model_fitness

November 20, 2021

[1]:
```
import matplotlib.pyplot as plt
import seaborn as sns

%matplotlib inline


from IPython.core.display import display, HTML


import numpy as np
import pandas as pd
import sklearn

# plot layouts
sns.set_style("whitegrid")
sns.set_context("talk", font_scale=1.5, rc={"lines.linewidth": 3.5})
sns.set_palette("dark")
```

[2]:
```
HTML('''<script>
code_show=true;
function code_toggle() {
 if (code_show){
 $('div.input').hide();
 } else {
 $('div.input').show();
 }
 code_show = !code_show
}
$( document ).ready(code_toggle);
</script>
<form action="javascript:code_toggle()"><input type="submit" value="Click here␣
 ↪to toggle on/off the raw code."></form>''')
```

[2]:
```
<IPython.core.display.HTML object>
```

1 Model Fitness

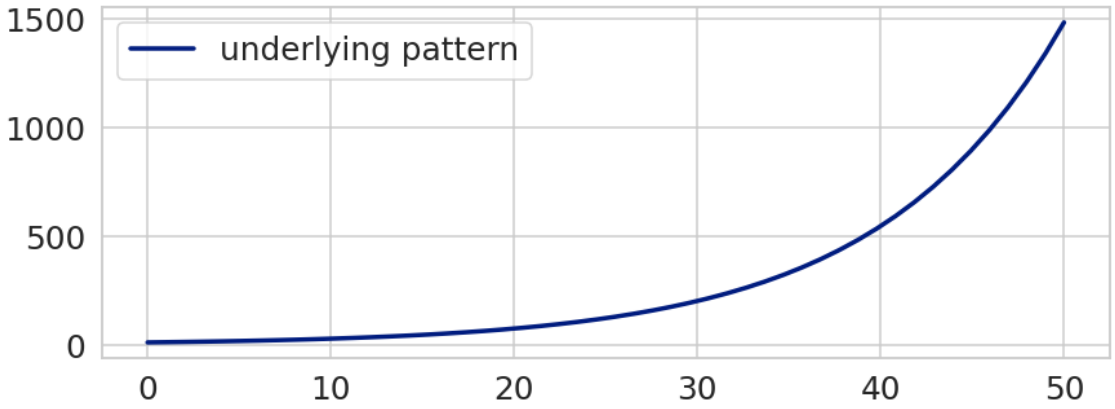
When training a model we want to resemble the data reasonable well. But what does that mean? Let's try to get some sort of intuition.

1.1 Underfitting

The model is not complex enought for the underlying phenomenon. So what is the phenomenon?

1.1.1 Example: Exponential Growth

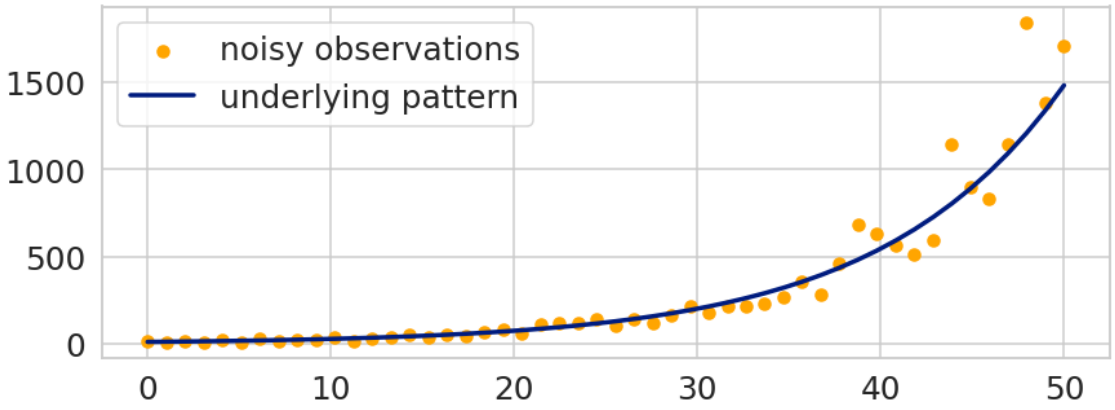
[3]:
```
def real_function(X):
    return np.exp(0.1 * X) * 10

plt.figure(figsize=(14, 5))

time = np.linspace(0, 50)

plt.plot(time, real_function(time), label='underlying pattern')
_ = plt.legend()
```



In real dataset we would never encounter the pure phenomenon. So let us mimick this situation with adding some noise.

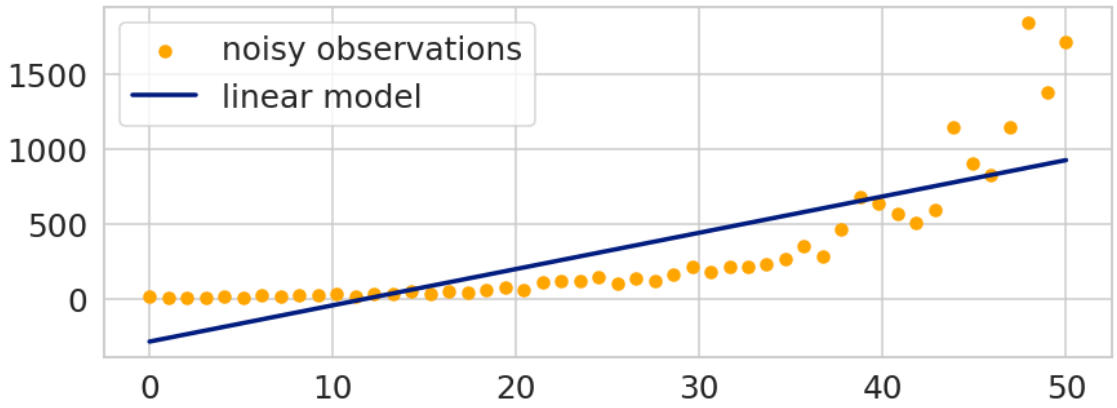
[4]:
```
plt.figure(figsize=(14, 5))

time = np.linspace(0, 50)
underlying_growth = real_function(time)

np.random.seed(1)

noisy_growth = (
    underlying_growth *
    (np.random.randn(len(underlying_growth)) * 0.25 + 1)
```

```
)

plt.scatter(time, noisy_growth, label='noisy observations', color='orange')

plt.plot(time, underlying_growth, label='underlying pattern')


_ = plt.legend()
```



Linear Regression Model Now let us try to fit the datapoints with a Linear Regression Model

[5]:
```
from sklearn.linear_model import LinearRegression

trained_lr_model = LinearRegression().fit(time[:, None], noisy_growth)

plt.figure(figsize=(14, 5))

plt.scatter(time, noisy_growth, label='noisy observations', color='orange')

plt.plot(time, trained_lr_model.predict(time[:, None]), label='linear model')

_ = plt.legend()
```



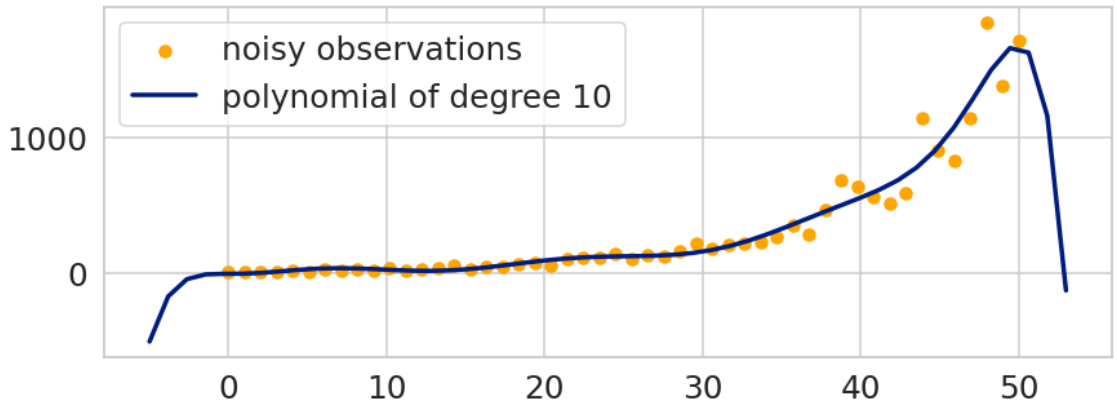
We observe * linear model does not match the pattern of the datapoints * best compromise, but too simple * not complex enough to fit the data properly.

This phenomenon is called **underfitting**.

1.2 Overfitting

Now let's approach the contrary phenomenon. We fit a polynomial of degree 10 to the datapoints.

[6]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

plt.figure(figsize=(14, 5))

trained_model = Pipeline(
    [
        ("polynomial_features", PolynomialFeatures(degree=10,␣
 ↪include_bias=True)),
        ("linear_regression", LinearRegression()),
    ]
).fit(time[:, None], noisy_growth)

plt.scatter(time, noisy_growth, label='noisy observations', color='orange')

extended_time = np.linspace(-5, 53)
plt.plot(extended_time, trained_model.predict(extended_time[:, None]),␣
 ↪label='polynomial of degree 10')

_ = plt.legend()
```



- within the time range from 0 to 50 the polynomial looks like $\exp(t)$
- outside that range the model does not follow an exponential growth
- the model fits the training data very well, but does not generalize
- no reasonable results for new data $t > 50$

These are all characteristics of a model **overfitting** to the training data.

1.3 Good Fitting

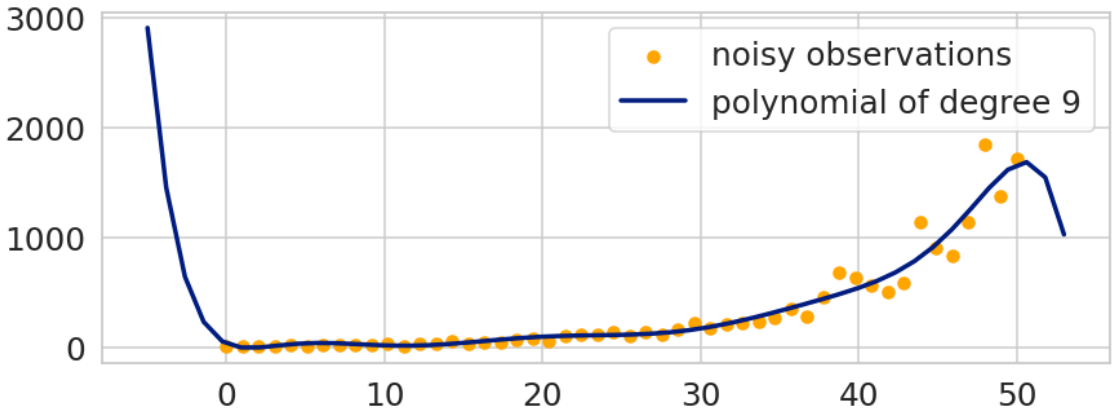
So we tried

- a polynomial of degree 1, not complex enough
- a polynomial of degree 10, is is too complex

How do we go on?

[7]:
```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

# feel free to change the degree
degree = 9

trained_model = Pipeline(
    [
        ("polynomial_features", PolynomialFeatures(degree=degree,␣
↪include_bias=True)),
        ("linear_regression", LinearRegression()),
    ]
).fit(time[:, None], noisy_growth)


plt.figure(figsize=(14, 5))
```

```
plt.scatter(time, noisy_growth, label='noisy observations', color='orange')

extended_time = np.linspace(-5, 53)
plt.plot(extended_time, trained_model.predict(extended_time[:, None]),␣
 ↪label='polynomial of degree %s' % degree)

_ = plt.legend()
```



What situation do we have?

- underfitting
- overfitting
- or appropriate fitting

1.4 Real Dataset: Diabetes

So how does it look like on a real dataset? How can we decide whether our model is an appropriate fit?

[8]:
```
import sklearn.datasets
diabetes_df = sklearn.datasets.load_diabetes(as_frame=True)['frame']
diabetes_df
```

[8]:

	age	sex	bmi	bp	s1	s2	s3	\
0	0.038076	0.050680	0.061696	0.021872	-0.044223	-0.034821	-0.043401	
1	-0.001882	-0.044642	-0.051474	-0.026328	-0.008449	-0.019163	0.074412	
2	0.085299	0.050680	0.044451	-0.005671	-0.045599	-0.034194	-0.032356	
3	-0.089063	-0.044642	-0.011595	-0.036656	0.012191	0.024991	-0.036038	
4	0.005383	-0.044642	-0.036385	0.021872	0.003935	0.015596	0.008142	
..	…	…	…	…	…	…	…	
437	0.041708	0.050680	0.019662	0.059744	-0.005697	-0.002566	-0.028674	
438	-0.005515	0.050680	-0.015906	-0.067642	0.049341	0.079165	-0.028674	
439	0.041708	0.050680	-0.015906	0.017282	-0.037344	-0.013840	-0.024993	

```
440 -0.045472 -0.044642  0.039062  0.001215  0.016318  0.015283 -0.028674
441 -0.045472 -0.044642 -0.073030 -0.081414  0.083740  0.027809  0.173816

           s4        s5        s6  target
0   -0.002592  0.019908 -0.017646   151.0
1   -0.039493 -0.068330 -0.092204    75.0
2   -0.002592  0.002864 -0.025930   141.0
3    0.034309  0.022692 -0.009362   206.0
4   -0.002592 -0.031991 -0.046641   135.0
..        …       …       …     …
437 -0.002592  0.031193  0.007207   178.0
438  0.034309 -0.018118  0.044485   104.0
439 -0.011080 -0.046879  0.015491   132.0
440  0.026560  0.044528 -0.025930   220.0
441 -0.039493 -0.004220  0.003064    57.0

[442 rows x 11 columns]
```

We are looking at a diabetes dataset of 442 patients

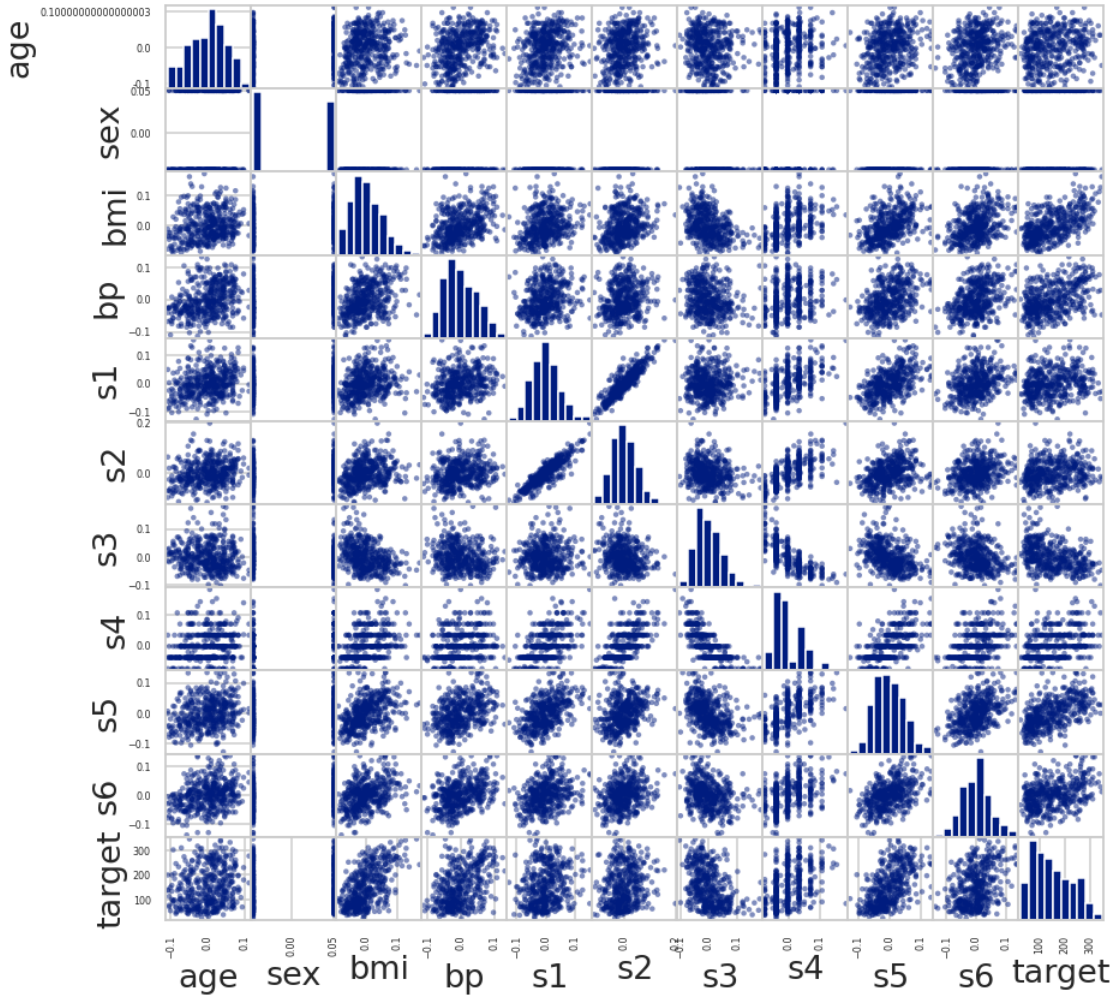
quanity of interest:

- "target" a quantitative measure of disease progression one year after baseline

features:

- age age in years
- sex
- bmi body mass index
- bp average blood pressure
- s1 tc, total serum cholesterol
- s2 ldl, low-density lipoproteins
- s3 hdl, high-density lipoproteins
- s4 tch, total cholesterol / HDL
- s5 ltg, possibly log of serum triglycerides level
- s6 glu, blood sugar level

[9]:
```
_ = pd.plotting.scatter_matrix(diabetes_df, alpha=0.5, figsize=(14, 14))
```



[10]:
```
diabetes_df.describe()
```

[10]:

	age	sex	bmi	bp	s1
count	4.420000e+02	4.420000e+02	4.420000e+02	4.420000e+02	4.420000e+02
mean	-3.639623e-16	1.309912e-16	-8.013951e-16	1.289818e-16	-9.042540e-17
std	4.761905e-02	4.761905e-02	4.761905e-02	4.761905e-02	4.761905e-02
min	-1.072256e-01	-4.464164e-02	-9.027530e-02	-1.123996e-01	-1.267807e-01
25%	-3.729927e-02	-4.464164e-02	-3.422907e-02	-3.665645e-02	-3.424784e-02
50%	5.383060e-03	-4.464164e-02	-7.283766e-03	-5.670611e-03	-4.320866e-03
75%	3.807591e-02	5.068012e-02	3.124802e-02	3.564384e-02	2.835801e-02
max	1.107267e-01	5.068012e-02	1.705552e-01	1.320442e-01	1.539137e-01

	s2	s3	s4	s5	s6
count	4.420000e+02	4.420000e+02	4.420000e+02	4.420000e+02	4.420000e+02
mean	1.301121e-16	-4.563971e-16	3.863174e-16	-3.848103e-16	-3.398488e-16
std	4.761905e-02	4.761905e-02	4.761905e-02	4.761905e-02	4.761905e-02

```
min   -1.156131e-01 -1.023071e-01 -7.639450e-02 -1.260974e-01 -1.377672e-01
25%   -3.035840e-02 -3.511716e-02 -3.949338e-02 -3.324879e-02 -3.317903e-02
50%   -3.819065e-03 -6.584468e-03 -2.592262e-03 -1.947634e-03 -1.077698e-03
75%    2.984439e-02  2.931150e-02  3.430886e-02  3.243323e-02  2.791705e-02
max    1.987880e-01  1.811791e-01  1.852344e-01  1.335990e-01  1.356118e-01
```

```
           target
count  442.000000
mean   152.133484
std     77.093005
min     25.000000
25%     87.000000
50%    140.500000
75%    211.500000
max    346.000000
```

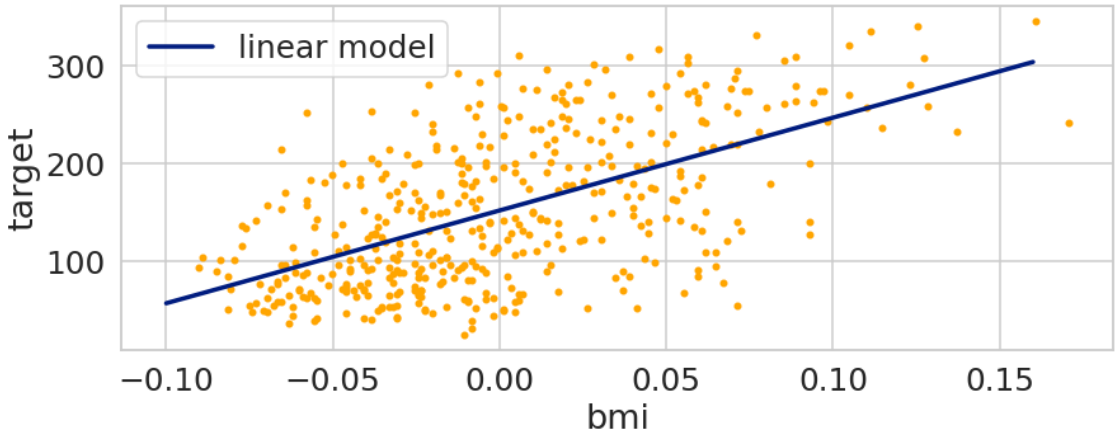
We need a metric to measure the quality of our predictions

[11]:
```
import math
from sklearn.metrics import mean_squared_error

ax = diabetes_df.plot.scatter(x='bmi', y='target', figsize=(14, 5),␣
↪color='orange')


trained_model = Pipeline(
    [
        ("polynomial_features", PolynomialFeatures(degree=1,␣
↪include_bias=True)),
        ("linear_regression", LinearRegression()),
    ]
).fit(diabetes_df['bmi'].values[:, None], diabetes_df['target'])


ax.plot(
    np.linspace(-0.1, 0.16, 100),
    trained_model.predict(np.linspace(-0.1, 0.16, 100)[:, None]),
    label='linear model'
)

print(
    'The Root Mean Squared Error on all data amounts to %4.2f' %␣
↪mean_squared_error(
        trained_model.predict(diabetes_df['bmi'].values[:, None]),
        diabetes_df['target'],
        squared=False,
    )
```

```
)

ax.legend()
```

```
The Root Mean Squared Error on all data amounts to 62.37
```

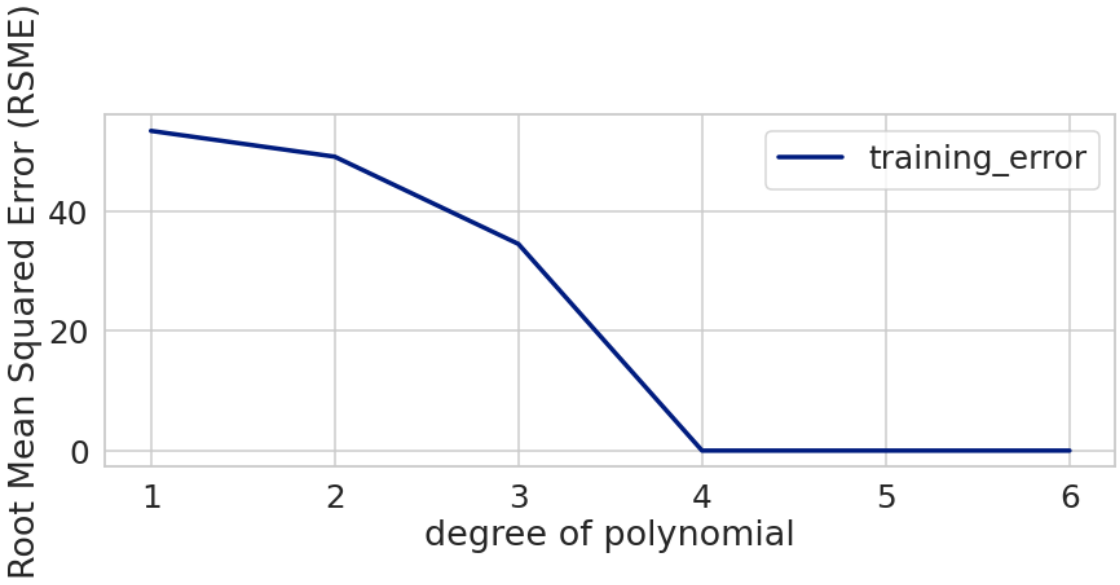
[11]:
```
<matplotlib.legend.Legend at 0x7fa23f7c30a0>
```



We can try to reduce that metric, by including more features.

[12]:
```
feature_names = list(diabetes_df.columns)[:-1]

features = diabetes_df[feature_names].values

polynom_degrees = list(range(1, 7))

trained_models = [
    Pipeline(
        [
            ("polynomial_features", PolynomialFeatures(degree=degree,␣
 ↪include_bias=True)),
            ("linear_regression", LinearRegression()),
        ]
    ).fit(features, diabetes_df['target'])
    for degree in polynom_degrees
]

training_errors = [
    mean_squared_error(
        model.predict(diabetes_df[feature_names].values),
        diabetes_df['target'],
        squared=False,
```

```
    )
    for model in trained_models
]


plt.figure(figsize=(14, 5))
plt.plot(
    polynom_degrees,
    training_errors,
    label='training_error',
)

plt.xlabel('degree of polynomial')
plt.ylabel('Root Mean Squared Error (RSME)')

_ = plt.legend()
```



We observe

- the higher the degree of the polynomial the lower the error
- model gets more complex training error reduces to zero

What kind of fitting do we see here?

1.5 Characteristics of Overfitting

- the model gets more complex and remembers the training data
- the model does not generalize to new data

1.6 Generalization Error

What would be the error on unseen data?

1.7 Data Splits

- need **test data** that the model never saw before
- we can use the test data exactly one time
- need to mimick the process of performance on unseen data
- so we need a **training** and a **validation** set

2 Cross Validation

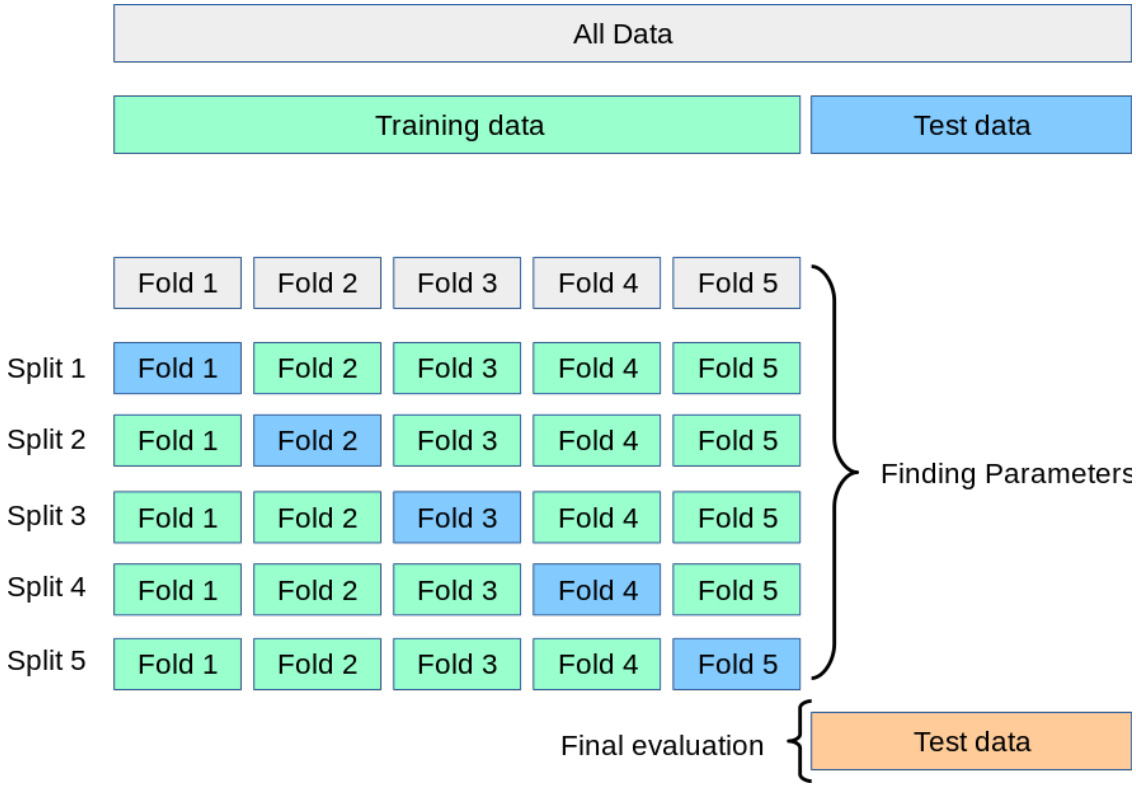


Image taken from Scikit-Learn

What is the essence of Cross Validation?

- mimick prediction on unseen data
- each fold of the training data is part of the validation set once

So how can we use cross validation for our diabetes model ?

2.1 Cross Validation on Diabetes Dataset

Seek:

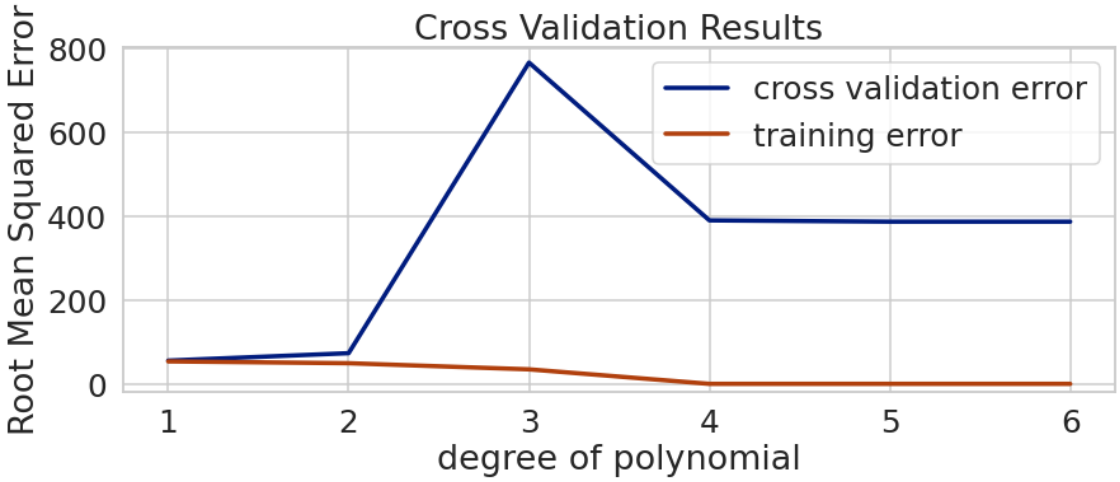
polynom degree that generalizes well

[13]:
```
from sklearn.model_selection import train_test_split

feature_names = list(diabetes_df.columns)[:-1]

features = diabetes_df[feature_names].values

labels = diabetes_df['target'].values

X_train, X_test, Y_train, Y_test = train_test_split(
    features,
    labels,
    test_size=0.4,
)

print('Size of the whole dataset: %s' % len(labels))
print('Size of the training set: %s' % len(Y_train))
print('Size of the test set: %s' % len(Y_test))
```

```
Size of the whole dataset: 442
Size of the training set: 265
Size of the test set: 177
```

[14]:
```
from sklearn.model_selection import cross_val_score

polynom_degrees = list(range(1, 7))

cv_scores = [
    np.mean(
        cross_val_score(
            Pipeline(
                [
                    ("polynomial_features", PolynomialFeatures(degree=degree,␣
 ↪include_bias=True)),
                    ("linear_regression", LinearRegression()),
                ]
            ),
            X_train,
            Y_train,
            cv=5,
            scoring='neg_root_mean_squared_error'
        )
    ) * -1.
    for degree in polynom_degrees
]
```

```
plt.figure(figsize=(14, 5))
plt.plot(
    polynom_degrees,
    cv_scores,
    label='cross validation error'
)

plt.plot(
    polynom_degrees,
    training_errors,
    label='training error'
)

plt.title('Cross Validation Results')
plt.xlabel('degree of polynomial')
plt.ylabel('Root Mean Squared Error')
plt.ylim(-20)

_ = plt.legend()
```



So which degree would you pick?

3 Regularization

Let us take a look at our current linear regression model

$\hat{y} = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_{10} x_{10}$

[15]:
```
model = LinearRegression().fit(X_train, Y_train)

for idx, (feature_name, value) in enumerate(
```

```
    zip(
        ['bias'] + feature_names,
        [float(model.intercept_)] + list(model.coef_)
    )
):
    print('w_%2s: % 8.2f  %5s' % (idx, value, feature_name))
```

```
w_ 0:   152.30   bias
w_ 1:    12.45    age
w_ 2:  -207.86    sex
w_ 3:   555.81    bmi
w_ 4:   359.92     bp
w_ 5: -1214.22     s1
w_ 6:   862.04     s2
w_ 7:   292.50     s3
w_ 8:   227.90     s4
w_ 9:   904.51     s5
w_10:   -21.00     s6
```

As all the features have the same mean and the same standard deviation, the absolute value of the coefficients gives an indication how important they are for the model.

3.1 Lasso

Now we want to find a model that is

- less complex (less features)
- compareable or better in terms of performance
- generalizes better

[16]:
```
from sklearn import linear_model

model = linear_model.Lasso(alpha=0.1).fit(X_train, Y_train)

for idx, (feature_name, value) in enumerate(
    zip(
        ['bias'] + feature_names,
        [float(model.intercept_)] + list(model.coef_)
    )
):
    print('w_%2s: % 8.2f  %5s' % (idx, value, feature_name))
```

```
w_ 0:   152.14   bias
w_ 1:     0.00    age
w_ 2:  -109.40    sex
w_ 3:   582.08    bmi
w_ 4:   305.36     bp
w_ 5:    -0.12     s1
w_ 6:    -0.00     s2
w_ 7:  -226.01     s3
```

```
w_ 8:     0.00     s4
w_ 9:   416.54     s5
w_10:     0.00     s6
```

We observe that the model choose to ignore the features:

- age
- s2
- s4
- s6

How did we achieve that?

Changed of loss function to

$\min_w \sum_{i=1}^{N} (\hat{y}_i - y_i)^2 + \alpha \sum_{k=0}^{K} |w_k|$

with

$\hat{y}_i = \sum_{k=0}^{K} w_k x_{ik}$

The new term in the the loss function $\alpha \sum_{k=0}^{K} |w_k|$ is called the **regularization** term for **Lasso**.

- α is the regularization strength
- changing α enables you to control the weight size
- sometimes we do not want to regularize the bias term

What happens when we decrease α?

[17]:
```
n_alphas = 200
alphas = np.logspace(-6, 1, n_alphas)

models = [
    linear_model.Lasso(alpha=a).fit(
        X_train, Y_train
    )
    for a in alphas
]

coefficients = np.array([
    [float(model.intercept_)] + list(model.coef_)
    for model in models
])

plt.figure(figsize=(14, 10))
ax = plt.gca()



for name, one_coef in zip(
    ['bias'] + feature_names,
    coefficients.T,
```

```
):
    ax.plot(alphas, one_coef, label=name)

ax.legend()

ax.set_xscale("log")
ax.set_xlim(ax.get_xlim()[::-1])  # reverse axis
plt.xlabel("regulariation strength alpha")
plt.ylabel("weights")
plt.title("Lasso coefficients as a function of alpha")
plt.axis("tight")
plt.show()
```