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:

```
==============================================================================
                       coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const              -18.3333      4.031     -4.548      0.000     -26.260     -10.407
weight              -0.0058      0.000    -22.015      0.000      -0.006      -0.005
model year           0.7516      0.049     15.266      0.000       0.655       0.848
origin_2             1.3483      1.199      1.124      0.262      -1.010       3.707
origin_3             1.9241      1.848      1.041      0.299      -1.710       5.559
make_buick           1.0599      1.024      1.036      0.301      -0.953       3.073
make_chevrolet       1.1589      0.795      1.459      0.146      -0.403       2.721
make_datsun          2.8444      2.030      1.402      0.162      -1.146       6.835
make_dodge           1.5970      0.891      1.792      0.074      -0.156       3.350
make_ford            0.5814      0.792      0.734      0.463      -0.975       2.138
make_honda           2.6716      2.117      1.262      0.208      -1.491       6.834
...
Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 8.49e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

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 MP

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:

2. 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()
```

```
X_centered = X_initial.copy()

for col in X_centered.columns:
    X_centered[col] = X_centered[col] - X_centered[col].mean()

X_centered.describe()
```

[26] ✓ 0.0s

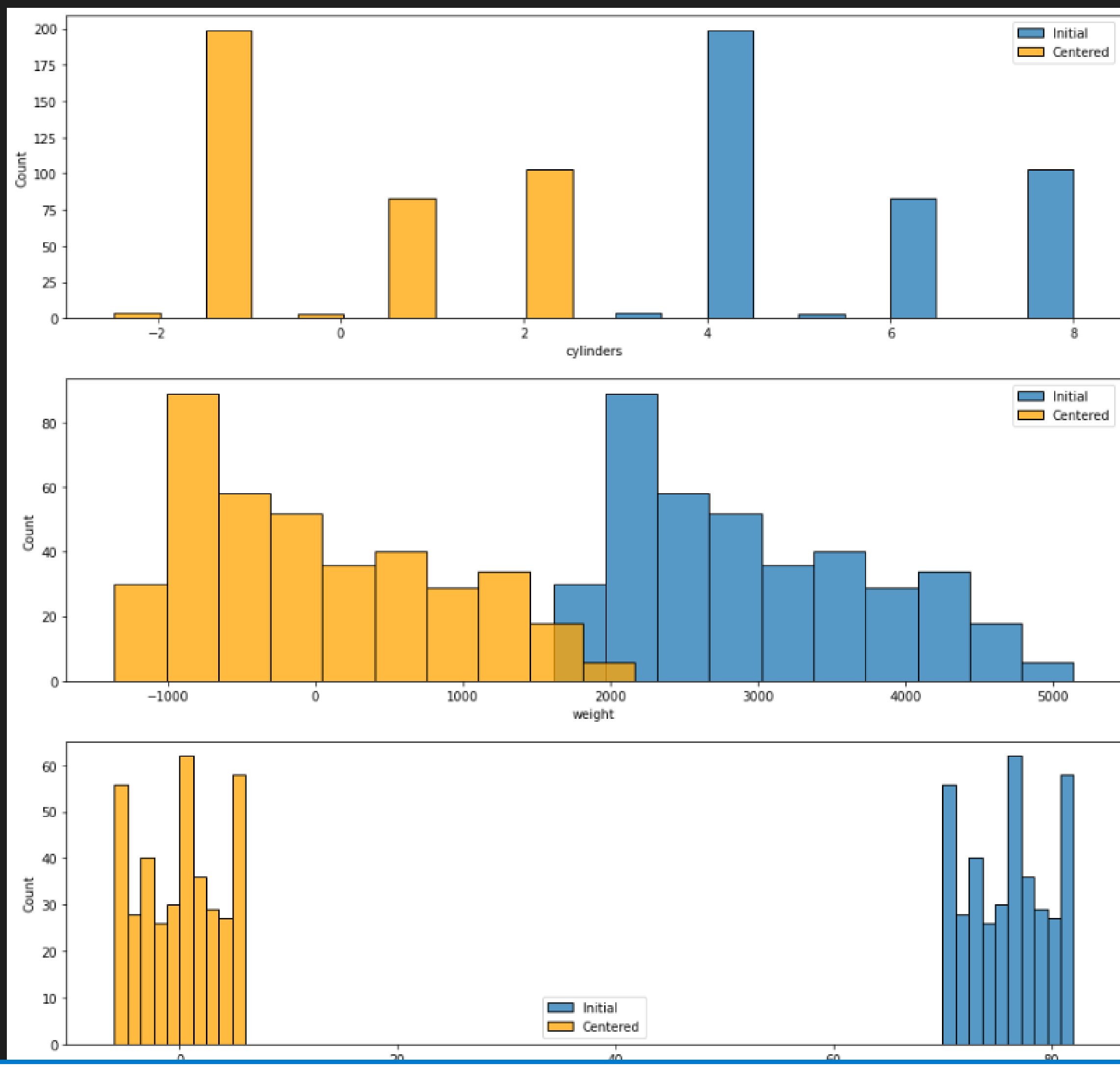
	cylinders	weight	model year
count	3.920000e+02	3.920000e+02	3.920000e+02
mean	-3.711317e-15	4.118248e-13	-4.640279e-15
std	1.705783e+00	8.494026e+02	3.683737e+00
min	-2.471939e+00	-1.364584e+03	-5.979592e+00
25%	-1.471939e+00	-7.523342e+02	-2.979592e+00
50%	-1.471939e+00	-1.740842e+02	2.040816e-02
75%	2.528061e+00	6.371658e+02	3.020408e+00
max	2.528061e+00	2.162416e+03	6.020408e+00

```
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()
```

Code + Markdown | Run All Restart Clear All Outputs | Variables Outline ··· learn-env (Python 3.8.5)

```python
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()
```

✓ 0.5s Python



On modelling we can now our coefficients are interpretable

```python
centered_model = sm.OLS(y_initial, sm.add_constant(X_centered))
centered_results = centered_model.fit()

print(f"""
Initial model adjusted R-Squared:  {initial_results.rsquared_adj}
Centered model adjusted R-Squared: {centered_results.rsquared_adj}
""")
```

[28] ✓ 0.0s Python

```
Initial model adjusted R-Squared:  0.8069069309563753
Centered model adjusted R-Squared: 0.8069069309563753

c:\Users\Gmwende\anaconda3\envs\learn-env\lib\site-packages\statsmodels\tsa\tsatools.py:142: FutureWarning: Ir
  x = pd.concat(x[::order], 1)
```

```python
initial_results.params
```

[29] ✓ 0.0s Python

```
const          -13.907606
cylinders       -0.151729
weight          -0.006366
model year       0.752020
dtype: float64
```

```python
centered_results.params
```

[30] ✓ 0.0s Python

```
const           23.445918
cylinders       -0.151729
weight          -0.006366
model year       0.752020
dtype: float64
```

As expected, our coefficients for the predictors are the same. For example, for each increase of 1 lb in weight, we see an associated decrease of about 0.006 in MPG.

However we now have a more meaningful intercept. In our initial model, the intercept interpretation was this:

> For a car with 0 cylinders, weighing 0 lbs, and built in 1900, we would expect an MPG of about -13.9

As expected, our coefficients for the predictors are the same. For example, for each increase of 1 lb in weight, we see an associated decrease of about 0.006 in MPG.

However we now have a more meaningful intercept. In our initial model, the intercept interpretation was this:

> For a car with 0 cylinders, weighing 0 lbs, and built in 1900, we would expect an MPG of about -13.9

That is an impossible MPG, for an impossible car.

In our zero-centered model, the intercept interpretation is this:

> For a car with the average number of cylinders, average weight, and average model year, we would expect an MPG of about 23.4

That makes a lot more sense! Now the intercept is something that might be worth reporting to stakeholders.

However you should also consider that this "average" car might be impossible as well. For example, if we look at the `cylinders` average, it is:

```python
data["cylinders"].mean()
```

[31] ✓ 0.0s Python

```
5.471938775510204
```

Can a car actually have 5.5 cylinders? Probably not! So this intercept interpretation is really only 100% realistic if all of the predictors are *continuous* variables. But you still may find it relevant for stakeholders, so long as you report it with the right caveats.

3. Standardization

Standardizing: Centering + Scaling

Standardization is a combination of zero-centering the variables and dividing by the standard deviation.

$$x' = \frac{x - \bar{x}}{\sigma}$$

After performing this transformation, x' will have mean of 0 and a standard deviation of 1.

```python
X_initial.describe()
```

[32] ✓ 0.0s

	cylinders	weight	model year
count	392.000000	392.000000	392.000000
mean	5.471939	2977.584184	75.979592
std	1.705783	849.402560	3.683737
min	3.000000	1613.000000	70.000000
25%	4.000000	2225.250000	73.000000
50%	4.000000	2803.500000	76.000000
75%	8.000000	3614.750000	79.000000
max	8.000000	5140.000000	82.000000

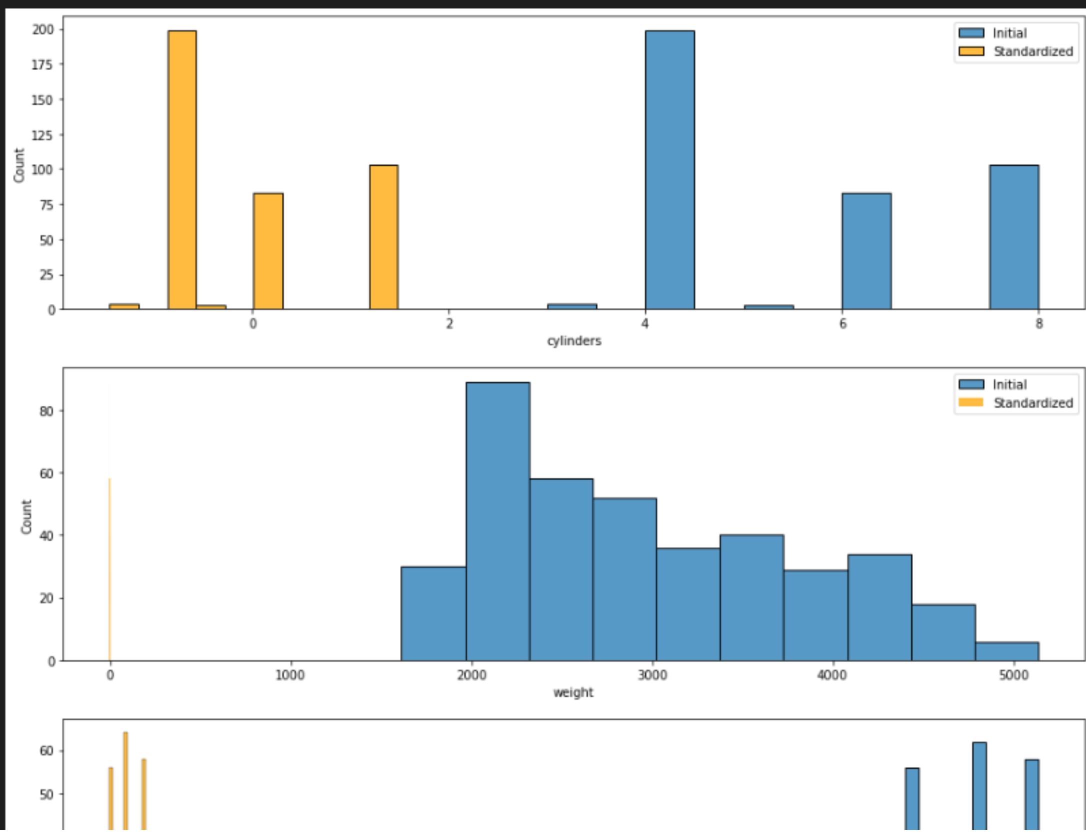
```python
X_standardized = X_initial.copy()

for col in X_standardized:
    X_standardized[col] = (X_standardized[col] - X_standardized[col].mean()) \
                          / X_standardized[col].std()


X_standardized.describe()
```

[33] ✓ 0.0s

	cylinders	weight	model year
count	3.920000e+02	3.920000e+02	3.920000e+02
mean	-4.316275e-16	1.281571e-16	3.002134e-16
std	1.000000e+00	1.000000e+00	1.000000e+00
min	1.449152e+00	1.606522e+00	1.633241e+00

```python
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_standardized,
        x=col,
        label="Standardized",
        color="orange",
        ax=axes[index]
    )
    axes[index].legend()
```

[34] ✓ 0.6s Python



In linear regression analysis, the most common reason for standardizing data is so that you can **compare the coefficients to each other**.

In our centered model, the coefficients are all using different units:

```
centered_results.params
```

[35] ✓ 0.0s Python

```
const          23.445918
cylinders      -0.151729
weight         -0.006366
model year      0.752020
dtype: float64
```

`model year` has the largest magnitude, but can we say that it "matters most" or "has the most impact"? Probably not, because it is measured in years whereas the other features are measured in cylinders and pounds. How can we compare those?

Standardization changes the units of the coefficients so that they are in standard deviations rather than the specific units of each predictor. This allows us to make just that comparison:

```
standardized_model = sm.OLS(y_initial, sm.add_constant(X_standardized))
standardized_results = standardized_model.fit()

print(f"""
Centered model adjusted R-Squared:     {centered_results.rsquared_adj}
Standardized model adjusted R-Squared: {standardized_results.rsquared_adj}
""")
```

[36] ✓ 0.0s Python

```
Centered model adjusted R-Squared:     0.8069069309563753
Standardized model adjusted R-Squared: 0.8069069309563753

c:\Users\Gmwende\anaconda3\envs\learn-env\lib\site-packages\statsmodels\tsa\tsatools.py:142: FutureWarning: In
  x = pd.concat(x[::order], 1)
```

```python
standardized_results.params
```

[37] ✓ 0.0s Python

```
const          23.445918
cylinders      -0.258817
weight         -5.407040
model year      2.770244
dtype: float64
```

We have the same intercept as the zero-centered model (since this model's features were also centered), but now the coefficients look quite different. We can interpret them like this:

> For each increase of 1 standard deviation in the number of cylinders, we see an associated decrease of about 0.26 MPG

> For each increase of 1 standard deviation in the weight, we see an associated decrease of about 5.4 MPG

> For each increase of 1 standard deviation in the model year, we see an associated increase of about 2.8 MPG

Comparing these three, we can conclude that `weight` "is the most important" or "has the most impact" because it has the largest coefficient. This might be surprising because the previous model had the *smallest* coefficient for `weight`, but that was because it was measured in pounds, with a much *larger* standard deviation than the other two predictors:

```python
data["weight"].std()
```

[38] ✓ 0.0s Python

```
849.4025600429494
```

```python
data["cylinders"].std()
```

[39] ✓ 0.0s Python

```
1.7057832474527843
```

```python
data["model year"].std()
```

[40] ✓ 0.0s Python

```
3.6837365435778318
```

(Every model is different, and sometimes the largest coefficient before standardizing will also be the largest after standardizing. This is just an example of how much of a difference it can make!)

Also, just like you can get transformed coefficients from un-transformed data by applying the inverse of the transformation, **you can get un-transformed coefficients from transformed data by applying the same transformation to the coefficient**.

For example, let's say you have this standardized model as your final model, because you knew that stakeholders would want to know which feature was most important:

```python
standardized_results.params
```

[41] ✓ 0.0s Python

```
const         23.445918
cylinders     -0.258817
weight        -5.407040
model year     2.770244
dtype: float64
```

You have answered the question about which is most important (`weight`) but now the stakeholder wants you to interpret the coefficient. You start to say "Each increase of 1 standard deviation..." but that is too confusing. A typical business stakeholder might not have a clear sense of what a "standard deviation" is.

Fortunately to get those coefficients to be the same as the un-transformed version (i.e. units of cylinders, pounds, and years respectively), just divide each of them by the standard deviation:

```python
standardized_results.params["cylinders"] / data["cylinders"].std()
```

[42] ✓ 0.0s Python

```
-0.15172901259380925
```

```python
standardized_results.params["weight"] / data["weight"].std()
```

[43] ✓ 0.0s Python

```
-0.006365697499915225
```

business stakeholder might not have a clear sense of what a "standard deviation" is.

Fortunately to get those coefficients to be the same as the un-transformed version (i.e. units of cylinders, pounds, and years respectively), just divide each of them by the standard deviation:

```python
standardized_results.params["cylinders"] / data["cylinders"].std()
```

✓ 0.0s Python

```
-0.15172901259380925
```

```python
standardized_results.params["weight"] / data["weight"].std()
```

✓ 0.0s Python

```
-0.006365697499915225
```

```python
standardized_results.params["model year"] / data["model year"].std()
```

✓ 0.0s Python

```
0.7520200488347166
```

These are now the same as the initial model params!

```python
initial_results.params[1:]
```

✓ 0.0s Python

```
cylinders    -0.151729
weight       -0.006366
model year    0.752020
dtype: float64
```

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()

```python
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()
# const         23.445918
# cylinders     -0.258817
# weight        -5.407040
# model year     2.770244
```

✓ 0.0s

OLS Regression Results

Dep. Variable:	mpg	R-squared:	0.808
Model:	OLS	Adj. R-squared:	0.807
Method:	Least Squares	F-statistic:	545.6
Date:	Sat, 07 Dec 2024	Prob (F-statistic):	8.73e-139
Time:	21:46:31	Log-Likelihood:	-1037.3
lo. Observations:	392	AIC:	2083.
Df Residuals:	388	BIC:	2099.
Df Model:	3		
Covariance Type:	nonrobust		

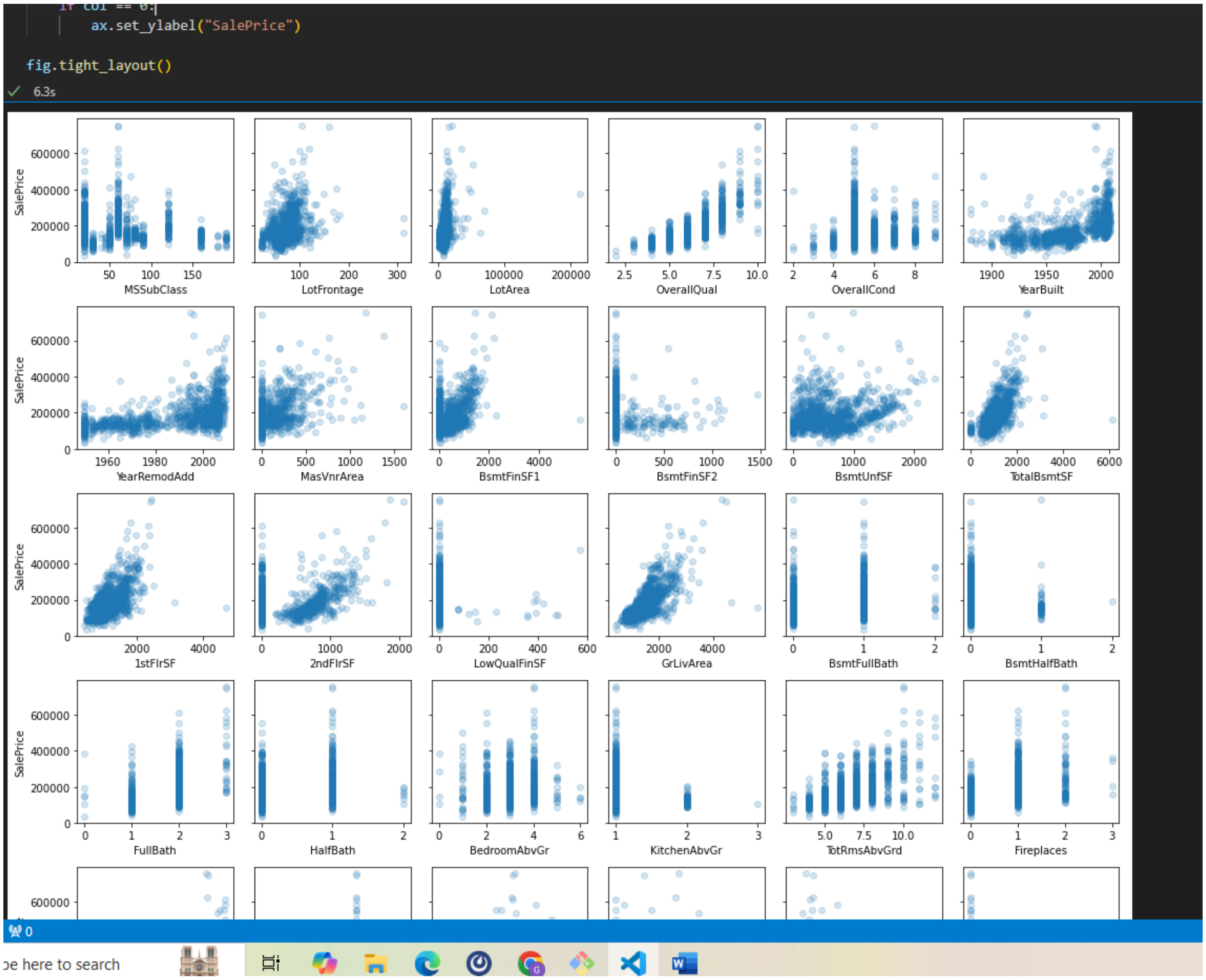
	coef	std err	t	P>|t|	[0.025	0.975]
onst	23.4459	0.173	135.349	0.000	23.105	23.786
x1	-0.2585	0.398	-0.649	0.517	-1.041	0.524
x2	-5.4001	0.393	-13.746	0.000	-6.173	-4.628
x3	2.7667	0.185	14.987	0.000	2.404	3.130

Omnibus:	43.326	Durbin-Watson:	1.231
rob(Omnibus):	0.000	Jarque-Bera (JB):	74.042
Skew:	0.678	Prob(JB):	8.35e-17
Kurtosis:	4.642	Cond. No.	4.54

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

```python
# 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()
```

Log Transformation

```
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()
```

[14] ✓ 0.0s

	log_disp	log_hp	log_wt
258	5.416100	4.700480	8.194229
182	4.941642	4.521789	7.852439
172	5.141664	4.574711	8.001020
63	5.762051	5.010635	8.327243
340	4.454347	4.158883	7.536364

```
#checking if same
np.log(X_train['displacement'])
```

[15] ✓ 0.0s

```
258    5.416100
182    4.941642
172    5.141664
63     5.762051
340    4.454347
         ...
71     5.717028
106    5.446737
270    5.017280
```

4. 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)
```

Test set

```
#encode test data
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()
```

	month_Apr	month_Aug	month_Dec	month_Feb	month_Jan	month_Jul	month_Jun	month_Mar	month_May	month_Nov	month_Oct	month_Sep
117	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
19	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
82	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
97	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
56	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

```
#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()
```

	year	month_Apr	month_Aug	month_Dec	month_Feb	month_Jan	month_Jul	month_Jun	month_Mar	month_May	month_Nov	month_Oct	month_Sep
117	1958	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
19	1950	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
82	1955	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
97	1957	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
56	1953	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

	month_Apr	month_Aug	month_Dec	month_Feb	month_Jan	month_Jul	month_Jun	month_Mar	month_May	month_Nov	month_Oct	month_Sep
117	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
19	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
82	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
97	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
56	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

```
#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()
```

	year	month_Apr	month_Aug	month_Dec	month_Feb	month_Jan	month_Jul	month_Jun	month_Mar	month_May	month_Nov	month_Oct	month_Sep
117	1958	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0
19	1950	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
82	1955	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0
97	1957	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
56	1953	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0

```
#Model score on Test
lr.score(df_test_concat,y_test)
```

```
0.9352318155740829
```

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()
```

One-Hot Encoding

```python
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()
```

[41] ✓ 0.0s Python

```
c:\Users\Gmwende\anaconda3\envs\learn-env\lib\site-packages\sklearn\preprocessing\_encoders.py:975: FutureWarnir
  warnings.warn(
```

	0	1
258	0.0	0.0
182	0.0	0.0
172	0.0	0.0
63	0.0	0.0
340	0.0	0.0

```python
# 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()
```

[42] ✓ 0.0s Python

	cylinders	acceleration	model year	log_disp	log_hp	log_wt	0	1
258	6	18.7	78	5.416100	4.700480	8.194229	0.0	0.0
182	4	14.9	76	4.941642	4.521789	7.852439	0.0	0.0

ENCODE TEST DATA AS WELL

```python
# 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()
```

```python
# Transform testing set
X_test_ohe = pd.DataFrame(ohe.transform(X_test[cat_columns]),
                          index=X_test.index)
X_test_ohe.head()
```

[43] ✓ 0.0s

	0	1
78	1.0	0.0
274	1.0	0.0
246	0.0	1.0
55	0.0	0.0
387	0.0	0.0

```python
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()
```

[44] ✓ 0.0s

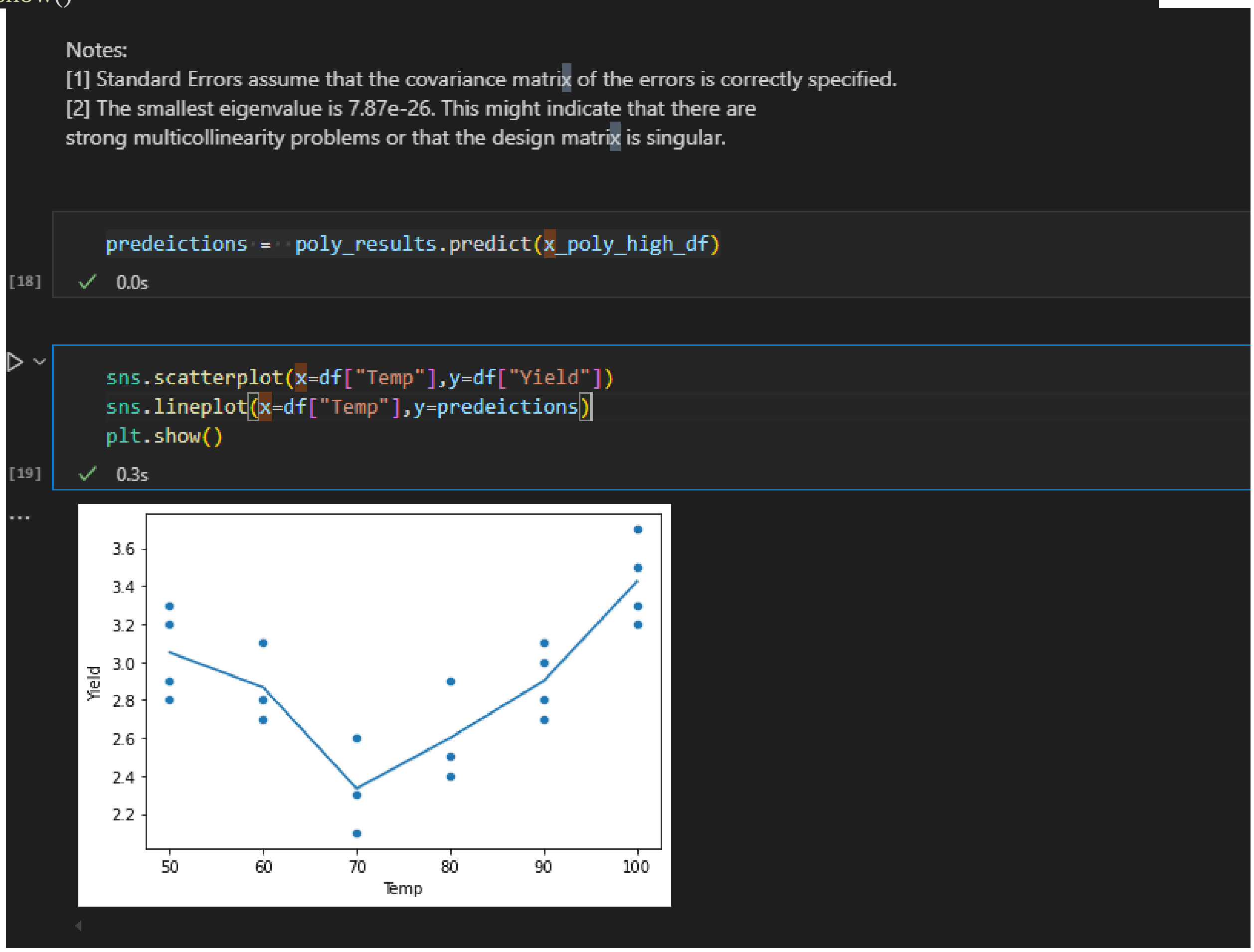
	cylinders	acceleration	model year	log_disp	log_hp	log_wt	0	1
78	4	18.0	72	4.564348	4.234107	7.691200	1.0	0.0
274	4	15.7	78	4.795791	4.744932	7.935587	1.0	0.0
246	4	16.4	78	4.510860	4.094345	7.495542	0.0	1.0
55	4	20.5	71	4.510860	4.248495	7.578145	0.0	0.0
387	4	15.6	82	4.941642	4.454347	7.933797	0.0	0.0

Building, Evaluating, and Validating a Model

5. POLYNOMIALS

```python
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()
```

```python
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()
```



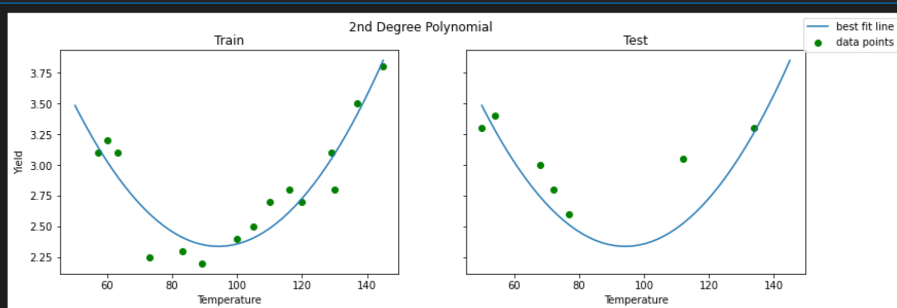
POLYNOMIAL WITH LINEAR REGRESSION 1

```python
# 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');
```

```
# 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)))}
```

That looks like a more reasonable model. Let's look at the MSE scores:

```python
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)))}
""")
```

Python

```
Simple Linear Regression
Train MSE: 0.18250661207533306
Test MSE:  0.21472110882651188

6th Degree Polynomial
Train MSE: 0.010446614261387478
Test MSE:  0.3871286896932373

2nd Degree Polynomial
Train MSE: 0.02786876093087891
Test MSE:  0.0612423773614243
```

The fit for the training set became worse with the 2nd degree polynomial compared to the 6th degree polynomial, but we can clearly see how the test set performance improved by looking at the mean squared error. Also, it seems like the results for training and test set are comparable, which is what you would want in general.

Based on this analysis, it appears that a 2nd degree polynomial achieves the correct degree of balance between underfitting (like the simple linear regression) and overfitting (like the 6th degree polynomial).

POLYNOMIAL WITH LINEAR REGRESSION2

```python
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)
```

Use the model to make predictions on both the training and test sets:

```python
# Training set predictions
poly_train_predictions = polyreg.predict(X_train_poly)

# Test set predictions
poly_test_predictions = polyreg.predict(X_test_poly)
```

[111] ✓ 0.0s

Plot predictions for the training set against the actual data:

```python
# Run this cell - vertical distance between the points and the line denote the errors
plt.figure(figsize=(8, 5))
plt.scatter(y_train, poly_train_predictions, label='Model')
plt.plot(y_train, y_train, label='Actual data')
plt.title('Model vs data for training set')
plt.legend();
```

[112] ✓ 0.1s

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

```python
# 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)
```

Building, Evaluating, and Validating a Model

Great, now that you have preprocessed all the columns, you can fit a linear regression 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)
```

[45] ✓ 0.0s Python

```
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)
```

[46] ✓ 0.0s Python

Look at the residuals and calculate the MSE for training and test sets:

```
train_residuals = y_hat_train - y_train
test_residuals = y_hat_test - y_test
```

[47] ✓ 0.0s Python

You can also do this directly using sklearn's `mean_squared_error()` function:

```
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)
```

[49] ✓ 0.0s Python

```
Train Mean Squared Error: 9.091818811315937
Test Mean Squared Error: 10.010059484009506
```

Great, there does not seem to be a big difference between the train and test MSE!

In other words, our evaluation process has indicated that we are **not** overfitting. In fact, we may be *underfitting* because linear regression is not a very complex model.

In other words, our evaluation process has indicated that we are **not** overfitting. In fact, we may be *underfitting* because linear regression is not a very complex model.

Overfitting with a Different Model

Just for the sake of example, here is a model that is overfit to the data. Don't worry about the model algorithm being shown! Instead, just look at the MSE for the train vs. test set, using the same preprocessed data:

```python
from sklearn.tree import DecisionTreeRegressor

other_model = DecisionTreeRegressor(random_state=42)
other_model.fit(X_train_tr, y_train)

other_train_mse = mean_squared_error(y_train, other_model.predict(X_train_tr))
other_test_mse = mean_squared_error(y_test, other_model.predict(X_test_tr))
print('Train Mean Squared Error:', other_train_mse)
print('Test Mean Squared Error:', other_test_mse)
```

[50] ✓ 0.0s Python

```
Train Mean Squared Error: 0.0
Test Mean Squared Error: 11.403164556962025
```

This model initially seems great...0 MSE for the training data! But then you see that it is performing worse than our linear regression model on the test data. This model **is** overfitting.

7. R2 SCORE AND MEAN SQUARED ERROR

```python
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)
```

```python
# Testing for overfit or underfit using linear regression
def linear_regression_intro(X_train, X_test, y_train, y_test):
    print("\n### Simple Linear Regression model ###")

    # Initialize and train the model
    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)

    print(f"Training MSE: {train_mse}")
    print(f"Testing MSE: {test_mse}")
    print(f"Training R^2 Score: {train_r2}")
    print(f"Testing R^2 Score: {test_r2}")

    # Visualization
    plt.figure(figsize=(10, 6))
    plt.scatter(y_test, y_test_pred, edgecolors=(0, 0, 0), label='Test Data')
    plt.scatter(y_train, y_train_pred, edgecolors=(0, 0, 0), label='Train Data', alpha=0.5)
    plt.plot([min(y_test), max(y_test)], [min(y_test), max(y_test)], 'r', lw=2)
    plt.xlabel('Actual')
    plt.ylabel('Predicted')
    plt.title('Linear Regression: Actual vs Predicted')
    plt.legend()
    plt.show()

linear_regression_intro(X_train, X_test, y_train, y_test)
```

8. Splitting data

```python
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)
```

9. Log transformation and one hot encoding together

```python
# 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)
```

10. CROSS VALIDATION

11.
```
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"])
```

12. <u>get mean of all the cross validation scores</u>

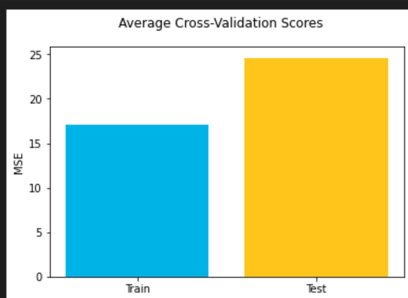
```
cross_val_results = cross_validate(linreg, X, y, scoring="neg_mean_squared_error",
return_train_score=True)
# Negative signs in front to convert back to MSE from -MSE
train_avg = -cross_val_results["train_score"].mean()
test_avg = -cross_val_results["test_score"].mean()

fig, ax = plt.subplots()
ax.bar(labels, [train_avg, test_avg], color=colors)
ax.set_ylabel("MSE")
fig.suptitle("Average Cross-Validation Scores");
```

Reporting Cross-Validation Scores

Often your stakeholders will want a single metric or visualization that represents model performance, not a list of scores like cross-validation produces.

One straightforward way to achieve this is to take the average:

```
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")

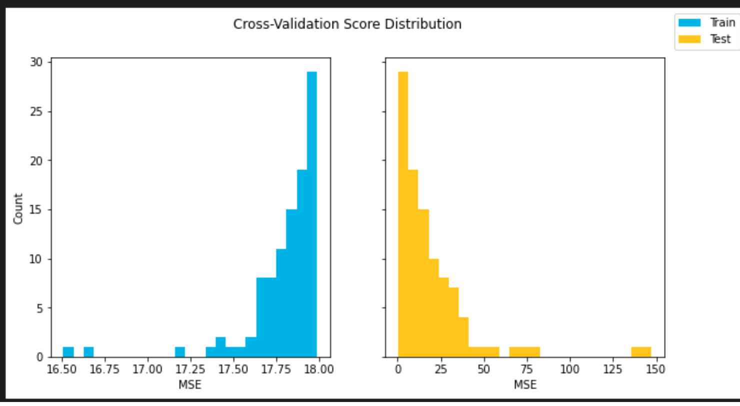
right.hist(test_scores, label=labels[1], bins=bins, color=colors[1])

right.set_xlabel("MSE")

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

fig.legend();

```
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();
```



13. 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']
```

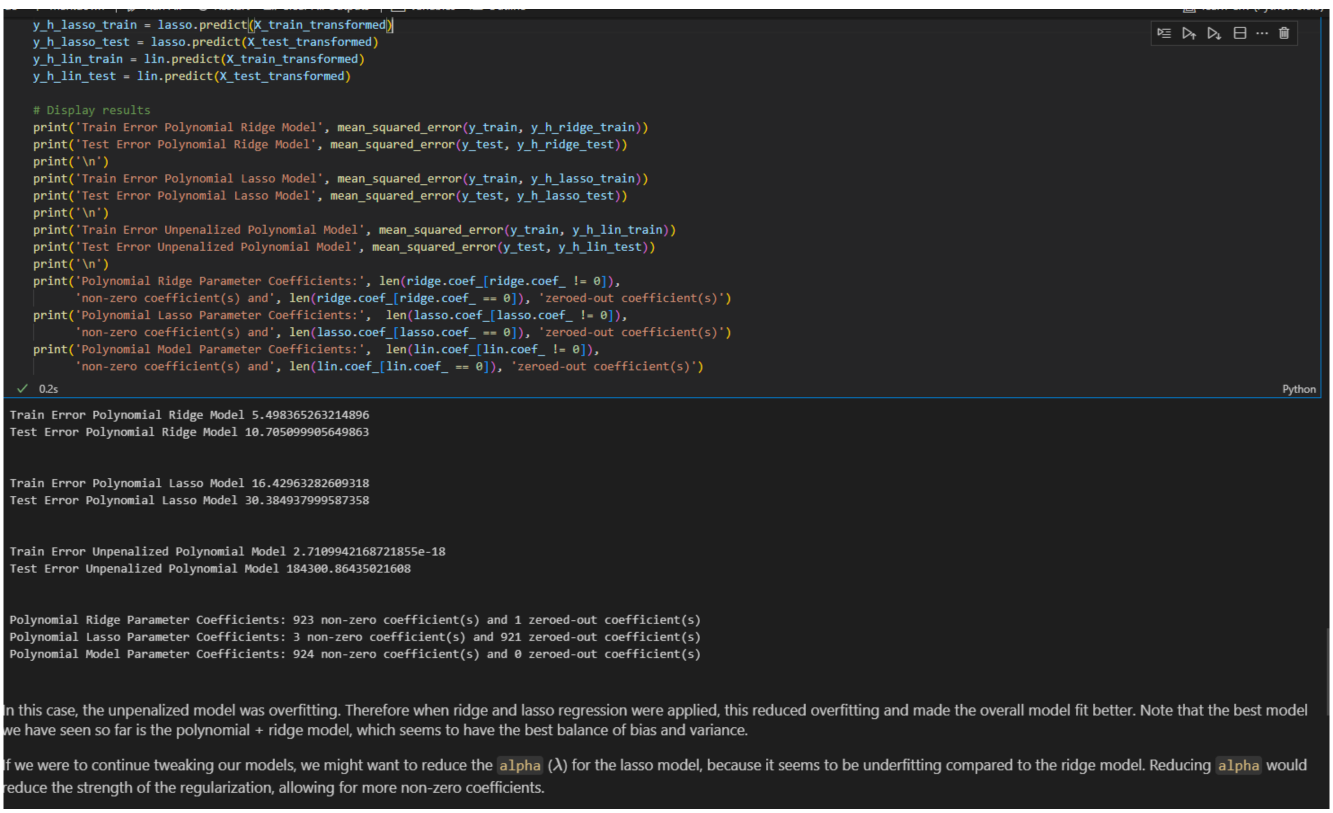
14. Another example of one hot encoding

```
ohe = OneHotEncoder(drop='first', sparse_output=False)
train_female = ohe.fit_transform(X_train[['SEX']]).flatten()
test_female = ohe.transform(X_test[['SEX']]).flatten()
```

15. 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)')
```



```
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)')
```

✓ 0.2s Python

```
Train Error Polynomial Ridge Model 5.498365263214896
Test Error Polynomial Ridge Model 10.705099905649863


Train Error Polynomial Lasso Model 16.42963282609318
Test Error Polynomial Lasso Model 30.384937999587358


Train Error Unpenalized Polynomial Model 2.7109942168721855e-18
Test Error Unpenalized Polynomial Model 184300.86435021608


Polynomial Ridge Parameter Coefficients: 923 non-zero coefficient(s) and 1 zeroed-out coefficient(s)
Polynomial Lasso Parameter Coefficients: 3 non-zero coefficient(s) and 921 zeroed-out coefficient(s)
Polynomial Model Parameter Coefficients: 924 non-zero coefficient(s) and 0 zeroed-out coefficient(s)
```

In this case, the unpenalized model was overfitting. Therefore when ridge and lasso regression were applied, this reduced overfitting and made the overall model fit better. Note that the best model we have seen so far is the polynomial + ridge model, which seems to have the best balance of bias and variance.

If we were to continue tweaking our models, we might want to reduce the `alpha` (λ) 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.

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.

16. Getting r squared

```python
print('Training r^2:', linreg.score(X_train_preprocessed, y_train))
print('Test r^2:    ', linreg.score(X_test_preprocessed, y_test))
```

```python
# Replace None with appropriate code
from sklearn.metrics import mean_squared_error
from sklearn.linear_model import LinearRegression

# Fit the model
linreg = LinearRegression()
linreg.fit(X_train_preprocessed,y_train)

# Print R2 and MSE for training and test sets
train_mse = mean_squared_error(y_train,linreg.predict(X_train_preprocessed))
test_mse  = mean_squared_error(y_test,linreg.predict(X_test_preprocessed))
print(f'train_mse {train_mse}')
print(f'test_mse {test_mse}')
print('Training r^2:', linreg.score(X_train_preprocessed, y_train))
print('Test r^2:    ', linreg.score(X_test_preprocessed, y_test))
```

✓ 0.0s Python

```
train_mse 402648264.5988585
test_mse 1.1573196494784997e+30
Training r^2: 0.9358944418347754
Test r^2:     -1.8139280124958058e+20
```

Notice the severe overfitting above; our training R-Squared is very high, but the test R-Squared is negative! Similarly, the scale of the test MSE is orders of magnitude higher than that of the training MSE.

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

```python
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()
```

18. 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()
```

<u>Get parameter estimates</u>

np.exp(result.params)

```
np.exp(result.params)
```

[32] 0.0s

```
const                       0.011977
Age                         1.039480
Race_Asian-Pac-Islander     2.715861
Race_Black                  1.198638
Race_Other                  0.891987
Race_White                  2.396965
Sex_Male                    3.343142
dtype: float64
```

You can also use scikit-learn to retrieve the parameter estimates. The disadvantage here though is that there are p-values for your parameter estimates!

```
logreg = LogisticRegression(fit_intercept = False, C = 1e15, solver='liblinear')
model_log = logreg.fit(X, y)
model_log
```

[56] 0.1s

```
LogisticRegression
LogisticRegression(C=1000000000000000.0, fit_intercept=False,
                   solver='liblinear')
```

```
model_log.coef_
```

[57] 0.0s

```
array([[-4.38706342,  0.03871011,  0.96178902,  0.14397983, -0.14384057,
         0.83689457,  1.2067121 ]])
```

19. 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
```

Fitting a Model

Now let's fit a model to the preprocessed training set. In scikit-learn, you do this by first creating an instance of the `LogisticRegression` class. From there, then use the `.fit()` method from your class instance to fit a model to the training data.

```
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
```

Python

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))
```

Model Evaluation

Now that we have a model, lets take a look at how it performs.

Performance on Training Data

First, how does it perform on the training data?

In the cell below, `0` means the prediction and the actual value matched, whereas `1` means the prediction and the actual value did not match.

```python
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))
```

Python

```
0    567
1    101
Name: Residuals (counts), dtype: int64

0    0.848802
1    0.151198
Name: Residuals (proportions), dtype: float64
```

Not bad; our classifier was about 85% correct on our training data!

Test Data

```python
# 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
```

Performance on Test Data

Now let's apply the same preprocessing process to our test data, so we can evaluate the model's performance on unseen data.

```python
# 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
```

Python

	Pclass	Age	SibSp	Fare	female	male	A10	A14	A16	A19	...	F33	F38	F4	G6	T	cabin_mi
495	1.0	0.368461	0.000	0.028221	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	
648	1.0	0.368461	0.000	0.014737	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	
278	1.0	0.079793	0.500	0.056848	0.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	
31	0.0	0.368461	0.125	0.285990	1.0	0.0	0.0	0.0	0.0	0.0		0.0	0.0	0.0	0.0	0.0	

```python
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))
```

```python
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))
```

Python

```
0    175
1     48
Name: Residuals (counts), dtype: int64

0    0.784753
1    0.215247
Name: Residuals (proportions), dtype: float64
```

And still about 78% accurate on our test data!

20. Fill missing values for multiple columns

```python
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()
```

21. Using Imputter to fill

```python
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
22.  X_train_fill_na.isna().sum()
```

```python
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
X_train_fill_na.isna().sum()
```

```
PassengerId    0
Pclass         0
Name           0
Sex            0
Age            0
SibSp          0
Parch          0
Ticket         0
Fare           0
Cabin          0
Embarked       0
dtype: int64
```

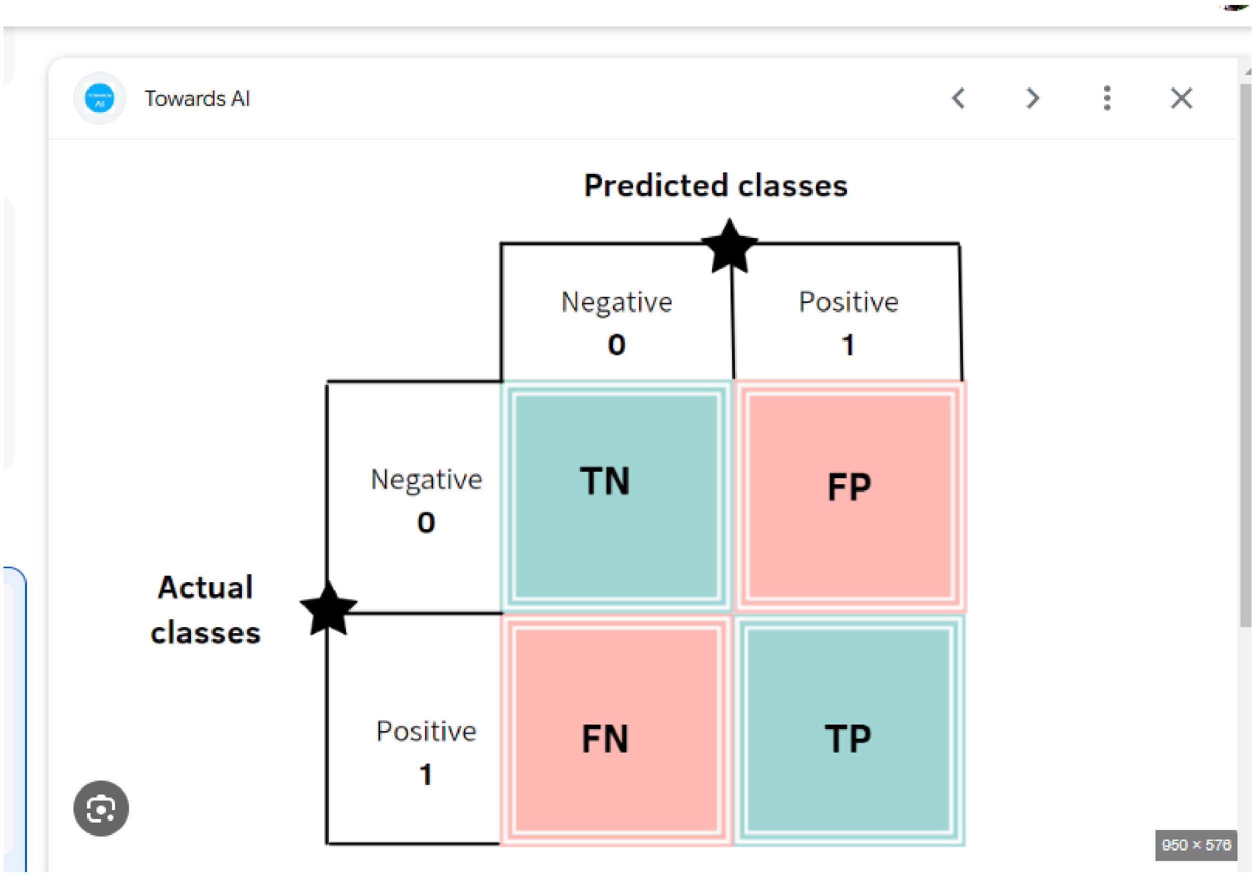
23. Select categorical columns only

```python
X_train_categorical = X_train_fill_na.select_dtypes(exclude=["int64", "float64"]).copy()
X_train_categorical
```

24. Scaling with MinMax scaler

```python
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
```

25. 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)
```

26. Confusion matrix 2

```
pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'],
margins=True)
```

```
#teachers code
pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'], margins=True)
```

✓ 0.2s

Predicted	0	1	All
True			
0	149	3	152
1	3	120	123
All	152	123	275

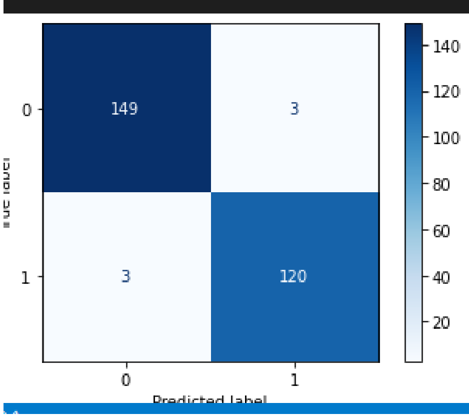
27. Confusion matrix 3

```
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

cnf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cnf_matrix,
display_labels=clf.classes_)
disp.plot(cmap=plt.cm.Blues)
```

```
# Alternative confusion matrix
from sklearn.metrics import ConfusionMatrixDisplay, confusion_matrix

cnf_matrix = confusion_matrix(y_test, y_pred)
disp = ConfusionMatrixDisplay(confusion_matrix=cnf_matrix, display_labels=clf.classes_)
disp.plot(cmap=plt.cm.Blues)
```

0.2s Pyth

```
klearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x210e8a26dc0>
```



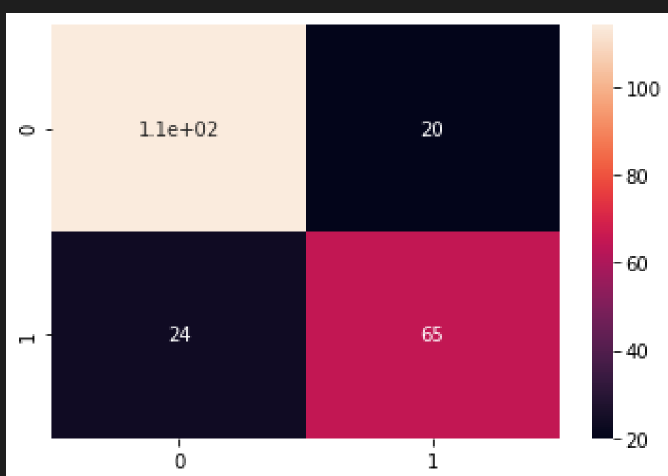
```
# example

from  sklearn.metrics import confusion_matrix
```
[41] ✓ 0.0s

```
cfn = confusion_matrix(y_true=y_test,y_pred=y_pred)
cfn
```
[42] ✓ 0.0s

```
array([[114,  20],
       [ 24,  65]], dtype=int64)
```

```
sns.heatmap(cfn,annot=True)
```
[43] ✓ 0.6s

```
<AxesSubplot:>
```



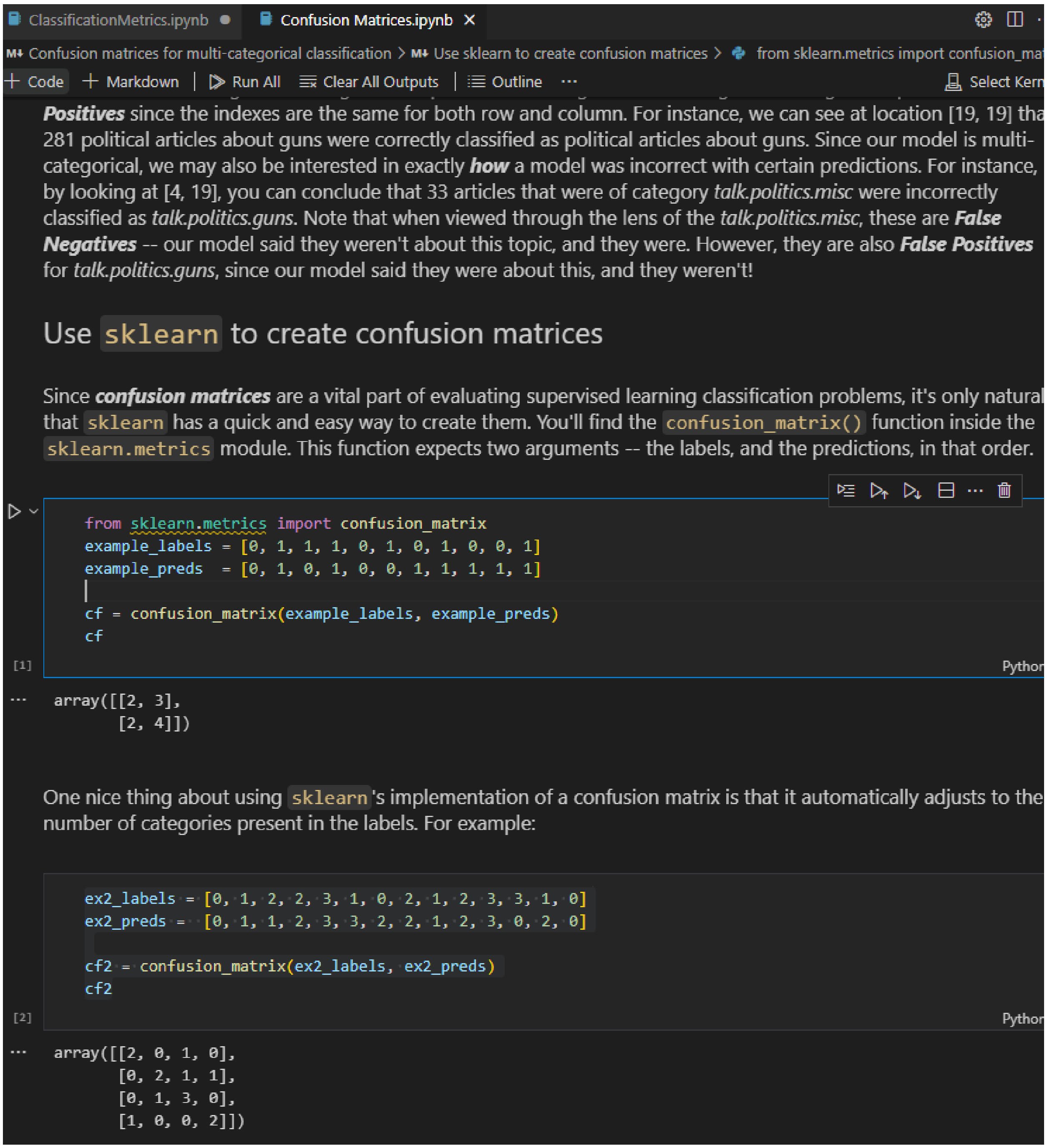

```
TP = cfn[1][1]
TN = cfn[0][0]
FP = cfn[0][1]
FN = cfn[1][0]
```

```
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
```

28.

D



ClassificationMetrics.ipynb ● Confusion Matrices.ipynb ×

M↓ Confusion matrices for multi-categorical classification > M↓ Use sklearn to create confusion matrices > from sklearn.metrics import confusion_ma

+ Code + Markdown | Run All Clear All Outputs | Outline ··· Select Kern

Positives since the indexes are the same for both row and column. For instance, we can see at location [19, 19] tha 281 political articles about guns were correctly classified as political articles about guns. Since our model is multi-categorical, we may also be interested in exactly ***how*** a model was incorrect with certain predictions. For instance, by looking at [4, 19], you can conclude that 33 articles that were of category *talk.politics.misc* were incorrectly classified as *talk.politics.guns*. Note that when viewed through the lens of the *talk.politics.misc*, these are ***False Negatives*** -- our model said they weren't about this topic, and they were. However, they are also ***False Positives*** for *talk.politics.guns*, since our model said they were about this, and they weren't!

Use `sklearn` to create confusion matrices

Since ***confusion matrices*** are a vital part of evaluating supervised learning classification problems, it's only natural that `sklearn` has a quick and easy way to create them. You'll find the `confusion_matrix()` function inside the `sklearn.metrics` module. This function expects two arguments -- the labels, and the predictions, in that order.

```
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
```

[1] Python

```
array([[2, 3],
       [2, 4]])
```

One nice thing about using `sklearn`'s implementation of a confusion matrix is that it automatically adjusts to the number of categories present in the labels. For example:

```
ex2_labels = [0, 1, 2, 2, 3, 1, 0, 2, 1, 2, 3, 3, 1, 0]
ex2_preds =  [0, 1, 1, 2, 3, 3, 2, 2, 1, 2, 3, 0, 2, 0]

cf2 = confusion_matrix(ex2_labels, ex2_preds)
cf2
```

[2] Python

```
array([[2, 0, 1, 0],
       [0, 2, 1, 1],
       [0, 1, 3, 0],
       [1, 0, 0, 2]])
```

29. Display confusion matrix

```python
# 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);
```

30. Precision,Recall,f1-score using classification report

```python
from sklearn.metrics import classification_report

report = classification_report(y_true=y_test,y_pred=y_pred)

print(report)
```

using Classification report

```
# example using classification report
from sklearn.metrics import classification_report

report = classification_report(y_true=y_test,y_pred=y_pred)

print(report)
```

```
              precision    recall  f1-score   support

           0       0.83      0.85      0.84       134
           1       0.76      0.73      0.75        89

    accuracy                           0.80       223
   macro avg       0.80      0.79      0.79       223
weighted avg       0.80      0.80      0.80       223
```

31. 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)
```

- The ROC curve is plotted by calculating TPR and FPR at various threshold values, and plotting TPR (y-axis) vs FPR (x-axis).

AUC:

The AUC is the area under the ROC curve, which quantifies the model's ability to distinguish between positive and negative classes.

```
# example
from sklearn.metrics import roc_curve
```

[28] ✓ 0.0s Python

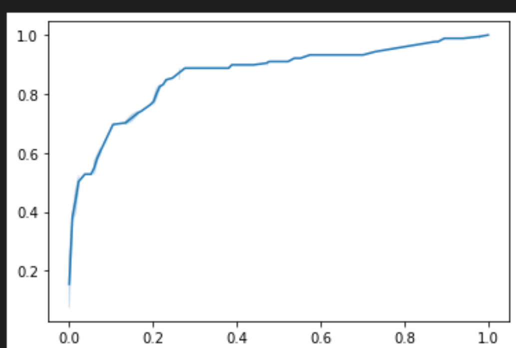
```
fpr1,tpr1,_ = roc_curve(y_true=y_test,y_score=model.decision_function(X_encoded_test))
```

[29] ✓ 0.0s Python

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

[30] ✓ 0.7s Python

```
<AxesSubplot:>
```



32. AUC

```
from sklearn.metrics import auc

area = auc(fpr1,tpr1)
area

#can be calculated also by roc_auc auc
y_pred = model_log.predict(X_test)
y_pred_proba = model_log.predict_proba(X_test)[:, 1] #used in decision trees
as well while decision function is nit available in decision tree

# Evaluation Metrics
f1 = f1_score(y_test, y_pred)
roc_auc = roc_auc_score(y_test, y_pred_proba)
```

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()
```

Train a classifier

```python
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()
```

[2] ✓ 0.0s Python

```
LogisticRegression(C=1000000000000.0, fit_intercept=False, solver='liblinear')
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

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)))

Draw the ROC curve

In practice, a good way to implement AUC and ROC is via `sklearn`'s built-in functions:

```
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)
```

[3] ✓ 0.0s

From there it's easy to calculate the AUC:

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

[4] ✓ 0.0s

```
AUC: 0.8823114869626498
```

33. ROC 2

```
clf = DecisionTreeClassifier(random_state=10)
clf.fit(X_train,y_train)
y_pred = clf.predict(X_test)
false_positive_rate, true_positive_rate, thresholds =
roc_curve(y_test,y_pred)
```

34.

<u>**Putting it all together**</u>

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)))

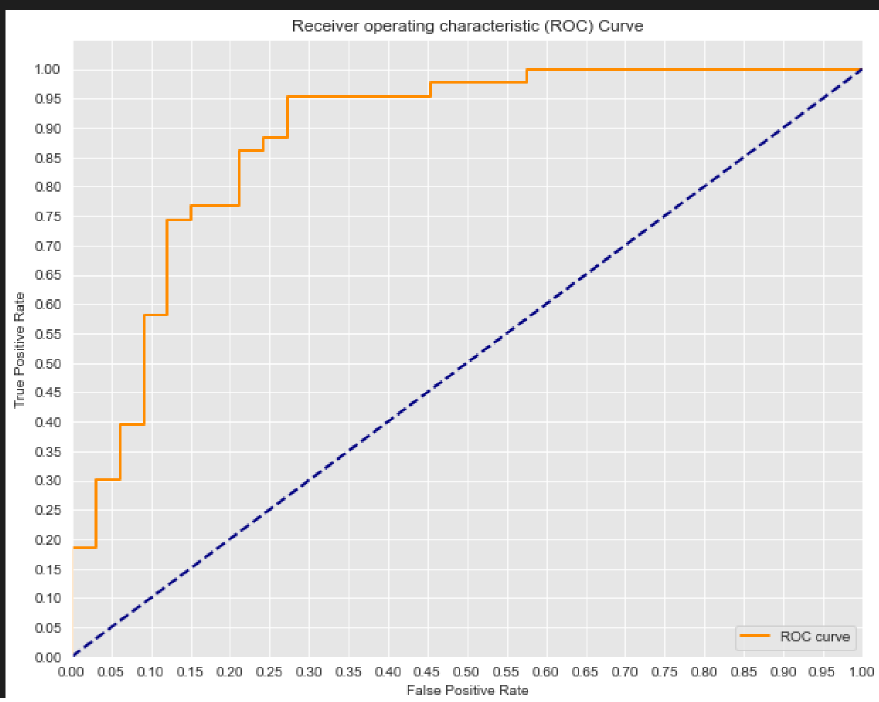
plt.figure(figsize=(10, 8))

```
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()
```

```python
# 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()
```

✓ 1.1s Python

```
AUC: 0.8823114869626498
```



LABS ROC AND AUC

```python
# Import roc_curve, auc
from sklearn.metrics import roc_curve,auc

# Calculate the probability scores of each point in the training set
y_train_score = model_log.decision_function(X_train)

# Calculate the fpr, tpr, and thresholds for the training set
train_fpr, train_tpr, thresholds = roc_curve(y_train,y_train_score)

# Calculate the probability scores of each point in the test set
```

```
y_score = model_log.decision_function(X_test)

# Calculate the fpr, tpr, and thresholds for the test set
fpr, tpr, thresholds = roc_curve(y_test,y_score)
```

35. **Prevent Class Imbalance using Smote**

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_encoded_train,y_train)

y_train_smote.value_counts()
model = LogisticRegression()

model.fit(X_train_smote,y_train_smote)
y_pred_smote = model.predict(X_encoded_test)

report2 = classification_report(y_true=y_test,y_pred=y_pred_smote)
print(report2)
```

```python
# imports

from imblearn.over_sampling import SMOTE
```
[35] Python

```python
# Smote
smote = SMOTE(random_state=42)
X_train_smote, y_train_smote = smote.fit_resample(X_encoded_train,y_train)

y_train_smote.value_counts()
```
[36] Python

```
1    415
0    415
Name: Survived, dtype: int64
```

```python
model = LogisticRegression()

model.fit(X_train_smote,y_train_smote)
```
[37] Python

```
▾ LogisticRegression
LogisticRegression()
```

```python
y_pred_smote = model.predict(X_encoded_test)

report2 = classification_report(y_true=y_test,y_pred=y_pred_smote)
print(report2)
```
[38] Python

```
              precision    recall  f1-score   support

           0       0.84      0.79      0.82       134
           1       0.71      0.78      0.74        89

    accuracy                           0.78       223
   macro avg       0.78      0.78      0.78       223
weighted avg       0.79      0.78      0.79       223
```

-Accurcay reduces to 78

fpr,tpr,_ = roc_curve(y_true=y_test,y_score=model.decision_function(X_encoded_test))

```python
sns.lineplot(x=fpr1,y=tpr1,color="red")
sns.lineplot(x=fpr,y=tpr)
area2 = auc(fpr,tpr)
area2
```

```
fpr,tpr,_ = roc_curve(y_true=y_test,y_score=model.decision_function(X_encoded_test))
```

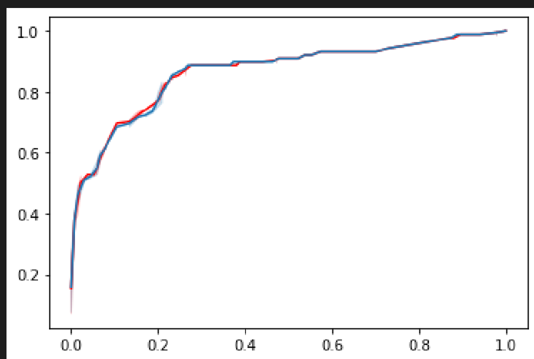
Python

and Debug (Ctrl+Shift+D)

```
sns.lineplot(x=fpr1,y=tpr1,color="red")
sns.lineplot(x=fpr,y=tpr)
```

Python

```
<AxesSubplot:>
```




```
area2 = auc(fpr,tpr)
area2
```

Python

```
0.8667616971323161
```

```
print(f"""
        Without Smote AUC = {area}

        with Smote  AUC = {area2}

      """)
```

Python

```
      Without Smote AUC = 0.86776790206272

      with Smote  AUC = 0.8667616971323161
```

36. SMOTE Using Random Forest

```
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=1)
model.fit(X_train_smote, y_train_smote)
predictions = model.predict(X_encoded_test)
report3 = classification_report(y_true=y_test,y_pred=predictions)
print(report3)
```

```python
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=1)
model.fit(X_train_smote, y_train_smote)
predictions = model.predict(X_encoded_test)
```

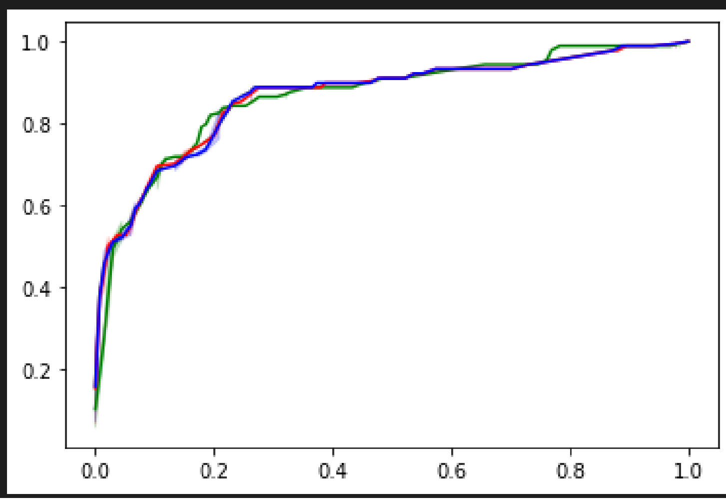
```python
report3 = classification_report(y_true=y_test,y_pred=predictions)
print(report3)
```

```
              precision    recall  f1-score   support

           0       0.83      0.88      0.85       134
           1       0.80      0.72      0.76        89

    accuracy                           0.82       223
   macro avg       0.81      0.80      0.80       223
weighted avg       0.82      0.82      0.81       223
```

```python
fpr2,tpr2,_ = roc_curve(y_test,model.predict_proba(X_encoded_test)[:,1])
sns.lineplot(x=fpr2,y=tpr2,color="green")
sns.lineplot(x=fpr1,y=tpr1,color="red")
sns.lineplot(x=fpr,y=tpr,color="blue")
plt.show()
```

```
fpr2,tpr2,_ = roc_curve(y_test,model.predict_proba(X_encoded_test)[:,1])
```

```
sns.lineplot(x=fpr2,y=tpr2,color="green")
sns.lineplot(x=fpr1,y=tpr1,color="red")
sns.lineplot(x=fpr,y=tpr,color="blue")
plt.show()
```



area3 = auc(fpr,tpr)
area3

```python
area3 = auc(fpr,tpr)
area3
```

```
0.8667616971323161
```

```python
print(f"""
    Without Smote AUC = {area}

    with Smote  AUC = {area2}

    Random forest Smote  AUC = {area3}

   """)
```

```
    Without Smote AUC = 0.86776790206272

    with Smote  AUC = 0.8667616971323161

    Random forest Smote  AUC = 0.8667616971323161
```

37. DECISION TREES

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import OneHotEncoder
from sklearn import tree
ohe = OneHotEncoder()

ohe.fit(X_train)
X_train_ohe = ohe.transform(X_train).toarray()
#Encode data as numbers
# Creating this DataFrame is not necessary its only to show the result of the
ohe
ohe_df = pd.DataFrame(X_train_ohe,
columns=ohe.get_feature_names_out(X_train.columns))

ohe_df.head()
```

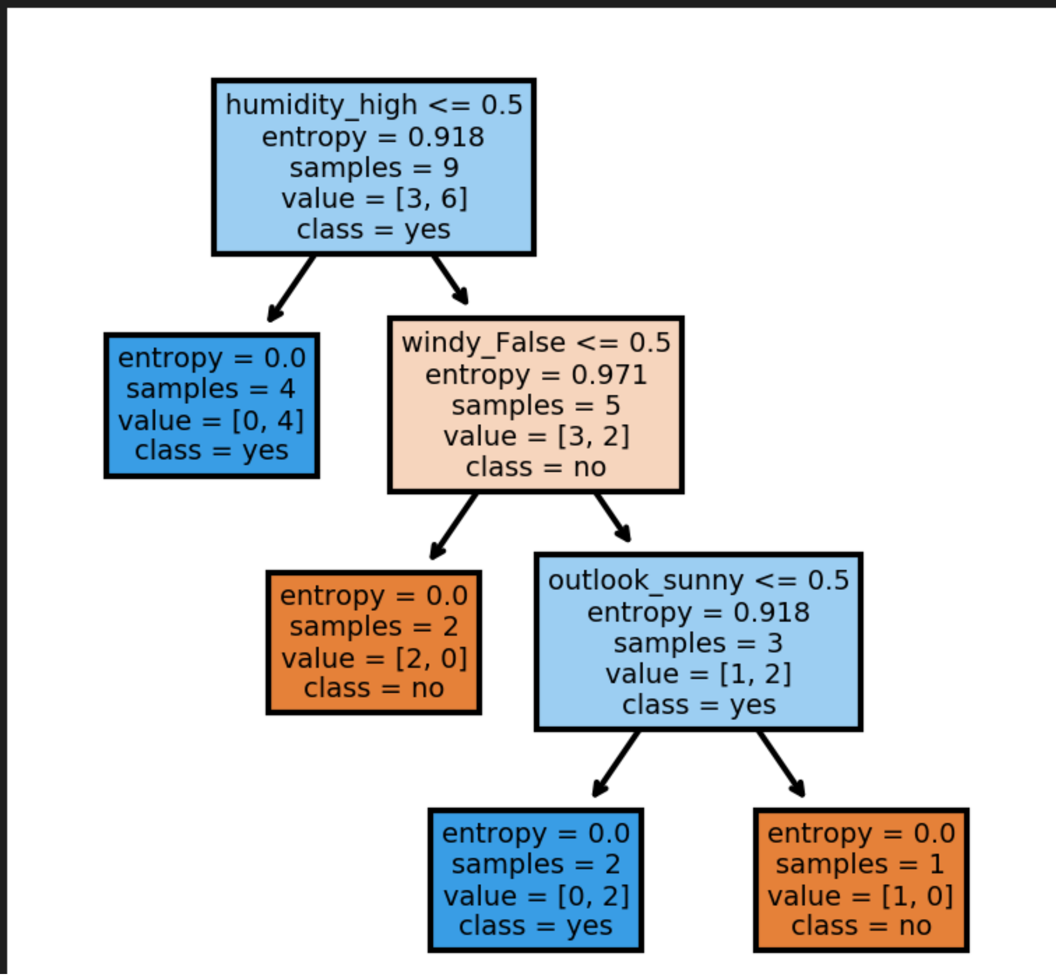
Train the Decision tree

```python
# Create the classifier, fit it on the training data and make predictions on the test set
clf = DecisionTreeClassifier(criterion='entropy')

clf.fit(X_train_ohe, y_train)
```

Plot the Decision Tree

```python
fig, axes = plt.subplots(nrows = 1,ncols = 1, figsize = (3,3), dpi=300)
tree.plot_tree(clf,
               feature_names = ohe_df.columns,
               class_names=np.unique(y).astype('str'),
               filled = True)
plt.show()
```

```python
fig, axes = plt.subplots(nrows = 1,ncols = 1, figsize = (3,3), dpi=300)
tree.plot_tree(clf,
               feature_names = ohe_df.columns,
               class_names=np.unique(y).astype('str'),
               filled = True)
plt.show()
```



38. Evaluate the predictive performance

```python
X_test_ohe = ohe.transform(X_test)
y_preds = clf.predict(X_test_ohe)

print('Accuracy: ', accuracy_score(y_test, y_preds))
```

Evaluate the predictive performance

Now that we have a trained model, we can generate some predictions, and go on to see how accurate our predictions are. We can use a simple accuracy measure, AUC, a confusion matrix, or all of them. This step is performed in the exactly the same manner, so it doesn't matter which classifier you are dealing with.

```
X_test_ohe = ohe.transform(X_test)
y_preds = clf.predict(X_test_ohe)

print('Accuracy: ', accuracy_score(y_test, y_preds))
```

[14] ✓ 0.0s Python

```
Accuracy:  0.6
```

Summary

In this lesson, we looked at how to grow a decision tree using `scikit-learn`. We looked at different stages of data processing, training, and evaluation that you would normally come across while growing a tree or training any other such classifier. We shall now move to a lab, where you will be required to build a tree for a given problem, following the steps shown in this lesson.

39. CART REGRESSOR

```
# Import the DecisionTreeRegressor class
from sklearn.tree import DecisionTreeRegressor

# Instantiate and fit a regression tree model to training data
regressor = DecisionTreeRegressor()
regressor.fit(X_train,y_train)
```

Make preidictions and calculate MAE,MSE,RMSE

```
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Make predictions on the test set
y_pred = regressor.predict(X_test)

# Evaluate these predictions
print('Mean Absolute Error:', mean_absolute_error(y_test,y_pred))
print('Mean Squared Error:', mean_squared_error(y_test,y_pred))
print('Root Mean Squared Error:',
mean_squared_error(y_test,y_pred,squared=False))
```

Make predictions and calculate the MAE, MSE, and RMSE

Use the above model to generate predictions on the test set.

Just as with decision trees for classification, there are several commonly used metrics for evaluating the performance of our model. The most common metrics are:

- Mean Absolute Error (MAE)
- Mean Squared Error (MSE)
- Root Mean Squared Error (RMSE)

If these look familiar, it's likely because you have already seen them before -- they are common evaluation metrics for any sort of regression model, and as we can see, regressions performed with decision tree models are no exception!

Since these are common evaluation metrics, `sklearn` has functions for each of them that we can use to make our job easier. You'll find these functions inside the `metrics` module. In the cell below, calculate each of the three evaluation metrics.

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error

# Make predictions on the test set
y_pred = regressor.predict(X_test)

# Evaluate these predictions
print('Mean Absolute Error:', mean_absolute_error(y_test,y_pred))
print('Mean Squared Error:', mean_squared_error(y_test,y_pred))
print('Root Mean Squared Error:', mean_squared_error(y_test,y_pred,squared=False))
```

✓ 0.1s Python

```
Mean Absolute Error: 94.3
Mean Squared Error: 17347.7
Root Mean Squared Error: 131.7106677532234
```

40. Display two dataframes at the same time

```python
display("Train", train.head(), train.shape, "Test", test.head(), test.shape)
```

```
test = pd.read_csv('data/Test.csv')
# Display the first few rows of the datasets and their shape
display("Train", train.head(), train.shape, "Test", test.head(), test.shape)
```

✓ 0.4s

'Train'

	ID	customer_id	country_id	tbl_loan_id	lender_id	loan_type	Total_Amount	Total_Amount
0	ID_266671248032267278	266671	Kenya	248032	267278	Type_1	8448.0	
1	ID_248919228515267278	248919	Kenya	228515	267278	Type_1	25895.0	
2	ID_308486370501251804	308486	Kenya	370501	251804	Type_7	6900.0	
3	ID_266004285009267278	266004	Kenya	285009	267278	Type_1	8958.0	
4	ID_253803305312267278	253803	Kenya	305312	267278	Type_1	4564.0	

(68654, 16)

'Test'

	ID	customer_id	country_id	tbl_loan_id	lender_id	loan_type	Total_Amount	Total_Amount
0	ID_269404226088267278	269404	Kenya	226088	267278	Type_1	1919.0	
1	ID_255356300042267278	255356	Kenya	300042	267278	Type_1	2138.0	
2	ID_257026243764267278	257026	Kenya	243764	267278	Type_1	8254.0	
3	ID_264617299409267278	264617	Kenya	299409	267278	Type_1	3379.0	
4	ID_247613296713267278	247613	Kenya	296713	267278	Type_1	120.0	

(18594, 15)

ZINDI HACKATHON(predict the likelihood of a customer defaulting on a loan based on their financial data)

Missing values

```
# Are there missing values in the train dataset ?
print(f"There are {train.isna().sum().sum()} missing values in the data.")
```

```
# Are there missing values in the train dataset ?
print(f"There are {train.isna().sum().sum()} missing values in the data.")
```

```
There are 0 missing values in the data.
```

Extract month, day, and year from the date columns / encoding/

```
▪ So here we are going to concatenate both the train and test so that we can
  do the processing once instead of repeating for each
```

```
data = pd.concat([train, test]).reset_index(drop=True)

# Convert the datetime columns appropriately
date_cols = ['disbursement_date', 'due_date']
for col in date_cols:
    data[col] = pd.to_datetime(data[col])
    # Extract month, day, and year from the date columns
    data[col+'_month'] = data[col].dt.month
    data[col+'_day'] = data[col].dt.day
    data[col+'_year'] = data[col].dt.year

# Select all categorical columns from the dataset and label encode them or
one hot encode
cat_cols = data.select_dtypes(include='object').columns
num_cols = [col for col in data.select_dtypes(include='number').columns if
col not in ['target']]
print(f"The categorical columns are: {cat_cols}.")
print("-"* 100)
print(f"The numerical columns are: {num_cols}")
print("-"* 100)
# we are going to one  hot encode the loan type
data = pd.get_dummies(data, columns=['loan_type'], prefix='loan_type',
drop_first=False)
# Convert all the columns with prefix loan_type_ to 0/1 instead of False/True
loan_type_cols = [col for col in data.columns if
col.startswith('loan_type_')]
data[loan_type_cols] = data[loan_type_cols].astype(int)

# Label-encoding for the other remaining categorical columns
le = LabelEncoder()
for col in [col for col in cat_cols if col not in ['loan_type', 'ID']]:
    data[col] = le.fit_transform(data[col])


# deal with numerical columns: we saw loan amount is  highly right skewed for
this we can log transform it
data['Total_Amount'] = np.log1p(data['Total_Amount']) # study other numerical
columns and see if they are skewed as well

# Splitting the data back into train and test
```

```python
train_df = data[data['ID'].isin(train['ID'].unique())]

test_df = data[data['ID'].isin(test['ID'].unique())]

# we are also going to drop the country id as we saw we have only one country
in train
features_for_modelling = [col for col in train_df.columns if col not in
date_cols + ['ID', 'target', 'country_id']]

# Check if the new datasets have the same rows as train and test datasets
print(f"The shape of train_df is: {train_df.shape}")
print(f"The shape of test_df is: {test_df.shape}")
print(f"The shape of train is: {train.shape}")
print(f"The shape of test is: {test.shape}")
print(f"The features for modelling are:\n{features_for_modelling}")
```

Train test split while avoding imbalance

```python
41.X_train, X_valid, y_train, y_valid =
   train_test_split(train_df[features_for_modelling], train['target'],
   stratify=train['target'], shuffle=True, random_state=42)
```

Modelling

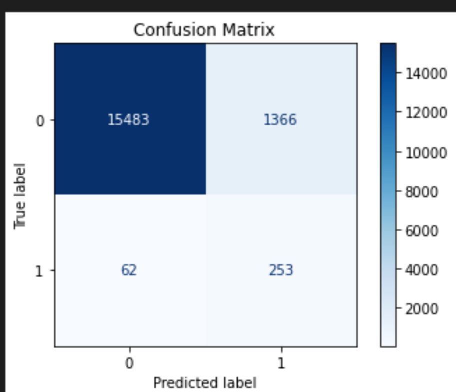
```python
# Standard Scaling
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_valid_scaled = scaler.transform(X_valid)

# Logistic Regression Classifier
clf = LogisticRegression(
    random_state=42,
    class_weight="balanced",  # Handle class imbalance
)
clf.fit(X_train_scaled, y_train)

# Predictions
y_pred = clf.predict(X_valid_scaled)
y_pred_proba = clf.predict_proba(X_valid_scaled)[:, 1]

# Evaluation Metrics
f1 = f1_score(y_valid, y_pred)
roc_auc = roc_auc_score(y_valid, y_pred_proba)

print(f"F1 Score: {f1:.4f}")
print(f"ROC AUC Score: {roc_auc:.4f}")
```

```python
print("\nClassification Report:\n", classification_report(y_valid, y_pred))

# Confusion Matrix
# Confusion Matrix Visualization
ConfusionMatrixDisplay.from_predictions(
    y_valid,
    y_pred,
    display_labels=clf.classes_,
    cmap=plt.cm.Blues
)
plt.title("Confusion Matrix")
plt.show()
```

```
    print(f"ROC AUC Score: {roc_auc:.4f}")
    print("\nClassification Report:\n", classification_report(y_valid, y_pred))

    # Confusion Matrix
    # Confusion Matrix Visualization
    ConfusionMatrixDisplay.from_predictions(
        y_valid,
        y_pred,
        display_labels=clf.classes_,
        cmap=plt.cm.Blues
    )
    plt.title("Confusion Matrix")
    plt.show()
```

✓ 1.3s

```
F1 Score: 0.2616
ROC AUC Score: 0.9285

Classification Report:
               precision    recall  f1-score   support

           0       1.00      0.92      0.96     16849
           1       0.16      0.80      0.26       315

    accuracy                           0.92     17164
   macro avg       0.58      0.86      0.61     17164
weighted avg       0.98      0.92      0.94     17164
```



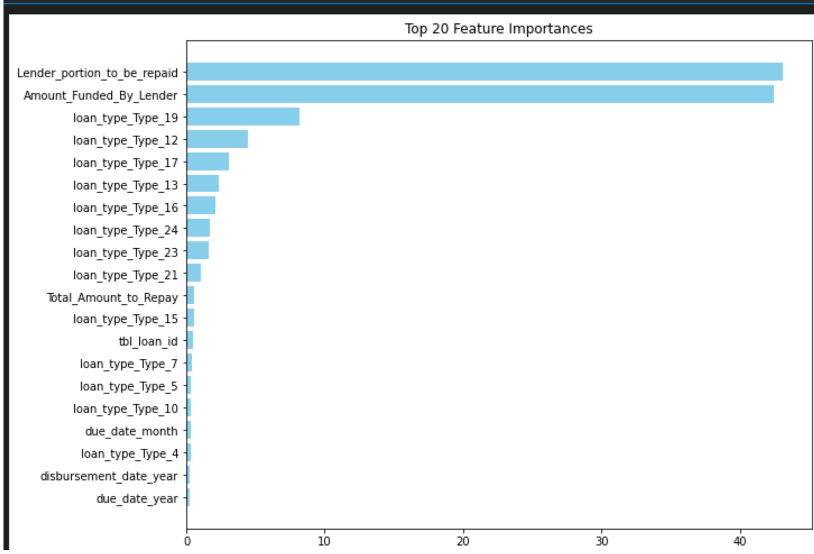
Select top 20 features

```
# Feature Importance

# Get the absolute values of the coefficients
feature_importances = np.abs(clf.coef_).flatten()

# Create a DataFrame for feature importance
importance_df = pd.DataFrame({
    'Feature': features_for_modelling,
    'Importance': feature_importances
})
```

```
# Sort by importance
importance_df = importance_df.sort_values(by='Importance',
ascending=False).head(20)

# Plot the top 20 feature importances
plt.figure(figsize=(10, 8))
plt.barh(importance_df['Feature'], importance_df['Importance'],
color='skyblue')
plt.gca().invert_yaxis()  # To display the most important feature at the top
plt.xlabel('Feature Importance')
plt.title('Top 20 Feature Importances')
plt.show()
```

```
# Get the absolute values of the coefficients
feature_importances = np.abs(clf.coef_).flatten()

# Create a DataFrame for feature importance
importance_df = pd.DataFrame({
    'Feature': features_for_modelling,
    'Importance': feature_importances
})

# Sort by importance
importance_df = importance_df.sort_values(by='Importance', ascending=False).head(20)

# Plot the top 20 feature importances
plt.figure(figsize=(10, 8))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='skyblue')
plt.gca().invert_yaxis()  # To display the most important feature at the top
plt.xlabel('Feature Importance')
plt.title('Top 20 Feature Importances')
plt.show()
```

0.2s



Model prediction and inference

```
# Make predictions on the test dataset
test_predictions = clf.predict(test_df[features_for_modelling])
```

```
test_predictions_proba =
clf.predict_proba(test_df[features_for_modelling])[:, 1]

# Save the predictions to a CSV file

test_df['target'] = test_predictions
sub =  test_df[['ID', 'target']]
sub.head()
```

Model Prediction & Inference

```
# Make predictions on the test dataset
test_predictions = clf.predict(test_df[features_for_modelling])
test_predictions_proba = clf.predict_proba(test_df[features_for_modelling])[:, 1]

# Save the predictions to a CSV file

test_df['target'] = test_predictions
sub =  test_df[['ID', 'target']]
sub.head()
```

	ID	target
68654	ID_269404226088267278	0
68655	ID_255356300042267278	0
68656	ID_257026243764267278	0
68657	ID_264617299409267278	0
68658	ID_247613296713267278	0

```
sub.to_csv('baseline_submission.csv', index=False)
```

42. Plot multiple columns at once

```
for i in train.columns:
    plt.hist(train[i])
    plt.title(i)
    plt.show()
```