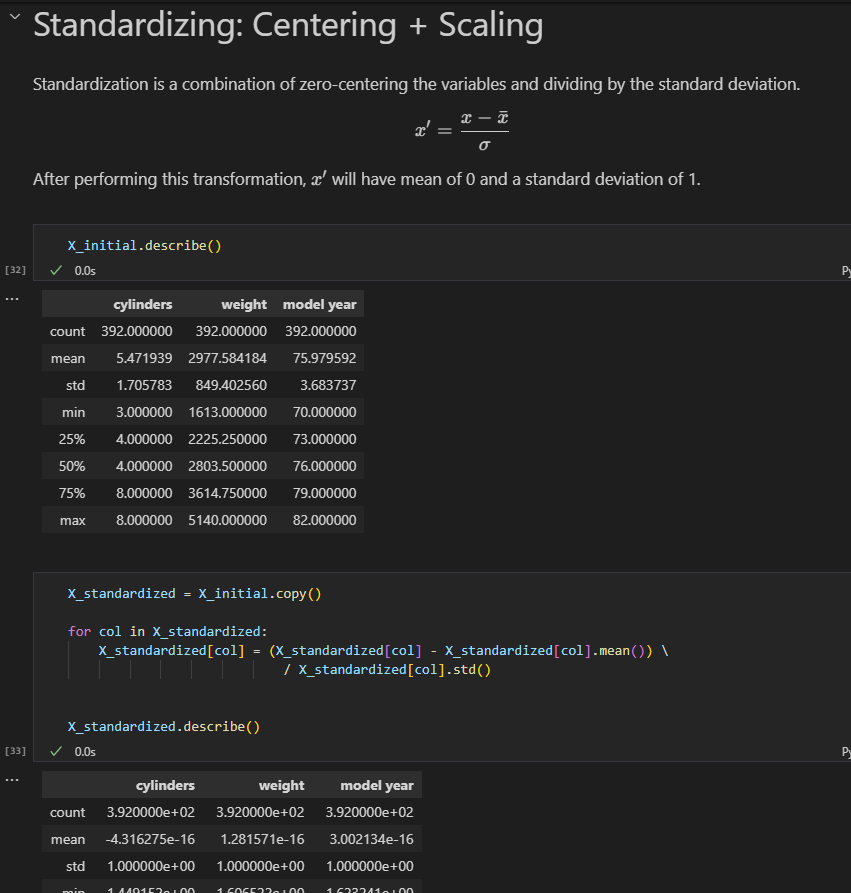


On modelling we can now our coefficients are interpretable

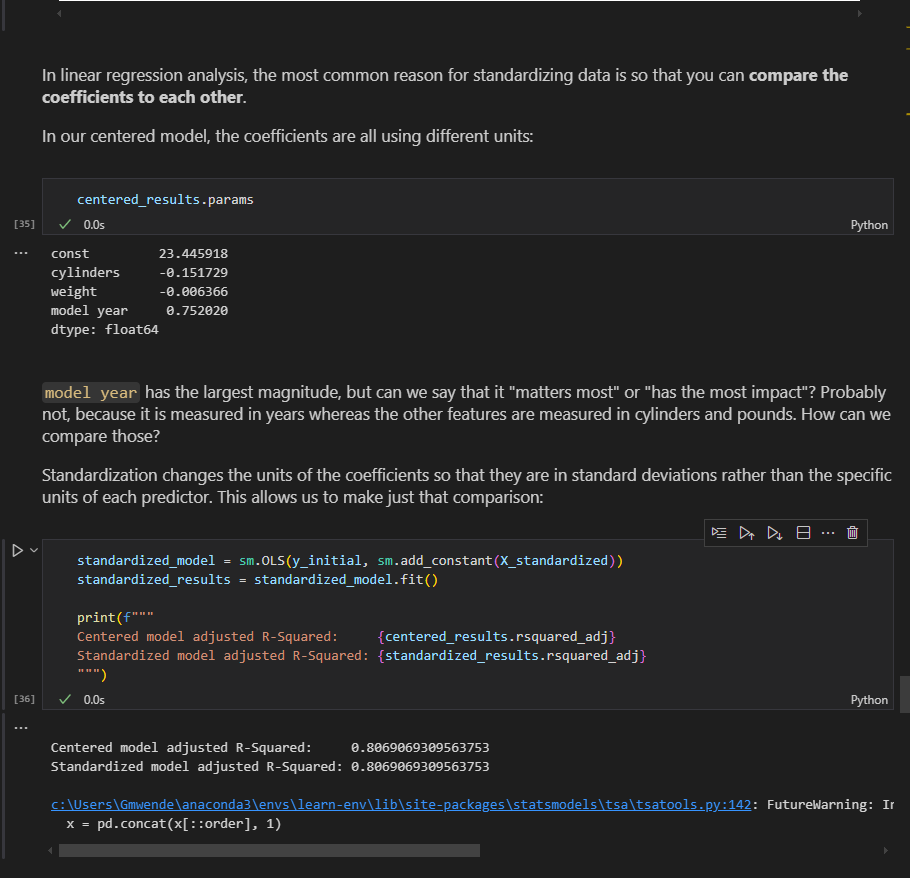


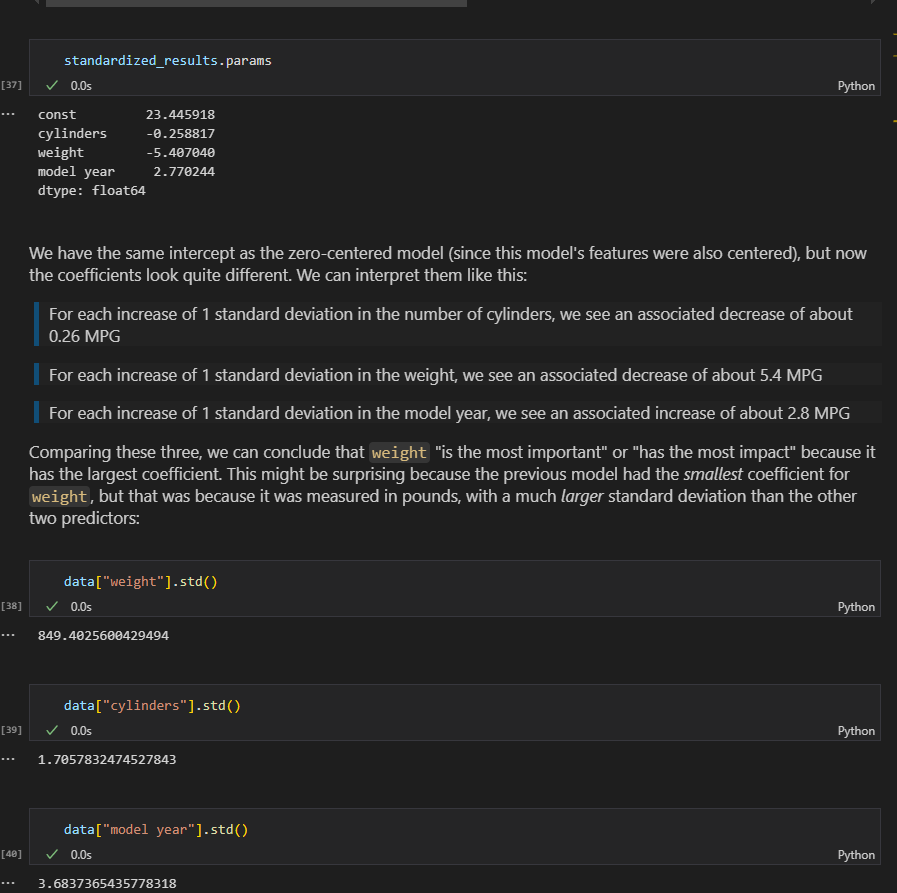


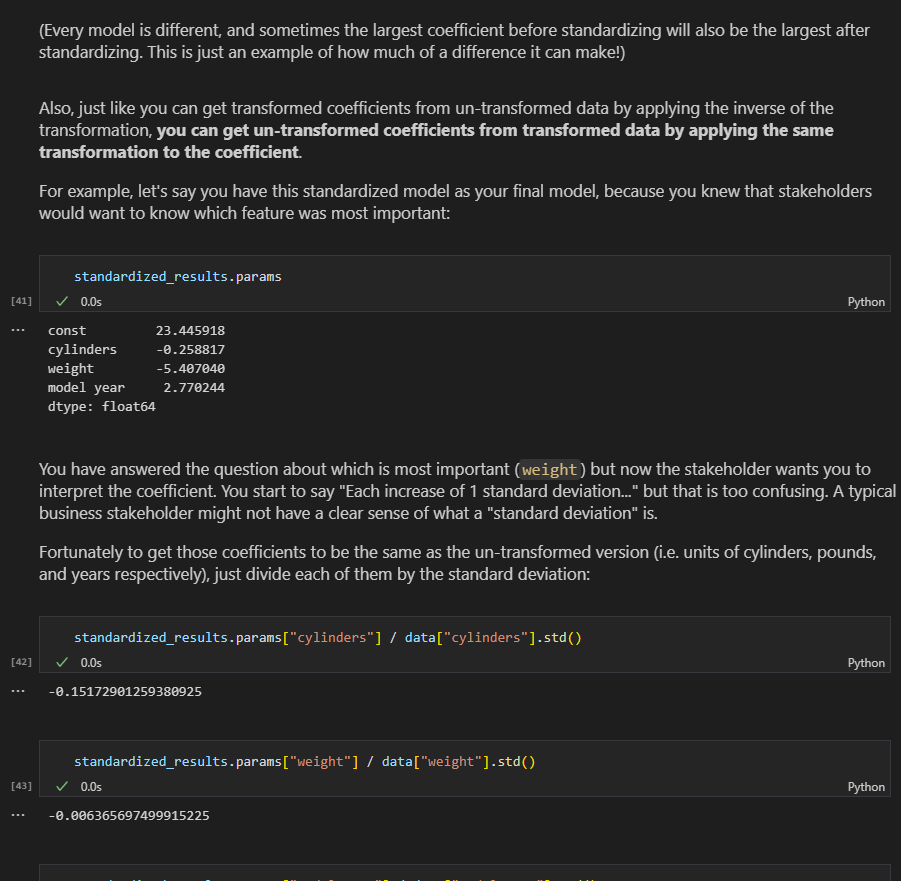
1. Standardization

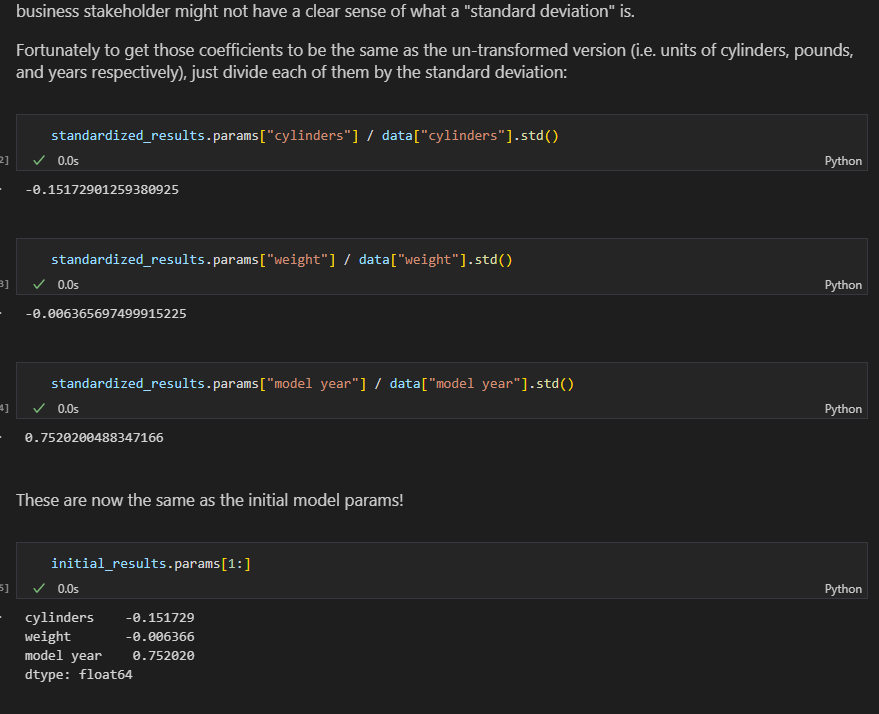












Standardization using sklearn(Standard Scaler)

from sklearn.preprocessing import StandardScaler

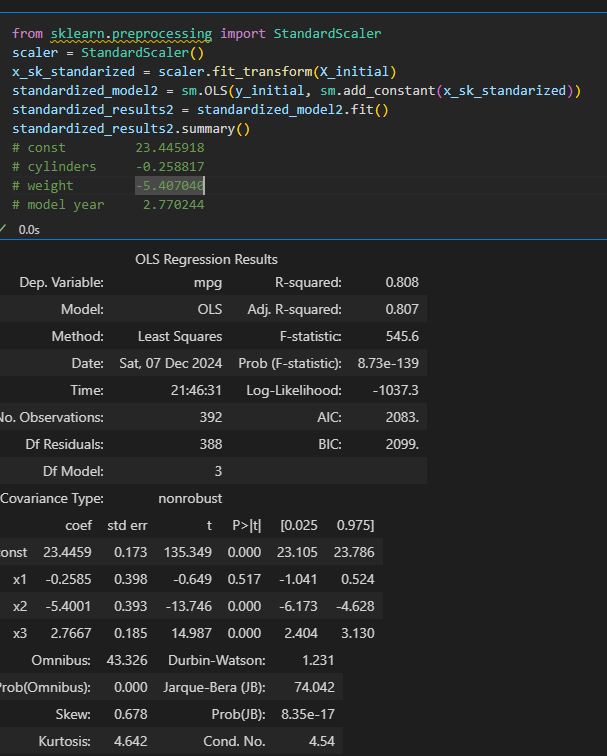
scaler = StandardScaler()

x\_sk\_standarized = scaler.fit\_transform(X\_initial)

standardized\_model2 = sm.OLS(y\_initial, sm.add\_constant(x\_sk\_standarized))

standardized\_results2 = standardized\_model2.fit()

standardized\_results2.summary()



**Draw Graph for multiple columns eg check good candidate for log transformation**

# Run this cell without changes

import matplotlib.pyplot as plt

import numpy as np

y = ames["SalePrice"]

X = ames.drop("SalePrice", axis=1)

fig, axes = plt.subplots(nrows=6, ncols=6, figsize=(15,15), sharey=True)

for i, column in enumerate(X.columns):

    # Locate applicable axes

    row = i // 6

    col = i % 6

    ax = axes[row][col]

    # Plot feature vs. y and label axes

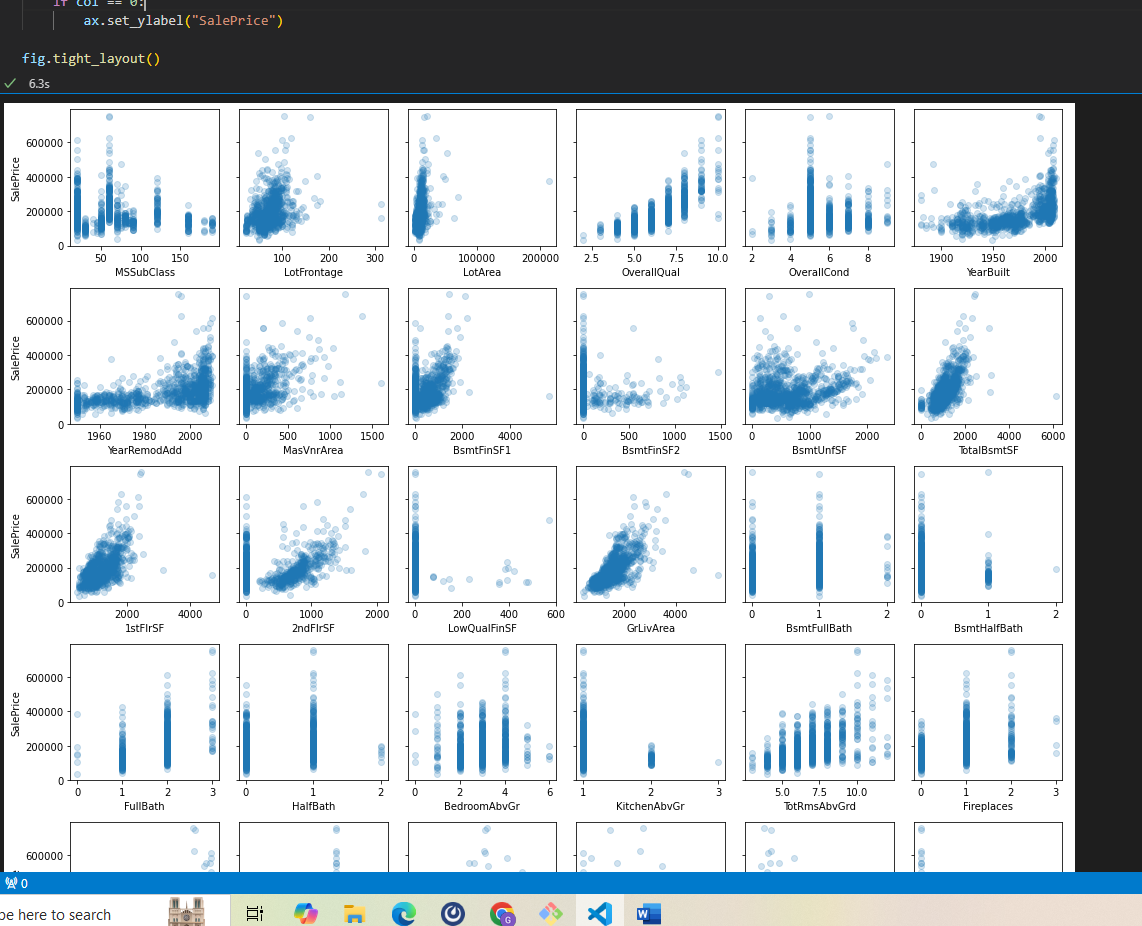
    ax.scatter(X[column], y, alpha=0.2)

    ax.set\_xlabel(column)

    if col == 0:

        ax.set\_ylabel("SalePrice")

fig.tight\_layout()



**Log transform Multiple columns**

from sklearn.preprocessing import FunctionTransformer

import numpy as np

# Instantiate a custom transformer for log transformation

log\_transformer = FunctionTransformer(np.log, validate=True)

# Columns to be log transformed

log\_columns = ['displacement', 'horsepower', 'weight']

# New names for columns after transformation

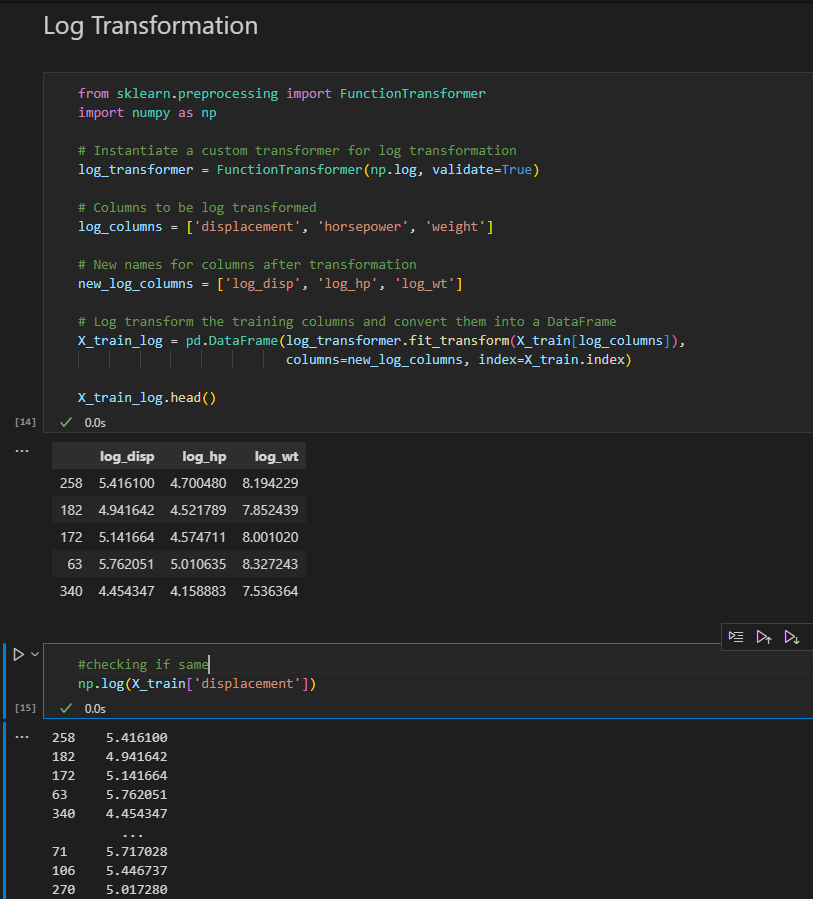
new\_log\_columns = ['log\_disp', 'log\_hp', 'log\_wt']

# Log transform the training columns and convert them into a DataFrame

X\_train\_log = pd.DataFrame(log\_transformer.fit\_transform(X\_train[log\_columns]),

                           columns=new\_log\_columns, index=X\_train.index)

X\_train\_log.head()



1. **One Hot Encoding using sklearn**

from sklearn.preprocessing import OneHotEncoder

#encode test data

ohe = OneHotEncoder()

columns\_to\_encode = ['month']

test\_encoded = ohe.transform(X\_test[columns\_to\_encode])

#Turn into a dataframe

new\_test\_df = pd.DataFrame(

              test\_encoded.todense(),

              columns= ohe.get\_feature\_names\_out(),

              index=X\_test.index

)

new\_test\_df.head()

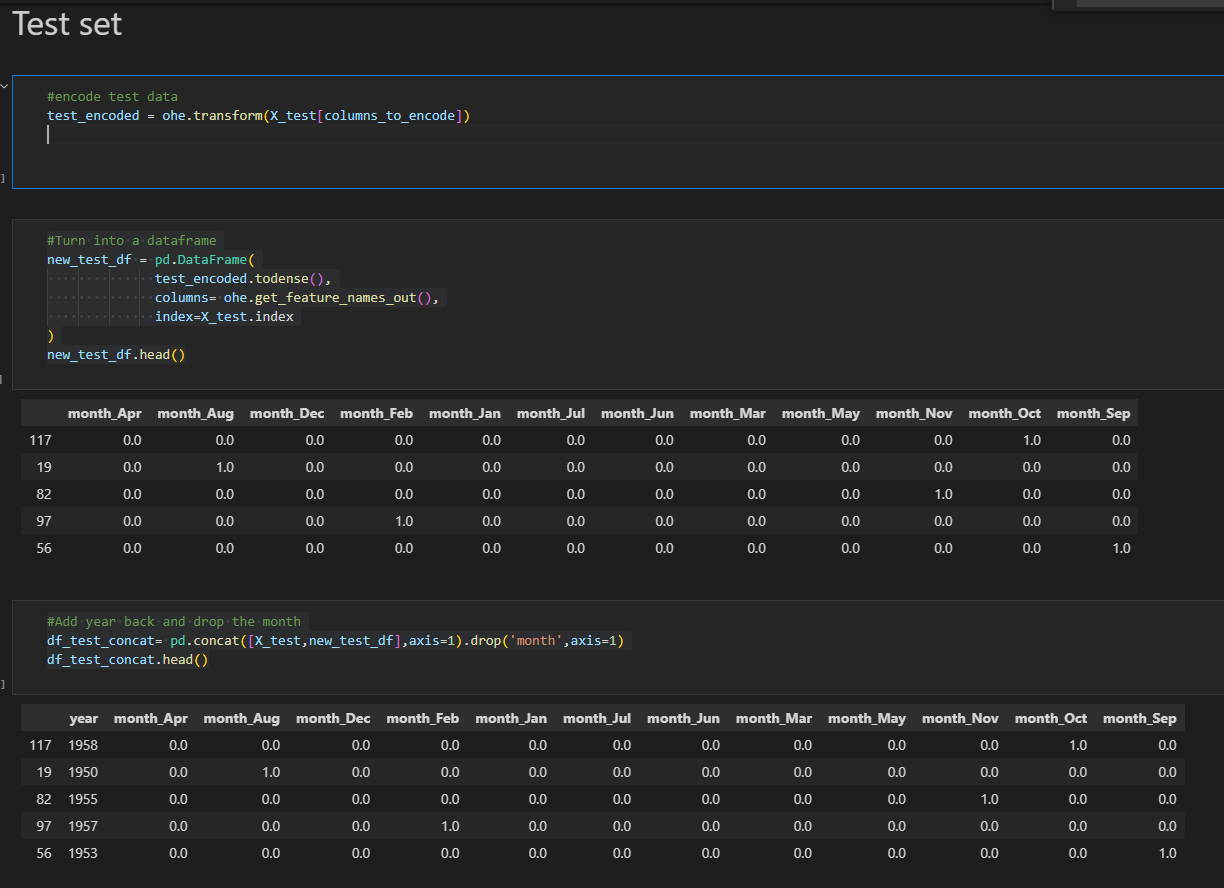
#Add year back and drop the month

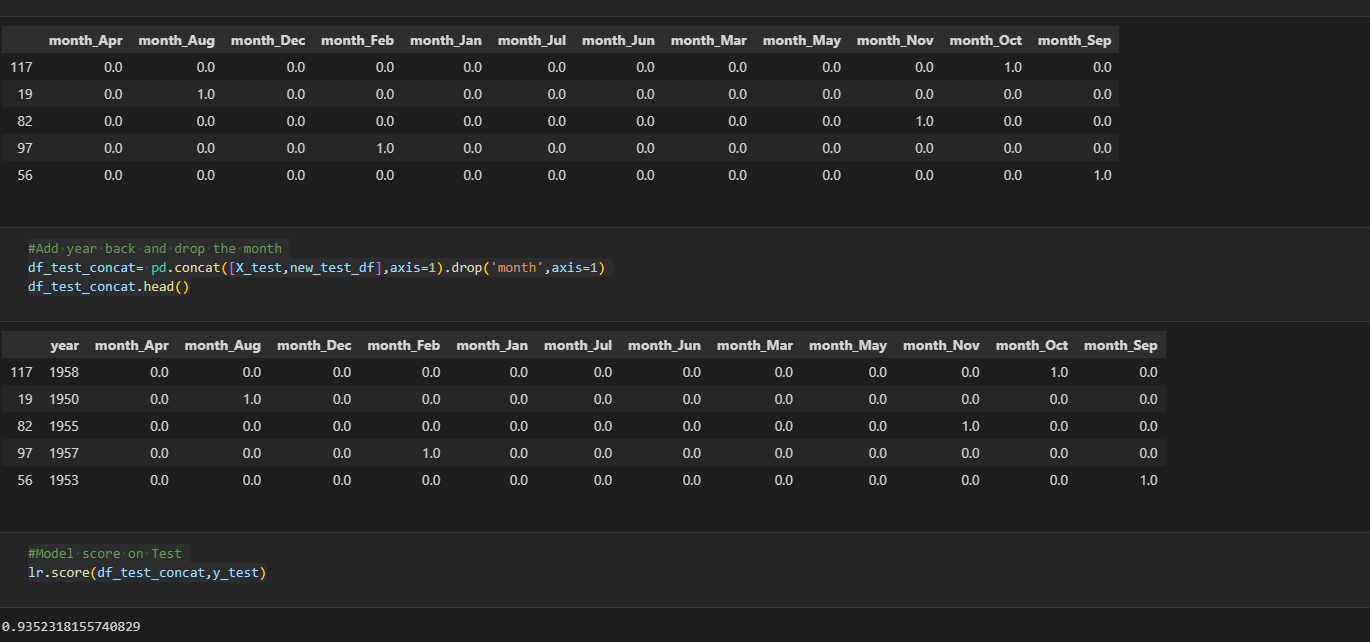
df\_test\_concat= pd.concat([X\_test,new\_test\_df],axis=1).drop('month',axis=1)

df\_test\_concat.head()

#Model score on Test

lr.score(df\_test\_concat,y\_test)





**ONE-HOT ENCODING 2**

from sklearn.preprocessing import OneHotEncoder

# Instantiate OneHotEncoder

# Need to use sparse\_output=False for sklearn 1.2 or greater

ohe = OneHotEncoder(drop='first', sparse=False)

# Create X\_cat which contains only the categorical variables

cat\_columns = ['origin']

X\_train\_cat = X\_train.loc[:, cat\_columns]

# Transform training set

X\_train\_ohe = pd.DataFrame(ohe.fit\_transform(X\_train\_cat),

                           index=X\_train.index)

X\_train\_ohe.head()

# Drop transformed columns

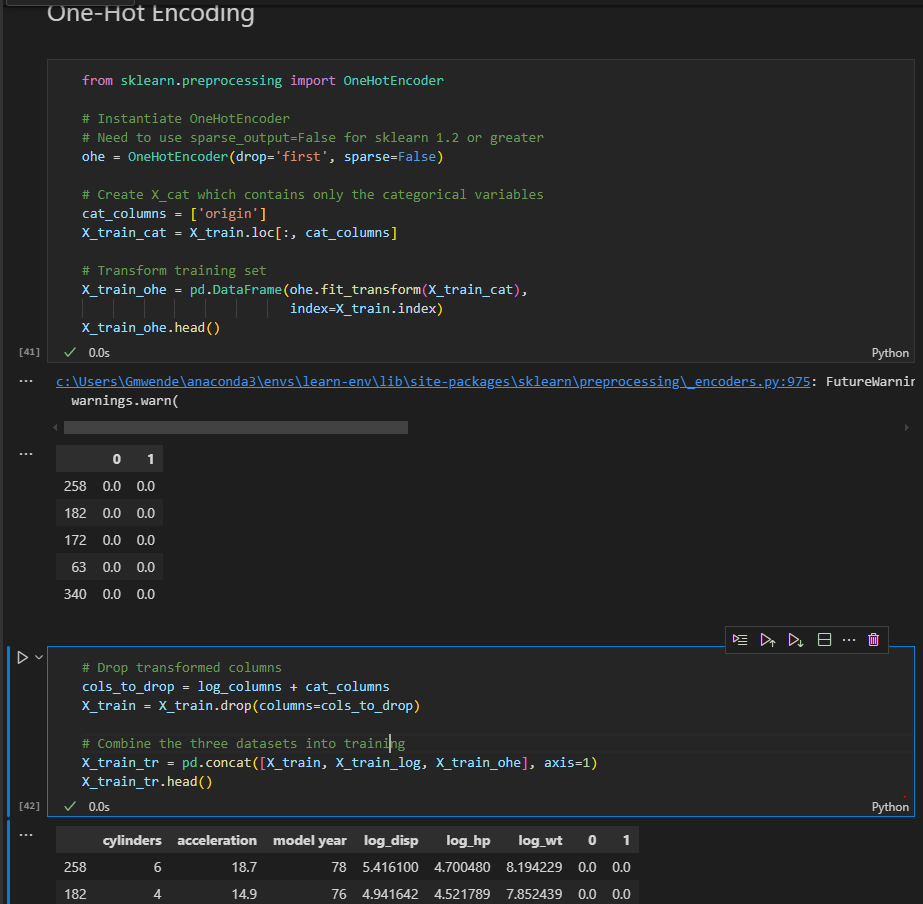
cols\_to\_drop = log\_columns + cat\_columns

X\_train = X\_train.drop(columns=cols\_to\_drop)

# Combine the three datasets into training

X\_train\_tr = pd.concat([X\_train, X\_train\_log, X\_train\_ohe], axis=1)

X\_train\_tr.head()



**ENCODE TEST DATA AS WELL**

# Transform testing set

X\_test\_ohe = pd.DataFrame(ohe.transform(X\_test[cat\_columns]),

                          index=X\_test.index)

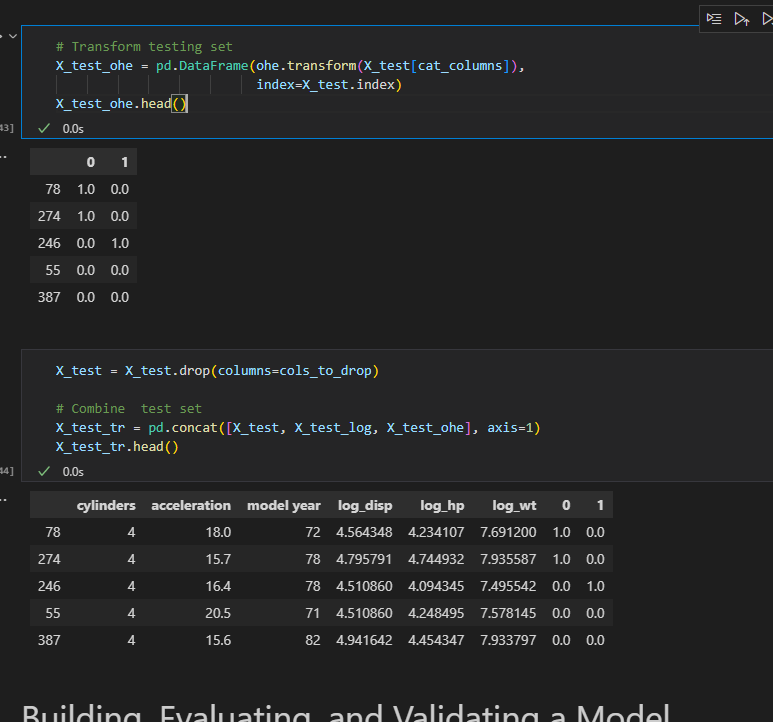
X\_test\_ohe.head()

X\_test = X\_test.drop(columns=cols\_to\_drop)

# Combine test set

X\_test\_tr = pd.concat([X\_test, X\_test\_log, X\_test\_ohe], axis=1)

X\_test\_tr.head()



1. **POLYNOMIALS**

from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(8)

X\_poly\_high = poly.fit\_transform(x)

X\_poly\_high

x\_poly\_high\_df = pd.DataFrame(X\_poly\_high,columns=poly.get\_feature\_names\_out(x.columns),index=x.index)

x\_poly\_high\_df

x\_poly\_high\_df.drop("1",axis=1,inplace=True)

poly\_results = sm.OLS(y, x\_poly\_high\_df).fit()

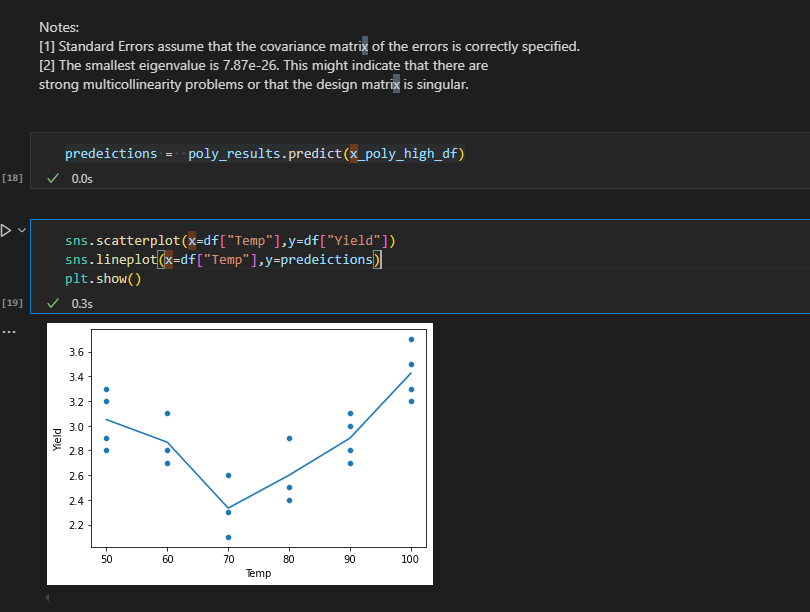
poly\_results.summary()

predeictions =  poly\_results.predict(x\_poly\_high\_df)

sns.scatterplot(x=df["Temp"],y=df["Yield"])

sns.lineplot(x=df["Temp"],y=predeictions)

plt.show()



**POLYNOMIAL WITH LINEAR REGRESSION 1**

# 2nd degree polynomial

poly\_2 = PolynomialFeatures(2)

reg\_poly\_2 = LinearRegression().fit(poly\_2.fit\_transform(X\_train), y\_train)

fig, axes = plt.subplots(ncols=2, figsize=(13,4), sharey=True)

axes[0].scatter(X\_train, y\_train, color='green', label="data points")

axes[0].plot(X\_linspace, reg\_poly\_2.predict(poly\_2.transform(X\_linspace)), label="best fit line")

axes[0].set\_xlabel('Temperature')

axes[0].set\_ylabel('Yield')

axes[0].set\_title('Train')

axes[1].scatter(X\_test, y\_test, color='green')

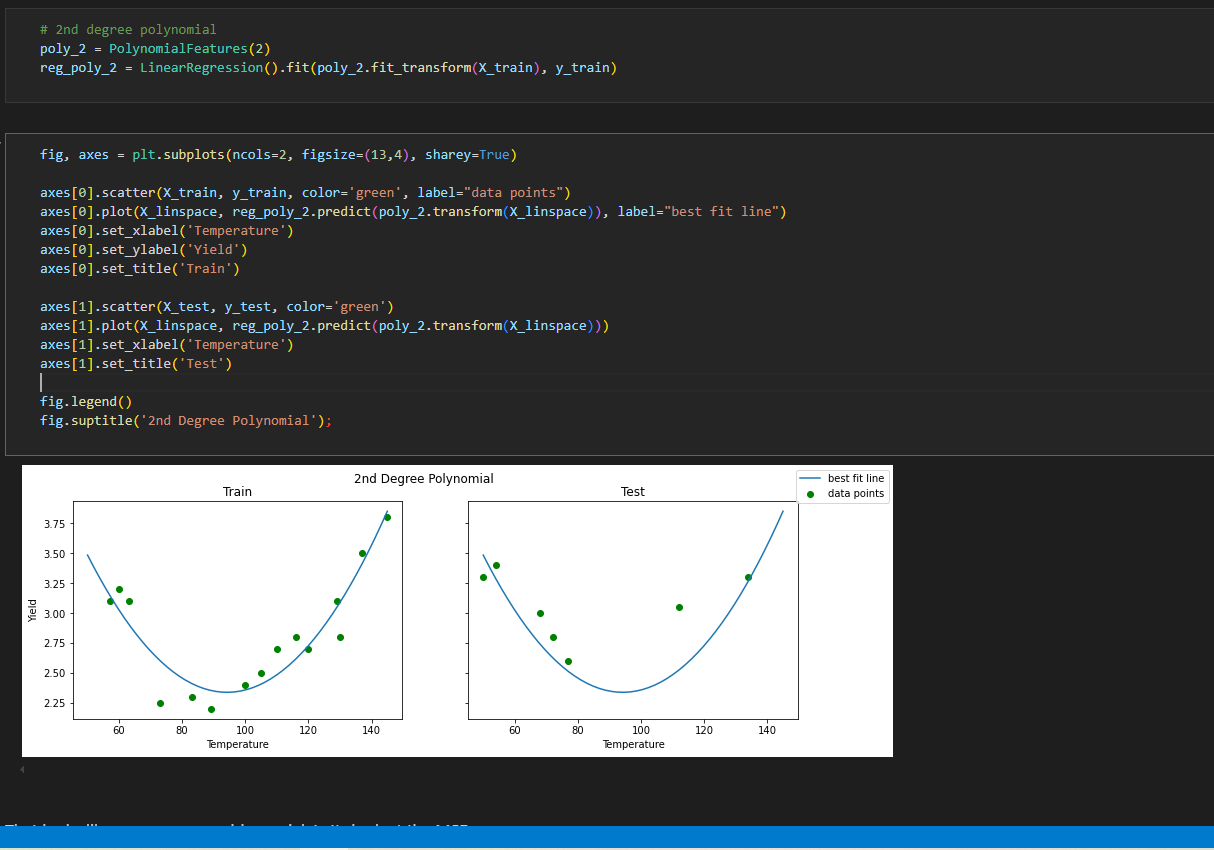
axes[1].plot(X\_linspace, reg\_poly\_2.predict(poly\_2.transform(X\_linspace)))

axes[1].set\_xlabel('Temperature')

axes[1].set\_title('Test')

fig.legend()

fig.suptitle('2nd Degree Polynomial');



print(f"""

Simple Linear Regression

Train MSE: {mean\_squared\_error(y\_train, reg.predict(X\_train))}

Test MSE:  {mean\_squared\_error(y\_test, reg.predict(X\_test))}

6th Degree Polynomial

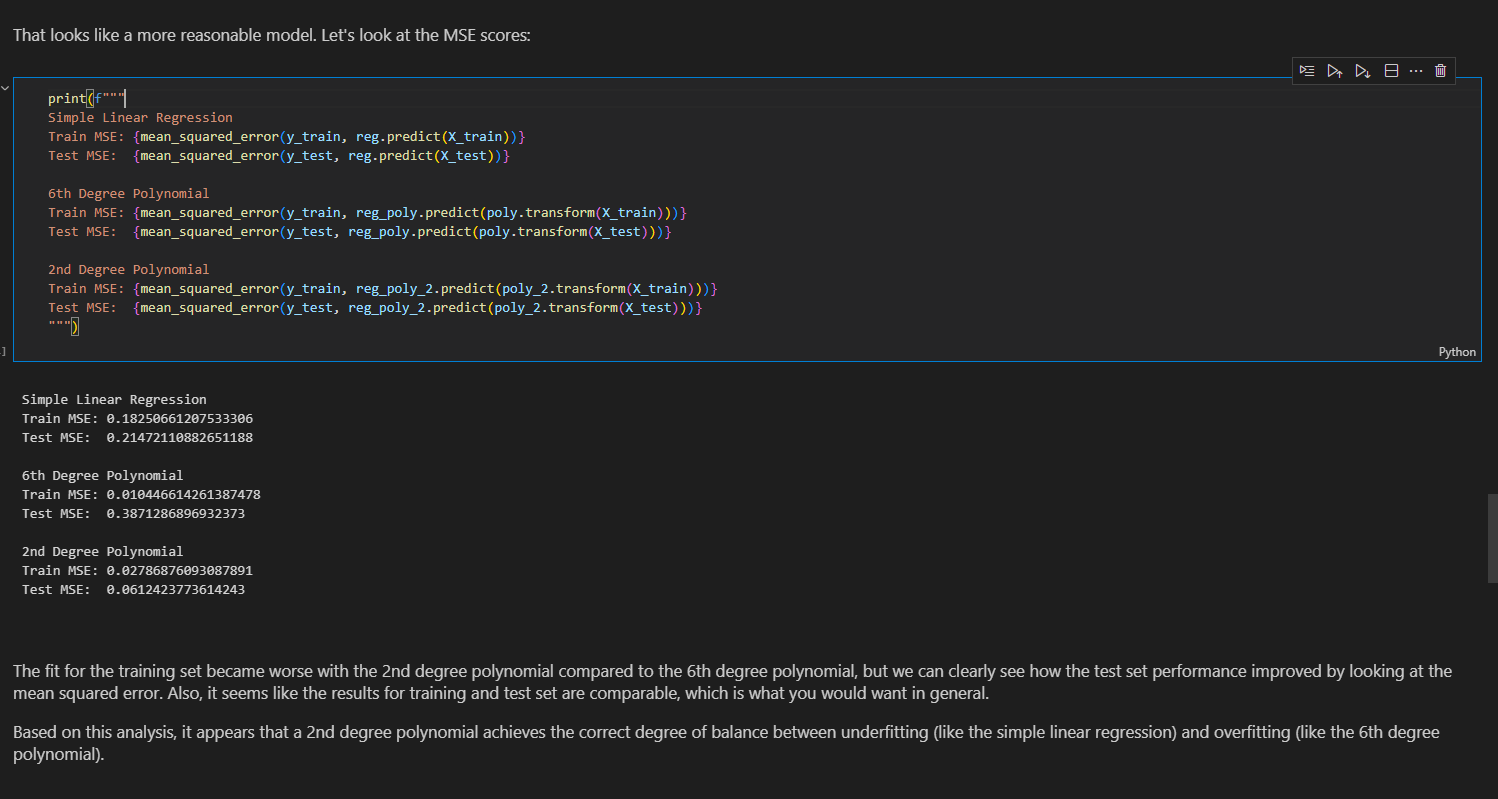
Train MSE: {mean\_squared\_error(y\_train, reg\_poly.predict(poly.transform(X\_train)))}

Test MSE:  {mean\_squared\_error(y\_test, reg\_poly.predict(poly.transform(X\_test)))}

2nd Degree Polynomial

Train MSE: {mean\_squared\_error(y\_train, reg\_poly\_2.predict(poly\_2.transform(X\_train)))}

Test MSE:  {mean\_squared\_error(y\_test, reg\_poly\_2.predict(poly\_2.transform(X\_test)))}



**POLYNOMIAL WITH LINEAR REGRESSION2**

poly = PolynomialFeatures(3)

X\_train\_poly = poly.fit\_transform(X\_train\_scaled)

X\_test\_poly = poly.transform(X\_test\_scaled)

polyreg = LinearRegression()

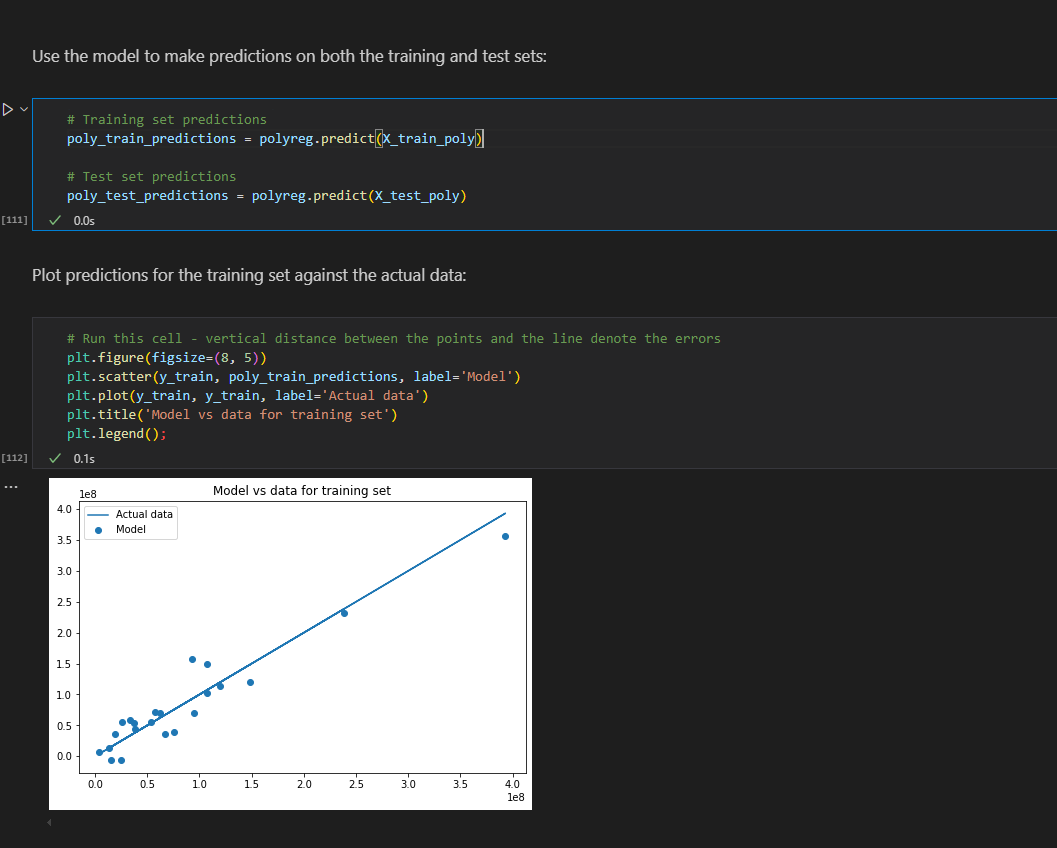
polyreg.fit(X\_train\_poly,y\_train)

# Training set predictions

poly\_train\_predictions = polyreg.predict(X\_train\_poly)

# Test set predictions

poly\_test\_predictions = polyreg.predict(X\_test\_poly)



1. **BUILDING,EVALUATING AND VALIDATING A MODEL**

# convert feature names to strings so there is not a TypeError with sklearn

X\_train\_tr.columns = X\_train\_tr.columns.astype(str)

X\_test\_tr.columns = X\_test\_tr.columns.astype(str)

from sklearn.linear\_model import LinearRegression

linreg = LinearRegression()

linreg.fit(X\_train\_tr, y\_train)

y\_hat\_train = linreg.predict(X\_train\_tr)

y\_hat\_test = linreg.predict(X\_test\_tr)

train\_residuals = y\_hat\_train - y\_train

test\_residuals = y\_hat\_test - y\_test

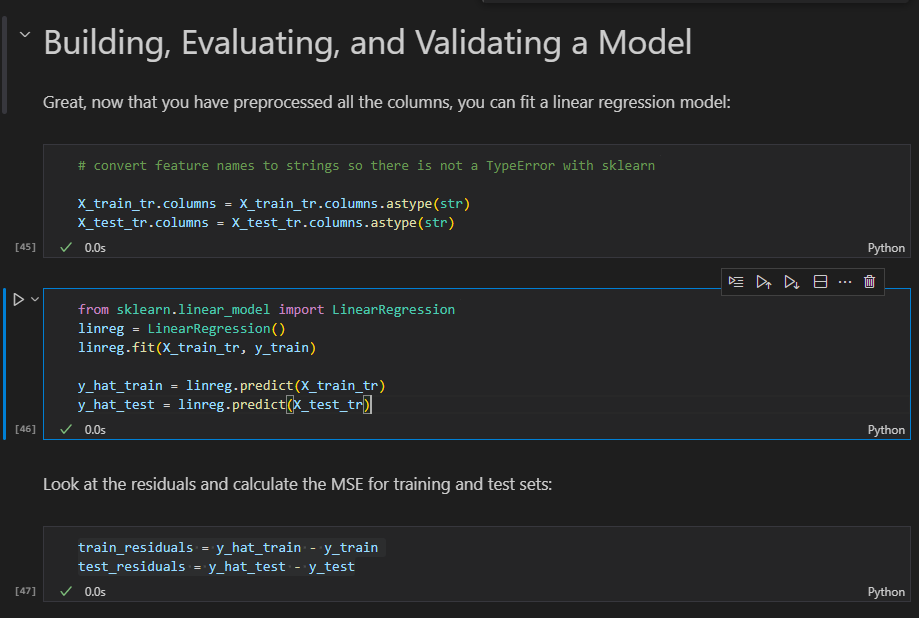
from sklearn.metrics import mean\_squared\_error

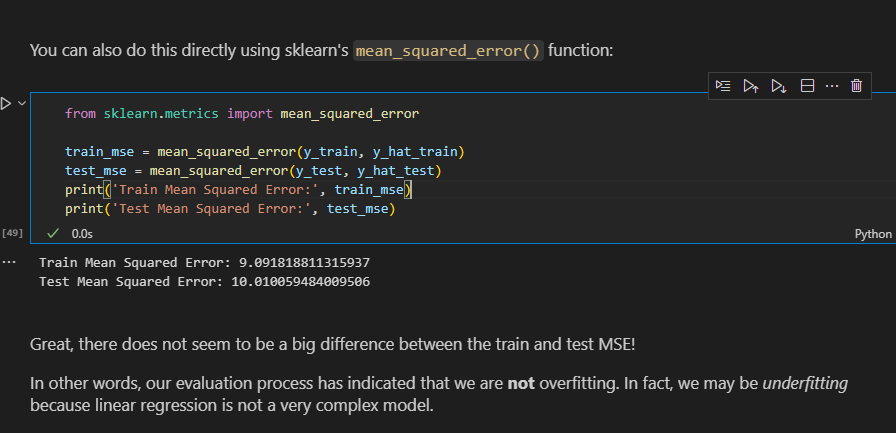
train\_mse = mean\_squared\_error(y\_train, y\_hat\_train)

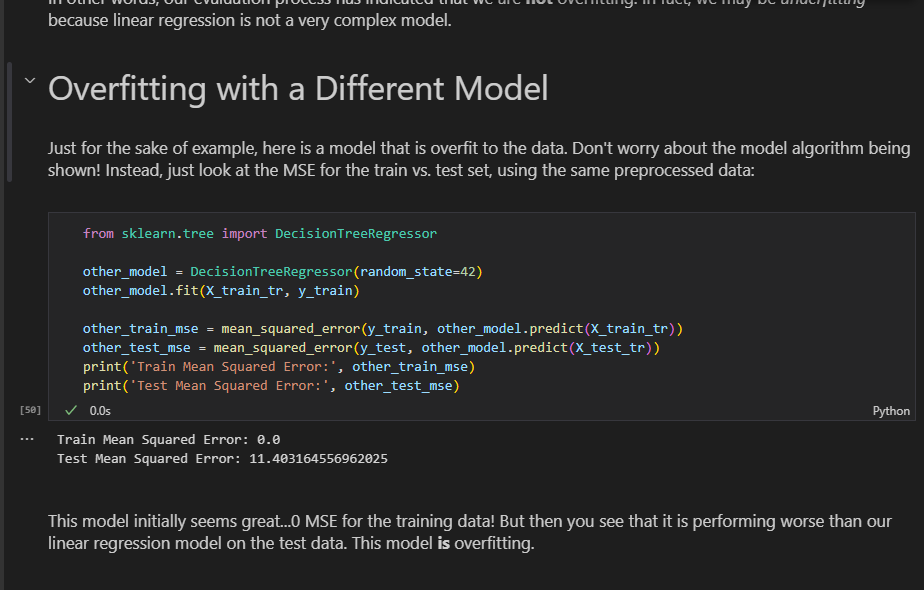
test\_mse = mean\_squared\_error(y\_test, y\_hat\_test)

print('Train Mean Squared Error:', train\_mse)

print('Test Mean Squared Error:', test\_mse)







1. **R2 SCORE AND MEAN SQUARED ERROR**

from sklearn.metrics import mean\_squared\_error, r2\_score

  lr = LinearRegression()

    lr.fit(X\_train, y\_train)

    # Predictions

    y\_train\_pred = lr.predict(X\_train)

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

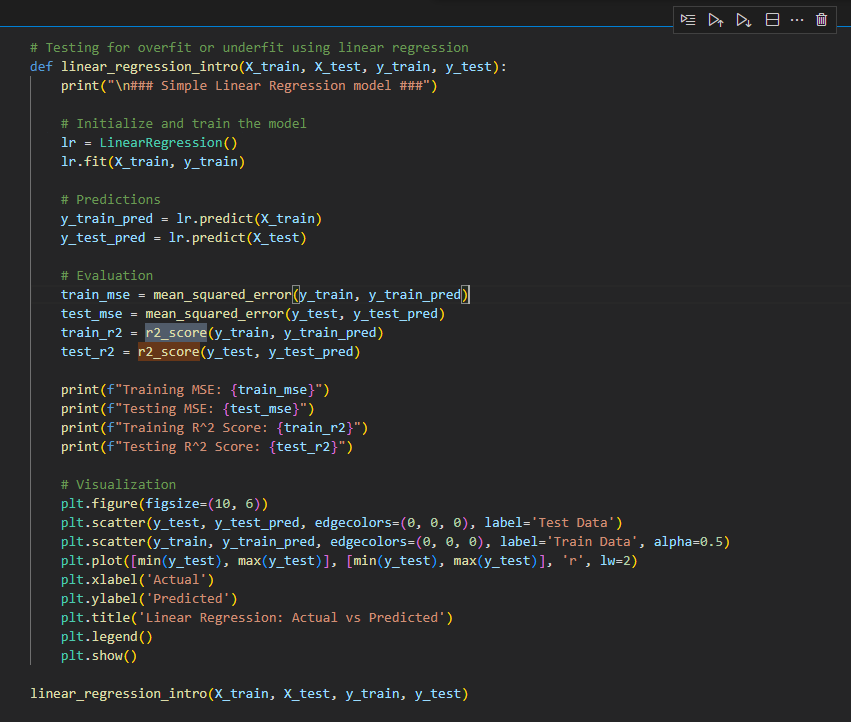
    # Evaluation

    train\_mse = mean\_squared\_error(y\_train, y\_train\_pred)

    test\_mse = mean\_squared\_error(y\_test, y\_test\_pred)

    train\_r2 = r2\_score(y\_train, y\_train\_pred)

    test\_r2 = r2\_score(y\_test, y\_test\_pred)



1. **Splitting data**

X = ames.drop('SalePrice',axis=1)

y = ames['SalePrice']

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

X\_train,X\_test,y\_train,y\_test = train\_test\_split(X,y,test\_size=0.33,random\_state=42)

1. **Log transformation and one hot encoding together**

# Run this cell without changes

from sklearn.preprocessing import FunctionTransformer, OneHotEncoder

continuous = ['LotArea', '1stFlrSF', 'GrLivArea']

categoricals = ['BldgType', 'KitchenQual', 'Street']

# Instantiate transformers

log\_transformer = FunctionTransformer(np.log, validate=True)

ohe = OneHotEncoder(drop='first', sparse=False)

# Fit transformers

log\_transformer.fit(X\_train[continuous])

ohe.fit(X\_train[categoricals])

# Transform training data

X\_train = pd.concat([

    pd.DataFrame(log\_transformer.transform(X\_train[continuous]), index=X\_train.index),

    pd.DataFrame(ohe.transform(X\_train[categoricals]), index=X\_train.index)

], axis=1)

# Transform test data

X\_test = pd.concat([

    pd.DataFrame(log\_transformer.transform(X\_test[continuous]), index=X\_test.index),

    pd.DataFrame(ohe.transform(X\_test[categoricals]), index=X\_test.index)

], axis=1)

1. **CROSS VALIDATION**
2. from sklearn.model\_selection import cross\_val\_score

cross\_val\_score(linreg, X, y, cv=10)

cross\_val\_score(linreg, X, y, scoring="neg\_mean\_squared\_error")#MSE instead of r2

#Scores for different metrics

from sklearn.model\_selection import cross\_validate

cross\_validate(linreg, X, y, scoring=["r2", "neg\_mean\_squared\_error"])

1. **get mean of all the cross validation scores**

cross\_val\_results = cross\_validate(linreg, X, y, scoring="neg\_mean\_squared\_error", return\_train\_score=True)

# Negative signs in front to convert back to MSE from -MSE

train\_avg = -cross\_val\_results["train\_score"].mean()

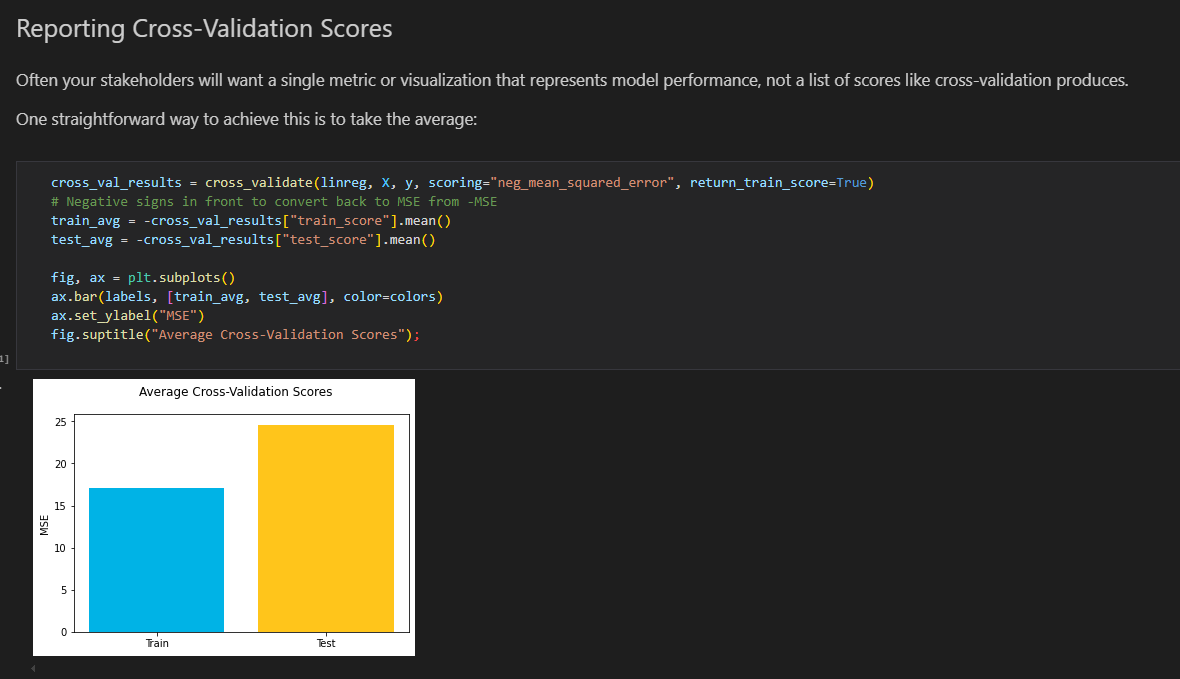
test\_avg = -cross\_val\_results["test\_score"].mean()

fig, ax = plt.subplots()

ax.bar(labels, [train\_avg, test\_avg], color=colors)

ax.set\_ylabel("MSE")

fig.suptitle("Average Cross-Validation Scores");



Another way, if you have enough folds to make it worthwhile, is to show the distribution of the train vs. test scores using a histogram or a box plot. *\*N.B.\**: The *\*x\**-axes are different scales, but the focus is on the different shapes of the respective distributions.

cross\_val\_results = cross\_validate(linreg, X, y, cv=100, scoring="neg\_mean\_squared\_error", return\_train\_score=True)

train\_scores = -cross\_val\_results["train\_score"]

test\_scores = -cross\_val\_results["test\_score"]

fig, (left, right) = plt.subplots(ncols=2, figsize=(10,5), sharey=True)

bins=25

left.hist(train\_scores, label=labels[0], bins=bins, color=colors[0])

left.set\_ylabel("Count")

left.set\_xlabel("MSE")

right.hist(test\_scores, label=labels[1], bins=bins, color=colors[1])

right.set\_xlabel("MSE")

fig.suptitle("Cross-Validation Score Distribution")

fig.legend();



1. **Log Transform and hot encoding in one place**

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

%matplotlib inline

ames = pd.read\_csv('data/ames.csv')

continuous = ['LotArea', '1stFlrSF', 'GrLivArea', 'SalePrice']

categoricals = ['BldgType', 'KitchenQual', 'SaleType', 'MSZoning', 'Street', 'Neighborhood']

ames\_cont = ames[continuous]

# log features

log\_names = [f'{column}\_log' for column in ames\_cont.columns]

ames\_log = np.log(ames\_cont)

ames\_log.columns = log\_names

# normalize (subract mean and divide by std)

def normalize(feature):

    return (feature - feature.mean()) / feature.std()

ames\_log\_norm = ames\_log.apply(normalize)

# one hot encode categoricals

ames\_ohe = pd.get\_dummies(ames[categoricals], prefix=categoricals, drop\_first=True)

preprocessed = pd.concat([ames\_log\_norm, ames\_ohe], axis=1)

X = preprocessed.drop('SalePrice\_log', axis=1)

y = preprocessed['SalePrice\_log']

1. **Another example of one hot encoding**

ohe = OneHotEncoder(drop='first', sparse\_output=False)

train\_female = ohe.fit\_transform(X\_train[['SEX']]).flatten()

test\_female = ohe.transform(X\_test[['SEX']]).flatten()

1. **Ridge and Lasso Regression**

# Prepare data

from sklearn.linear\_model import Lasso, Ridge, LinearRegression

poly = PolynomialFeatures(degree=6)

X\_train\_poly = poly.fit\_transform(X\_train)

X\_test\_poly = poly.transform(X\_test)

X\_train\_transformed = scale.fit\_transform(X\_train\_poly)

X\_test\_transformed = scale.transform(X\_test\_poly)

ridge = Ridge(alpha=0.5)

ridge.fit(X\_train\_transformed, y\_train)

lasso = Lasso(alpha=0.5)

lasso.fit(X\_train\_transformed, y\_train)

lin = LinearRegression()

lin.fit(X\_train\_transformed, y\_train)

# Fit models

ridge.fit(X\_train\_transformed, y\_train)

lasso.fit(X\_train\_transformed, y\_train)

lin.fit(X\_train\_transformed, y\_train)

# Generate predictions

y\_h\_ridge\_train = ridge.predict(X\_train\_transformed)

y\_h\_ridge\_test = ridge.predict(X\_test\_transformed)

y\_h\_lasso\_train = lasso.predict(X\_train\_transformed)

y\_h\_lasso\_test = lasso.predict(X\_test\_transformed)

y\_h\_lin\_train = lin.predict(X\_train\_transformed)

y\_h\_lin\_test = lin.predict(X\_test\_transformed)

# Display results

print('Train Error Polynomial Ridge Model', mean\_squared\_error(y\_train, y\_h\_ridge\_train))

print('Test Error Polynomial Ridge Model', mean\_squared\_error(y\_test, y\_h\_ridge\_test))

print('\n')

print('Train Error Polynomial Lasso Model', mean\_squared\_error(y\_train, y\_h\_lasso\_train))

print('Test Error Polynomial Lasso Model', mean\_squared\_error(y\_test, y\_h\_lasso\_test))

print('\n')

print('Train Error Unpenalized Polynomial Model', mean\_squared\_error(y\_train, y\_h\_lin\_train))

print('Test Error Unpenalized Polynomial Model', mean\_squared\_error(y\_test, y\_h\_lin\_test))

print('\n')

print('Polynomial Ridge Parameter Coefficients:', len(ridge.coef\_[ridge.coef\_ != 0]),

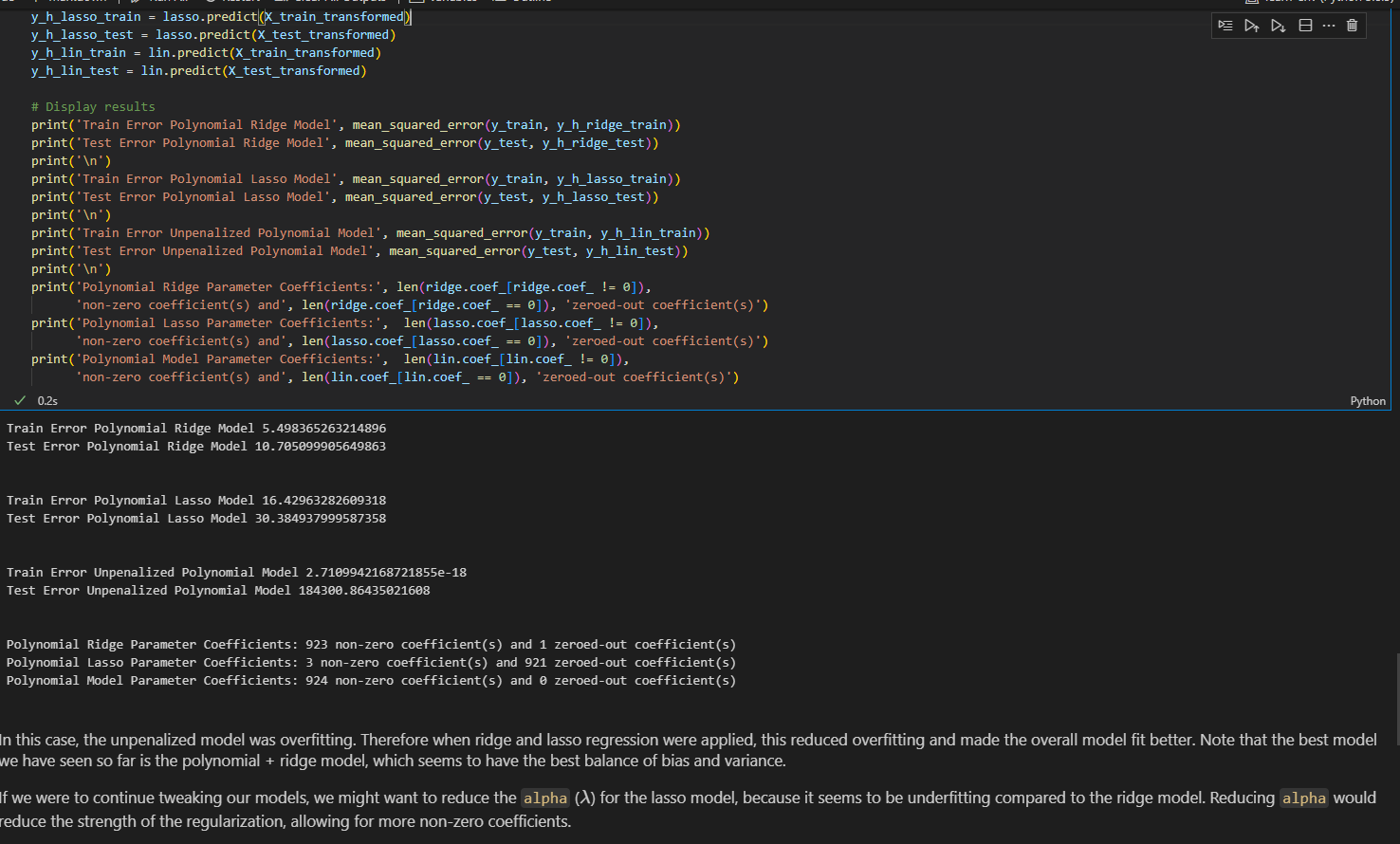
      'non-zero coefficient(s) and', len(ridge.coef\_[ridge.coef\_ == 0]), 'zeroed-out coefficient(s)')

print('Polynomial Lasso Parameter Coefficients:',  len(lasso.coef\_[lasso.coef\_ != 0]),

      'non-zero coefficient(s) and', len(lasso.coef\_[lasso.coef\_ == 0]), 'zeroed-out coefficient(s)')

print('Polynomial Model Parameter Coefficients:',  len(lin.coef\_[lin.coef\_ != 0]),

      'non-zero coefficient(s) and', len(lin.coef\_[lin.coef\_ == 0]), 'zeroed-out coefficient(s)')

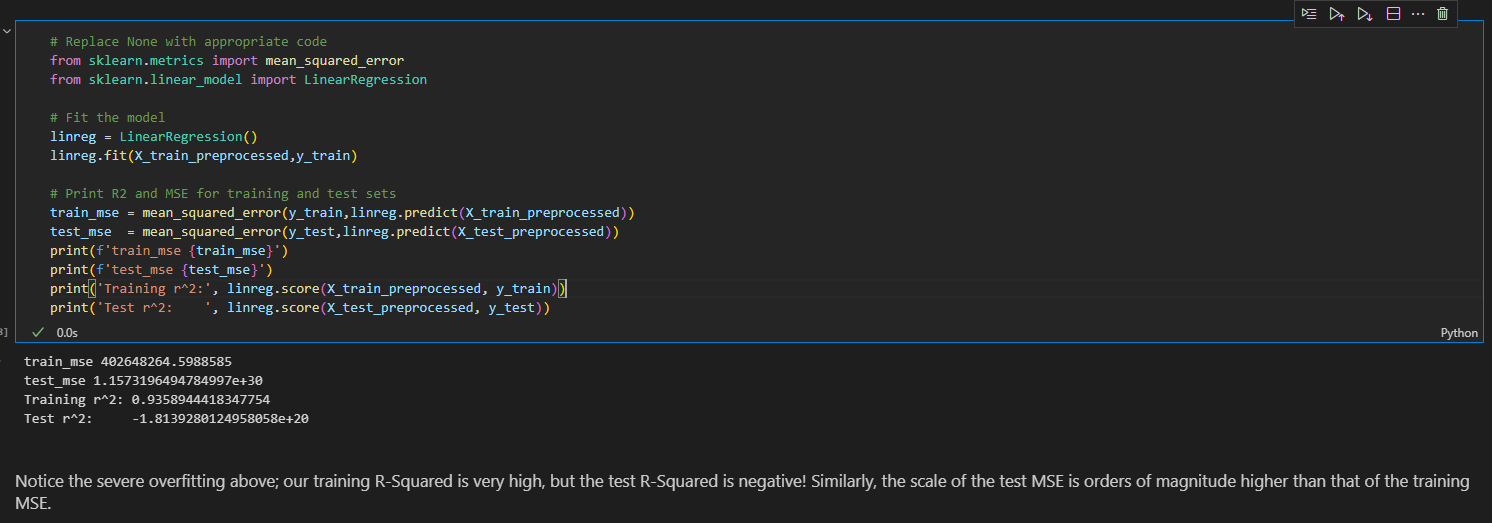


If we were to continue tweaking our models, we might want to reduce the alpha for the lasso model, because it seems to be underfitting compared to the ridge model. Reducing alpha would reduce the strength of the regularization, allowing for more non-zero coefficients.

1. **Getting r squared**

print('Training r^2:', linreg.score(X\_train\_preprocessed, y\_train))

print('Test r^2:    ', linreg.score(X\_test\_preprocessed, y\_test))



1. **Scale and add back to data frame(df) for modelling**

target = df['Y']

features = df.drop(columns='Y')

# Split the data

X\_train, X\_test, y\_train, y\_test = train\_test\_split(features, target, random\_state=20, test\_size=0.2)

# Create dummy variable for sex

ohe = OneHotEncoder(drop='first', sparse\_output=False)

train\_female = ohe.fit\_transform(X\_train[['SEX']]).flatten()

test\_female = ohe.transform(X\_test[['SEX']]).flatten()

# Initialize the scaler

scaler = StandardScaler()

# Scale every feature except the binary column - female

transformed\_training\_features = scaler.fit\_transform(X\_train.iloc[:,:-1])

transformed\_testing\_features = scaler.transform(X\_test.iloc[:,:-1])

# Convert the scaled features into a DataFrame

X\_train\_transformed = pd.DataFrame(scaler.transform(X\_train.iloc[:,:-1]),

                                   columns=X\_train.columns[:-1],

                                   index=X\_train.index)

X\_test\_transformed = pd.DataFrame(scaler.transform(X\_test.iloc[:,:-1]),

                                  columns=X\_train.columns[:-1],

                                  index=X\_test.index)

# Add binary column back in

X\_train\_transformed['female'] = train\_female

X\_test\_transformed['female'] = test\_female

X\_train\_transformed

poly = PolynomialFeatures(degree=2, interaction\_only=False, include\_bias=False)

X\_poly\_train = pd.DataFrame(poly.fit\_transform(X\_train\_transformed),

                            columns=poly.get\_feature\_names\_out(X\_train\_transformed.columns))

X\_poly\_test = pd.DataFrame(poly.transform(X\_test\_transformed),

                           columns=poly.get\_feature\_names\_out(X\_test\_transformed.columns))

X\_poly\_train.head()

1. **Logistic Regression using statsmodels**

relevant\_columns = ['Pclass', 'Age', 'SibSp', 'Fare', 'Sex', 'Embarked', 'Survived']

dummy\_dataframe = pd.get\_dummies(df[relevant\_columns],drop\_first=True,dtype=float)

y = dummy\_dataframe['Survived']

X = dummy\_dataframe.drop('Survived',axis=1)

import statsmodels.api as sm

# Create intercept term required for sm.Logit, see documentation for more information

X = sm.add\_constant(X)

# Fit model

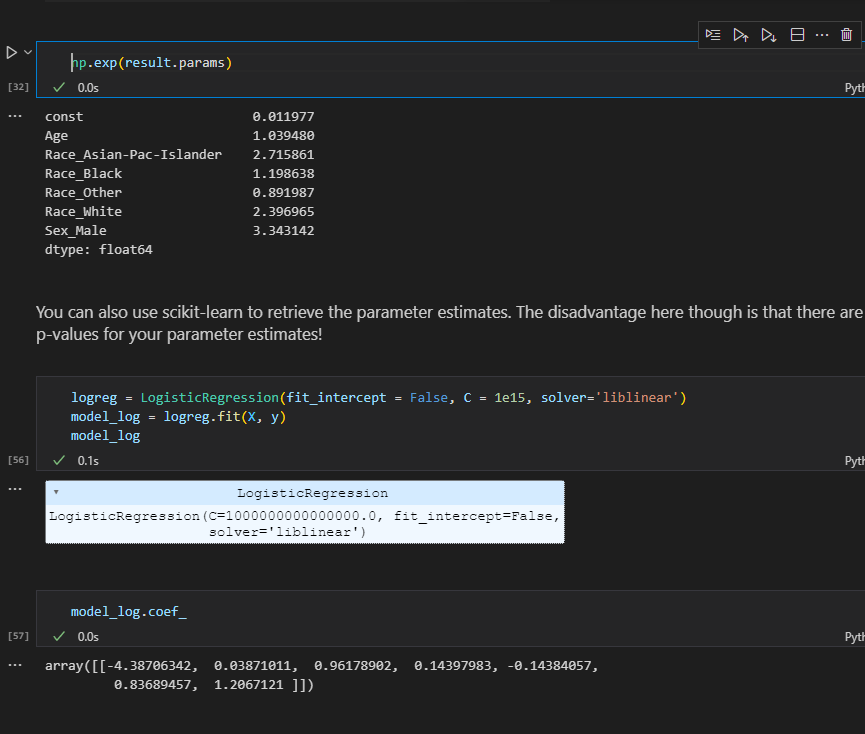
logit\_model = sm.Logit(y, X)

# Get results of the fit

result = logit\_model.fit()

**Get parameter estimates**

**np.exp(result.params)**



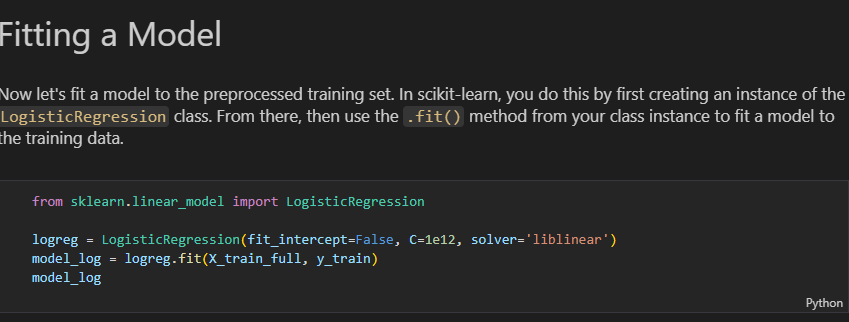
1. **Logistic Regression using scikit learn**

from sklearn.linear\_model import LogisticRegression

logreg = LogisticRegression(fit\_intercept=False, C=1e12, solver='liblinear')

model\_log = logreg.fit(X\_train\_full, y\_train)

model\_log



**MODEL EVALUATION**

**Train data**

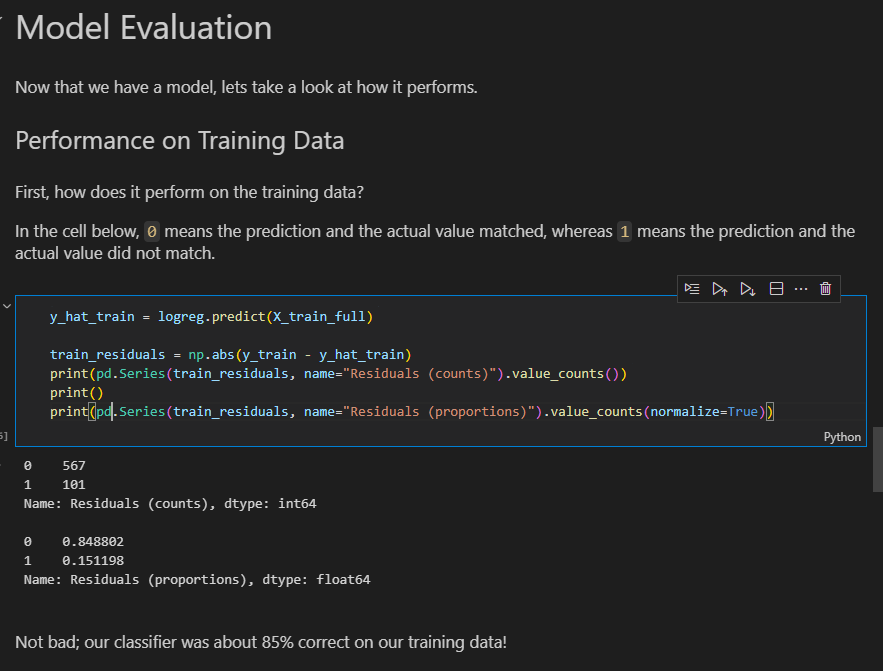
y\_hat\_train = logreg.predict(X\_train\_full)

train\_residuals = np.abs(y\_train - y\_hat\_train)

print(pd.Series(train\_residuals, name="Residuals (counts)").value\_counts())

print()

print(pd.Series(train\_residuals, name="Residuals (proportions)").value\_counts(normalize=True))



**Test Data**

# Filling in missing categorical data

X\_test\_fill\_na = X\_test.copy()

X\_test\_fill\_na.fillna({"Cabin":"cabin\_missing", "Embarked":"embarked\_missing"}, inplace=True)

# Filling in missing numeric data

test\_age\_imputed = pd.DataFrame(

    imputer.transform(X\_test\_fill\_na[["Age"]]),

    index=X\_test\_fill\_na.index,

    columns=["Age"]

)

X\_test\_fill\_na["Age"] = test\_age\_imputed

# Handling categorical data

X\_test\_categorical = X\_test\_fill\_na[categorical\_features].copy()

X\_test\_ohe = pd.DataFrame(

    ohe.transform(X\_test\_categorical),

    index=X\_test\_categorical.index,

    columns=np.hstack(ohe.categories\_)

)

# Normalization

X\_test\_numeric = X\_test\_fill\_na[numeric\_features].copy()

X\_test\_scaled = pd.DataFrame(

    scaler.transform(X\_test\_numeric),

    index=X\_test\_numeric.index,

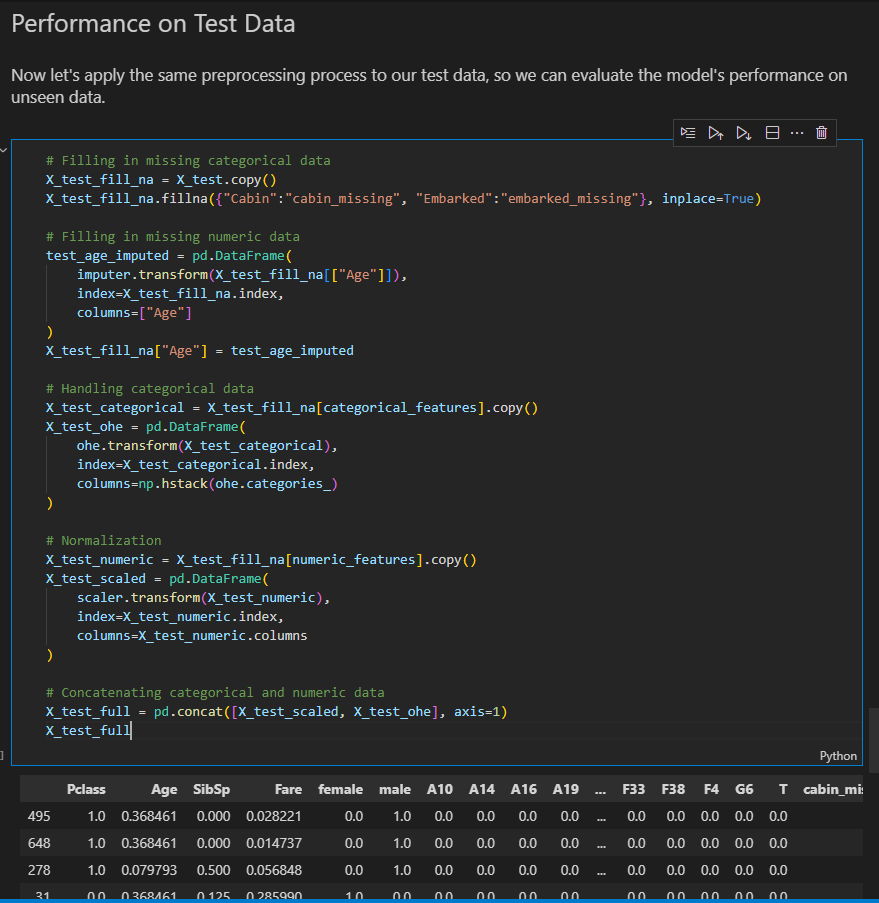
    columns=X\_test\_numeric.columns

)

# Concatenating categorical and numeric data

X\_test\_full = pd.concat([X\_test\_scaled, X\_test\_ohe], axis=1)

X\_test\_full



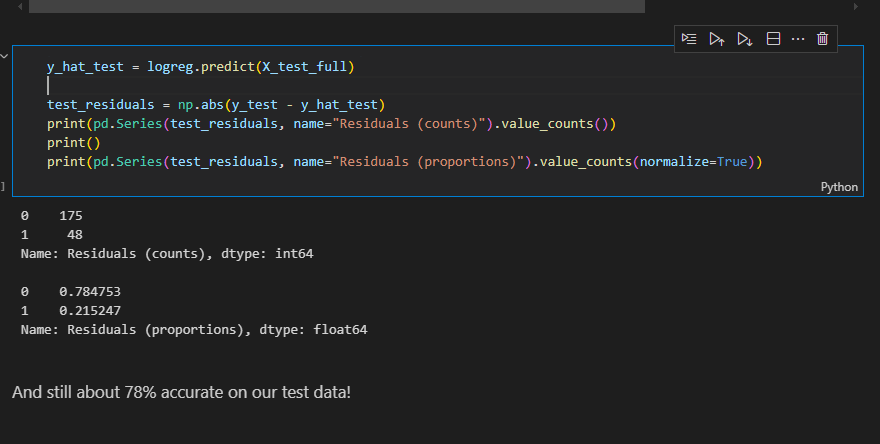
y\_hat\_test = logreg.predict(X\_test\_full)

test\_residuals = np.abs(y\_test - y\_hat\_test)

print(pd.Series(test\_residuals, name="Residuals (counts)").value\_counts())

print()

print(pd.Series(test\_residuals, name="Residuals (proportions)").value\_counts(normalize=True))



1. **Fill missing values for multiple columns**

X\_train\_fill\_na = X\_train.copy()

X\_train\_fill\_na.fillna({"Cabin":"cabin\_missing", "Embarked":"embarked\_missing"}, inplace=True)

X\_train\_fill\_na.isna().sum()

1. **Using Imputter to fill**

from sklearn.impute import SimpleImputer

imputer = SimpleImputer()

imputer.fit(X\_train\_fill\_na[["Age"]])

age\_imputed = pd.DataFrame(

    imputer.transform(X\_train\_fill\_na[["Age"]]),

    # index is important to ensure we can concatenate with other columns

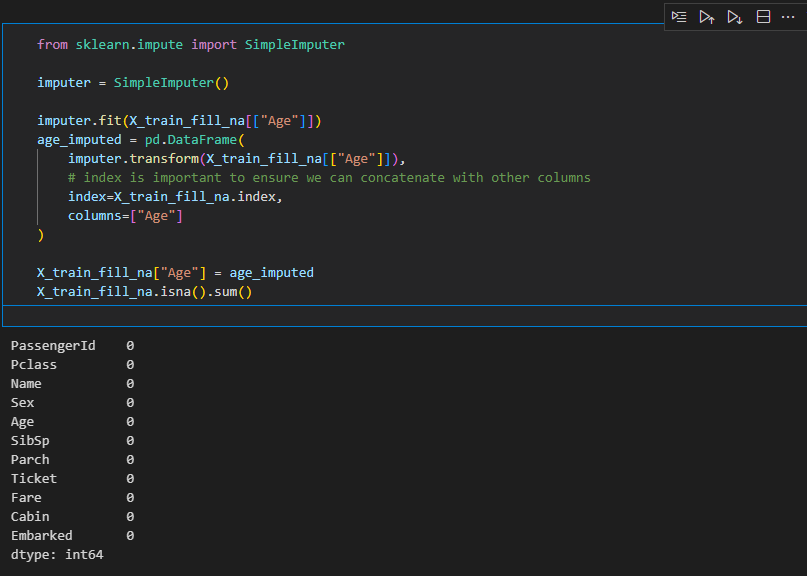
    index=X\_train\_fill\_na.index,

    columns=["Age"]

)

X\_train\_fill\_na["Age"] = age\_imputed

1. X\_train\_fill\_na.isna().sum()



1. **Select categorical columns only**

X\_train\_categorical = X\_train\_fill\_na.select\_dtypes(exclude=["int64", "float64"]).copy()

X\_train\_categorical

1. **Scaling with MinMax scaler**

from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaler.fit(X\_train\_numeric)

X\_train\_scaled = pd.DataFrame(

    scaler.transform(X\_train\_numeric),

    # index is important to ensure we can concatenate with other columns

    index=X\_train\_numeric.index,

    columns=X\_train\_numeric.columns

)

X\_train\_scaled

1. **Confusion Matrix**

X = df.drop(columns=["Survived"])

y = df["Survived"]

X\_encoded = pd.get\_dummies(X,columns=["Sex"],drop\_first=True,dtype=int)

model =  LogisticRegression()

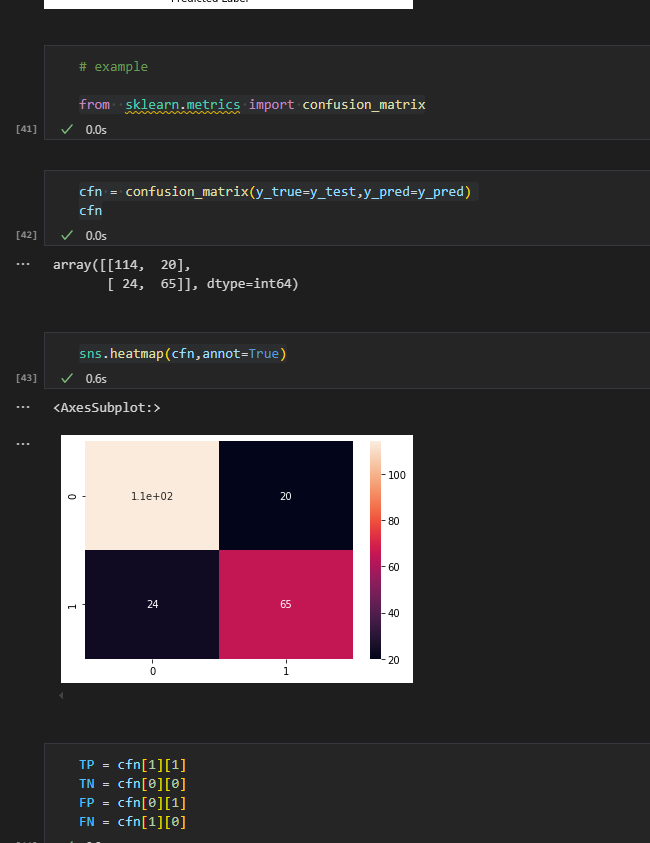
model.fit(X\_encoded\_train,y\_train)

y\_pred = model.predict(X\_encoded\_test)

from  sklearn.metrics import confusion\_matrix

cfn = confusion\_matrix(y\_true=y\_test,y\_pred=y\_pred)

sns.heatmap(cfn,annot=True)



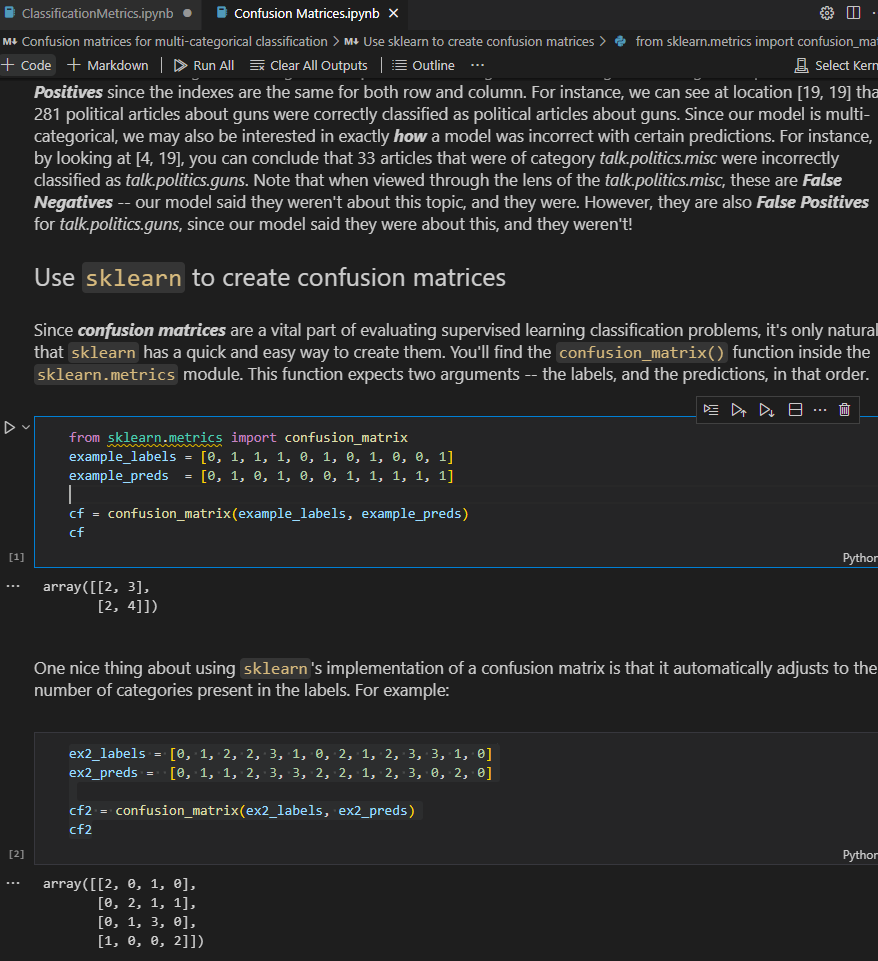
from sklearn.metrics import confusion\_matrix

example\_labels = [0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1]

example\_preds  = [0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1]

cf = confusion\_matrix(example\_labels, example\_preds)

cf

D

1. **Display confusion matrix**

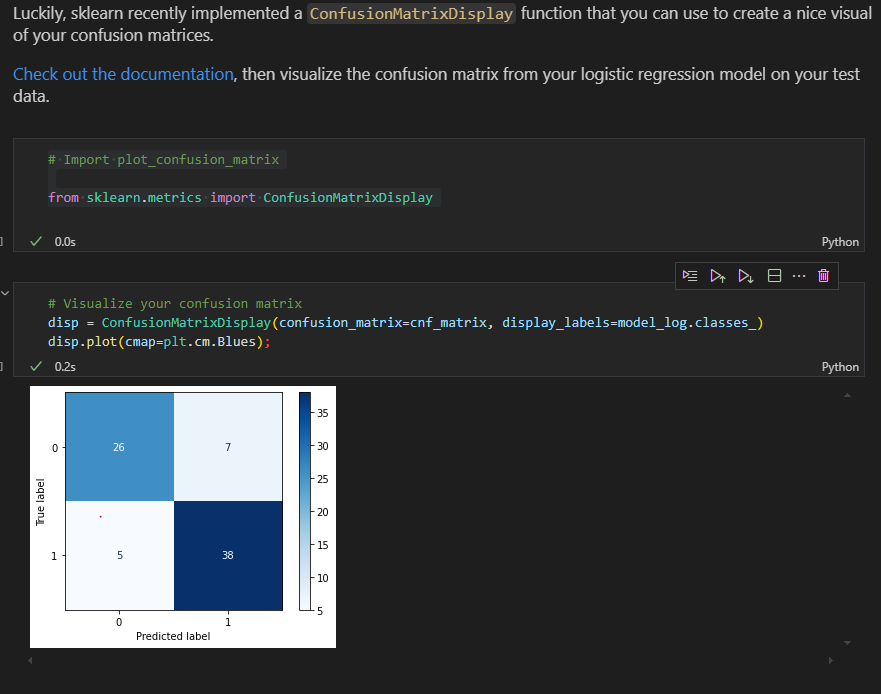
# Import plot\_confusion\_matrix

from sklearn.metrics import ConfusionMatrixDisplay

# Visualize your confusion matrix

disp = ConfusionMatrixDisplay(confusion\_matrix=cnf\_matrix, display\_labels=model\_log.classes\_)

disp.plot(cmap=plt.cm.Blues);

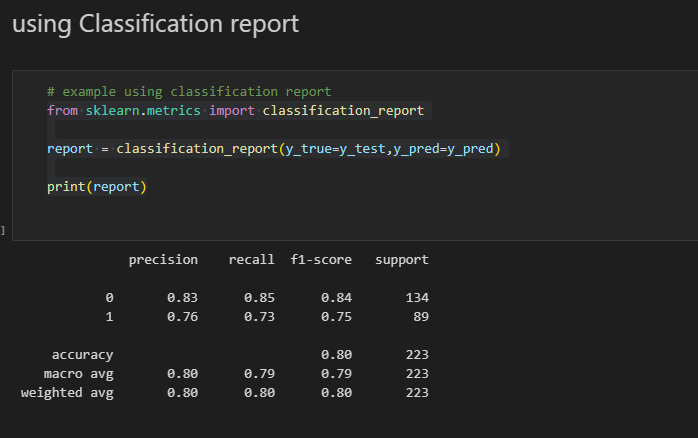


1. **Precision,Recall,f1-score using classification report**

from sklearn.metrics import classification\_report

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

print(report)



1. **ROC CURVE**

The **\*\*ROC (Receiver Operating Characteristic)\*\*** curve and **\*\*AUC (Area Under the Curve)\*\*** are used to evaluate the performance of a classification model, especially for binary classification problems.

- **\*\*ROC Curve\*\***: The ROC curve is a graphical representation of a model's performance at all classification thresholds. It plots the **\*\*True Positive Rate (Recall)\*\*** against the **\*\*False Positive Rate\*\***.

- **\*\*AUC (Area Under the Curve)\*\***: The AUC is the area under the ROC curve, and it quantifies the overall ability of the model to distinguish between the positive and negative classes.

from sklearn.metrics import roc\_curve

fpr1,tpr1,\_ = roc\_curve(y\_true=y\_test,y\_score=model.decision\_function(X\_encoded\_test))

sns.lineplot(x=fpr1,y=tpr1)



1. **AUC**

from sklearn.metrics import auc

area = auc(fpr1,tpr1)

area

**REAL LIFE EXAMPLE**

**from sklearn.linear\_model import LogisticRegression**

**from sklearn.model\_selection import train\_test\_split**

**import pandas as pd**

**# Load the data**

**df = pd.read\_csv('data/heart.csv')**

**# Define appropriate X and y**

**y = df['target']**

**X = df.drop(columns='target', axis=1)**

**# Normalize the Data**

**X = X.apply(lambda x : (x - x.min()) /(x.max() - x.min()),axis=0)**

**# Split the data into train and test sets.**

**X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, random\_state=0)**

**# Fit a model**

**logreg = LogisticRegression(fit\_intercept=False, C=1e12, solver='liblinear')**

**logreg.fit(X\_train, y\_train)**

**print(logreg) # Preview model params**

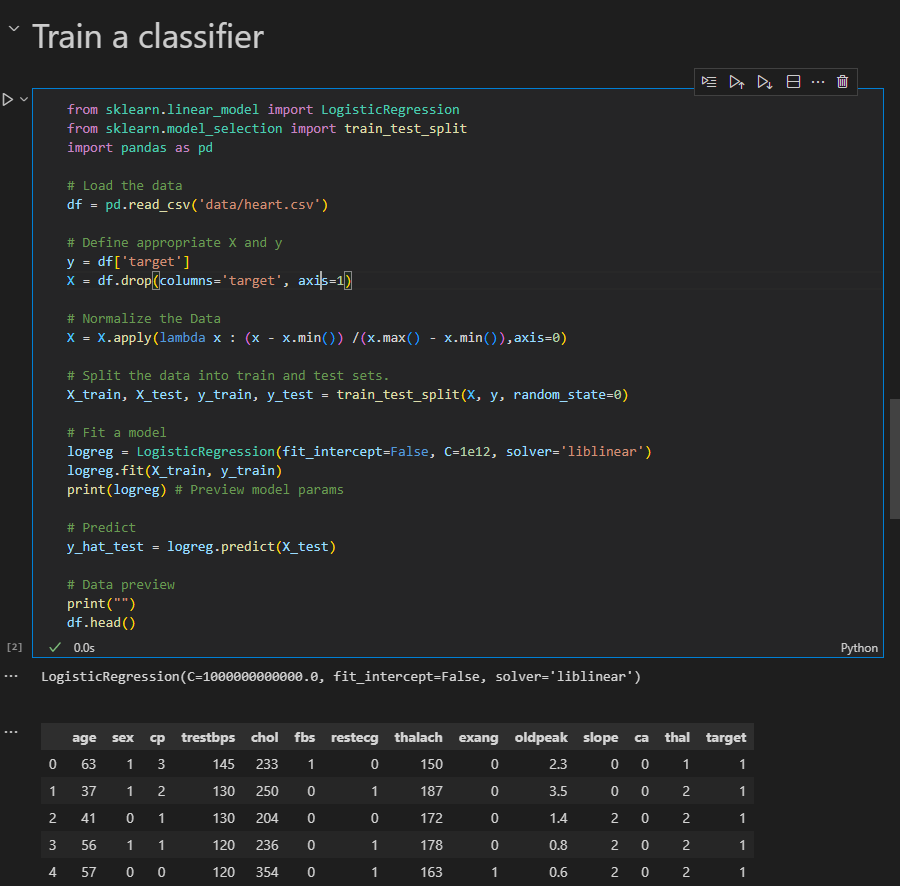
**# Predict**

**y\_hat\_test = logreg.predict(X\_test)**

**# Data preview**

**print("")**

**df.head()**



**Draw the AUC curve**

**from sklearn.metrics import roc\_curve, auc**

**# Scikit-learn's built in roc\_curve method returns the fpr, tpr, and thresholds**

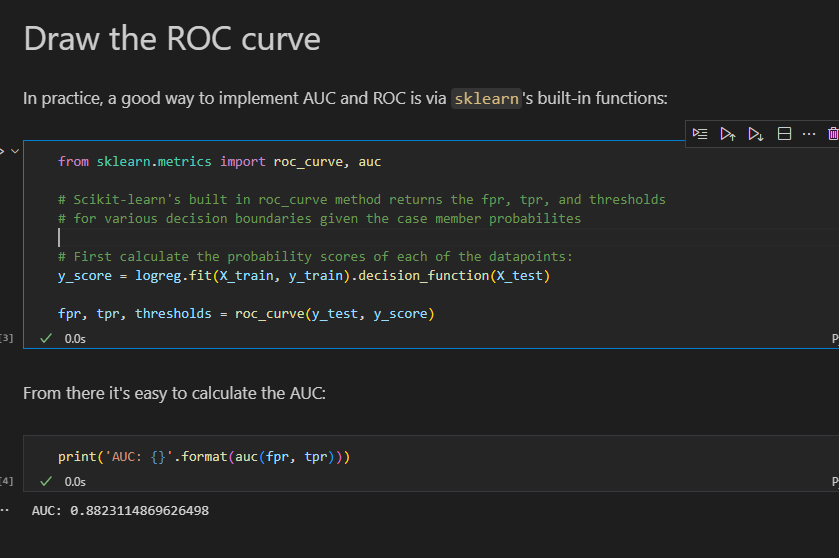
**# for various decision boundaries given the case member probabilites**

**# First calculate the probability scores of each of the datapoints:**

**y\_score = logreg.fit(X\_train, y\_train).decision\_function(X\_test)**

**fpr, tpr, thresholds = roc\_curve(y\_test, y\_score)**

**print('AUC: {}'.format(auc(fpr, tpr)))**



**Putting it all together**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**%matplotlib inline**

**# Seaborn's beautiful styling**

**sns.set\_style('darkgrid', {'axes.facecolor': '0.9'})**

**print('AUC: {}'.format(auc(fpr, tpr)))**

**plt.figure(figsize=(10, 8))**

**lw = 2**

**plt.plot(fpr, tpr, color='darkorange',**

**lw=lw, label='ROC curve')**

**plt.plot([0, 1], [0, 1], color='navy', lw=lw, linestyle='--')**

**plt.xlim([0.0, 1.0])**

**plt.ylim([0.0, 1.05])**

**plt.yticks([i/20.0 for i in range(21)])**

**plt.xticks([i/20.0 for i in range(21)])**

**plt.xlabel('False Positive Rate')**

**plt.ylabel('True Positive Rate')**

**plt.title('Receiver operating characteristic (ROC) Curve')**

**plt.legend(loc='lower right')**

**plt.show()**

