chapter-2-2

February 2, 2022

Run in Google Colab

1 Validation and Testing

TODO - Per completare questa parte vedere i notebooks di Flavio per il workshop

1.1 Understanding the scikit-learn estimator API

1.1.1 Transformers

The so-called ***transformer*** classes in scikit-learn, which are used for data transformation, have two essential methods: **fit** and **transform**. The fit method is used to learn the parameters from the training data, and the transform method uses those parameters to transform the data. Any data array that is to be transformed needs to have the same number of features as the data array that was used to fit the model.

1.1.2 Estimators

The ***estimators*** classes in scikit-learn, have an API that is conceptually very similar to the transformer class. Estimators have a **predict** method but can also have a transform method. We also used the fit method to learn the parameters of a model when we trained those estimators for classification. However, in supervised learning tasks, we additionally provide the class labels for fitting the model, which can then be used to make predictions about new, unlabeled data examples via the predict method, as illustrated in the following figure:

1.2 Validation and Testing

When data is used for forecasting there is a danger that the machine learning model will work very well for data, but will not generalize well to other data. An obvious point is that it is important that the data used in a machine learning model be representative of the situations to which the model is to be applied. It is also important to test a model out-of-sample, by this we mean that the model should be tested on data that is different from the sample data used to determine the parameters of the model.

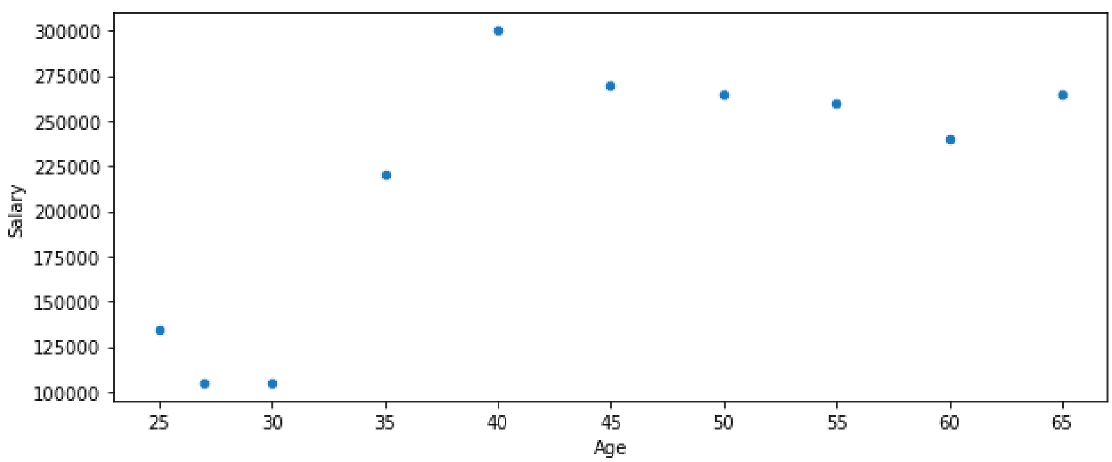
Data scientist refer to the sample data as the **training set** and the data used to determine the accuracy of the model as the **test set**, often a **validation set** is used as well as we explain later;

[32]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[33]:
```
# Load the Pandas libraries with alias 'pd'
import pandas as pd
# Read data from file 'salary_vs_age_1.csv'
# (in the same directory that your python process is based)
# Control delimiters, with read_table
df1 = pd.read_table(path + "salary_vs_age_1.csv", sep=";")
# Preview the first 5 lines of the loaded data
print(df1.head())
```

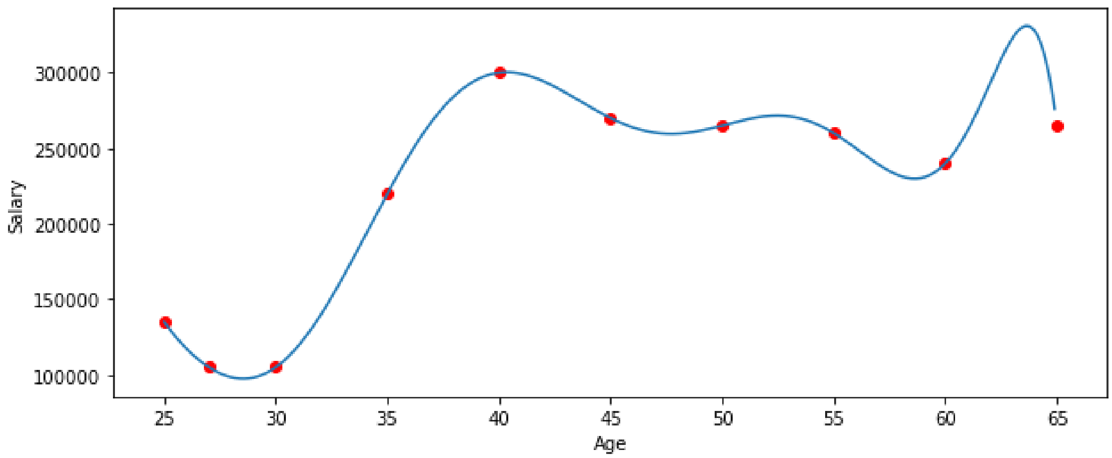
```
   Age  Salary
0   25  135000
1   27  105000
2   30  105000
3   35  220000
4   40  300000
```

[34]:
```
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [10, 4]
ax=plt.gca()

df1.plot(x ='Age', y='Salary', kind = 'scatter', ax=ax)
plt.show()
```



polynomial fitting with pandas

[35]:
```
import numpy as np

x1 = df1['Age']
y1 = df1['Salary']

n = len(x1)

degree = 9

weights = np.polyfit(x1, y1, degree)
model   = np.poly1d(weights)

xx1 = np.arange(x1[0], x1[n-1], 0.1)
plt.plot(xx1, model(xx1))
plt.xlabel("Age")
plt.ylabel("Salary")
plt.scatter(x1,y1, color='red')
plt.show()
```



[36]:
```
y1  = np.array(y1)
yy1 = np.array(model(x1))

rmse = np.sqrt(np.sum((y1-yy1)**2)/(n-1))

print('Root Mean Square Error:')
print(rmse)
```

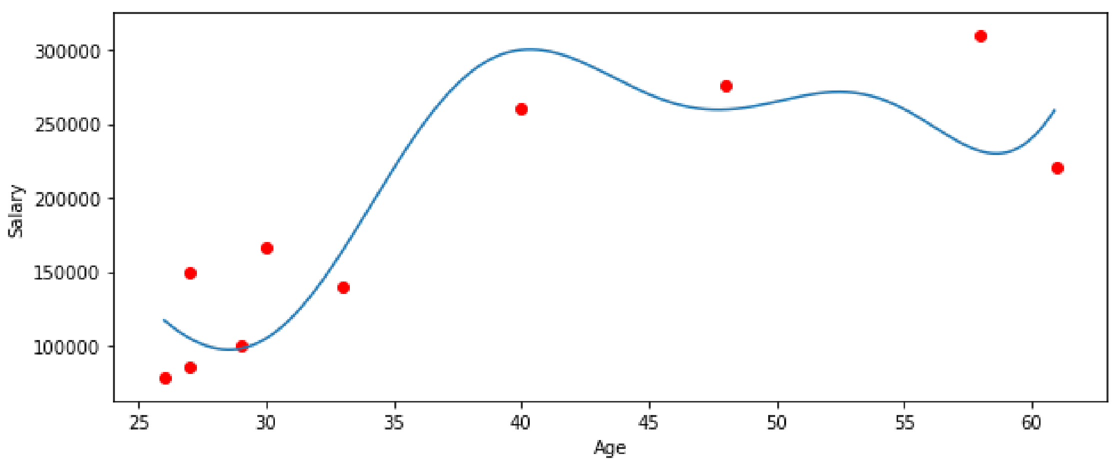
```
Root Mean Square Error:
0.0007184405154295555
```

[37]:
```python
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[38]:
```python
df2 = pd.read_table(path + "salary_vs_age_2.csv", sep=";")
x2 = df2['Age']
y2 = df2['Salary']
n  = len(x2)

y2  = np.array(y2)
yy2 = np.array(model(x2))

rmse = np.sqrt(np.sum((y2-yy2)**2)/(n-1))

print('Root Mean Square Error:')
print(rmse)
```

```
Root Mean Square Error:
44726.74305949611
```

[39]:
```python
xx2 = np.arange(x2[0], x2[n-1], 0.1)
plt.plot(xx2, model(xx2))
plt.xlabel("Age")
plt.ylabel("Salary")
plt.scatter(x2,y2, color='red')
plt.show()
```



- The root mean squared error (rmse) for the training data set is $12,902
- The rmse for the test data set is $38,794

We conclude that the model overfits the data. The complexity of the model should be increased only until out-of-sample tests indicate that it does not generalize well.

1.3 Bias and Variance

Suppose there is a relationship between an independent variable x and a dependent variable y:

$$y = f(x) + \epsilon \tag{1}$$

Where ϵ is an error term with mean zero and variance σ^2. The error term captures either genuine randomness in the data or noise due to measurement error.

Suppose we find a deterministic model for this relationship:

$$y = \hat{f}(x) \tag{2}$$

Now it comes a new data point x' not in the training set and we want to predict the corresponding y'. The error we will observe in our model at point x' is going to be

$$\hat{f}(x') - f(x') - \epsilon \tag{3}$$

There are two different sources of error in this equation. The first one is included in the factor ϵ, the second one, more interesting, is due to what is in our training set. A robust model should give us the same prediction whatever data we used for training out model. Let's look at the average error:

$$E\left[\hat{f}(x')\right] - f(x') \tag{4}$$

where the expectation is taken over random samples of training data (having the same distributio as the training data).

This is the definition of the **bias**

$$\text{Bias}\left[\hat{f}(x')\right] = E\left[\hat{f}(x')\right] - f(x') \tag{5}$$

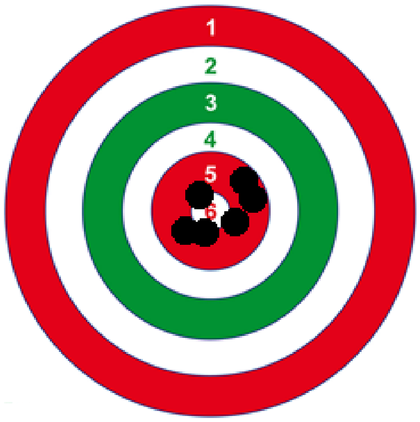
We can also look at the mean square error

$$E\left[\left(\hat{f}(x') - f(x') - \epsilon\right)^2\right] = \left[\text{Bias}\left(\hat{f}(x')\right)\right]^2 + \text{Var}\left[\hat{f}(x')\right] + \sigma^2 \tag{6}$$

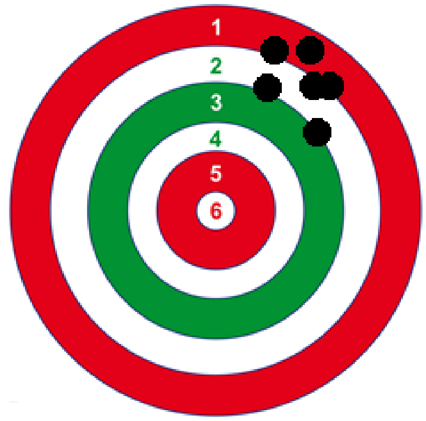
Where we remember that $\hat{f}(x')$ and ϵ are independent.

This show us that there are two important quantities, the **bias** and the **variance** that will affect our results and that we can control to some extent.

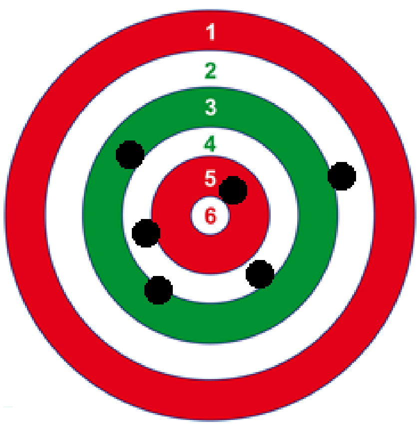
FIGURE 1.1 - A good model should have low bias and low variance



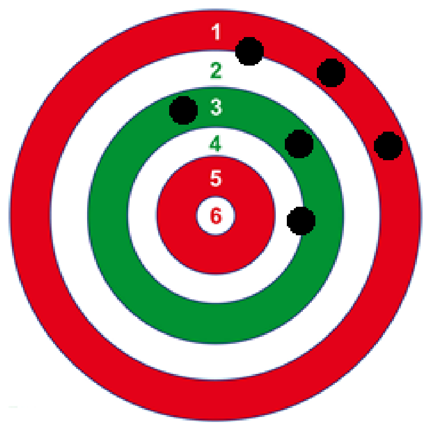
Low Bias and Low Variance



High Bias and Low Variance



Low Bias and High Variance

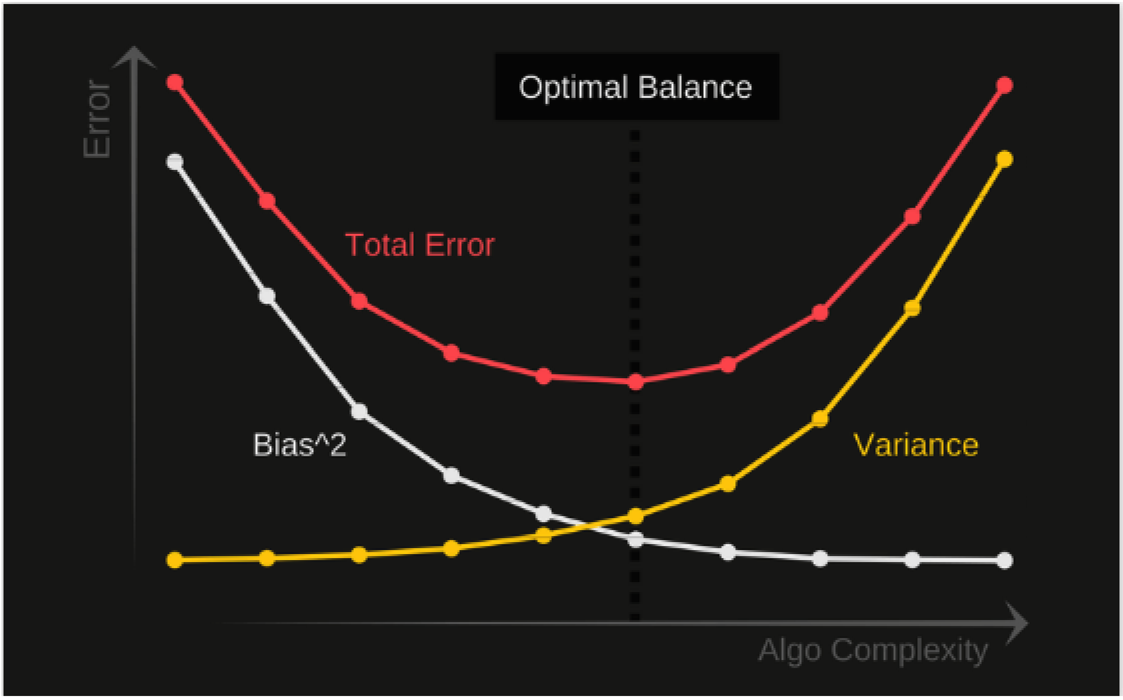


High Bias and High Variance

Bias is how far away the trained model is from the correct result on average. Where *on average* means over many goes at training the model using different data. And **Variance is a measure of the magnitude of that error**.

Unfortunately, we often find that there is a trade-off between bias and variance. As one is reduced, the other is increased. This is the matter of over- and under-fitting.

Overfitting is when we train our algorithm too well on training data, perhaps having too many parameters for fitting.



1.4 Partitioning a dataset into training and test datasets

[40]:
```
import pandas as pd
```

[41]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[42]:
```
# Both features and target have already been scaled: mean = 0; SD = 1
# See Chapter 4-0 for a descrition of this dataset
df = pd.read_csv(path + 'Houseprice_data_scaled.csv')
df.head()
```

[42]:
```
    LotArea  OverallQual  OverallCond  YearBuilt  YearRemodAdd  BsmtFinSF1  \
0 -0.199572     0.652747    -0.512407   1.038851      0.875754    0.597837
1 -0.072005    -0.072527     2.189741   0.136810     -0.432225    1.218528
2  0.111026     0.652747    -0.512407   0.972033      0.827310    0.095808
3 -0.077551     0.652747    -0.512407  -1.901135     -0.722887   -0.520319
4  0.444919     1.378022    -0.512407   0.938624      0.730423    0.481458

   BsmtUnfSF  TotalBsmtSF  1stFlrSF  2ndFlrSF  ...   OLDTown     SWISU  \
0  -0.937245    -0.482464 -0.808820  1.203988  ... -0.286942 -0.136621
1  -0.635042     0.490326  0.276358 -0.789421  ... -0.286942 -0.136621
```

```
2  -0.296754    -0.329118 -0.637758  1.231999  ... -0.286942 -0.136621
3  -0.057698    -0.722067 -0.528171  0.975236  ... -0.286942 -0.136621
4  -0.170461     0.209990 -0.036366  1.668495  ... -0.286942 -0.136621

   Sawyer   SawyerW   Somerst   StoneBr    Timber    Veenker  Bsmt Qual  \
0 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644   0.584308
1 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  10.905682   0.584308
2 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644   0.584308
3 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644  -0.577852
4 -0.2253 -0.214192 -0.268378 -0.127929 -0.152629  -0.091644   0.584308

   Sale Price
0    0.358489
1    0.008849
2    0.552733
3   -0.528560
4    0.895898

[5 rows x 48 columns]
```

A convenient way to randomly partition this dataset into separate test and training datasets is to use the train_test_split function from scikit-learn's model_selection submodule:

[43]:
```
ncol = df.shape[1]

X = df.iloc[:, :ncol-1].values
y = df.iloc[:, ncol-1].values
```

[44]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test =\
    train_test_split(X, y,
    test_size=0.3,
    random_state=0)
```

First, we assigned the NumPy array representation of the feature columns from 0 to ncol-1 to the variable X and we assigned the class labels from the last column to the variable y. Then, we used the train_test_split function to randomly split X and y into separate training and test datasets. By setting test_size=0.3, we assigned 30 percent of the wine examples to X_test and y_test, and the remaining 70 percent of the examples were assigned to X_train and y_train, respectively.

1.5 Selecting meaningful features: Regularization

The reason for the overfitting is that our model is too complex for the given training data. Common solutions to reduce the generalization error are as follows:

- Collect more training data (Easier said that done…)

- Introduce a penalty for complexity via regularization (see section ...)
- Choose a simpler model with fewer parameters
- Reduce the dimensionality of the data

In the following sections, we will look at common ways to reduce overfitting by regularization, which leads to simpler models by requiring fewer parameters to be fitted to the data.

1.5.1 Ridge Regression

Ridge regression is a regularization technique where we change the function that is to be minimize. Reduce magnitude of regression coefficients by choosing a parameter λ and minimizing

$$\frac{1}{2N}\sum_{n=1}^{N}\left[h_\theta\left(x^{(n)}\right)-y^{(n)}\right]^2+\lambda\sum_{n=1}^{N}\theta_i^2$$

This change has the effect of encouraging the model to keep the weights b_j as small as possibile. The Ridge regression should only be used for determining model parameters using the training set. Once the model parameters have been determined the penalty term should be removed for prediction.

[62]:
```
import matplotlib.pyplot as plt
```

[63]:
```
#
# Here we have to load the file 'salary_vs_age_1.csv'
#
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[64]:
```
# Load the Pandas libraries with alias 'pd'
import pandas as pd
# Read data from file 'salary_vs_age_1.csv'
# (in the same directory that your python process is based)
# Control delimiters, with read_table
df1 = pd.read_table(path + "salary_vs_age_1.csv", sep=";")
# Preview the first 5 lines of the loaded data
print(df1.head())
```

```
   Age  Salary
0   25  135000
1   27  105000
2   30  105000
3   35  220000
4   40  300000
```

[65]:
```
columns_titles = ["Salary","Age"]
df2=df1.reindex(columns=columns_titles)
df2
```

[65]:

	Salary	Age
0	135000	25
1	105000	27
2	105000	30
3	220000	35
4	300000	40
5	270000	45
6	265000	50
7	260000	55
8	240000	60
9	265000	65

[66]:
```
df2['Salary'] = df2['Salary']/1000
df2['Age2']=df2['Age']**2
df2['Age3']=df2['Age']**3
df2['Age4']=df2['Age']**4
df2['Age5']=df2['Age']**5
df2
```

[66]:

	Salary	Age	Age2	Age3	Age4	Age5
0	135.0	25	625	15625	390625	9765625
1	105.0	27	729	19683	531441	14348907
2	105.0	30	900	27000	810000	24300000
3	220.0	35	1225	42875	1500625	52521875
4	300.0	40	1600	64000	2560000	102400000
5	270.0	45	2025	91125	4100625	184528125
6	265.0	50	2500	125000	6250000	312500000
7	260.0	55	3025	166375	9150625	503284375
8	240.0	60	3600	216000	12960000	777600000
9	265.0	65	4225	274625	17850625	1160290625

We can compute the z-score in Pandas using the .mean() and std() methods.

[67]:
```
# apply the z-score method in Pandas using the .mean() and .std() methods
def z_score(df):
    # copy the dataframe
    df_std = df.copy()
    # apply the z-score method
    for column in df_std.columns:
        df_std[column] = (df_std[column] - df_std[column].mean()) /␣
↪df_std[column].std()

    return df_std
```

```
# call the z_score function
df2_standard = z_score(df2)
df2_standard['Salary'] = df2['Salary']
df2_standard
```

[67]:

```
   Salary       Age      Age2      Age3      Age4      Age5
0   135.0 -1.289948 -1.128109 -0.988322 -0.873562 -0.782128
1   105.0 -1.148195 -1.045510 -0.943059 -0.849996 -0.770351
2   105.0 -0.935566 -0.909699 -0.861444 -0.803378 -0.744782
3   220.0 -0.581185 -0.651577 -0.684372 -0.687799 -0.672266
4   300.0 -0.226804 -0.353745 -0.448740 -0.510508 -0.544103
5   270.0  0.127577 -0.016202 -0.146184 -0.252677 -0.333075
6   265.0  0.481958  0.361052  0.231663  0.107030 -0.004250
7   260.0  0.836340  0.778017  0.693166  0.592463  0.485972
8   240.0  1.190721  1.234693  1.246690  1.229979  1.190828
9   265.0  1.545102  1.731080  1.900602  2.048447  2.174155
```

[68]:

```
y = df2_standard['Salary']
X = df2_standard.drop('Salary',axis=1)
```

[69]:

```
print(y)
```

```
0    135.0
1    105.0
2    105.0
3    220.0
4    300.0
5    270.0
6    265.0
7    260.0
8    240.0
9    265.0
Name: Salary, dtype: float64
```

[70]:

```
print(X)
```

```
        Age      Age2      Age3      Age4      Age5
0 -1.289948 -1.128109 -0.988322 -0.873562 -0.782128
1 -1.148195 -1.045510 -0.943059 -0.849996 -0.770351
2 -0.935566 -0.909699 -0.861444 -0.803378 -0.744782
3 -0.581185 -0.651577 -0.684372 -0.687799 -0.672266
4 -0.226804 -0.353745 -0.448740 -0.510508 -0.544103
5  0.127577 -0.016202 -0.146184 -0.252677 -0.333075
6  0.481958  0.361052  0.231663  0.107030 -0.004250
7  0.836340  0.778017  0.693166  0.592463  0.485972
8  1.190721  1.234693  1.246690  1.229979  1.190828
9  1.545102  1.731080  1.900602  2.048447  2.174155
```

Now we implement the Ridge regularization method using the scikit-learn package. Scikit-learn is **one of the most popular Python library for machine learning**.

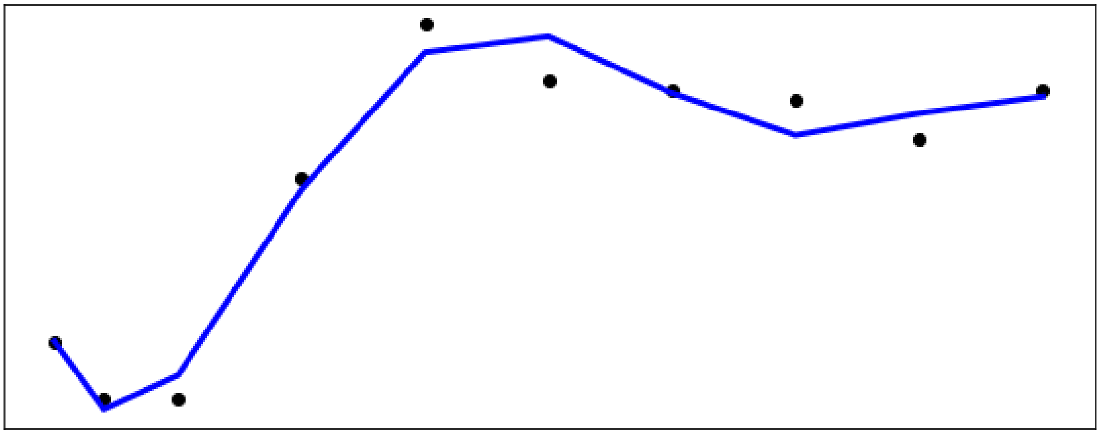
Why this library is one of the best choices for machine learning projects?

- It has a **high level of support** and **strict governance for the development** of the library which means that it is an incredibly robust tool.
- There is **a clear, consistent code style** which ensures that your machine learning code is easy to understand and reproducible, and also vastly lowers the barrier to entry for coding machine learning models.
- It is **well integrated with the major components of the Python scientific stack**: numpy, pandas, scipy and matplotlib.
- It is **widely supported by third-party tools** so it is possible to enrich the functionality to suit a range of use cases.

[71]:
```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.metrics import mean_squared_error, r2_score

lr = LinearRegression()
lr.fit(X, y)

y_pred = lr.predict(X)

# The coefficients
print('Coefficients: \n', lr.coef_)
# The mean squared error
print('Mean squared error: %.2f'
      % mean_squared_error(y, y_pred))
```

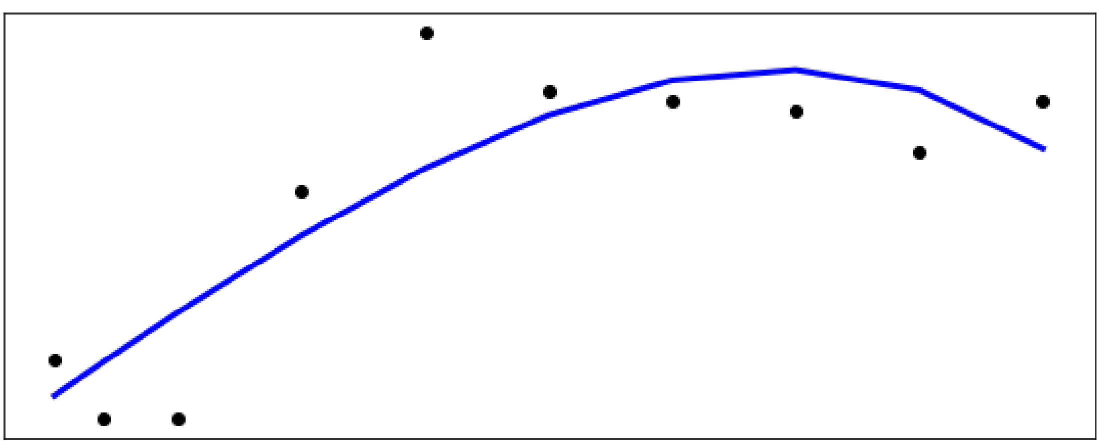
```
Coefficients:
 [ -32622.57240727  135402.73116519 -215493.11781297  155314.61367273
  -42558.76209732]
Mean squared error: 149.82
```

[72]:
```
# Plot outputs
plt.scatter(X['Age'], y,  color='black')
plt.plot(X['Age'], y_pred, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```



[73]:
```
rr = Ridge(alpha=0.01, normalize=True)
# higher the alpha value, more restriction on the coefficients; low alpha > more␣
↪generalization,
# in this case linear and ridge regression resembles
rr.fit(X, y)

y_pred_r = rr.predict(X)
```

[74]:
```
# Plot outputs
plt.scatter(X['Age'], y,  color='black')
plt.plot(X['Age'], y_pred_r, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```



[75]:
```
# The coefficients
print('Coefficients: \n', rr.coef_)
# The mean squared error
print('Mean squared error: %.2f'
      % mean_squared_error(y, y_pred_r))
```

```
Coefficients:
 [119.85726826  32.07576023 -24.12692453 -45.195683   -35.65836346]
Mean squared error: 1148.95
```

1.5.2 Lasso Regression

Lasso is short for *Least Absolute Shrinkage and Selection Operator*. It is similar to ridge regression except we minimize

$$\frac{1}{2N}\sum_{n=1}^{N}\left[h_\theta\left(x^{(n)}\right)-y^{(n)}\right]^2+\lambda\sum_{n=1}^{N}|b_n|$$

This function cannot be minimized analytically and so a variation on the gradient descent algorithm must be used. Lasso regression also has the effect of simplifying the model. It does this by setting the weights of unimportant features to zero. When there are a large number of features, Lasso can identify a relatively small subset of the features that form a good predictive model.

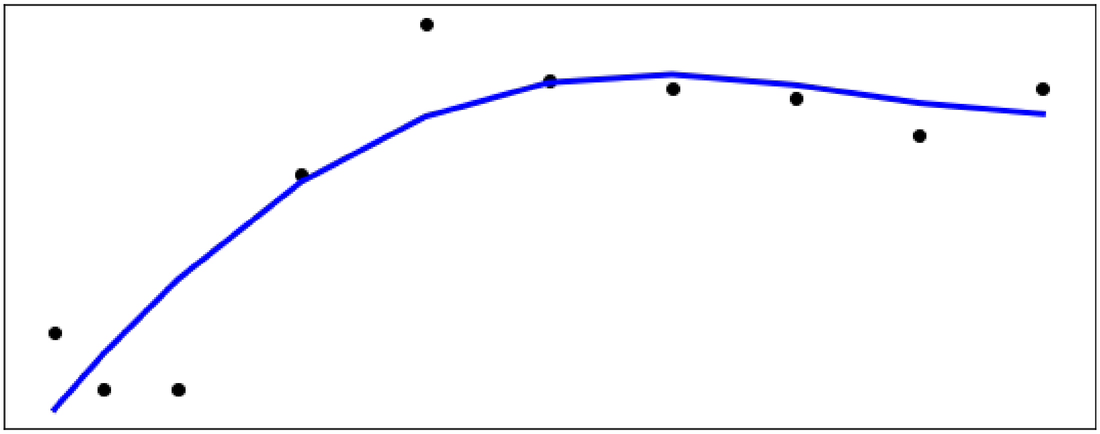
[76]:
```
from sklearn.linear_model import Lasso

lsr = Lasso(alpha=.02, normalize=True, max_iter=1000000)
# higher the alpha value, more restriction on the coefficients; low alpha > more␣
↪generalization,
# in this case linear and ridge regression resembles
lsr.fit(X, y)

y_pred_lsr = lsr.predict(X)
```

[77]:
```
# The coefficients
print('Coefficients: \n', lsr.coef_)
# The mean squared error
print('Mean squared error: %.2f'
      % mean_squared_error(y, y_pred_lsr))
```

```
Coefficients:
 [ 344.99709034   -0.         -471.80600937   -0.          183.42041303]
Mean squared error: 854.75
```

[78]:
```
# Plot outputs
plt.scatter(X['Age'], y,  color='black')
```

```
plt.plot(X['Age'], y_pred_lsr, color='blue', linewidth=3)

plt.xticks(())
plt.yticks(())

plt.show()
```



1.5.3 Elastic Net Regression

Middle ground between Ridge and Lasso. Minimize

$$\frac{1}{2N}\sum_{n=1}^{N}\left[h_\theta\left(x^{(n)}\right)-y^{(n)}\right]^2+\lambda_1\sum_{n=1}^{N}b_n^2+\lambda_2\sum_{n=1}^{N}|b_n|$$

In Lasso some weights are reduced to zero but others may be quite large. In Ridge, weights are small in magnitude but they are not reduced to zero. The idea underlying Elastic Net is that we may be able to get the best of both by making some weights zero while reducing the magnitude of the others.

[43]:
```
from sklearn.linear_model import ElasticNet

# define model
model = ElasticNet(alpha=1.0, l1_ratio=0.5)
```

[ ]:
```
```

1.6 References

A. GÃl'ron, "*Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*", 2nd Edition. O'Reilly Media, 2019

S. Raschka and V. Mirjalili, "*Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*", 3rd Edition. Packt Publishing Ltd, 2019.

S. Raschka, "*Model Evaluation, Model Selection and Algorithm Selection in Machine Learning*", downlodable here

Scikit-Learn web site