Nonlinear Regression: Feature Transformation and Basis Functions

One trick we can use to adapt linear regression for nonlinear data is to **transform the features according to some functions**. To apply such transformation systematically, we will often use some **basis functions** for $\phi(x)$. For example, a polynomial regression:

(1) $\phi(x) = \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_m x^m = \sum_{i=1}^{m} \alpha_i x^i$

(2) $y \approx w_0 + w_1 \phi(x) \approx w_0 + w_1 (\sum_{i=1}^{m} \alpha_i x^i)$

Remark: again, $y \approx w_0 + w_1 \phi(x)$ remains linear with respect to w_1.

In addition to polynomial basis functions, another popular choice of basis functions is Gaussian basis functions (or radial basis functions, RBF):

(3) $\phi(x) = \alpha_1 e^{-\frac{x-\mu_1}{2s^2}} + \alpha_2 e^{-\frac{x-\mu_2}{2s^2}} + \cdots + \alpha_m e^{-\frac{x-\mu_m}{2s^2}} = \sum_{i=1}^{m} \alpha_i e^{-\frac{x-\mu_i}{2s^2}}$

(4) $y \approx w_0 + w_1 \phi(x) \approx w_0 + w_1 (\sum_{i=1}^{m} \alpha_i e^{-\frac{x-\mu_i}{2s^2}})$

where the μ_i are the locations of the basis functions in input space. The parameter s governs their spatial "coverage" in input space.

We will briefly introduce the usage and Python implementation of these two frequently used basis functions.

Polynomial basis functions

This polynomial projection is useful enough that it is built into `Scikit-Learn`, using the `PolynomialFeatures` transformer:

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, np.newaxis])
```

```
array([[  2.,   4.,   8.],
       [  3.,   9.,  27.],
       [  4.,  16.,  64.]])
```

Notice that the `np.newaxis` object is used to reshape a one-dimensional array into a one-dimensional column vector as we have explained and the parameter `include_bias:boolean` is `True` (default), then the transformation will include a bias column, the feature in which all polynomial powers are zero (i.e. a column of ones - acts as an intercept term in a linear model).

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3)
poly.fit_transform(x[:, np.newaxis])
```

```
array([[  1.,   2.,   4.,   8.],
       [  1.,   3.,   9.,  27.],
       [  1.,   4.,  16.,  64.]])
```

We see here that the transformer has converted our one-dimensional array into a three-dimensional array by taking the polynomial basis of each value. This new, **higher dimensional data representation** can then be plugged into a linear regression.

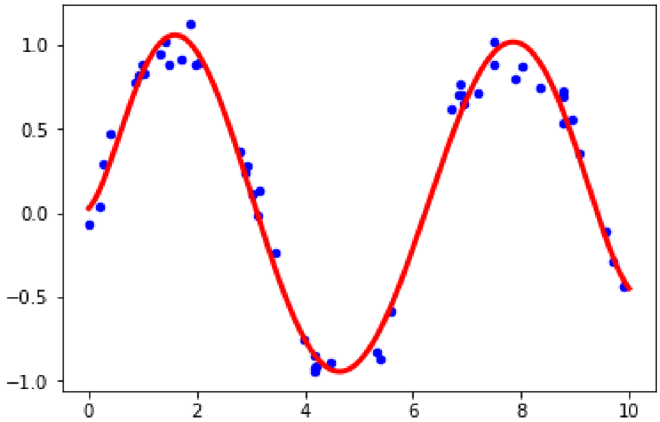
The cleanest way to accomplish this in Python is to use a pipeline. Let's make a 7th-degree polynomial model in this way:

```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7), LinearRegression())
```

With this transform in place, we can use the linear model to fit much more complicated relationships between x and y. For example, here is a sine wave with noise

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
import matplotlib.pyplot as plt
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression

poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

poly_model.fit(x[:, np.newaxis], y)
xfit = np.linspace(0, 10, 1000)
```

```
yfit = poly_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y, c='b', marker='o', s=20)
plt.plot(xfit, yfit, c='r', lw='3')
plt.show()
```

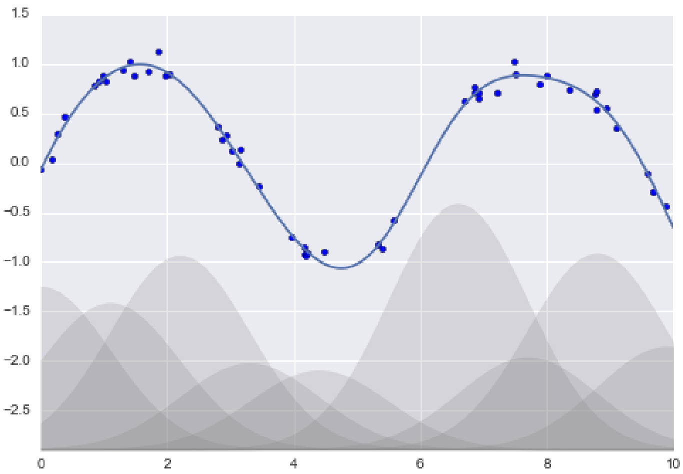


Notice that `np.random` is random sampling in NumPy. `np.random.RandomState` exposes several methods for generating random numbers drawn from various probability distributions. `randn` is a method that returns a sample (or samples) from the "standard normal" distribution.

Using 7th-order polynomial basis functions, our linear model can provide an excellent fit to this non-linear data! You can download the above Python codes `FT_PolyBasis.ipynb` from the course website.

Gaussian basis functions (Radial basis functions, RBF)

Of course, other basis functions are possible. For example, one useful pattern is to fit a model that is not a sum of polynomial bases but **a sum of Gaussian basis** (or radial basis). The results are:

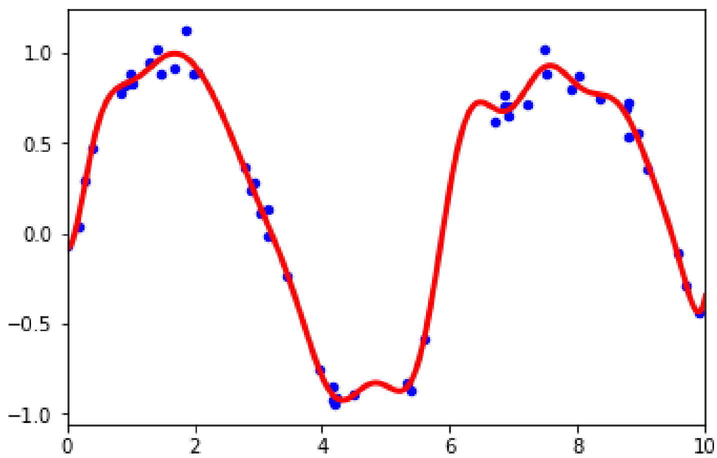


The shaded regions in the plot are the **scaled basis functions**, and when added together they reproduce the smooth curve through the data.

These Gaussian basis functions are not built into `Scikit-Learn`, but we can write a custom transformer that will create them (`Scikit-Learn` transformers are implemented as Python classes; reading `Scikit-Learn`'s source is a good way to see how they can be created):

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LinearRegression

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] -
self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
```

```
                              LinearRegression())
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)
gauss_model.fit(x[:, np.newaxis], y)
xfit = np.linspace(0, 10, 1000)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10)
plt.show()
```



You can download the above Python codes `FT_GaussianBasis.ipynb` from the course website.

Remark: If you have some sort of intuition (domain knowledge, physics etc.) into the generating process of your data that makes you think <u>one basis or another</u> might be appropriate, you should use it.