Extensions to Linear Models - Lab

Introduction

In this lab, you'll practice many concepts you have learned so far, from adding interactions and polynomials to your model to regularization!

Summary

You will be able to:

- Build a linear regression model with interactions and polynomial features
- Use feature selection to obtain the optimal subset of features in a dataset

Let's Get Started!

Below we import all the necessary packages for this lab.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings('ignore')
from itertools import combinations

from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression, Lasso
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, PolynomialFeatures
```

Load the data.

```
# Load data from CSV
df = pd.read_csv("ames.csv")
# Subset columns
df = df[['LotArea', 'OverallQual', 'OverallCond', 'TotalBsmtSF',
         '1stFlrSF', '2ndFlrSF', 'GrLivArea', 'TotRmsAbvGrd',
         'GarageArea', 'Fireplaces', 'SalePrice']]

# Split the data into X and y
y = df['SalePrice']
X = df.drop(columns='SalePrice')

# Split into train, test, and validation sets
X_train, X_val, y_train, y_val = train_test_split(X, y, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, random_state=0
```

Build a Baseline Housing Data Model

Above, we imported the Ames housing data and grabbed a subset of the data to use in this analysis.

Next steps:

- Scale all the predictors using `StandardScaler`, then convert these scaled features back into DataFrame objects
- Build a baseline `LinearRegression` model using *scaled variables* as predictors and use the R^2 score to evaluate the model

```
# Scale X_train and X_test using StandardScaler
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Ensure X_train and X_test are scaled DataFrames
# (hint: you can set the columns using X.columns)
X_train = pd.DataFrame(X_train_scaled, columns=X.columns)
X_test = pd.DataFrame(X_test_scaled, columns=X.columns)

X_train
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	LotArea	OverallQual	OverallCond	TotalBsmtSF	1stFlrSF	2nd
0	-0.114710	-0.099842	-0.509252	-0.639316	-0.804789	1.26
1	-0.176719	0.632038	-0.509252	0.838208	0.641608	-0.80
2	-0.246336	-0.831723	1.304613	-0.012560	-0.329000	-0.80
3	-0.378617	-0.831723	1.304613	-0.339045	-0.609036	-0.80
4	-0.010898	-1.563603	1.304613	-2.531499	-1.315922	0.55
...	...	...	...	...	...	...
816	-0.532331	-0.099842	-0.509252	-0.510628	-0.897228	-0.80
817	-0.309245	-0.099842	-0.509252	0.514106	0.315353	-0.80
818	0.119419	0.632038	-0.509252	-0.513011	-0.899947	1.68
819	-0.002718	-0.099842	1.304613	-0.889542	-1.329516	0.78
820	0.086287	-0.099842	0.397681	0.433080	0.179414	-0.80

821 rows × 10 columns

```
# Create a LinearRegression model and fit it on scaled training data
regression = LinearRegression()
regression.fit(X_train, y_train)

# Calculate a baseline r-squared score on training data
baseline = regression.score(X_train, y_train)
baseline
```

```
0.7868344817421309
```

Add Interactions

Instead of adding all possible interaction terms, let's try a custom technique. We are only going to add the interaction terms that increase the R^2 score as much as possible. Specifically we are going to look for the 7 interaction terms that each cause the most increase in the coefficient of determination.

Find the Best Interactions

Look at all the possible combinations of variables for interactions by adding interactions one by one to the baseline model. Create a data structure that stores the pair of columns used as well as the R^2 score for each combination.

Hint: We have imported the `combinations` function from `itertools` for you (documentation here). Try applying this to the columns of `X_train` to find all of the possible pairs.

Print the 7 interactions that result in the highest R^2 scores.

```
# Set up data structure
# (Here we are using a list of tuples, but you could use a dictionary,
# a list of lists, some other structure. Whatever makes sense to you.)
interactions = []

# Find combinations of columns and loop over them
column_pairs = list(combinations(X_train.columns, 2))
for (col1, col2) in column_pairs:
    # Make copies of X_train and X_test
    features_train = X_train.copy()
    features_test = X_test.copy()
```

```
        # Add interaction term to data
        features_train["interaction"] = features_train[col1] * features_train[col2]
        features_test["interaction"] = features_test[col1] * features_test[col2]

        # Find r-squared score (fit on training data, evaluate on test data)
        score = LinearRegression().fit(features_train, y_train).score(features_test, y_t

        # Append to data structure
        interactions.append(((col1, col2), score))

    # Sort and subset the data structure to find the top 7
    top_7_interactions = sorted(interactions, key=lambda record: record[1], reverse=True
    print("Top 7 interactions:")
    print(top_7_interactions)
```

```
Top 7 interactions:
[(('LotArea', '1stFlrSF'), 0.7211105666140574), (('LotArea', 'TotalBsmtSF'),
0.7071649207050104), (('LotArea', 'GrLivArea'), 0.6690980823779029), (('LotArea',
'Fireplaces'), 0.6529699515652587), (('2ndFlrSF', 'TotRmsAbvGrd'),
0.647299489040519), (('OverallCond', 'TotalBsmtSF'), 0.6429019879233769),
(('OverallQual', '2ndFlrSF'), 0.6422324294284367)]
```

Add the Best Interactions

Write code to include the 7 most important interactions in `X_train` and `X_test` by adding 7 columns. Use the naming convention `"col1_col2"`, where `col1` and `col2` are the two columns in the interaction.

```
# Loop over top 7 interactions
for record in top_7_interactions:
    # Extract column names from data structure
    col1, col2 = record[0]

    # Construct new column name
    new_col_name = col1 + "_" + col2

    # Add new column to X_train and X_test
    X_train[new_col_name] = X_train[col1] * X_train[col2]
    X_test[new_col_name] = X_test[col1] * X_test[col2]

X_train
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	LotArea	OverallQual	OverallCond	TotalBsmtSF	1stFlrSF	2nd
0	-0.114710	-0.099842	-0.509252	-0.639316	-0.804789	1.261
1	-0.176719	0.632038	-0.509252	0.838208	0.641608	-0.80
2	-0.246336	-0.831723	1.304613	-0.012560	-0.329000	-0.80
3	-0.378617	-0.831723	1.304613	-0.339045	-0.609036	-0.80
4	-0.010898	-1.563603	1.304613	-2.531499	-1.315922	0.550
...	...	...	...	...	...	...
816	-0.532331	-0.099842	-0.509252	-0.510628	-0.897228	-0.80
817	-0.309245	-0.099842	-0.509252	0.514106	0.315353	-0.80
818	0.119419	0.632038	-0.509252	-0.513011	-0.899947	1.684
819	-0.002718	-0.099842	1.304613	-0.889542	-1.329516	0.783
820	0.086287	-0.099842	0.397681	0.433080	0.179414	-0.80

821 rows × 17 columns

Add Polynomials

Now let's repeat that process for adding polynomial terms.

Find the Best Polynomials

Try polynomials of degrees 2, 3, and 4 for each variable, in a similar way you did for interactions (by looking at your baseline model and seeing how R^2 increases). Do understand that when going for a polynomial of degree 4 with `PolynomialFeatures`, the particular column is raised to the power of 2 and 3 as well in other terms.

We only want to include "pure" polynomials, so make sure no interactions are included.

Once again you should make a data structure that contains the values you have tested. We recommend a list of tuples of the form:

`(col_name, degree, R2)`, so eg. `('OverallQual', 2, 0.781)`

```
# Set up data structure
polynomials = []

# Loop over all columns
for col in X_train.columns:
    # Loop over degrees 2, 3, 4
    for degree in (2, 3, 4):

        # Make a copy of X_train and X_test
        features_train = X_train.copy().reset_index()
        features_test = X_test.copy().reset_index()

        # Instantiate PolynomialFeatures with relevant degree
        poly = PolynomialFeatures(degree, include_bias=False)

        # Fit polynomial to column and transform column
        # Hint: use the notation df[[column_name]] to get the right shape
        # Hint: convert the result to a DataFrame
        col_transformed_train = pd.DataFrame(poly.fit_transform(features_train[[col]
        col_transformed_test = pd.DataFrame(poly.transform(features_test[[col]]))

        # Add polynomial to data
        # Hint: use pd.concat since you're combining two DataFrames
        # Hint: drop the column before combining so it doesn't appear twice
        features_train = pd.concat([features_train.drop(col, axis=1), col_transforme
        features_test = pd.concat([features_test.drop(col, axis=1), col_transformed_

        # Find r-squared score
        score = LinearRegression().fit(features_train, y_train).score(features_test,

        # Append to data structure
        polynomials.append((col, degree, score))

# Sort and subset the data structure to find the top 7
top_7_polynomials = sorted(polynomials, key=lambda record: record[-1], reverse=True)
```

```
print("Top 7 polynomials:")
print(top_7_polynomials)
```

```
Top 7 polynomials:
[('GrLivArea', 3, 0.8283186230750728), ('OverallQual_2ndFlrSF', 3,
0.8221477940922196), ('LotArea_Fireplaces', 4, 0.8124290394772224),
('LotArea_Fireplaces', 3, 0.8122028721735164), ('OverallQual', 3,
0.8068150958879932), ('OverallQual_2ndFlrSF', 2, 0.8057158750082858),
('OverallQual', 4, 0.8033460977380442)]
```

Add the Best Polynomials

If there are duplicate column values in the results above, don't add multiple of them to the model, to avoid creating duplicate columns. (For example, if column `A` appeared in your list as both a 2nd and 3rd degree polynomial, adding both would result in `A` squared being added to the features twice.) Just add in the polynomial that results in the highest R-Squared.

```
# Convert to DataFrame for easier manipulation
top_polynomials = pd.DataFrame(top_7_polynomials, columns=["Column", "Degree", "R^2"
top_polynomials
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Column	Degree	R^2
0	GrLivArea	3	0.828319
1	OverallQual_2ndFlrSF	3	0.822148
2	LotArea_Fireplaces	4	0.812429

	Column	Degree	R^2
3	LotArea_Fireplaces	3	0.812203
4	OverallQual	3	0.806815
5	OverallQual_2ndFlrSF	2	0.805716
6	OverallQual	4	0.803346

```
# Drop duplicate columns
top_polynomials.drop_duplicates(subset="Column", inplace=True)
top_polynomials
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	Column	Degree	R^2
0	GrLivArea	3	0.828319
1	OverallQual_2ndFlrSF	3	0.822148
2	LotArea_Fireplaces	4	0.812429
4	OverallQual	3	0.806815

```
# Loop over remaining results
for (col, degree, _) in top_polynomials.values:
    # Create polynomial terms
    poly = PolynomialFeatures(degree, include_bias=False)
    col_transformed_train = pd.DataFrame(poly.fit_transform(X_train[[col]]),
                                         columns=poly.get_feature_names([col]))
    col_transformed_test = pd.DataFrame(poly.transform(X_test[[col]]),
                                        columns=poly.get_feature_names([col]))
    # Concat new polynomials to X_train and X_test
    X_train = pd.concat([X_train.drop(col, axis=1), col_transformed_train], axis=1)
    X_test = pd.concat([X_test.drop(col, axis=1), col_transformed_test], axis=1)
```

Check out your final data set and make sure that your interaction terms as well as your polynomial terms are included.

```
X_train.head()
```

<style scoped> .dataframe tbody tr th:only-of-type { vertical-align: middle; }

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

</style>

	LotArea	OverallCond	TotalBsmtSF	1stFlrSF	2ndFlrSF	TotRmsAl
0	-0.114710	-0.509252	-0.639316	-0.804789	1.261552	0.250689
1	-0.176719	-0.509252	0.838208	0.641608	-0.808132	-0.365525
2	-0.246336	1.304613	-0.012560	-0.329000	-0.808132	-0.981739
3	-0.378617	1.304613	-0.339045	-0.609036	-0.808132	-0.981739
4	-0.010898	1.304613	-2.531499	-1.315922	0.550523	0.250689

5 rows × 26 columns

Full Model R-Squared

Check out the R^2 of the full model with your interaction and polynomial terms added. Print this value for both the train and test data.

```
lr = LinearRegression()
lr.fit(X_train, y_train)
```

```
print("Train R^2:", lr.score(X_train, y_train))
print("Test R^2: ", lr.score(X_test, y_test))
```

```
Train R^2: 0.8571817758242435
Test R^2:  0.6442143449157876
```

It looks like we may be overfitting some now. Let's try some feature selection techniques.

Feature Selection

First, test out `RFE` (documentation here) with several different `n_features_to_select` values. For each value, print out the train and test R^2 score and how many features remain.

```
for n in [5, 10, 15, 20, 25]:
    rfe = RFE(LinearRegression(), n_features_to_select=n)
    X_rfe_train = rfe.fit_transform(X_train, y_train)
    X_rfe_test = rfe.transform(X_test)

    lr = LinearRegression()
    lr.fit(X_rfe_train, y_train)

    print("Train R^2:", lr.score(X_rfe_train, y_train))
    print("Test R^2: ", lr.score(X_rfe_test, y_test))
    print(f"Using {n} out of {X_train.shape[1]} features")
    print()
```

```
Train R^2: 0.776039994126505
Test R^2:  0.6352981725272363
Using 5 out of 26 features

Train R^2: 0.8191862278324273
Test R^2:  0.6743476159860743
Using 10 out of 26 features

Train R^2: 0.8483321237427194
Test R^2:  0.704013767108713
Using 15 out of 26 features

Train R^2: 0.8495176468836853
Test R^2:  0.7169477986870836
Using 20 out of 26 features

Train R^2: 0.8571732578218183
```

```
Test R^2:  0.6459291693655327
Using 25 out of 26 features
```

Now test out `Lasso` (documentation here) with several different `alpha` values.

```python
for alpha in [1, 10, 100, 1000, 10000]:
    lasso = Lasso(alpha=alpha)
    lasso.fit(X_train, y_train)

    print("Train R^2:", lasso.score(X_train, y_train))
    print("Test R^2: ", lasso.score(X_test, y_test))
    print(f"Using {sum(abs(lasso.coef_) < 10**(-10))} out of {X_train.shape[1]} feat
    print("and an alpha of", alpha)
    print()
```

```
Train R^2: 0.857153074119144
Test R^2:  0.6485699116355144
Using 0 out of 26 features
and an alpha of 1

Train R^2: 0.8571373079024015
Test R^2:  0.6480527180183058
Using 0 out of 26 features
and an alpha of 10

Train R^2: 0.856958744623801
Test R^2:  0.6471042867008598
Using 1 out of 26 features
and an alpha of 100

Train R^2: 0.8506404012942795
Test R^2:  0.7222278677869791
Using 9 out of 26 features
and an alpha of 1000

Train R^2: 0.7790529223548714
Test R^2:  0.7939567393897818
Using 14 out of 26 features
and an alpha of 10000
```

Compare the results. Which features would you choose to use?

```
"""
For RFE the model with the best test R-Squared was using 20 features
For Lasso the model with the best test R-Squared was using an alpha of 10000

The Lasso result was a bit better so let's go with that and the 14 features
that it selected
"""
```

Evaluate Final Model on Validation Data

Data Preparation

At the start of this lab, we created `X_val` and `y_val`. Prepare `X_val` the same way that `X_train` and `X_test` have been prepared. This includes scaling, adding interactions, and adding polynomial terms.

```
# Scale X_val
X_val_scaled = scaler.transform(X_val)
X_val = pd.DataFrame(X_val_scaled, columns=X.columns)

# Add interactions to X_val
for record in top_7_interactions:
    col1, col2 = record[0]
    new_col_name = col1 + "_" + col2
    X_val[new_col_name] = X_val[col1] * X_val[col2]

# Add polynomials to X_val
for (col, degree, _) in top_polynomials.values:
    poly = PolynomialFeatures(degree, include_bias=False)
    col_transformed_val = pd.DataFrame(poly.fit_transform(X_val[[col]]),
                                       columns=poly.get_feature_names([col]))
    X_val = pd.concat([X_val.drop(col, axis=1), col_transformed_val], axis=1)
```

Evaluation

Using either `RFE` or `Lasso`, fit a model on the complete train + test set, then find R-Squared and MSE values for the validation set.

```
final_model = Lasso(alpha=10000)
final_model.fit(pd.concat([X_train, X_test]), pd.concat([y_train, y_test]))

print("R-Squared:", final_model.score(X_val, y_val))
print("MSE:", mean_squared_error(y_val, final_model.predict(X_val)))
```

```
R-Squared: 0.7991273457324125
MSE: 1407175023.3081572
```

Level Up Ideas (Optional)

Create a Lasso Path

From this section, you know that when using `Lasso`, more parameters shrink to zero as your regularization parameter goes up. In scikit-learn there is a function `lasso_path()` which visualizes the shrinkage of the coefficients while $alpha$ changes. Try this out yourself!

https://scikit-learn.org/stable/auto_examples/linear_model/plot_lasso_coordinate_descent_path.html#sphx-glr-auto-examples-linear-model-plot-lasso-coordinate-descent-path-py

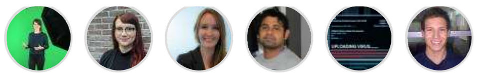
AIC and BIC for Subset Selection

This notebook shows how you can use AIC and BIC purely for feature selection. Try this code out on our Ames housing data!

https://xavierbourretsicotte.github.io/subset_selection.html

Summary

Congratulations! You now know how to apply concepts of bias-variance tradeoff using extensions to linear models and feature selection.



Languages

- ● **Jupyter Notebook** 100.0%