


# Data Science Lab

## Scikit-learn
## Regression

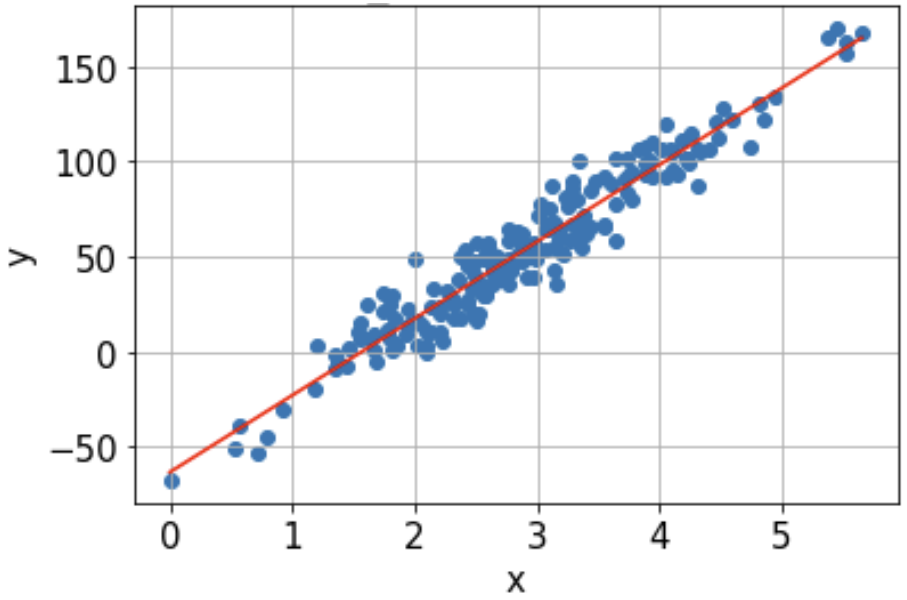
Andrea Pasini
Flavio Giobergia

DataBase and Data Mining Group

# Linear regression

- Regression with Scikit-learn

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_train, y_train)
y_test_pred = reg.predict(X_test)
```



# Evaluating regression

PoliTo DBMG

- Evaluation metrics for regression:
  - MAE (Mean Absolute Error)
  - MSE (Mean Squared Error)
  - $R^2$

```
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

# Compute R2, MAE and MSE:
r2 = r2_score(y_test, y_test_pred)
mae = mean_absolute_error(y_test, y_test_pred)
mse = mean_squared_error(y_test, y_test_pred)
```

# Evaluating regression

PoliTo DBMG

- Evaluation with cross_val_score()

```
from sklearn.model_selection import cross_val_score

reg = LinearRegression()
r2 = cross_val_score(reg, X, y, cv=5, scoring='r2')
```

- Parameters:
  - cv = number of partitions for cross-validation
  - scoring = scoring function for the evaluation
    - E.g. 'r2', '**neg**_mean_squared_error'
- Similarly, we can use cross_val_predict()

# Notebook Examples

- 4b-Scikitlearn-Linear-Regression.ipynb
  - 1. Simple linear regression
  - 2. Linear regression with multiple input features

# Polynomial regression

- Polynomial regression
  - Useful when the data does not follow a linear trend
- **It consists of:**
  - Computing new **features** that are power functions of the input features
  - Applying **linear** regression on the new features

input vector = $[x_1, x_2]$

↓

degree(2) features = $[1, x_1, x_2, x_1^2, x_2^2, x_1x_2]$

↓

$f(x) = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_2^2 + w_5x_1x_2$

# Polynomial regression



- Extracting polynomial features

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(5)
X_poly = poly.fit_transform(X)
```

- Parameters:
  - degree
    - Integer: the maximal degree of the computed features
    - Tuple: min and max degrees of the computed features
  - interaction_only: if True, only include interaction terms
    - Exclude terms with power $\geq 2$ of the same input feature
  - include_bias: if True, add a bias column (column of ones)
- Output of fit_transform():
  - A 2D **NumPy array** with the new feature matrix

# Polynomial regression

PoliTo DBMG

- Building a **pipeline** with polynomial features and linear regression

```
from sklearn.pipeline import make_pipeline
reg = make_pipeline(PolynomialFeatures(5), LinearRegression())
reg.fit(X_train, y_train)
y_test_pred = ret.predict(X_test)
```

- Pipelines are objects that allow concatenating multiple Scikit-learn models
  - N-1 transformers, 1 final predictor

# Polynomial regression



- **Ridge:**

```
from sklearn.linear_model import Ridge
reg = Ridge(alpha=0.5)
```

- **Lasso:**

```
from sklearn.linear_model import Lasso
reg = Lasso(alpha=0.5)
```

# Notebook Examples

- 4c-Scikitlearn-Polynomial-Regression.ipynb
  - 1. Polynomial regression
  - 2. Overfitting and regularization

# Hyperparameters selection

- Hyperparameters vs parameters
  - Hyperparameters are selected by the user
  - Parameters are computed by the algorithm during training
- **Important**: hyperparameters cannot be set by finding the values that give the best results on the test set
  - This methodology **will overfit the test set**
  - We would be using **information from the test data** to select some **training** hyperparameters

# Hyperparameters selection



- Step 1. define the grid of all hyperparameters
  - A **dictionary** with the **parameter values** that to tune
  - E.g. for Ridge regression:

```
param_grid = {'alpha' : [0.1, 0.2],
              'fit_intercept' : [True, False]}
```

  - ParameterGrid() enumerates all possible combinations
  - Scikit-learn will try all the **combinations**:
    - {alpha=0.1,fit_intercept=True}
    - {alpha=0.1,fit_intercept=False}
    - {alpha=0.2,fit_intercept=True}
    - {alpha=0.2,fit_intercept=False}

# Hyperparameters selection

- Step 2. Define a model, and call GridSearchCV

```
from sklearn.model_selection import GridSearchCV
reg = Ridge()
gridsearch = GridSearchCV(reg, param_grid, scoring='r2', cv=5)
gridsearch.fit(X_train, y_train)
```

- This code will pick the best configuration of the param grid, for Ridge model,
  - According to the **$R^2$** score
  - Using a cross validation with **k=5** partitions
- Passing `n_jobs=-1`, parallelizes across all available processors

# Hyperparameters selection

- The best parameter configuration can be found in the *best_params_* attribute of the gridsearch object
- An instance of the model with the best configuration is available in best_estimator_
  - **Important**: it is trained on the **whole dataset**!

```
...
gridsearch.fit(X_train, y_train)
print(gridsearch.best_params_['alpha'])
print(gridsearch.best_params_['fit_intercept')


best_configured_model = gridsearch.best_estimator_
```

# Other strategies

- RandomizedSearchCV()
  - Same params as GridSearchCV()
  - An additional `n_iter=[number]` defines the total number of configurations to try
- BayesSearchCV()
  - Not in scikit-learn, but in scikit-optimize
    - https://scikit-optimize.github.io/
  - Uses Bayesian optimization
  - Same params as RandomizedSearchCV()
  - The optimizer can be configured via optimizer_kwargs

# Notebook Examples

- 4c-Scikitlearn-Polynomial-Regression.ipynb
  - 3. Grid-search to select model hyperparameters