w3-ML-regressions

March 11, 2021

1 Big Data for Public Policy

1.1 Machine Learning - Regressions

1.2 Malka Guillot

1.3 ETH Zürich | 860-0033-00L

Regression belongs like classification to the field of **supervised learning**.

Regression predicts numerical values in contrast to classification which predicts categories.

Other differences are:

Accuracy in measured differently

Other algorithms

[ ]:
```
# Common imports
import numpy as np
import os
import pandas as pd

# To plot pretty figures
%matplotlib inline
%config InlineBackend.figure_format = 'retina'
import matplotlib as mpl
import matplotlib.pyplot as plt
#%matplotlib notebook
mpl.rc('axes', labelsize=14)
mpl.rc('xtick', labelsize=12)
mpl.rc('ytick', labelsize=12)

import seaborn as sns
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)
warnings.filterwarnings('ignore', category=DeprecationWarning)
warnings.filterwarnings = lambda *a, **kw: None
```

```python
# to make this notebook's output identical at every run
np.random.seed(42)
```

1.4 General ML Procedure

0. Look at the data
1. Select a ML method (eg. LASSO)
2. Draw randomly a hold-out sample from the data
3. Estimate the ML model using different hyperparameters
4. Select the optimal hyperparameters
5. Predict $\hat{Y}$ using hyperparameters and extrapolated the fitted values to the retarded hold-out-sample
6. Evaluation the prediction power of the ML in the hold-out-sample

1.5 Scikit-Learn Design Overview

1.5.1 Estimator: an object that can estimate parameters

- e.g. `linear_models.LinearRegression`
- Estimation performed by `fit()` method
- Exogenous parameters (provided by the researcher) are called `hyperparameters`

1.5.2 Transformer (preprocessor): An object that transforms a data set.

- e.g. `preprocessing.StandardScaler`
- Transformation is performed by the `transform()` method.
- The convenience method `fit_transform()` both fits an estimator and returns the transformed input data set.

1.5.3 Predictor: An object that forms a prediction from an input data set.

- e.g. `LinearRegression`, after training
- The `predict()` method forms the predictions.
- It also has a `score()` method that measures the quality of the predictions given a test set.

1.5.4 Miscellaneous

- **Inspection**: Hyperparameters and parameters are accessible. Learned parameters have an underscore suffix (e.g.`lin_reg.coef_`)
- **Non-proliferation of classes**: Use native Python data types; existing building blocks are used as much as possible.
- **Sensible defaults**: Provides reasonable default values for hyperparameters – easy to get a good baseline up and running

1.6 Set up and load data

1.7 Boston housing data

```
[ ]: # Scikit-Learn  0.20 is required
     import sklearn
```

We use as an example the **Boston housing data** (from `sklearn`), which contains 13 attributes of housing markets around Boston. The data was collected in 1978 and each of the 506 entries represent aggregated data about 14 features for homes from various suburbs in Boston, Massachusetts.

The objective is to **predict the value of prices** of the house using the given features

```
[ ]: from sklearn.datasets import load_boston
     data = load_boston() # object is a dictionary
     data.keys()
```

Data Set Characteristics:

```
[ ]: print(data['DESCR'])
```

1.7.1 Create X and y

```
[ ]: X_full, y_full = data.data, data.target
     n_samples = X_full.shape[0]
     n_features = X_full.shape[1]
```

```
[ ]: X_df=pd.DataFrame(X_full, columns=data['feature_names']) # to dataframe format
     X_df.head()
```

1.7.2 Look for null values in the dataset

```
[ ]: X_df.isnull().sum()
```

There is none

1.8 Exploratory Data Analysis

1.8.1 Quantity to predict= price (`target` or y)

Before the regression, let us inspect the features and their distributions.

```
[ ]: y_full.shape
```

[ ]:
```
sns.set(rc={'figure.figsize':(11.7,8.27)})
plt.hist(y_full, bins=30)
plt.xlabel("House prices in $1000", size=15)
plt.ylabel('count', size=15)
plt.title('Distribution of median price in each neighborhood', size=20)
plt.show()
```

1.8.2 Features (X) used for prediction

[ ]:
```
X_full.shape
```

Distributions **Histogram plots** to look at the distribution

[ ]:
```
X_df.hist(bins=50, figsize=(20,15))
plt.show()
```

1.8.3 Correlations

Boston Correlation Heatmap Example with Seaborn

The seaborn package offers a heatmap that will allow a two-dimensional graphical representation of the Boston data. The heatmap will represent the individual values that are contained in a matrix are represented as colors.

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
sns.set(rc={'figure.figsize':(8.5,5)})
correlation_matrix = X_df.corr().round(2)
sns.heatmap(correlation_matrix) #annot=True
plt.show()
```

1.8.4 Check for multicolinearity

An important point in selecting features for a linear regression model is to check for **multicolinearity**.

The features RAD, TAX have a correlation of 0.91. These feature pairs are strongly correlated to each other. This can affect the model.

Same goes for the features DIS and AGE which have a correlation of -0.75.

1.8.5 Correlation plots

```
[ ]: from pandas.plotting import scatter_matrix
     scatter_matrix(X_df[['DIS', 'AGE','RAD', 'TAX']], figsize=(12, 8))
```

1.8.6 Scatter plot relative to the target (price)

```
[ ]: for feature_name in X_df.columns:
         plt.figure(figsize=(4, 3))
         plt.scatter(X_df[feature_name], y_full, alpha=0.1)
         plt.ylabel('Price', size=15)
         plt.xlabel(feature_name, size=15)
         plt.tight_layout()
```

1.8.7 What can we say ?

- The prices increase as the value of RM increases linearly. There are few outliers and the data seems to be capped at 50.
- The prices tend to decrease with an increase in LSTAT. Though it doesn't look to be following exactly a linear line.

1.9 Prepare the data for ML algorithms

1.9.1 Drop some labeled observations:

Drop the observations with price >=50 (because of the right censure)

```
[ ]: mask=y_full<50

     y_full=y_full[mask==True]
     X_full=X_full[mask==True]
     X_df=X_df[mask==True]
```

1.10 Split train test sets

1.10.1 Using train_test_split

Pure ramdomness of the sampling method

```
[ ]: from sklearn.model_selection import train_test_split
     X_train, X_test, y_train, y_test = train_test_split(X_full, y_full,test_size=0.
      ↪2, random_state=1)
     print("train data", X_train.shape, y_train.shape)
print("test data", X_test.shape,  y_test.shape)
```

1.10.2 Data cleaning

The missing features should be: 1. dropped 2. imputed to some value (zero, the mean, the median…)

1.10.3 Feature Scaling

Most common scaling methods: - **standardization**= normalization by substracting the mean and dividing by the standard deviation (values are not bounded) - **Min-max scaling**= normalization by substracting the minimum and dividing by the maximum (values between `0` and `1`)

```
[ ]: from sklearn.preprocessing import StandardScaler, MinMaxScaler
     scaler = StandardScaler().fit(X_train)
     X_train_scaled = scaler.transform(X_train)
     X_test_scaled = scaler.transform(X_test)
```

1.11 Select and Train a Model

Regression algorithm (we consider firs the `LinearRegression`, more algorithms will be discussed later): ### First algorithm: Simple Linear Regression

```
[ ]: # our first machine learning model
     from sklearn.linear_model import LinearRegression
     lin_reg = LinearRegression()
```

`<i class="fa fa-warning"></i>&nbsp;<strong><code>scikit-learn</code> API</strong>`

In scikit-learn all regression algorithms have:

a fit() method to learn from data, and

and a subsequent predict() method for predicting numbers from input features.

```
[ ]: lin_reg.fit(X_train, y_train)
     print("R-squared for training dataset:{}".
           format(np.round(lin_reg.score(X_train, y_train), 2)))
```

```
[ ]: lin_reg.fit(X_train_scaled, y_train)
     print("R-squared for training dataset & scaled features:{}".
           format(np.round(lin_reg.score(X_train_scaled, y_train), 2)))
```

1.11.1 Coefficients of the linear regression

```
[ ]: features = list(X_df.columns)

     print('The coefficients of the features from the linear model:')
     print(dict(zip(features, [round(x, 2) for x in lin_reg.coef_])))
```

1.12 Metrics / error measures

`scikit-learn` offers the following metrics for measuring regression quality:

1.12.1 Mean absolute error

Taking absolute values before adding up the deviatons assures that deviations with different signs can not cancel out.

`<i class="fa fa-info-circle"></i>&nbsp; <strong>mean absolute error</strong> is defined as`

$$\frac{1}{n}\left(\,|y_1 - \hat{y}_1| \,+\, |y_2 - \hat{y}_2| \,+\, \ldots \,+\, |y_n - \hat{y}_n|\,\right)$$

`neg_mean_absolute_error` in `scikit-learn`.

1.12.2 Mean squared error

Here we replace the absolute difference by its squared difference. Squaring also insures positive differeces.

`<i class="fa fa-info-circle"></i>&nbsp; <strong>mean squared error</strong> is defined as`

$$\frac{1}{n}\left(\,(y_1 - \hat{y}_1)^2 \,+\, (y_2 - \hat{y}_2)^2 \,\ldots \,+\, (y_n - \hat{y}_n)^2\,\right)$$

This measure is more sensitive to **outliers**: A few larger differences contribute more significantly to a larger mean squared error.

`neg_mean_squared_error` in `scikit-learn`.

1.12.3 Median absolute error

Here we replace mean calculation by median.

`<i class="fa fa-info-circle"></i>&nbsp; <strong>median absolute error</strong> is defined as`

$$\text{median}\left(|y_1 - \hat{y}_1|, |y_2 - \hat{y}_2|, \ldots, |y_n - \hat{y}_n|\right)$$

This measure is less sensitive to outliers: A few larger differences will not contribute significantly to a larger error value.

`neg_median_absolute_error` in `scikit-learn`.

1.12.4 Mean squared log error

The formula for this metric can be found here.

This metric is recommended **when your target values are distributed over a huge range of values**, like popoluation numbers.

The previous error metrics would put a larger weight on large target values.

The name is `neg_mean_squared_log_error`

1.12.5 In-sample performance with MSE

```
[ ]: from sklearn.metrics import mean_squared_error

y_train_pred = lin_reg.predict(X_train_scaled)
train_mse = mean_squared_error(y_train, y_train_pred)
train_rmse = np.sqrt(train_mse)
print("RMSE: %s" % train_rmse) # = np.sqrt(np.mean((predicted - expected) **␣
 ↪2))
```

Your turn

```
Compute:
<ul>
    <li>the out-of-sample mean squared error
    </li>
    <li> mean  absolute error (using $mean\ absolute\ error$ from $sklearn.metrics$: you need
    </li>
</ul>
```

```
[ ]:
```

```
[ ]:
```

1.12.6 Explained variance and R^2-score

Two other scores to mention are *explained variance* and R^2-score. For both larger values indicate better regression results.

The $R^2$-score corresponds to **the proportion of variance (of y) that has been explained by the independent variables in the model**. It takes values in the range 0..1. The name within `scikit-learn` is R2.

<i class="fa fa-info-circle"></i>&nbsp; <strong>$R^2$</strong> is defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

The formula for explained variance, the score takes values up to 1. The name within `scikit-learn` is `explained_variance`.

```
[ ]: from sklearn.metrics import r2_score
     r2=round(r2_score(y_test, y_test_pred), 2)
     print("R2: %s" % r2)
```

1.12.7 Binned Regression Plots

```
[ ]: # Regplot
     g=sns.regplot(x= y_test_pred, y=y_test, x_bins=100)
     g=g.set_title("Test sample")

     plt.xlabel("Predicted prices: $\hat{Y}_i$")
     plt.ylabel("Prices: $Y_i$")
     plt.annotate('R2={}'.format(r2),
                  xy=(1, 0),  xycoords='axes fraction',
                  horizontalalignment='right',
                  verticalalignment='bottom')
     plt.annotate('Notes: 100 binned',
                  xy=(0, 0),  xycoords='figure fraction',
                  horizontalalignment='left',
                  verticalalignment='bottom')
     plt.plot([0, 50], [0, 50], '--k')
     plt.axis('tight')
     plt.tight_layout()
     plt.show(g)
```

1.12.8 Plotting Regression Residuals

```
[ ]: #Let us plot how good given and predicted values match on the training data set␣
      ↪(sic !).
     def plot_fit_quality(values_test, predicted):

         plt.figure(figsize=(12, 4))
         plt.subplot(1, 2, 1)
```

```
    x = np.arange(len(predicted))
    plt.scatter(x, predicted - values_test, color='steelblue', marker='o')

    plt.plot([0, len(predicted)], [0, 0], "k:")

    max_diff = np.max(np.abs(predicted - values_test))
    plt.ylim([-max_diff, max_diff])

    plt.ylabel("error")
    plt.xlabel("sample id")

    plt.subplot(1, 2, 2)

    plt.scatter(x, (predicted - values_test) / values_test, color='steelblue',␣
 ↪marker='o')
    plt.plot([0, len(predicted)], [0, 0], "k:")
    plt.ylim([-.5, .5])

    plt.ylabel("relative error")
    plt.xlabel("sample id")

plot_fit_quality(y_test, y_test_pred)
```

Your turn

`Train a ridge model and look at the goodeness of fit`

[ ]:
```
from sklearn.linear_model import Ridge
```

[ ]:
```

```

[ ]:
```
print('The coefficients of the features from the Ridge model:')
print(dict(zip(features, [round(x, 2) for x in ridge_reg.coef_])))
```

Polynomial regression

[ ]:
```
from sklearn.preprocessing import PolynomialFeatures
poly_features=PolynomialFeatures(degree=2)
X_train_poly=poly_features.fit_transform(X_train)
X_test_poly=poly_features.fit_transform(X_test)

lin_reg = LinearRegression()
lin_reg.fit(X_train_poly, y_train)

y_test_pred = lin_reg.predict(X_test_poly)
test_rmse = mean_squared_error(y_test,y_test_pred)
test_rmse = np.sqrt(test_rmse)
print("test RMS: %s" % test_rmse)
print("train R2: %s" % round(r2_score(y_train, y_train_pred), 2))
print("test R2: %s" % round(r2_score(y_test, y_test_pred), 2))
```

1.12.9 Lasso regression

[ ]:
```
from sklearn.linear_model import Lasso
lasso_reg=Lasso(alpha=1)
lasso_reg.fit(X_train, y_train)

y_test_pred = lasso_reg.predict(X_test)
test_mse = mean_squared_error(y_test, y_test_pred)
test_rmse = np.sqrt(test_mse)
print("test RMS: %s" % test_rmse)
print("train R2: %s" % round(r2_score(y_train, y_train_pred), 2))
print("test R2: %s" % round(r2_score(y_test, y_test_pred), 2))
```

1.12.10 Lasso regression – with scaled X

[ ]:
```
from sklearn.linear_model import Lasso
lasso_reg=Lasso(alpha=1)
lasso_reg.fit(X_train_scaled, y_train)

y_test_pred = lasso_reg.predict(X_test_scaled)
test_mse = mean_squared_error(y_test, y_test_pred)
test_rmse = np.sqrt(test_mse)
print("test RMS: %s" % test_rmse)
print("train R2: %s" % round(r2_score(y_train, y_train_pred), 2))
print("test R2: %s" % round(r2_score(y_test, y_test_pred), 2))
```

[ ]:
```
print('The coefficients of the features from the Lasso model:')
print(dict(zip(features, [round(x, 2) for x in lasso_reg.coef_])))
```

Elastic Net

[ ]:
```
from sklearn.linear_model import ElasticNet
elanet_reg=ElasticNet(random_state=0)
elanet_reg.fit(X_train_scaled, y_train)

y_test_pred = elanet_reg.predict(X_test_scaled)
test_mse = mean_squared_error(y_test, y_test_pred)
test_rmse = np.sqrt(test_mse)
print("test RMS: %s" % test_rmse)
print("train R2: %s" % round(r2_score(y_train, y_train_pred), 2))
print("test R2: %s" % round(r2_score(y_test, y_test_pred), 2))
```

[ ]:
```
print('The coefficients of the features from the Lasso model:')
print(dict(zip(features, [round(x, 2) for x in elanet_reg.coef_])))
```

1.13 Setting the regularization parameter: generalized Cross-Validation.

[ ]:
```
alphas=np.logspace(-6, 6, 13)
```

[ ]:
```
from sklearn import linear_model
lassocv_reg = linear_model.LassoCV(alphas=alphas)
lassocv_reg.fit(X_train, y_train)
alpha=lassocv_reg.alpha_
print("Best alpha", alpha)
```

1.13.1 Then re-run the model using the best α

[ ]:
```
lasso_reg=Lasso(alpha=alpha)

lasso_reg.fit(X_train_scaled, y_train)

y_train_pred=lasso_reg.predict(X_train_scaled)
y_test_pred = lasso_reg.predict(X_test_scaled)
test_mse = mean_squared_error(y_test, y_test_pred)
test_rmse = np.sqrt(test_mse)
print("test RMS: %s" % test_rmse)
print("train R2: %s" % round(r2_score(y_train, y_train_pred), 2))
print("test R2: %s" % round(r2_score(y_test, y_test_pred), 2))
```

1.14 Fine-tuning of the Model

1.14.1 Model Evaluation using Cross-Validation

```
<i class="fa fa-info-circle"></i>&nbsp; cross_val_score expect a utility function rather than a
```

[ ]:
```
from sklearn.model_selection import cross_val_score, cross_val_predict
# Perform 6-fold cross validation
scores = cross_val_score(elanet_reg, X_train_scaled, y_train,
                         scoring="neg_mean_squared_error", cv=5)
scores
```

1.14.2 Make cross validated predictions

```
[ ]: y_train_pred_cv = cross_val_predict(elanet_reg, X_train_scaled, y_train, cv=5)
```

```
[ ]: ## plt.figure(figsize=(4, 3))
     plt.scatter(y_train, y_train_pred_cv)
     plt.plot([0, 50], [0, 50], '--k')
     plt.axis('tight')
     plt.xlabel('True price ($1000s)')
     plt.ylabel('Predicted price ($1000s)')
     plt.tight_layout()
```

```
[ ]: accuracy =r2_score(y_train, y_train_pred_cv)
     print('Cross-Predicted Accuracy:', accuracy)
```

1.14.3 Hyperparameters tuning

```
[ ]: from sklearn.model_selection import GridSearchCV
     param_grid = [
       {'alpha': [0.0001, 0.001, 0.01, 0.1 ,1, 10],
           'l1_ratio':[.1,.5,.9,1]}

     ]
     # train across 5 folds, that's a total of (12+6)*5=90 rounds of training
     grid_search = GridSearchCV(elanet_reg, param_grid, cv=5,
                                scoring='neg_mean_squared_error',
                                return_train_score=True)
     grid_search.fit(X_train, y_train)
```

1.14.4 The best hyperparameter combination found:

```
[ ]: grid_search.best_params_
```

```
[ ]: grid_search.best_estimator_
```

1.14.5 Score of each hyperparameter combination tested during the grid search

```
[ ]: cvres = grid_search.cv_results_
     for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
         print(np.sqrt(-mean_score), params)
```

1.14.6 In a DataFrame

[ ]:
```
df_cvres=pd.DataFrame(cvres)
df_cvres['mean_test_score_pos_sqrt']=df_cvres['mean_test_score'].apply(lambda x:
 ↪ np.sqrt(-x))
df_cvres['log_param_alpha']=df_cvres['param_alpha'].apply(lambda x: np.log(x))
df_cvres.head()
```

1.14.7 Vizualize the grid search results

[ ]:
```
_, ax = plt.subplots(1,1)
plt.plot(df_cvres["log_param_alpha"], df_cvres["mean_test_score_pos_sqrt"])
ax.set_title("Grid Search", fontsize=20, fontweight='bold')
ax.set_xlabel("$\log (alpha)$", fontsize=16)
ax.set_ylabel('Avg. mean test score', fontsize=16)
```

1.14.8 Other possibility: for randomized search of hyperparameters

[ ]:
```
from sklearn.model_selection import RandomizedSearchCV
```

1.15 What is not covered today:

- more advanced regression algorithms (gradient boosting, random forest)
- classification algorithm
- pipelines