01-introduction-to-machine-learning

October 7, 2021

Run in Google Colab

1 Introduction to Machine Learning

1.1 What is Machine Learning?

Two definitions of Machine Learning are offered. Arthur Samuel described it as: "the field of study that gives computers the ability to learn without being explicitly programmed." This is an older, informal definition.

Tom Mitchell provides a more modern definition: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Example: playing checkers.

E = the experience of playing many games of checkers

T = the task of playing checkers.

P = the probability that the program will win the next game.

To use machine learning effectively **you have to understand how the underlying algorithms work**. It is tempting to learn a language such as Python or R and apply various packages to your data without really understanding what the packages are doing or even how the results should be interpreted. This would be a bit like a finance specialist using the Black and Scholes model to value options without understanding where it comes from or its limitations.

1.1.1 Type of Machine Learning Models

There are four main categories of machine learning models:

- Supervised Learning
- Unsupervised Learning
- Semi-Supervised Learning
- Reinforcement Learning

Supervised learning is concerned with using data to make predictions. We can distinguish between supervised learning models that are used to predict a variable and models that are used for classification.

Unsupervised learning is concerned with recognizing patterns in data. The main object is not to forecast a particular variable, rather it is to understand the data environment better.

1.1.2 Jargon

The data for supervised learning contains whare are referred to as **features** and **labels**. The **labels** are the values of the target that is to be predicted. The **features** are the variables from which the predictions are to be made. For example when predicting the price of a house the **features** could be the swuare meters of living space, the number of bedrooms, the number of bathrooms, the size of the garage and so on. The **label** would be the house price.

The data for unsupervised learning consists of features but no labels because the model is being used to identify patterns not to forecast something.

1.1.3 Type of Data

There are two types of data:

- Numerical
- Categorical

Numerical data consists of numbers. Categorical data is data which can fall into a number of different categories, for example data to predict a house price might categorize driveways as asphalt, concrete, grass, etc. Categorical data must be converted to numbers for the purposes of analysis.

The standard way of dealing with categorical features is to create a dummy variable for each category. The value of this variable is 1 if the feature is in the category and 0 otherwise. For example in the situation in which individuals are categorized as male or female, we could create two dummy variables. For man the first dummy variable would be 1 and the second would be 0. The opposite for women. This procedure is appropriate when there is no natural ordering between the feature values.

When there is a natural ordering, we can reflect this in the numbers assigned. For example if the size of an order is classified as small, medium or large, we can replace the feature by a numerical variable where *small = 1*, *medium = 2* and *large = 3*.

1.2 Feature Normalization

The success of a machine learning algorithm highly depends on the quality of the data fed into the model. Real-world data is often dirty containing outliers, missing values, wrong data types, irrelevant features, or non-standardized data. The presence of any of these will prevent the machine learning model to properly learn. For this reason, transforming raw data into a useful format is an essential stage in the machine learning process. One technique you will come across multiple times when pre-processing data is feature normalization.

Data Normalization is a common practice in machine learning which consists of transforming numeric columns to a common scale. In machine learning, some feature values differ from others multiple times. The features with higher values will dominate the leaning process. However, it

does not mean those variables are more important to predict the outcome of the model. Data normalization transforms multiscaled data to the same scale. After normalization, all variables have a similar influence on the model, improving the stability and performance of the learning algorithm.

There are multiple normalization techniques in statistics. In this notebook, we will cover the most important ones:

- The maximum absolute scaling
- The min-max feature scaling
- The z-score method

1.2.1 The maximum absolute scaling

The maximum absolute scaling rescales each feature between -1 and 1 by dividing every observation by its maximum absolute value.

$$x_{new} = \frac{x_{old}}{\max |x_{old}|}$$

1.2.2 The min-max feature scaling

The min-max approach (often called normalization) rescales the feature to a fixed range of [0,1] by subtracting the minimum value of the feature and then dividing by the range:

$$x_{new} = \frac{x_{old} - x_{min}}{x_{max} - x_{min}}$$

1.2.3 Z-Score

The z-score method (often called standardization) transforms the data into a distribution with a mean of 0 and a standard deviation of 1. Each standardized value is computed by subtracting the mean of the corresponding feature and then dividing by the standard deviation.

$$x_{new} = \frac{x_{old} - \mu}{\sigma}$$

Unlike min-max scaling, the z-score does not rescale the feature to a fixed range. The z-score typically ranges from -3.00 to 3.00 (more than 99% of the data) if the input is normally distributed.

It is important to bear in mind that z-scores are not necessarily normally distributed. They just scale the data and follow the same distribution as the original input. This transformed distribution has a mean of 0 and a standard deviation of 1 and is going to be the standard normal distribution only if the input feature follows a normal distribution.

1.3 Cost Functions

1.3.1 Linear Cost Function

In Machine Learning a cost function or loss function is used to represent how far away a mathematical model is from the real data. One adjusts the mathematical model, usually by varying parameters within the model, so as to minimize the cost function.

Let's take for example the simple case of a linear fitting. We want to find a relationship of the form

$$y = \theta_0 + \theta_1 x \tag{1}$$

where the θs are the parameters that we want to find to give us the best fit to the data. We call this linear function $h_\theta(x)$ to emphasize the dependence on both the variable x and the two parameters θ_0 and θ_1.

We want to measure how far away the data, the $y^{(n)}$s, are from the function $h_\theta(x)$. A common way to do this is via the quadratic *cost function*

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

This is called *Ordinary Least Squares*.

In this case, the minimum is easily find analitically, differentiate (2) with respect to both θs and set the result to zero:

$$\begin{aligned} \frac{\partial J}{\partial \theta_0} &= \sum_{n=1}^{N} \left(\theta_0 + \theta_1 x^{(n)} - y^{(n)} \right) = 0 \\ \frac{\partial J}{\partial \theta_1} &= \sum_{n=1}^{N} x^{(n)} \left(\theta_0 + \theta_1 x^{(n)} - y^{(n)} \right) = 0 \end{aligned} \tag{3}$$

The solution is trivially obtained for both θs

$$\begin{aligned} \theta_0 &= \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{N(\sum x^2)(\sum x)^2} \\ \theta_1 &= \frac{N(\sum xy) - (\sum y)(\sum x)}{N(\sum x^2)(\sum x)^2} \end{aligned} \tag{4}$$

1.4 Gradient Descent

The scheme works as follow: start with an initial guess for each parameter θ_k. Then move θ_k in the direction of the slope:

$$\theta_k^{new} = \theta_k^{old} + \beta \frac{\partial J}{\partial \theta_k} \tag{5}$$

Update all θ_k simultaneously and repet until convergence. Here β is a *learning factor* that governs how far you move. if β is too small it will take a long time to converge, if too large it will overshoot and might not converge at all.

The loss function J is a function of all of the data points. In the above description of gradient descent we have used all of the data points simultaneously. This is called *batch gradient* descent. But rather than use all of the data in the parameter updating we can use a technique called *stochastic gradient descent*. This is like batch gradient descent except that you only update using *one* of the data points each time. And that data point is chosen randomly.

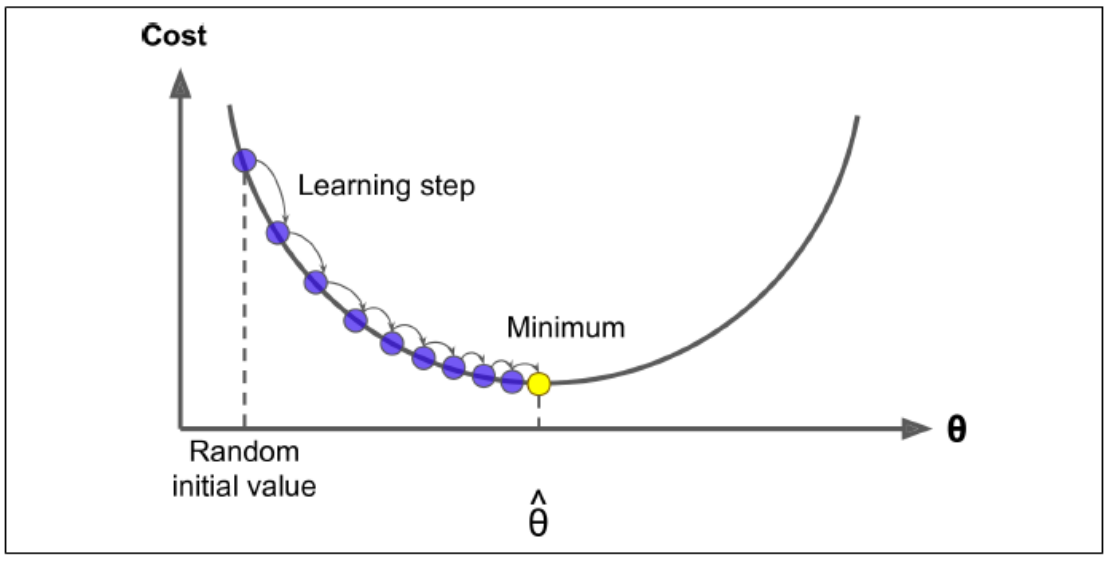
$$J(\theta) = \sum_{n=1}^{N} J_n(\theta) \tag{6}$$

Stochastic gradient descent means pick an n at random and then update according to

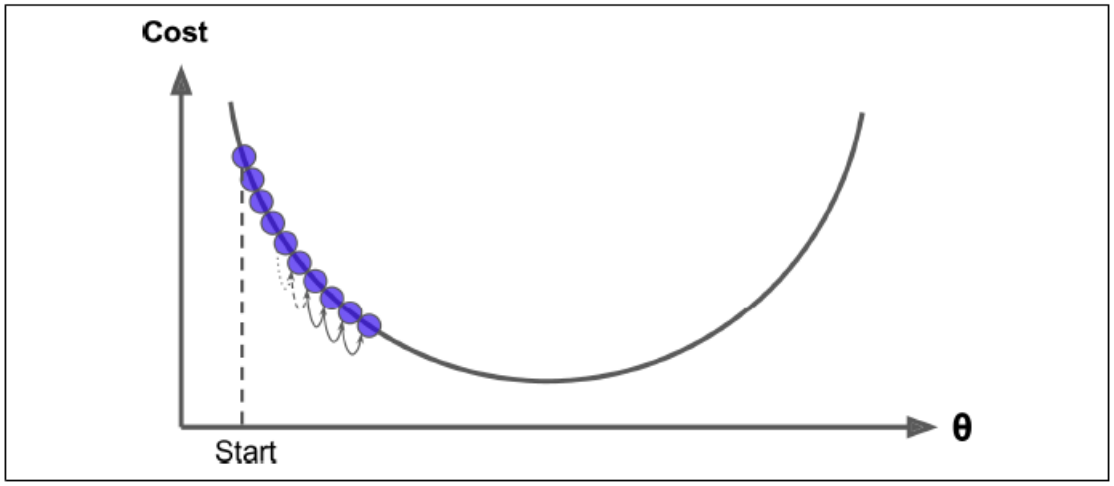
$$\theta_k^{new} = \theta_k^{old} + \beta \frac{\partial J_n}{\partial \theta_k} \tag{7}$$

Repeat, picking another data point at random, etc.

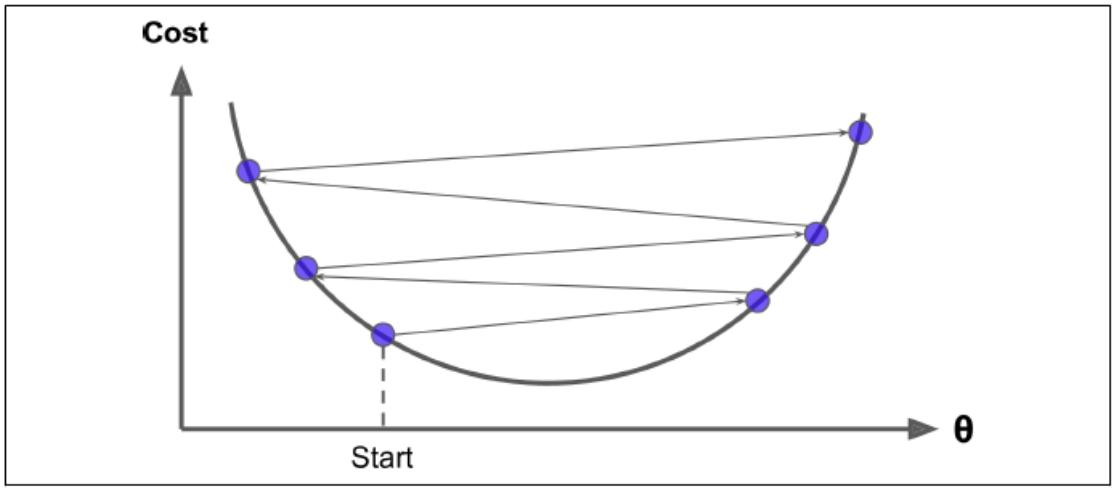
An important parameter in Gradient Descent is the size of the steps, determined by the learning rate hyperparameter.



If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time…



… on the other hand, if the learning rate is too high, you might jump across the valley. This might make the algorithm diverge failing to find a good solution.



1.5 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;

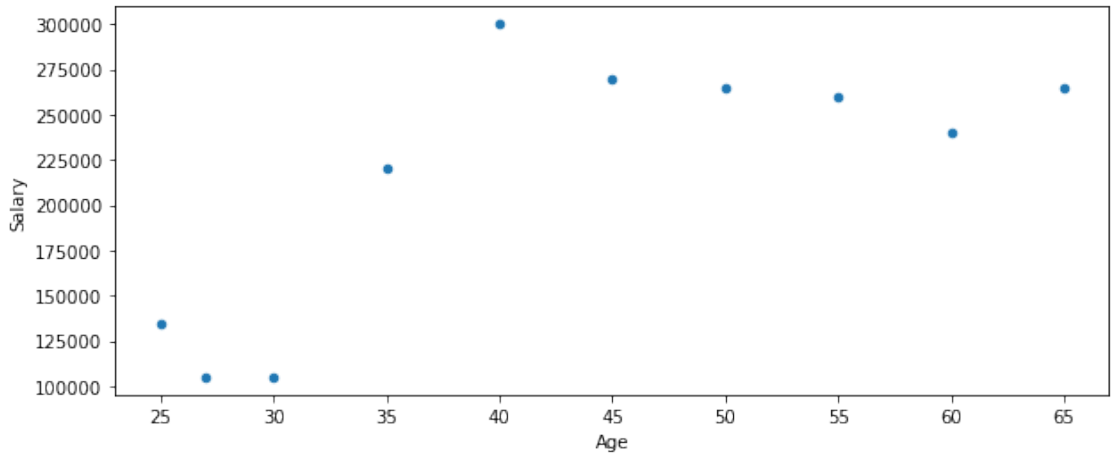
[14]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
```

```
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[15]:
```
# 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())
```

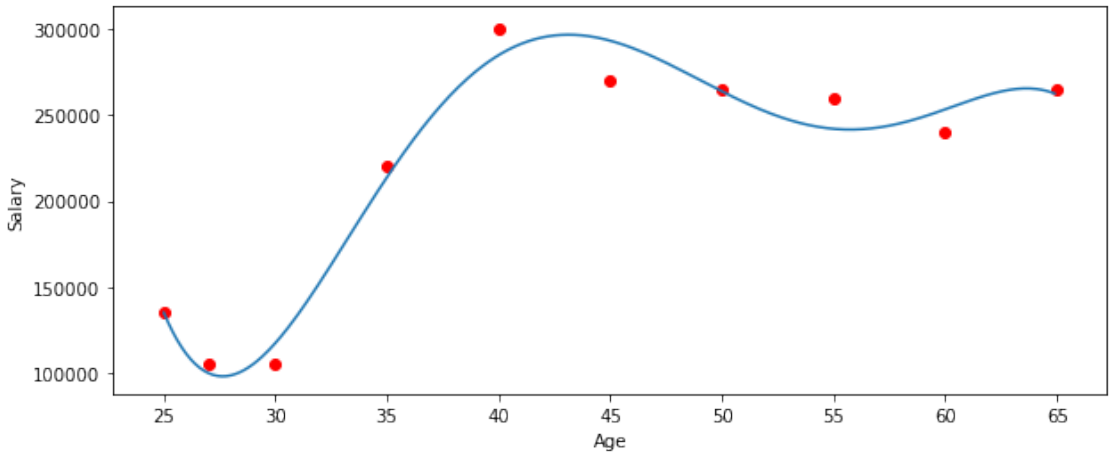
```
   Age  Salary
0   25  135000
1   27  105000
2   30  105000
3   35  220000
4   40  300000
```

[16]:
```
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

[17]:
```
import numpy as np

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

n = len(x1)

degree = 5

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()
```



[18]:
```
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)
```

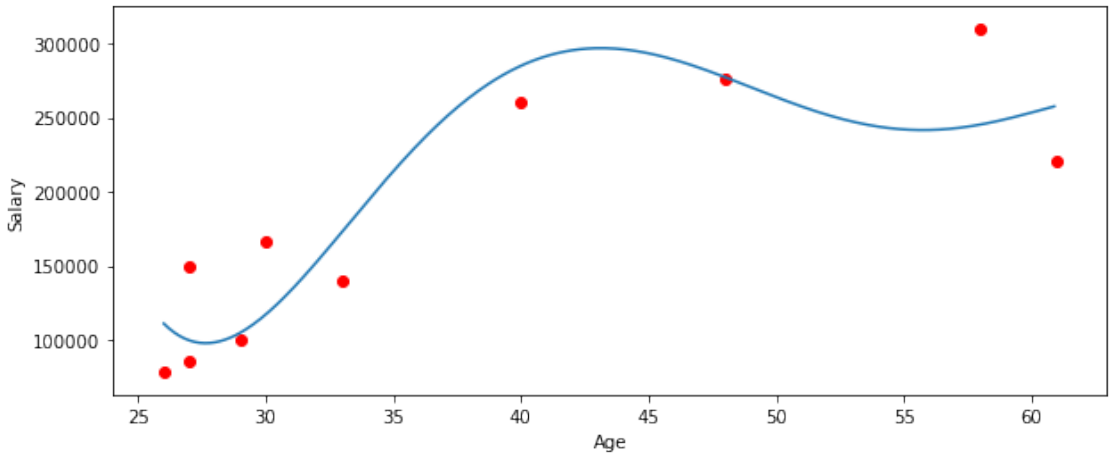
```
Root Mean Square Error:
12902.203044361002
```

[19]:
```
if 'google.colab' in str(get_ipython()):
    from google.colab import files
    uploaded = files.upload()
    path = ''
else:
    path = './data/'
```

[20]:
```
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:
38825.22050917512
```

[21]:
```
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.

1.6 Bias and Variance

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

$$y = f(x) + \epsilon \tag{8}$$

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:

$$y = \hat{f}(x) \tag{9}$$

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

$$\hat{f}(x') - f(x') - \epsilon \tag{10}$$

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:

$$E\left[\hat{f}(x')\right] - f(x') \tag{11}$$

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**

$$\text{Bias}\left[\hat{f}(x')\right] = E\left[\hat{f}(x')\right] - f(x') \tag{12}$$

We can also look at the mean square error

$$E\left[\left(\hat{f}(x') - f(x') - \epsilon\right)^2\right] = \left[\text{Bias}\left(\hat{f}(x')\right)\right]^2 + \text{Var}\left[\hat{f}(x')\right] + \sigma^2 \tag{13}$$

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

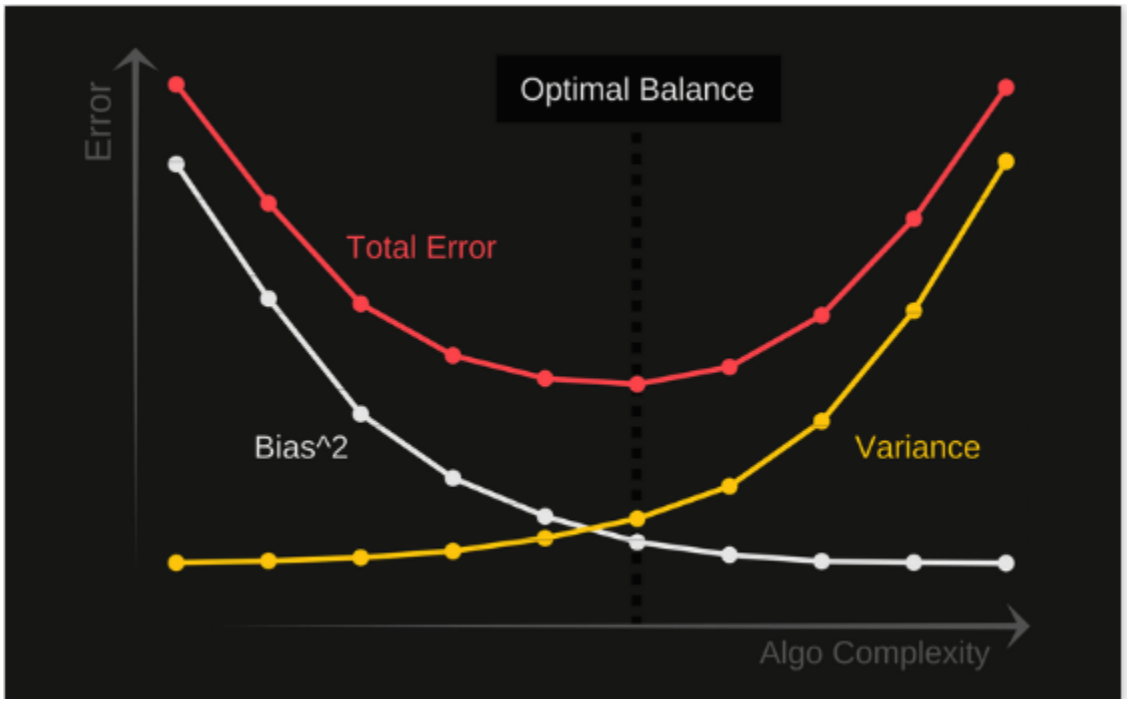
Low Bias and High Variance

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.



1.7 What is scikit-learn?

Scikit-learn - started as a Google Summer of Code project in 2007 - is **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.

[ ]:

1.8 Regularization

1.8.1 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.

[22]:
```
columns_titles = ["Salary","Age"]
df2=df1.reindex(columns=columns_titles)
df2
```

[22]:

	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

[23]:
```
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
```

[23]:

	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.

[24]:
```
# 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
```

[24]:

	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

[25]:
```
y = df2_standard['Salary']
X = df2_standard.drop('Salary',axis=1)
```

[26]:
```
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
```

[27]:
```
print(X)
```

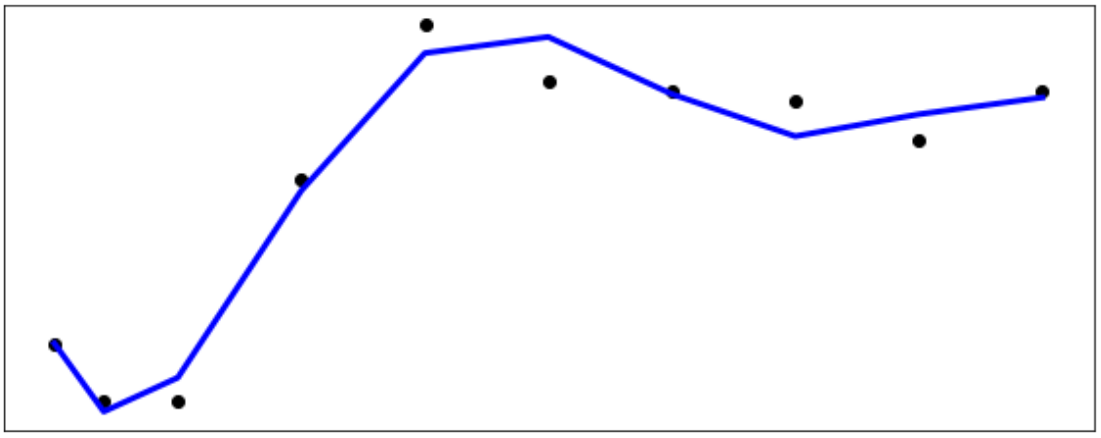
```
        Age      Age2      Age3      Age4      Age5
0 -1.289948 -1.128109 -0.988322 -0.873562 -0.782128
```

```
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
```

[28]:
```
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))
```

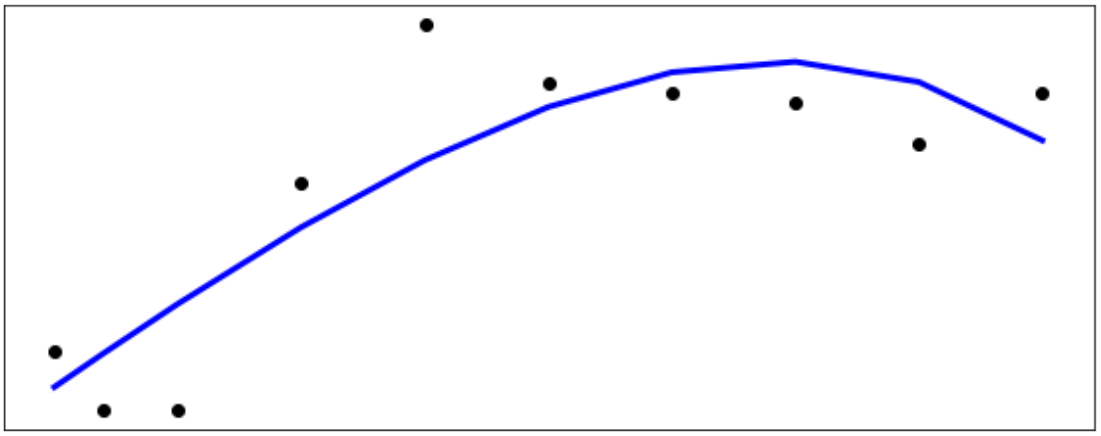
```
Coefficients:
 [ -32622.57240727  135402.73116519 -215493.11781297  155314.61367273
  -42558.76209732]
Mean squared error: 149.82
```

[29]:
```
# 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()
```



[30]:
```
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)
```

[31]:
```
# 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()
```



[32]:
```
# 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
```

1.8.2 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.

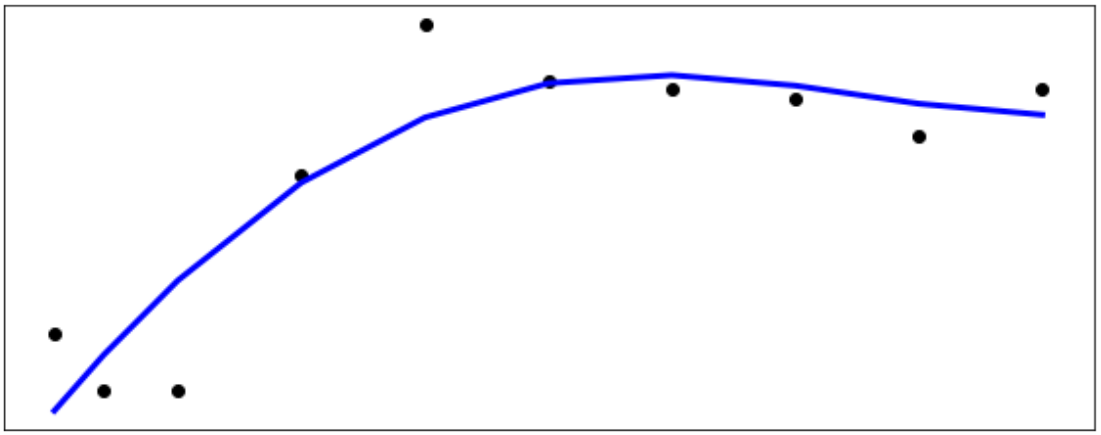
[33]:
```
from sklearn.linear_model import Lasso

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)
```

[34]:
```
# 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
```

[35]:
```
# 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()
```



1.8.3 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.

[36]:
```
from sklearn.linear_model import ElasticNet

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

1.9 References

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.

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

Scikit-Learn web site