Workshop 12: Models for regression and classification

FIE463: Numerical Methods in Macroeconomics and Finance using Python

Richard Foltyn

NHH Norwegian School of Economics

April 3, 2025

See GitHub repository for notebooks and data:

https://github.com/richardfoltyn/FIE463-V25

Exercise 1: Predicting house prices with linear models

In this exercise, you will work with the Ames house data set which we already encountered in the lectures. Your task is to evaluate the following three linear models in terms of their performance when predicting house prices:

1. Linear regression without any regularization
2. Ridge regression
3. Lasso

Data description

The data is stored in in the file `data/ames_houses.csv` and can be loaded as follows:

[1]:
```
import pandas as pd

# Use this path to use the CSV file from the data/ directory
file = '../../data/ames_houses.csv'

df = pd.read_csv(file, sep=',')

# Variables used in the analysis
variables = [
    'LotArea',
    'LivingArea',
    'Bathrooms',
    'Bedrooms',
    'SalePrice',
    'OverallQuality',
    'BuildingType',
    'YearBuilt',
    'CentralAir',
]

# Drop rows with any missing observation
df = df.dropna(subset=variables)
```

```
# Drop observations with large living or lot area
df = df.query('LivingArea <= 350 & LotArea <= 5000')

print(f'Number of observations: {df.shape[0]:,d}')
```

```
Number of observations: 2,755
```

The included variables are a simplified subset of the original data (see here for a detailed description of the original variables):

1. `LotArea`: Lot size in square meters
2. `Neighborhood`: Physical locations within Ames city limits
3. `OverallQuality`: Rates the overall material and finish of the house (1 = very poor, 10 = excellent)
4. `OverallCondition`: Rates the overall condition of the house (1 = very poor, 10 = excellent)
5. `YearBuilt`: Original construction date
6. `YearRemodeled`: Remodel date (same as construction date if no remodeling or additions)
7. `BuildingType`: Type of dwelling
8. `CentralAir`: Central air conditioning (string, Y/N)
9. `LivingArea`: Above grade (ground) living area in square meters
10. `Bathrooms`: Full bathrooms above grade
11. `Bedrooms`: Bedrooms above grade (does not include basement bedrooms)
12. `Fireplaces`: Number of fireplaces
13. `SalePrice`: Sale price in thousands of USD
14. `YearSold`: Year sold
15. `MonthSold`: Month sold
16. `HasGarage`: Flag whether property has a garage

Part 1 — Data preprocessing

Apply the following steps to preprocess the data before estimation:

1. Recode the string values in column `CentralAir` into numbers such that `'N'` is mapped to 0 and `'Y'` is mapped to 1.
2. Recode the string values in column `BuildingType` and create the new variable `IsSingleFamily` which takes on the value 1 whenever a house is a single-family home and 0 otherwise.
3. Convert the variables `SalePrice`, `LivingArea` and `LotArea` to (natural) logs. Name the transformed columns `logSalePrice`, `logLivingArea` and `logLotArea`.
4. Plot the histograms of `SalePrice`, `LivingArea`, and `LotArea`. In a new figure, plot the histograms of `logSalePrice`, `logLivingArea` and `logLotArea`. Which set of variables os better suited for model fitting?

Part 2 — Model features

Model specification

You are now asked to estimate the following model of house prices as a function of house characteristics:

$$\begin{aligned}\log(SalePrice_i) = \alpha + f\Big(&\log(LivingArea_i),\ \log(LotArea_i), OverallQuality_i,\\ &Bathrooms_i,\ Bedrooms_i\Big)\\ &+ \gamma_0 YearBuilt_i + \gamma_1 CentralAir_i + \gamma_3 IsSingleFamily_i + \epsilon_i\end{aligned}$$

where i indexes observations and ϵ is an additive error term. The function $f(\bullet)$ is a *polynomial of degree 3* in its arguments, i.e., it includes all terms and interactions of the given variables where the exponents sum to 3 or less:

$$\begin{aligned}f(\log(LivingArea_i), \log(LotArea_i), \dots) = &\beta_0 \log(LivingArea_i) + \beta_1 \log(LivingArea_i)^2\\ &+ \beta_2 \log(LivingArea_i)^3 + \beta_3 \log(LotArea_i)\\ &+ \beta_4 \log(LotArea_i)^2 + \beta_5 \log(LotArea_i)^3\\ &+ \beta_6 \log(LivingArea_i) \log(LotArea_i)\\ &+ \beta_7 \log(LivingArea_i)^2 \log(LotArea_i)\\ &+ \beta_8 \log(LivingArea_i) \log(LotArea_i)^2\\ &+ \cdots\end{aligned}$$

Creating model features and outcomes

1. Complete the template code below to create a feature matrix `X` which contains all polynomial interactions as well as the remaining non-interacted variables.

   Hints:

   - Use the `PolynomialFeatures` transformation to create the polynomial terms and interactions from the columns `logLivingArea`, `logLotArea`, `OverallQuality`, `Bathrooms` and `Bedrooms`.
   - Make sure that the generated polynomial does *not* contain a constant ("bias"). You should include the intercept when estimating a model instead.
   - You can use `np.hstack()` to concatenate two matrices (the polynomials and the remaining covariates) along the column dimension.
   - The complete feature matrix `X` should contain a total of 55 columns (52 polynomial interactions and 3 non-polynomial features).

2. Split the data into a training and a test subset such that the training sample contains 70% of observations.

   Hint:

   - Use the function `train_test_split()` to split the sample. Pass the argument `random_state=1234` to get reproducible results.
   - Make sure to define the training and test samples only *once* so that they are identical for all estimators used below.

[9]:
```
# Random state (for train/test split and cross-validation)
RANDOM_STATE = 1234

# Name of target variable
target = 'logSalePrice'
```

```python
# Features included as polynomials
features_poly = [
    'logLivingArea',
    'logLotArea',
    'OverallQuality',
    'Bathrooms',
    'Bedrooms',
]

# Other features not included in polynomials
features_other = ['YearBuilt', 'CentralAir', 'IsSingleFamily']
features = features_poly + features_other

# Keep only columns that are used to estimate model
columns = [target] + features
df = df[columns]

# Response variable
y = df[target]

# TODO: Create polynomial features

# TODO: Merge polynomial features and non-polynomial features into single matrix X

# TODO: Split data into training and test sets
```

Part 3 — Linear regression

Perform the following tasks:

1. Estimate the above specification using the linear regression model `LinearRegression` on the training sub-set.
    - Do you need to standardize features before estimating a linear regression model?
    - Does the linear regression model have any hyperparameters?
2. Compute and report the root mean squared error (RMSE) and the R^2 on the test sample.

Hints:

- The root mean squared error can be computed with `root_mean_squared_error()`.
- The R^2 can be computed with r2_score().

Part 4 — Ridge regression

Next, you want to estimate a Ridge regression which has the regularization strength α as a hyperparameter.

1. Use `RidgeCV` to determine the best α on the training sub-sample. You can use the MSE metric (the default) to find the optimal α. Report the optimal α and the corresponding MSE.
    - Does Ridge regression require feature standardization? If so, don't forget to apply it before fitting the model.
2. Use the function `plot_validation_curve()` defined below to plot the MSE (averaged over folds on the training sub-sample) against the regularization strength α.
3. Compute and report the RMSE and the R^2 on the test sample.

Hints:

- When running RidgeCV, use a grid of 1,000 α's which are spaced uniformly in logs:

  `alphas = np.logspace(np.log10(1.0e-6), np.log10(100), 1000)`

- Recall that the (negative!) best MSE is stored in the attribute `best_score_` after cross-validation is complete.

[14]:
```python
import matplotlib.pyplot as plt

def plot_validation_curve(alphas, mse_mean, title=None):
    """
    Plot validation curve for Ridge or Lasso.

    Parameters
    ----------
    alphas : array-like
        Regularization strengths.
    mse_mean : array-like
        Cross-validated MSE (averaged over folds).
    title : str, optional
        Title of the plot.
    """

    # Index of MSE-minimizing alpha
    imin = np.argmin(mse_mean)

    # Plot MSE against alphas, highlight minimum MSE
    plt.plot(alphas, mse_mean)
    plt.xlabel(r'Regularisation strength $\alpha$ (log scale)')
    plt.ylabel('Cross-validated MSE')
    plt.scatter(alphas[imin], mse_mean[imin], s=15, c='black', zorder=100)
    plt.axvline(alphas[imin], ls=':', lw=0.75, c='black')
    plt.title(title)
    plt.xscale('log')
```

Part 5 — Lasso

Next, you want to estimate a Lasso model which also has a regularization strength hyperparameter α:

1. Use `LassoCV` to determine the best α on the training sub-sample using cross-validation with 5 folds. You can use the MSE metric (the default) to find the optimal α. Report the optimal α and the corresponding MSE.
   - Does Lasso require feature standardization? If so, don't forget to apply it before fitting the model.
2. Use the function `plot_validation_curve()` to plot the MSE (averaged over folds on the training sub-sample) against the regularization strength α.
3. Compute and report the RMSE and the R^2 on the test sample for the model using the optimal α.
4. Report the number of non-zero coefficients for the model using the optimal α.

Hints:

- Getting Lasso to converge may require some experimentation. The following settings should help:
  1. Increase the max. number of iterations to `max_iter=100_000`.
  2. Use `selection='random'` and set `random_state=1234` to get reproducible results.
- Use `eps=1.0e-4` as an argument to `LassoCV` to specify the ratio of the smallest to the largest α.
- After cross-validation is complete, the MSE for each value of α and each fold are stored in the attribute `mse_path_` which is an array with shape `(N_ALPHA, N_FOLDS)`.

Part 6 — Compare estimation results

Create a table which contains the MSE and R^2 computed on the test sample for all three models (using their optimal hyperparameters). Which model performs best?

Exercise 2: Classification of above-average houses

We continue with the setup from the previous exercise, but now use classification to predict whether a house was sold for more than the average price in its neighborhood.

Use the same initial data processing steps as before, which are repeated here for convenience:

[ ]:
```
import pandas as pd
import numpy as np

# Use this path to use the CSV file from the data/ directory
file = '../../data/ames_houses.csv'

df = pd.read_csv(file, sep=',')

# Drop rows with any missing observation
df = df.dropna()

# Drop observations with large living or lot area
df = df.query('LivingArea <= 350 & LotArea <= 5000')

# Create log-transformed variables
df['logLivingArea'] = np.log(df['LivingArea'])
df['logLotArea'] = np.log(df['LotArea'])

# Create indicator variable for single family homes
df['IsSingleFamily'] = (df['BuildingType'] == 'Single-family').astype(int)

# Create indicator variable for central air
df['CentralAir'] = df['CentralAir'].map({'Y': 1, 'N': 0})

print(f'Number of observations: {df.shape[0]:,d}')
```

Part 1 — Data preprocessing

Perform the following additional data processing steps:

1. Drop all neighborhoods with less than 40 observations.
2. Create a new variable `MoreExpensive` which is 1 whenever the sale price is above the average sale price in the neighborhood.
3. Split the data set into two data frames, `df_train` and `df_test`, where the test sample should contain 20% of the observations. Stratify the train-test split using the indicator `MoreExpensive`.

Part 2 — Logistic regression

Using the template code below, create the feature matrix for the logistic regression as follows:

1. Create polynomials of degree 3 using the variables `LivingArea`, `LotArea`, `OverallQuality`, `OverallCondition`, `Bathrooms`, `Bedrooms`, `Fireplaces`, and `YearRemodeled`
2. Add the non-interacted features `CentralAir`, and `IsSingleFamily` to the feature matrix.

Then perform the following steps to fit and evaluate the model:

1. Fit the logistic regression with `LogisticRegression()`, using the indicator `MoreExpensive` as the target variable.

   - Does the logistic regression require feature standardization? If so, you need to transform the features using `StandardScaler()`.
   - You can use the default parameters for `LogisticRegression`, but you might need to increase the maximum number of iterations (e.g., `max_iter=10_000`).

2. After you have fitted the model, use the function `tabulate_classifier_metrics()` defined below to tabulate the accuracy, precision, recall, and the F1 store on the test sample.

3. After you have fitted the model, use the function `plot_confusion_matrix()` defined below to plot the confusion matrix on the test sample.

   This function calls ConfusionMatrixDisplay.from_estimator() to create a confusion matrix graph.

[ ]:
```
from sklearn.preprocessing import PolynomialFeatures

# Target variable name
target = 'MoreExpensive'

# Features included as polynomials (in logistic regression)
features_poly = [
    'LivingArea',
    'LotArea',
    'OverallQuality',
    'OverallCondition',
    'Bathrooms',
    'Bedrooms',
    'Fireplaces',
    'YearRemodeled'
]

# Other features not included in polynomials
features_other = ['CentralAir', 'IsSingleFamily']
features = features_poly + features_other

# Response variable
y_train = df_train[target]
y_test = df_test[target]

# TODO: Create polynomial features for training sample

# TODO: Create polynomial features for test sample

# TODO: Merge polynomial features and non-polynomial features into X_train

# TODO: Merge polynomial features and non-polynomial features into X_test

# TODO: Standardize features

# TODO: Fit logistic regression model

# TODO: Tabulate metrics on test sample using tabulate_classification_metrics()

# TODO: Plot confusion matrix using plot_confusion_matrix()
```

[30]:
```
from sklearn.metrics import accuracy_score, f1_score, precision_score, recall_score


def tabulate_classifier_metrics(estimator, X, y):
```

```python
    """
    Tabulate classification metrics (accuracy, precision, recall, F1).

    Parameters
    ----------
    estimator : object
        Fitted classifier.
    X : array-like
        Feature matrix.
    y : array-like
        Target variable.
    """

    # Predict outcome
    y_pred = estimator.predict(X)

    # Compute scores
    acc = accuracy_score(y, y_pred)
    pre = precision_score(y, y_pred)
    rec = recall_score(y, y_pred)
    f1 = f1_score(y, y_pred)

    # Combine scores into a single Series
    index = pd.Index(
        ['Accuracy', 'Precision [TP/(TP+FP)]', 'Recall [TP/P]', 'F1'], name='Metric'
    )
    stats = pd.Series([acc, pre, rec, f1], index=index)

    stats = stats.round(3)

    return stats
```

[31]:
```python
from sklearn.metrics import ConfusionMatrixDisplay

def plot_confusion_matrix(estimator, X, y, title='Confusion matrix'):
    """
    Plot confusion matrix for classification model.

    Parameters
    ----------
    estimator : estimator
        Fitted classification model.
    X : array-like
        Feature matrix.
    y : array-like
        Target variable.
    title : str
        Title of the plot.
    """

    cm = ConfusionMatrixDisplay.from_estimator(
        estimator=estimator,
        X=X,
        y=y,
        values_format=',d',
        cmap='Blues',
        colorbar=False,
        text_kw={'fontsize': 10, 'fontweight': 'bold'},
    )
    cm.ax_.set_title(title)
```

Part 3 — Logistic regression CV

Instead of using the default regularization strength `C=1`, perform cross-validation to find the optimal value of C:

1. Run the cross-validation with LogisticRegressionCV.

   Create a log-spaced grid of candidate values as follows:

   ```
   C_grid = np.logspace(-2, 2, 500)
   ```

2. Report the optimal value of C.
3. After you have fitted the model, use the function `tabulate_classifier_metrics()` to tabulate the accuracy, precision, recall, and the F1 store on the test sample.
4. After you have fitted the model, use the function `plot_confusion_matrix()` defined below to plot the confusion matrix on the test sample.

Part 4 — Random forest

You now want to investigate how other classifiers perform on this task compared to logistic regression.

1. Fit the Random forest classifier implemented in RandomForestClassifier to the data. Use the default parameters for now.
   - Do you need to include polynomial interactions with Random forest?
   - Do you need to standardize the features with Random forest?
2. After you have fitted the model, use the function `tabulate_classifier_metrics()` to tabulate the accuracy, precision, recall, and the F1 store on the test sample.
3. After you have fitted the model, use the function `plot_confusion_matrix()` defined below to plot the confusion matrix on the test sample.

Part 5 — Random forest CV

In the previous part, you used the standard hyperparameters for the Random forest (e.g., the number of trees to grow and the maximum depth).

1. Perform cross-validation of these parameters with `GridSearchCV`, using the parameter grids defined in the template below.
2. After you have fitted the model, use the function `tabulate_classifier_metrics()` to tabulate the accuracy, precision, recall, and the F1 store on the test sample.
3. After you have fitted the model, use the function `plot_confusion_matrix()` defined below to plot the confusion matrix on the test sample.

[43]:
```python
from sklearn.model_selection import GridSearchCV

# Define hyperparameter grid
param_grid = {
    'n_estimators': np.arange(100, 201, 10),
    'max_depth': np.arange(3, 20),
}

# TODO: Call GridSearchCV to find optimal hyperparameters

# TODO: Report optimal number of estimators stored in best_params_

# TODO: Report optimal max depth stored in best_params_
```

Part 6 — Compare estimation results

Combine the accuracy, presion, recall, and F1 metrics for all the models you estimated and report them in a single table. Which estimator does best on the classification task?