Nonlinear Regression: Regularization

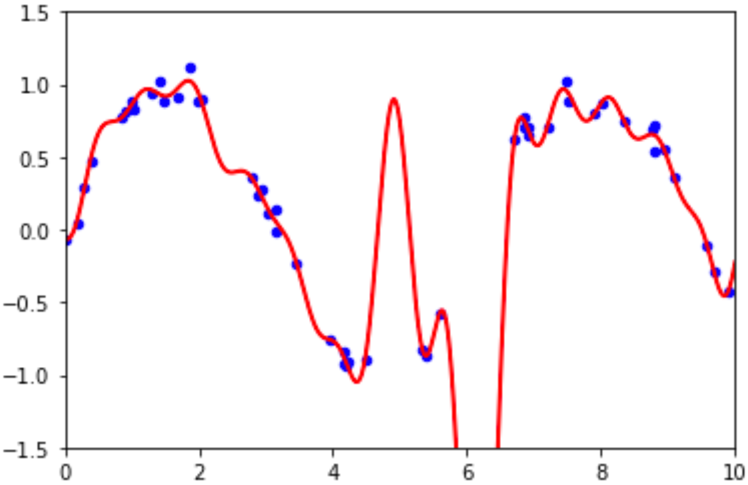
The kernel method allows us to easily extend linear models into nonlinear models. Nonlinear models are powerful but tend to **overfit**. For example, if we choose too many Gaussian basis functions, we end up with results that don't look so good:

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

model = make_pipeline(GaussianFeatures(30),
                      LinearRegression())
model.fit(x[:, np.newaxis], y)

plt.scatter(x, y, c='b', marker='o', s=20)
xfit = np.linspace(0, 10, 1000)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]), c='r', lw='2')

plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);
plt.show()
```



Q: what do you observe?

A:

overfit

With the data projected to the **30** Gaussian basis, the model has far too much flexibility.

The model goes to extreme values between locations where it is constrained by data.

We can see the reason for this if we plot the coefficients of the Gaussian bases α_i with respect to their locations μ_i (x-axis):

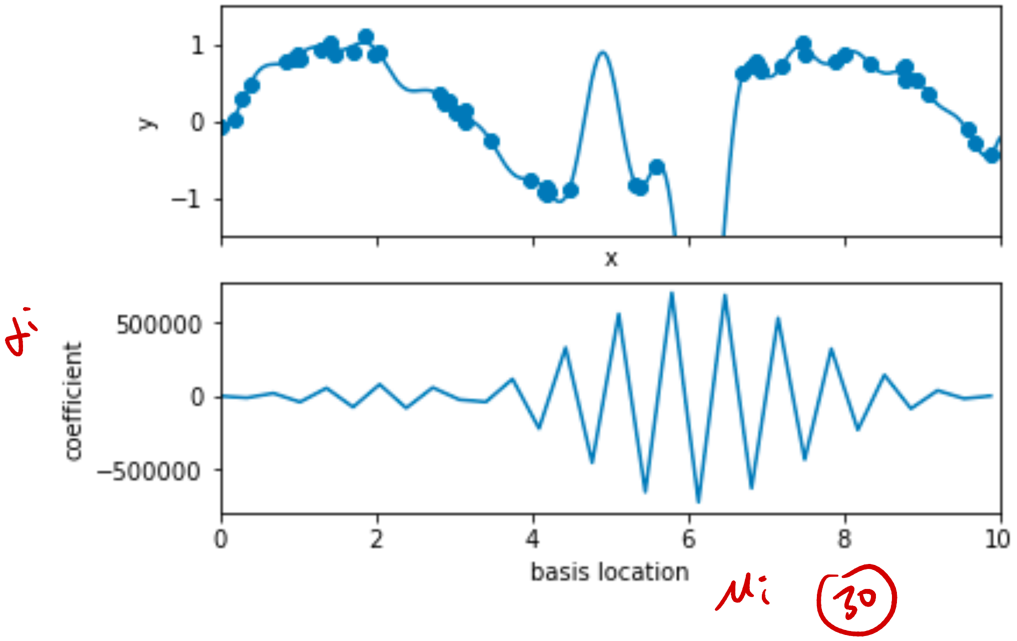
$$y \approx w_0 + w_1\phi(x) \approx w_0 + w_1\Big(\sum_{i=1}^{\overset{\boxed{30}}{m}} \alpha_i e^{-\frac{x-\mu_i}{2s^2}}\Big)$$

```
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:
        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))
    plt.show()

model = make_pipeline(GaussianFeatures(30), LinearRegression())
basis_plot(model)
```



The figures above show the amplitude of the basis function at each location.

Q: what do you observe?

A:

This is typical overfitting behavior when basis functions overlap: t**he coefficients of adjacent basis**

functions blow up and cancel each other out.

We know that such behavior is problematic, and it would be nice if we could limit such spikes explicitly in the model by **penalizing** large values of the model parameters. Such a penalty is known as **regularization**, and the two best-known techniques are **ridge** regression and the **lasso** regression.

1. Ridge Regression

Theoretical Minimum

The hypothesis or function for linear regression model is:

$h(\mathbf{x}) = w_0 + \sum_{i=1}^{d} w_i x_i = \mathbf{w}^{\mathrm{T}}\mathbf{x}$

where $\mathbf{w}$ is the column vector of the weights $[w_0\ w_1 \cdots w_d]^T$ for d distinct features. Consider a dataset $(\mathbf{x}_1,\ y_1), (\mathbf{x}_2,\ y_2), \cdots, (\mathbf{x}_n,\ y_n)$ and recall that the least squares fitting procedure estimates $w_0, w_1, \cdots, w_d$ using the dataset that minimizes:

$$E(\mathbf{w}) = \sum_{i=1}^{n} \left(y_i - \overset{\hat{y}}{h(\mathbf{x})}\right)^2$$

Ridge regression is very similar to least squares, except that the coefficients are estimated by minimizing a slightly different quantity. In particular, the ridge regression coefficient estimates are the values that minimize:

$$E_{\text{aug}}(\mathbf{w}) = E(\mathbf{w}) + \alpha \mathbf{w}^{\mathrm{T}}\mathbf{w}$$

where $\alpha \geq 0$ is a tuning parameter, to be determined separately.

Remarks:

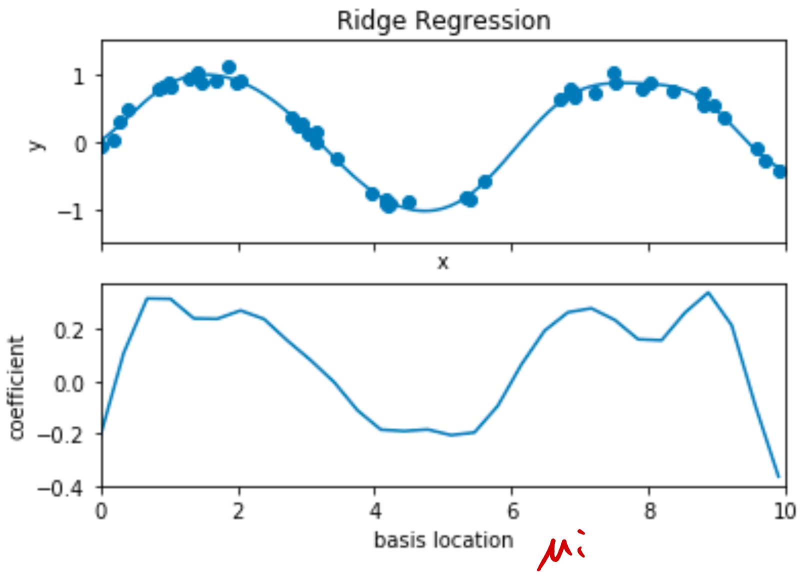
1. Ridge regression trades off two different criteria. As with least squares, ridge regression seeks coefficient estimates that fit the data well, by making the $E(\mathbf{w})$ small. However, the second term, $\alpha\mathbf{w}^{\mathrm{T}}\mathbf{w}$ called a **penalty**, is shrinkage small when $w_0, w_1, \cdots, w_d$ are close to zero, and **so it has the effect of shrinking penalty the estimates of $\mathbf{w}$ towards zero**.
2. The tuning parameter α serves to control the relative impact of these two terms on the regression coefficient estimates. When $\alpha = 0$, the penalty term has no effect, and ridge regression will produce the least squares estimates. However, as $\alpha \to \infty$, the impact of the shrinkage penalty

grows, and the $w_0, w_1, \cdots, w_d$ will approach zero.

Python Example

We can now run the overfitting case of 30 Gaussian basis with the ridge regression. The ridge regression is built into `Scikit-Learn` with the `Ridge` estimator.

```
from sklearn.linear_model import Ridge
model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
basis_plot(model, title='Ridge Regression')
```



The α parameter is essentially a knob controlling the complexity of the resulting model. In the limit $\alpha = 0$, we recover the standard linear regression result; in the limit $\alpha \to \infty$, all model responses will be suppressed. Ridge regression can be computed very efficiently—at hardly more computational cost than the original linear regression model.

2. Lasso Regression

Theoretical Minimum

Another very common type of regularization is known as lasso, and involves penalizing the sum of absolute values (1-norms)[1] of regression coefficients:

[1] Ridge regression is known as L_2 normalization and lasso L_1 normalization.

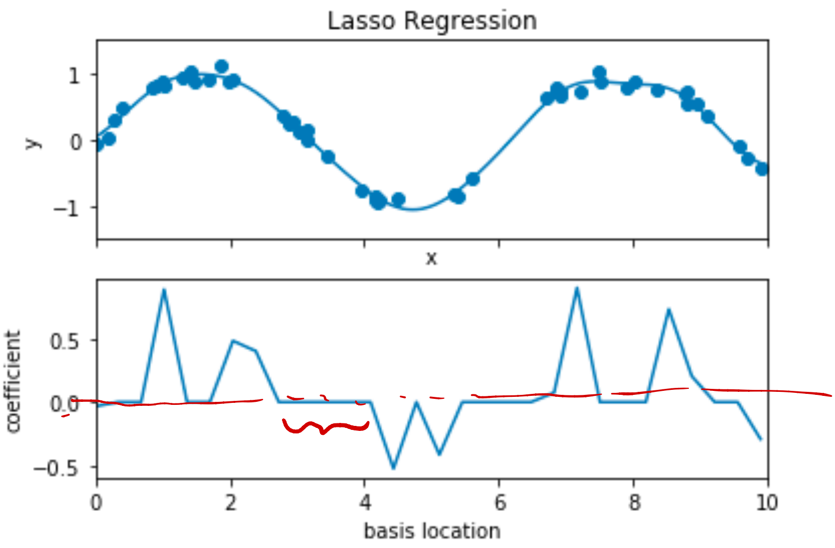
$$E_{\text{aug}}(\mathbf{w}) = E(\mathbf{w}) + \underline{\alpha} \sum_{i=1}^{d} \underline{|w_i|}$$

Though this is conceptually very similar to ridge regression, the results can differ surprisingly: for example, due to geometric reasons lasso regression preferentially sets model coefficients to **exactly zero**.

We can see this behavior in duplicating the ridge regression figure, but using $\underline{L_1}$-normalized coefficients:

```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001,
max_iter=10000))
basis_plot(model, title='Lasso Regression')
```

α_i



μ_i

Q: what do you observe?

A:

With the lasso regression penalty, the majority of the coefficients are exactly zero, with the functional behavior being modeled by a small subset of the available basis functions.

As with ridge regularization, the α parameter tunes the strength of the penalty, and should be determined via, for example, cross-validation.

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