Train Test Splits, Cross Validation, and Linear Regression

Lregressions.png

Learning Objectives

- Explain the difference between over-fitting and under-fitting a model
- Describe Bias-variance tradeoffs
- Find the optimal training and test data set splits, cross-validation, and model complexity versus error
- Apply a linear regression model for supervised learning
- Apply Intel® Extension for Scikit-learn* to leverage underlying compute capabilities of hardware

scikit-learn*

Frameworks provide structure that Data Scientists use to build code. Frameworks are more than just libraries, because in addition to callable code, frameworks influence how code is written.

A main virtue of using an optimized framework is that code runs faster. Code that runs faster is just generally more convenient but when we begin looking at applied data science and AI models, we can see more material benefits. Here you will see how optimization, particularly hyperparameter optimization can benefit more than just speed.

These exercises will demonstrate how to apply **the Intel® Extension for Scikit-learn*,** a seamless way to speed up your Scikit-learn application. The acceleration is achieved through the use of the Intel® oneAPI Data Analytics Library (oneDAL). Patching is the term used to extend scikit-learn with Intel optimizations and makes it a well-suited machine learning framework for dealing with real-life problems.

To get optimized versions of many Scikit-learn algorithms using a patch() approach consisting of adding these lines of code prior to importing sklearn:

- **from sklearnex import patch_sklearn**
- **patch_sklearn()**

This exercise relies on installation of Intel® Extension for Scikit-learn*

If you have not already done so, follow the instructions from Week 1 for instructions

Introduction

We will be working with a data set based on [housing prices in Ames, Iowa](). It was compiled for educational use to be a modernized and expanded alternative to the well-known Boston Housing dataset. This version of the data set has had some missing values filled for convenience.

There are an extensive number of features, so they've been described in the table below.

Predictor

- SalePrice: The property's sale price in dollars.

Features

- MoSold: Month Sold
- YrSold: Year Sold

```
<li>SaleType: Type of sale</li>
<li>SaleCondition: Condition of sale</li><br>

<li>MSSubClass: The building class</li>
<li>MSZoning: The general zoning classification</li><br>

<li>Neighborhood: Physical locations within Ames city limits</li>
<li>Street: Type of road access</li>
<li>Alley: Type of alley access</li><br>

<li>LotArea: Lot size in square feet</li>
<li>LotConfig: Lot configuration</li>
<li>LotFrontage: Linear feet of street connected to property</li>
<li>LotShape: General shape of property</li><br>

<li>LandSlope: Slope of property</li>
```

```
        <li>LandContour: Flatness of the property</li><br>

        <li>YearBuilt: Original construction date</li>
        <li>YearRemodAdd: Remodel date</li>
        <li>OverallQual: Overall material and finish quality</li>
        <li>OverallCond: Overall condition rating</li><br>

        <li>Utilities: Type of utilities available</li>
        <li>Foundation: Type of foundation</li>
        <li>Functional: Home functionality rating</li><br>

        <li>BldgType: Type of dwelling</li>
        <li>HouseStyle: Style of dwelling</li><br>

        <li>1stFlrSF: First Floor square feet</li>
        <li>2ndFlrSF: Second floor square feet</li>
        <li>LowQualFinSF: Low quality finished square feet (all floors)</li>
        <li>GrLivArea: Above grade (ground) living area square feet</li>
        <li>TotRmsAbvGrd: Total rooms above grade (does not include bathrooms)</li><br>

        <li>Condition1: Proximity to main road or railroad</li>
        <li>Condition2: Proximity to main road or railroad (if a second is present)</li><br>

        <li>RoofStyle: Type of roof</li>
        <li>RoofMatl: Roof material</li><br>

        <li>ExterQual: Exterior material quality</li>
        <li>ExterCond: Present condition of the material on the exterior</li>
        <li>Exterior1st: Exterior covering on house</li>
        <li>Exterior2nd: Exterior covering on house (if more than one material)</li><br><br>

      </ul>
    </td>

    <td valign="top">
      <ul>
        <li>MasVnrType: Masonry veneer type</li>
        <li>MasVnrArea: Masonry veneer area in square feet</li><br>

        <li>WoodDeckSF: Wood deck area in square feet</li>
        <li>OpenPorchSF: Open porch area in square feet</li>
        <li>EnclosedPorch: Enclosed porch area in square feet</li>
        <li>3SsnPorch: Three season porch area in square feet</li>
        <li>ScreenPorch: Screen porch area in square feet</li><br>

        <li>PoolArea: Pool area in square feet</li>
        <li>PoolQC: Pool quality</li>
        <li>Fence: Fence quality</li>
        <li>PavedDrive: Paved driveway</li><br>

        <li>GarageType: Garage location</li>
        <li>GarageYrBlt: Year garage was built</li>
        <li>GarageFinish: Interior finish of the garage</li>
        <li>GarageCars: Size of garage in car capacity</li>
        <li>GarageArea: Size of garage in square feet</li>
        <li>GarageQual: Garage quality</li>
        <li>GarageCond: Garage condition</li><br>

        <li>Heating: Type of heating</li>
        <li>HeatingQC: Heating quality and condition</li>
        <li>CentralAir: Central air conditioning</li>
        <li>Electrical: Electrical system</li><br>

        <li>FullBath: Full bathrooms above grade</li>
        <li>HalfBath: Half baths above grade</li><br>

        <li>BedroomAbvGr: Number of bedrooms above basement level</li><br>

        <li>KitchenAbvGr: Number of kitchens</li>
        <li>KitchenQual: Kitchen quality</li><br>

        <li>Fireplaces: Number of fireplaces</li>
        <li>FireplaceQu: Fireplace quality</li><br>

        <li>MiscFeature: Miscellaneous feature not covered in other categories</li>
        <li>MiscVal: Value of miscellaneous feature</li><br>

        <li>BsmtQual: Height of the basement</li>
        <li>BsmtCond: General condition of the basement</li>
        <li>BsmtExposure: Walkout or garden level basement walls</li>
        <li>BsmtFinType1: Quality of basement finished area</li>
        <li>BsmtFinSF1: Type 1 finished square feet</li>
        <li>BsmtFinType2: Quality of second finished area (if present)</li>
        <li>BsmtFinSF2: Type 2 finished square feet</li>
        <li>BsmtUnfSF: Unfinished square feet of basement area</li>
        <li>BsmtFullBath: Basement full bathrooms</li>
        <li>BsmtHalfBath: Basement half bathrooms</li>
        <li>TotalBsmtSF: Total square feet of basement area</li>
      </ul>
    </td>
  </tr>
```

```
from __future__ import print_function
import os
data_path = [ r'G:\MITADT\2023-24 Sem-II\AIEC Machine Learning lab\Resources\Class3-Train_Test_Splits_Validation_Linear_Regression\Class3-Train_Test

from sklearnex import patch_sklearn
patch_sklearn()

from sklearn.preprocessing import OneHotEncoder, LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler, MinMaxScaler, MaxAbsScaler
```

```
Intel(R) Extension for Scikit-learn* enabled (https://github.com/intel/scikit-learn-intelex)
```

Question 1

- Import the data using Pandas and examine the shape. There are 79 feature columns plus the predictor, the sale price (`SalePrice`).
- There are three different types: integers (`int64`), floats (`float64`), and strings (`object`, categoricals). Examine how many there are of each data type.

```
import pandas as pd
import numpy as np

filepath = os.sep.join(data_path + ['Ames_Housing_Sales.csv'])
data = pd.read_csv(r'G:\MITADT\2023-24 Sem-II\AIEC Machine Learning lab\Resources\Class3-Train_Test_Splits_Validation_Linear_Regression\Class3-Tra

print(data.shape)
```

```
(1379, 80)
```

```
data.dtypes.value_counts()
```

```
object     43
float64    21
int64      16
dtype: int64
```

Question 2

As discussed in the lecture, a significant challenge, particularly when dealing with data that have many columns, is ensuring each column gets encoded correctly.

This is particularly true with data columns that are ordered categoricals (ordinals) vs unordered categoricals. Unordered categoricals should be one-hot encoded, however this can significantly increase the number of features and creates features that are highly correlated with each other.

Determine how many total features would be present, relative to what currently exists, if all string (object) features are one-hot encoded. Recall that the total number of one-hot encoded columns is `n-1`, where `n` is the number of categories.

```
# Select the object (string) columns
mask = data.dtypes == object
categorical_cols = data.columns[mask]


num_ohc_cols = (data[categorical_cols]
                .apply(lambda x: x.nunique())
                .sort_values(ascending=False))


small_num_ohc_cols = num_ohc_cols.loc[num_ohc_cols>1]

small_num_ohc_cols -= 1

small_num_ohc_cols.sum()
```

```
215
```

Question 3

Let's create a new data set where all of the above categorical features will be one-hot encoded. We can fit this data and see how it affects the results.

- Used the dataframe `.copy()` method to create a completely separate copy of the dataframe for one-hot encoding
- On this new dataframe, one-hot encode each of the appropriate columns and add it back to the dataframe. Be sure to drop the original column.
- For the data that are not one-hot encoded, drop the columns that are string categoricals.

For the first step, numerically encoding the string categoricals, either Scikit-learn;s `LabelEncoder` or `DictVectorizer` can be used. However, the former is probably easier since it doesn't require specifying a numerical value for each category, and we are going to one-hot encode all of the numerical values anyway. (Can you think of a time when `DictVectorizer` might be preferred?)

```
data_ohc = data.copy()

le = LabelEncoder()
ohc = OneHotEncoder()

for col in num_ohc_cols.index:

    dat = le.fit_transform(data_ohc[col]).astype(int)

    data_ohc = data_ohc.drop(col, axis=1)

    new_dat = ohc.fit_transform(dat.reshape(-1,1))

    n_cols = new_dat.shape[1]
    col_names = ['_'.join([col, str(x)]) for x in range(n_cols)]

    new_df = pd.DataFrame(new_dat.toarray(),
                          index=data_ohc.index,
                          columns=col_names)

   data_ohc = pd.concat([data_ohc, new_df], axis=1)
```

```
data_ohc.shape[1] - data.shape[1]
```

```
215
```

```
print(data.shape[1])

data = data.drop(num_ohc_cols.index, axis=1)

print(data.shape[1])
```

```
80
37
```

Question 4

- Create train and test splits of both data sets. To ensure the data gets split the same way, use the same `random_state` in each of the two splits.
- For each data set, fit a basic linear regression model on the training data.
- Calculate the mean squared error on both the train and test sets for the respective models. Which model produces smaller error on the test data and why?

```
y_col = 'SalePrice'

feature_cols = [x for x in data.columns if x != y_col]
X_data = data[feature_cols]
y_data = data[y_col]

X_train, X_test, y_train, y_test = train_test_split(X_data, y_data,
                                                    test_size=0.3, random_state=42)

feature_cols = [x for x in data_ohc.columns if x != y_col]
X_data_ohc = data_ohc[feature_cols]
y_data_ohc = data_ohc[y_col]

X_train_ohc, X_test_ohc, y_train_ohc, y_test_ohc = train_test_split(X_data_ohc, y_data_ohc,
                                                    test_size=0.3, random_state=42)
```

```
# Compare the indices to ensure they are identical
(X_train_ohc.index == X_train.index).all()
```

```
True
```

```
LR = LinearRegression()

error_df = list()

LR = LR.fit(X_train.values, y_train.values)
y_train_pred = LR.predict(X_train.values)
y_test_pred = LR.predict(X_test.values)

error_df.append(pd.Series({'train': mean_squared_error(y_train, y_train_pred),
                           'test' : mean_squared_error(y_test,  y_test_pred)},
                           name='no enc'))
LR = LinearRegression()
LR = LR.fit(X_train_ohc.values, y_train_ohc.values)
y_train_ohc_pred = LR.predict(X_train_ohc.values)
y_test_ohc_pred = LR.predict(X_test_ohc.values)

error_df.append(pd.Series({'train': mean_squared_error(y_train_ohc, y_train_ohc_pred),
                           'test' : mean_squared_error(y_test_ohc,  y_test_ohc_pred)},
                          name='one-hot enc'))
error_df = pd.concat(error_df, axis=1)
error_df
```

	no enc	one-hot enc
train	1.131507e+09	3.177266e+08
test	1.372182e+09	4.777314e+20

Note that the error values on the one-hot encoded data are very different for the train and test data. In particular, the errors on the test data are much higher. Based on the lecture, this is because the one-hot encoded model is overfitting the data. We will learn how to deal with issues like this in the next lesson.

Question 5

For each of the data sets (one-hot encoded and not encoded):

- Scale the all the non-hot encoded values using one of the following: `StandardScaler`, `MinMaxScaler`, `MaxAbsScaler`.
- Compare the error calculated on the test sets

Be sure to calculate the skew (to decide if a transformation should be done) and fit the scaler on *ONLY* the training data, but then apply it to both the train and test data identically.

```
pd.options.mode.chained_assignment = None
```

```
scalers = {'standard': StandardScaler(),
           'minmax': MinMaxScaler(),
           'maxabs': MaxAbsScaler()}

training_test_sets = {
    'not_encoded': (X_train, y_train, X_test, y_test),
    'one_hot_encoded': (X_train_ohc, y_train_ohc, X_test_ohc, y_test_ohc)}

mask = X_train.dtypes == float
float_columns = X_train.columns[mask]

errors = {}
for encoding_label, (_X_train, _y_train, _X_test, _y_test) in training_test_sets.items():
    for scaler_label, scaler in scalers.items():
        trainingset = _X_train.copy()
        testset = _X_test.copy()
        trainingset[float_columns] = scaler.fit_transform(trainingset[float_columns])
        testset[float_columns] = scaler.transform(testset[float_columns])

        LR = LinearRegression()
        LR.fit(trainingset.values, _y_train.values)
        predictions = LR.predict(testset.values)
        key = encoding_label + ' - ' + scaler_label + 'scaling'
        errors[key] = mean_squared_error(_y_test, predictions)

errors = pd.Series(errors)
print(errors.to_string())
print('-' * 80)
for key, error_val in errors.items():
    print('{} {:9.8g}'.format(key, error_val))
```

```
not_encoded - standardscaling        1.372182e+09
not_encoded - minmaxscaling          1.372182e+09
not_encoded - maxabsscaling          1.372182e+09
one_hot_encoded - standardscaling    8.065328e+09
one_hot_encoded - minmaxscaling      8.065328e+09
one_hot_encoded - maxabsscaling      8.065328e+09
--------------------------------------------------------------------------------
not_encoded - standardscaling 1.3721824e+09
not_encoded - minmaxscaling 1.3721824e+09
not_encoded - maxabsscaling 1.3721824e+09
one_hot_encoded - standardscaling 8.0653276e+09
one_hot_encoded - minmaxscaling 8.0653276e+09
one_hot_encoded - maxabsscaling 8.0653276e+09
```

Question 6

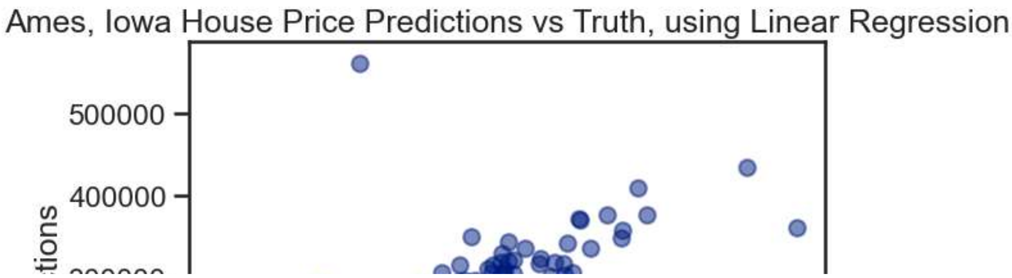
Plot predictions vs actual for one of the models.

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

sns.set_context('talk')
sns.set_style('ticks')
sns.set_palette('dark')

ax = plt.axes()
ax.scatter(y_test, y_test_pred, alpha=.5)

ax.set(xlabel='Ground truth',
       ylabel='Predictions',
       title='Ames, Iowa House Price Predictions vs Truth, using Linear Regression');
```


Ames, Iowa House Price Predictions vs Truth, using Linear Regression
500000
400000