Fitting Models to Data

In this example, we want to find the function of a given form that best fits a large number of data points. In order to do that, we use Python to compute the least squares approximation to a linear system.

In [13]:
```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
import math
```

Toronto Stock Exchange Index

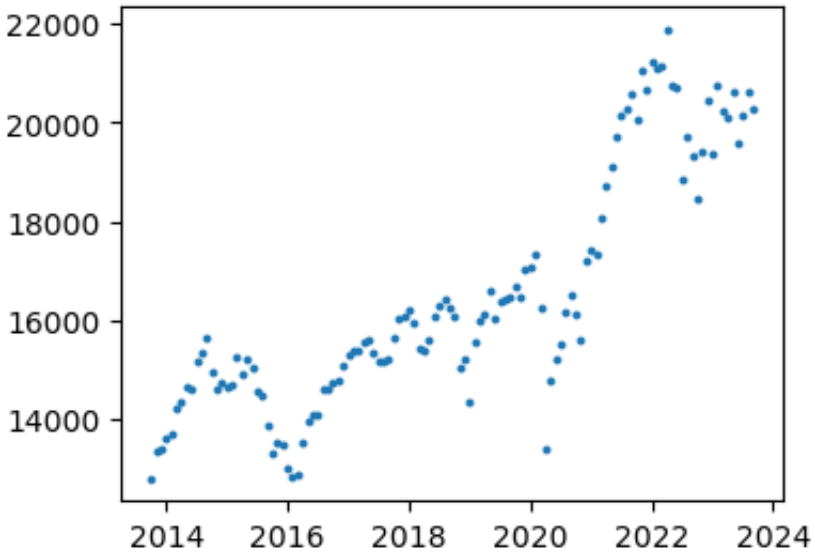
We will use data from the Toronto Stock Exchange (available via open.canada.ca). More specifically, we use monthly values of the S&P/TSX Composite Index over the past 10 years (2013-10 to 2023-09). This index is based on around 250 of the largest Canadian companies, so it can be used as an indicator of the overall strength of the Canadian economy.

First, we input our 120 data points in Python as a 120x2 matrix, with time points in the first column and the corresponding index values in the second column.

In [14]:
```
TSX = np.array([[2013.75, 12787.20], [2013.83333333333, 13361.26], [2013.916
```

Next, let's plot the data to see how the TSX index has changed over the last ten years.

In [15]:
```
t = TSX[:,[0]]
y = TSX[:,[1]]
plt.scatter(t,y,alpha=1,lw=0,s=8);
plt.show()
```



Linear regression

Suppose we want to find a linear function

$$y(t) = c_0 + c_1 t$$

that describes the value of the TSX index as a function of time. Obviously, we cannot find a single line that fits all the 120 points. Instead, we determine the line that gives the best possible fit. This means that we want to approximate the system $A\vec{c} = \vec{y}$, where

$$A = \begin{bmatrix} 1 & t_1 \\ 1 & t_2 \\ \vdots & \vdots \\ 1 & t_{120} \end{bmatrix}, \ \vec{c} = \begin{bmatrix} c_0 \\ c_1 \end{bmatrix}, \text{ and } \ \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{120} \end{bmatrix},$$

to find the coefficients c_0 and c_1 of the linear function.

Typically we consider the best fit to be the line with the smallest sum of squared errors,

$$SSE = \sum_{i=1}^{120} (y_i - (c_0 + c_1 t_i))^2 = \|\vec{y} - A\vec{c}\|^2.$$

To find this least squares solution, we can either solve the *normal equations* or use the *QR decomposition*. Let's take the normal equation approach for now: in this case we find the unique solution to the system

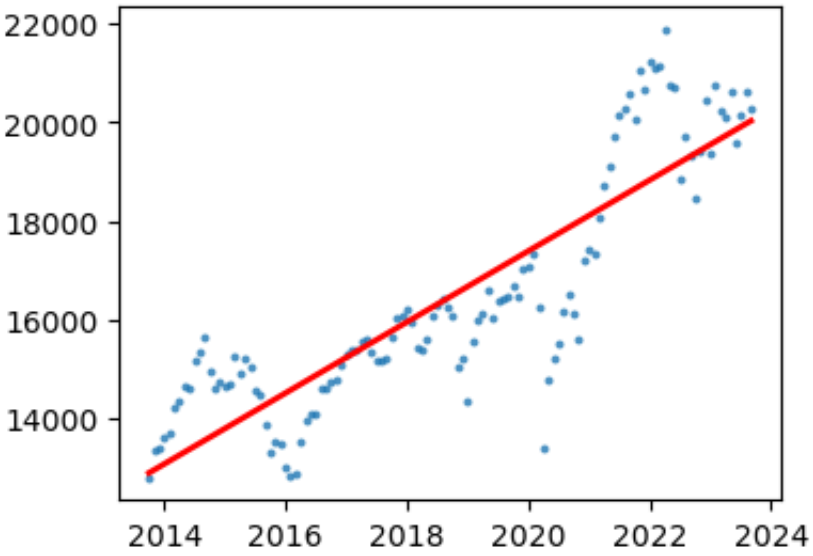
$$A^T A\vec{c} = A^T \vec{y}.$$

In [16]:
```
A = np.column_stack([np.ones(120),t])
c = la.solve(A.T @ A, A.T @ y)
print(c)
```

```
[[-1.43877955e+06]
 [ 7.20874856e+02]]
```

We have now found the coefficients c_0 and c_1 for the line with best fit. Let's plot this line together with the data points.

In [17]:
```
tl = np.linspace(2013.75,2023.75,120, endpoint=False)
yl = c[0] + c[1]*tl
plt.plot(tl,yl,'r',linewidth=2)
plt.scatter(t,y,alpha=0.8,lw=0,s=8);
plt.show()
```



As we can see, the red line gives a fairly good linear approximation to our large set of data points (in fact, the best possible line with respect to the sum of squared errors). Note that the index value shows quite significant fluctuations, though, so the linear function is not necessarily a very accurate description of the index price at all times. However, it could still be useful as a description of the general trend of the stock market. (When making long-term investments, we maybe do not care much about the smaller fluctuations, but rather we are interested in the long-term trend of the market.)

We can compute the norm of the residual vector (the difference between values on the line and the data points) to measure how well the line fits the data. The value won't maybe tell us much on its own, but we can use it to compare with the fit of other functions to the same data.

In [18]:
```
np.sqrt(np.sum((yl-y.T)**2))
```

Out[18]:
```
13138.774067621729
```

Polynomial regression

We can also use the same method to find the polynomial function of a certain degree that best fits the TSX data. For instance, let's consider finding a quartic function $y(t) = c_0 + c_1 t + c_2 t^2 + c_3 t^3 + c_4 t^4$, this time using the QR decomposition (numpy.linalg.qr).

The least squares approximation $\vec{c}$ is the solution to

$$R_1 \vec{c} = Q_1^T \vec{y},$$

where $A_2 = Q_1 R_1$ is the thin QR decomposition of the Vandermonde matrix

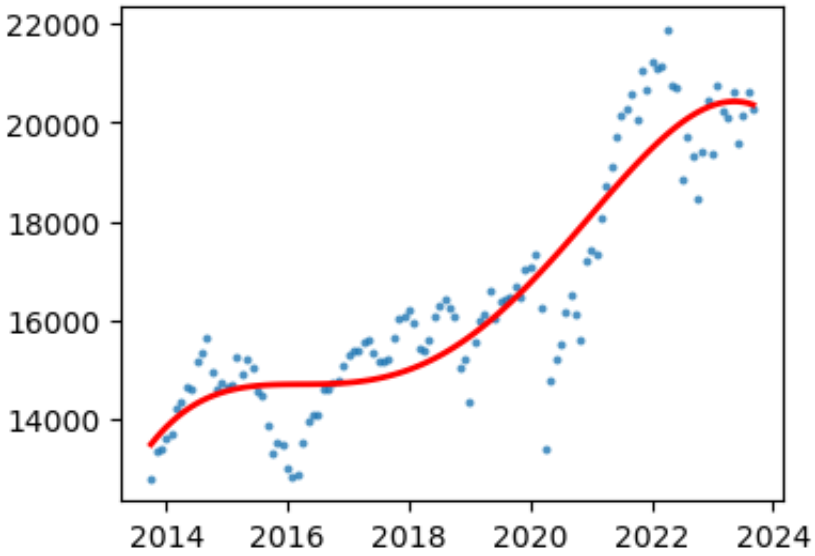
$$A_2 = \begin{bmatrix} 1 & t_1 & t_1^2 & t_1^3 & t_1^4 \\ 1 & t_2 & t_2^2 & t_2^3 & t_2^4 \\ \vdots & \vdots & & & \\ 1 & t_{120} & t_{120}^2 & t_{120}^3 & t_{120}^4 \end{bmatrix}.$$

In [19]:
```
A2 = np.column_stack([np.ones(120),t,t**2,t**3,t**4])
Q1,R1 = la.qr(A2,mode='economic')
b = Q1.T @ y
cc = la.solve(R1,b[:5])
print(cc)
```

```
[[-1.09728099e+14]
 [ 2.17434694e+11]
 [-1.61573634e+08]
 [ 5.33616227e+04]
 [-6.60873332e+00]]
/tmp/ipykernel_88/2384024634.py:4: LinAlgWarning: Ill-conditioned matrix (rco
nd=1.71132e-25): result may not be accurate.
  cc = la.solve(R1,b[:5])
```

We get a warning that the matrix is ill-conditioned, but we will ignore the warning for now, since this is only an illustration.

In [20]:
```
tq = np.linspace(2013.75,2023.75,120,endpoint=False)
yq = cc[0] + cc[1]*tq + cc[2]*tq**2 + cc[3]*tq**3 + cc[4]*tq**4
plt.plot(tq,yq,'r',linewidth=2)
plt.scatter(t,y,alpha=0.8,lw=0,s=8);
plt.show()
```



In [21]:
```
la.norm(yq-y.T)
```

Out[21]:
```
10756.545083258956
```

Generally, the higher the degree of the polynomial function, the more closely it can fit the data. That is usually a good thing, but it is worth keeping in mind that:

- Higher accuracy comes at the price of more calculations.
- If we continue to increase the polynomial degree, we run the risk of "overfitting", meaning that the approximating function is too closely modelled after the particular data used and might therefore not work well on another data set.
- Ultimately, one should not *only* look for a close fit but also consider what would be a suitable model function.
    - Is the function reasonable? Even though we could fit a 119-degree polynomial exactly to the data points in this example (good for interpolation between the points), it would not make sense to use it for predicting future outcomes.
    - What will we use the function for? Are we aiming for a sophisticated but complicated model or rather a linear model that is obviously very crude but also very easy to interpret and use?

Other types of regression

We could also try to fit other types of functions to the data. When studying financial markets like the Toronto Stock Exchange, we might want to consider an exponential function

$$y(t) = \alpha e^{\beta t}.$$

In this case, the coefficients α and β are not directly given as the solution to a linear system. However, we can make the system linear by taking the logarithm on both sides:

$$\ln(y(t)) = \ln(\alpha) + \beta t.$$

Let $\vec{z}$ be the vector with $z_i = \ln(y_i)$ and write $c_0 = \ln(\alpha)$ and $c_1 = \beta$. Then we can find c_0 and c_1 as before by solving

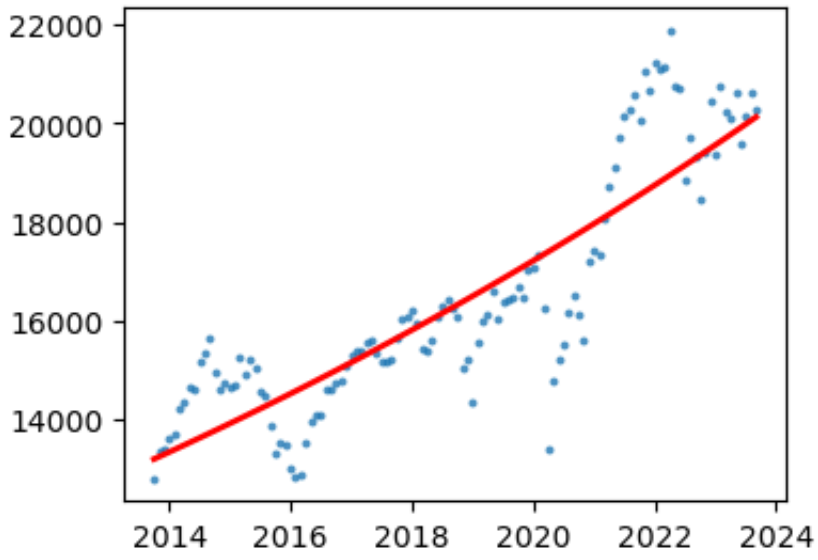
$$A^T A\vec{c} = A^T\vec{z}.$$

In [22]:
```
cexp = la.solve(A.T @ A, A.T @ np.log(y))
print(cexp)
```

```
[[-7.64273431e+01]
 [ 4.26637328e-02]]
```

Let's plot the exponential function to see how well it fits the data points.

In [23]:
```
texp = np.linspace(2013.75,2023.75,120,endpoint=False)
yexp = np.exp(cexp[0] + cexp[1]*texp)
plt.plot(texp,yexp,'r',linewidth=2)
plt.scatter(t,y,alpha=0.8,lw=0,s=8);
plt.show()
```



In [24]:
```
la.norm(yexp-y.T)
```

Out[24]: 12596.021396691656

The fit is somewhat better than with linear regression, although the difference is not very big. The coefficient β is of particular interest here, since

$$\frac{y(t+1)-y(t)}{y(t)} = \frac{\alpha e^{\beta(t+1)}}{\alpha e^{\beta t}} - 1 = e^{\beta} - 1$$

gives the annual return of the stock exchange index under this exponential model.

In [25]:
```
math.exp(cexp[1]) -1
```

Out[25]: 0.043586911748686896

The annual return is around 4.4%, which is important information when we want to consider different investment options.

Note! When we transform the system by taking the logarithm, we actually do not get the least squares solution. Instead of minimizing the sum of the squared errors, we minimize the sum of the squares of the *logarithms* of the errors. This means that the fit will be better for smaller values than for large. In this case, it would actually be better to use a *weighted least squares* method instead, but we won't discuss that here, so you can find out more about that on your own if you wish.