Python for Machine Learning and Data Analysis

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Spring 2019

Machine Learning 03

Lecture 9

Content

- Naive Bayes Classification
  - Bayesian Classification
  - Gaussian Naïve Bayes
  - Multinomial Naïve Bayes
  - When to use Naïve Bayes
- Linear Regression
  - Simple Linear Regression
  - Basis Function Regression
  - Regularization
  - Example:

1. Bayesian Classification

- Bayesian Classification relies on Bayes's theorem, which is an equation describing the relationship of conditional probabilities of statistical quantities.
- In Bayesian classification, we're interested in finding the probability of a label given some observed features, which we can write as $P(L\sim \mid\sim features)$
- Bayes' theorem tells us how to express this in terms of quantities we can compute more directly:

$$P(L\sim \mid\sim features) = \frac{P(features\sim \mid\sim L)\, P(L)}{P(features)}$$

- Naive Bayes classifiers are built on Bayesian classification methods.

- If we are trying to decide between two labels — let's call them $L1$ and $L2$ — then one way to make this decision is to compute the ratio of the posterior probabilities for each label:

$$\frac{P(L_1 \sim \mid features \sim)}{P(L_2 \sim \mid features \sim)} = \frac{P(features \sim \mid \sim L_1)\, P(L_1)}{P(features \sim \mid \sim L_2)\, P(L_2)}$$

- All we need now is some model by which we can compute $P(features \sim \mid \sim L_i)$ for each label. Such a model is called a **generative model** because it specifies the hypothetical random process that generates the data.

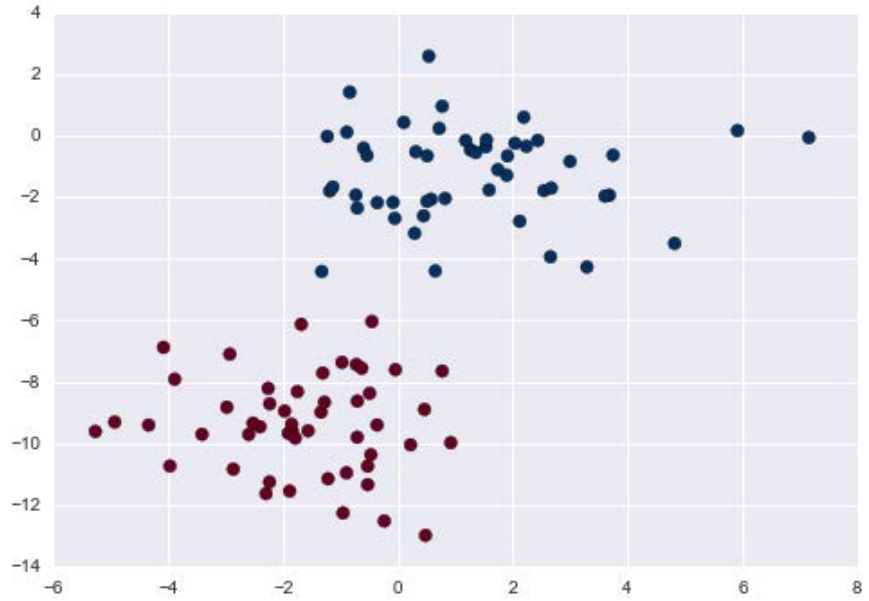
- Specifying this generative model for each label is the main piece of the training of such a Bayesian classifier. The general version of such a training step is a very difficult task, but we can make it simpler through the use of some simplifying assumptions about the form of this model.

- This is where the "naive" in "naive Bayes" comes in: if we make very naive assumptions about the generative model for each label, we can find a rough approximation of the generative model for each class, and then proceed with the Bayesian classification.

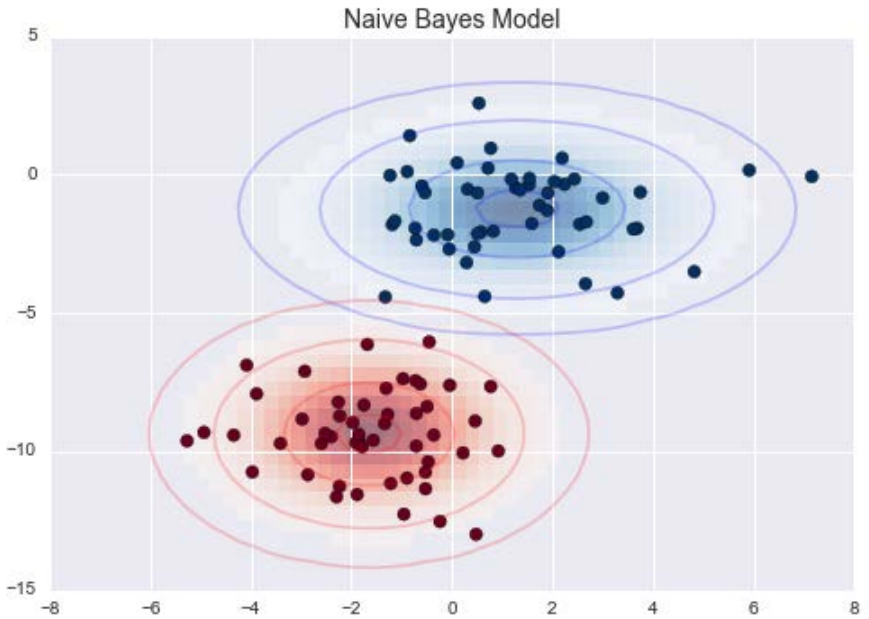
- Different types of naive Bayes classifiers rest on different naive assumptions about the data, and we will examine a few of these in the following sections.
  - Gaussian Naive Bayes
  - Multinomial Naive Bayes

1.1 Gaussian Naïve Bayes

- The easiest naive Bayes classifier to understand is Gaussian naive Bayes. In this classifier, the assumption is that *data from each label is drawn from a simple Gaussian distribution*.



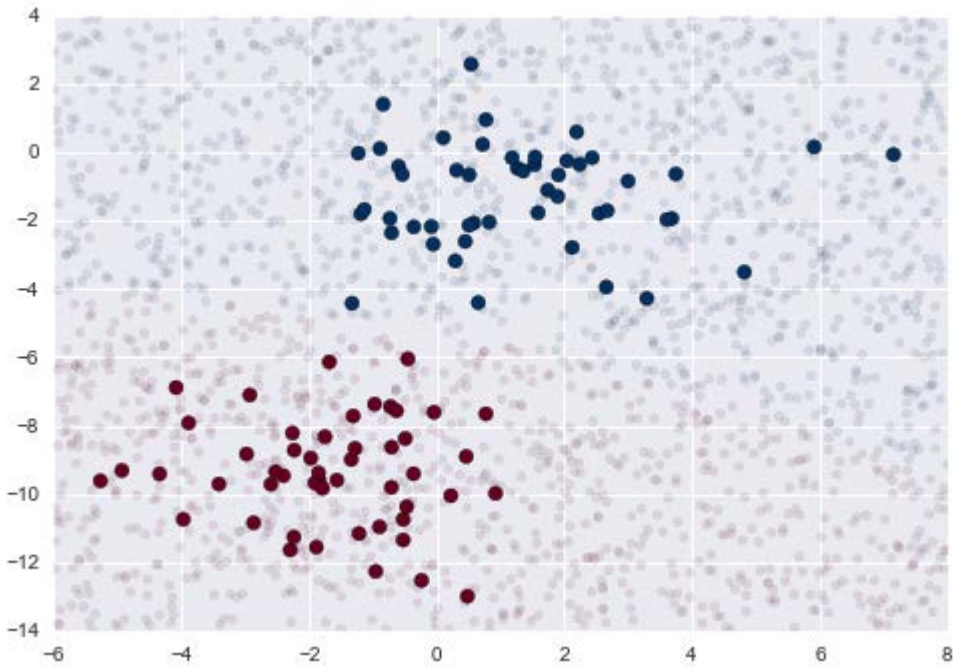
- One extremely fast way to create a simple model is to assume that the data is described by a Gaussian distribution with no covariance between dimensions.
- This model can be fit by simply finding the mean ar standard deviation of the points within each label, which is all you need to define such a distribution.



- The ellipses here represent the Gaussian generative model for each label, with larger probability toward the center of the ellipses.

- With this generative model in place for each class, we have a simple recipe to compute the likelihood $P(features\sim|\sim L_1)$ for any data point, and thus we can quickly compute the posterior ratio and determine which label is the most probable for a given point.

- This procedure is implemented in Scikit-Learn's sklearn.naive_bayes.GaussianNB estimator



- We see a slightly curved boundary in the classifications—in general, the boundary in Gaussian naive Bayes is quadratic.

- A nice piece of this Bayesian formalism is that it naturally allows for probabilistic classification, which we can compute using the **predict_proba** method:

```
array([[ 0.89,  0.11],
       [ 1.  ,  0.  ],
       [ 1.  ,  0.  ],
       [ 1.  ,  0.  ],
       [ 1.  ,  0.  ],
       [ 1.  ,  0.  ],
       [ 0.  ,  1.  ],
       [ 0.15,  0.85]])
```

- The columns give the posterior probabilities of the first and second label, respectively.
- If you are looking for estimates of uncertainty in your classification, Bayesian approaches like this can be a useful approach.

- Of course, the final classification will only be as good as the model assumptions that lead to it, which is why Gaussian naive Bayes often does not produce very good results.
- Still, in many cases—especially as the number of features becomes large—this assumption is not detrimental enough to prevent Gaussian naive Bayes from being a useful method.

1.2 Multinomial Naïve Bayes

- Multinomial naive Bayes is another useful assumption to be used to specify the generative distribution for each label.
- The features of multinomial naive Bayes are assumed to be generated from a simple multinomial distribution.
- The multinomial distribution describes the probability of observing counts among a number of categories, and thus multinomial naive Bayes is most appropriate for features that represent counts or count rates.
- The idea is precisely the same as before, except that instead of modeling the data distribution with the best-fit Gaussian, we model the data distribution with a best-fit multinomial distribution.

Example: Classifying Text

- One place where multinomial naive Bayes is often used is in text classification, where the features are related to word counts or frequencies within the documents to be classified.

- Here we will use the sparse word count features from the 20 Newsgroups corpus to show how we might classify these short documents into categories.

1. Let's download the data and take a look at the target names.
2. For simplicity here, select just a few of these categories, and download the training and testing set.
3. Give a representative entry from the data.
4. In order to use this data for machine learning, need to be able to convert the content of each string into a vector of numbers. For this, use the TF-IDF vectorizer (discussed in Feature Engineering), and create a pipeline that attaches it to a multinomial naive Bayes classifier.
5. With this pipeline, apply the model to the training data, and predict labels for the test data.
6. Now that we have predicted the labels for the test data, we can evaluate them to learn about the performance of the estimator. For example, use the confusion matrix between the true and predicted labels for the test data.

- Evidently, even this very simple classifier can successfully separate space talk from computer talk, but it gets confused between talk about religion and talk about Christianity. This is perhaps an expected area of confusion!

- The very cool thing here is that we now have the tools to determine the category for any string, using the predict() method of this pipeline. Here's a quick utility function that will return the prediction for a single string:

  predict_category('sending a payload to the ISS')
  'sci.space'

- Remember that this is nothing more sophisticated than a simple probability model for the (weighted) frequency of each word in the string; nevertheless, the result is striking. Even a very naive algorithm, when used carefully and trained on a large set of high-dimensional data, can be surprisingly effective.

1.3 When to use Naïve Bayes

- Because naive Bayesian classifiers make such stringent assumptions about data, they will generally not perform as well as a more complicated model. That said, they have several advantages:
  - They are extremely fast for both training and prediction
  - They provide straightforward probabilistic prediction
  - They are often very easily interpretable
  - They have very few (if any) tunable parameters
- These advantages mean a naive Bayesian classifier is often a good choice as an initial baseline classification.
  - If it performs suitably, then congratulations: you have a very fast, very interpretable classifier for your problem.
  - If it does not perform well, then you can begin exploring more sophisticated models, with some baseline knowledge of how well they should perform.

- Naive Bayes classifiers tend to perform especially well in one of the following situations:
  - When the naive assumptions actually match the data (very rare in practice)
  - For very well-separated categories, when model complexity is less important
  - For very high-dimensional data, when model complexity is less important
- The last two points seem distinct, but they actually are related: as the dimension of a dataset grows, it is much less likely for any two points to be found close together (after all, they must be close in *every single dimension* to be close overall).
- This means that clusters in high dimensions tend to be more separated, on average, than clusters in low dimensions, assuming the new dimensions actually add information.
- For this reason, simplistic classifiers like naive Bayes tend to work as well or better than more complicated classifiers as the dimensionality grows: once you have enough data, even a simple model can be very powerful.

2. Linear Regression

- Naïve Bayes is a good starting point for classification tasks.
- Linear regression models are a good starting point for regression tasks.
  - Such models are popular because they can be fit very quickly, and are very interpretable.
  - The simplest form of a linear regression model is to fit a straight line to data; and can be extended to model more complicated data behavior.

- In this section we will start with a quick intuitive walk-through of the mathematics behind this well-known problem, before moving on to see how linear models can be generalized to account for more complicated patterns in data.

2.1 Simple Linear Regression

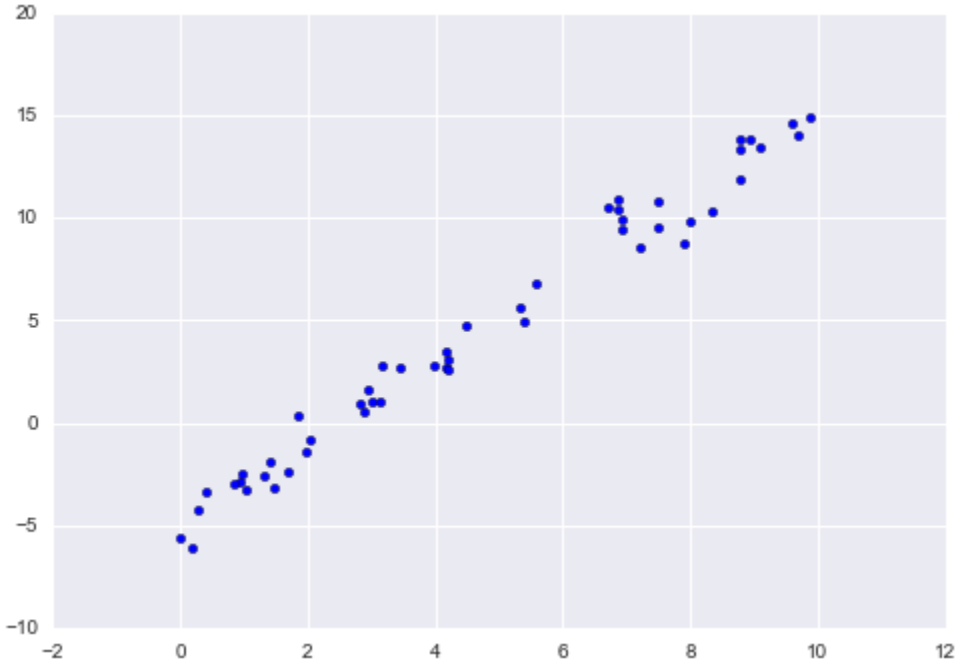
- Start with the most familiar linear regression, a straight-line fit to data.
- A straight-line fit is a model of the form

$$y = ax + b$$

where *a* is commonly known as the *slope*, and *b* is commonly known as the *intercept*.

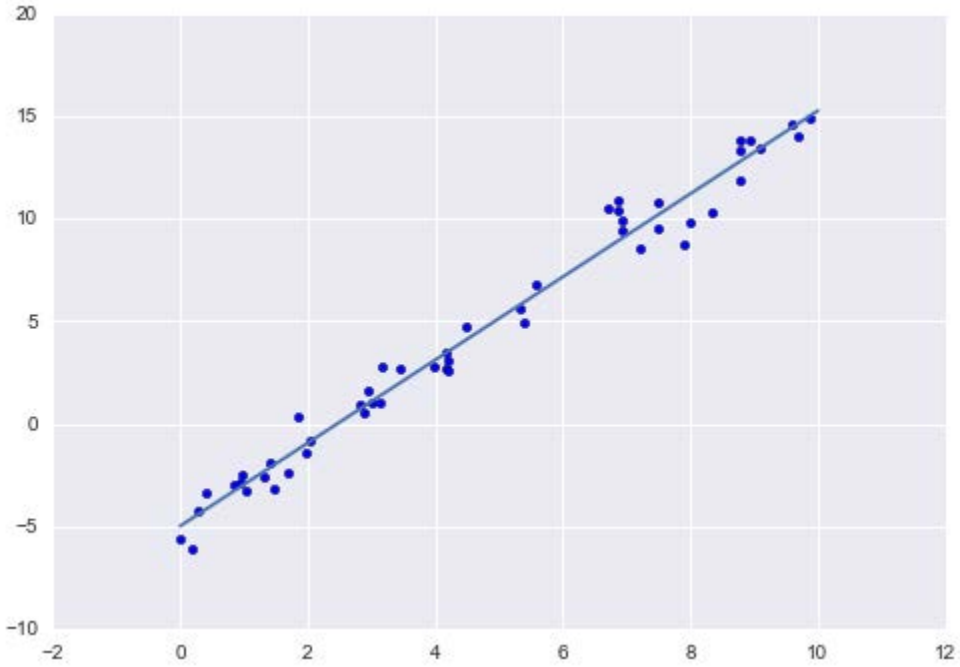
Consider the data, which is scattered about a line with *a slope of a=2* and *an intercept of b= -5:*

- use Scikit-Learn's LinearRegression estimator to fit this data and construct the best-fit line:



The slope and intercept of the data are contained in the model's fit parameters, which in Scikit-Learn are always marked by a trailing underscore. Here the relevant parameters are coef_ and intercept_:

Model slope: 2.02720881036
Model intercept: -4.99857708555



- The LinearRegression estimator is much more capable than the simple straight-line fits, it can also handle multidimensional linear models of the form.

$$y = a_0 + a_1 x_1 + a_2 x_2 + \cdots$$

where there are multiple x values. Geometrically, this is akin to fitting a plane to points in three dimensions, or fitting a hyper-plane to points in higher dimensions.

The multidimensional nature of such regressions makes them more difficult to visualize, but we can see one of these fits in action by building some example data, using NumPy's matrix multiplication operator:

```
rng = np.random.RandomState(1)
X = 10 * rng.rand(100, 3)
y = 0.5 + np.dot(X, [1.5, -2., 1.])

model.fit(X, y)
print(model.intercept_)
print(model.coef_)
```

Here the y data is constructed from three random x values, and the linear regression recovers the coefficients used to construct the data.

In this way, we can use the single LinearRegression estimator to fit lines, planes, or hyperplanes to our data. It still appears that this approach would be limited to strictly linear relationships between variables, but it turns out we can relax this as well.

2.2 Basis Function Regression

One method you can use to adapt linear regression to nonlinear relationships between variables is to transform the data according to basis functions.

The idea is to take the multidimensional linear model:

$$y = a_0 + a_1 x_1 + a_2 x_2 + a_3 x_3 + \cdots$$

and build the $x_1,\ x_2,\ x_3$ and so on, from our single-dimensional input x. That is, we let $x_n = f_n(x)$, where $f_n(\,)$ is some function that transforms our data.

For example, if $f_n(x) = x^n$, our model becomes a polynomial regression:

$$y = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots$$

Notice that this is *still a linear model* — the linearity refers to the fact that the coefficients a_n never multiply or divide each other. What we have effectively done is taken our one-dimensional x values and projected them into a higher dimension, so that a linear fit can fit more complicated relationships between x and y.

2.2.1 Polynomial basis functions

- This polynomial projection is useful enough that it is built into Scikit-Learn, using the PolynomialFeatures transformer:
- The transformer has converted the one-dimensional array into a three-dimensional array by taking the exponent of each value. This new, higher-dimensional data representation can then be plugged into a linear regression.

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])
Out[6]:
array([[ 2.,  4.,  8.],
       [ 3.,  9., 27.],
       [ 4., 16., 64.]])
```

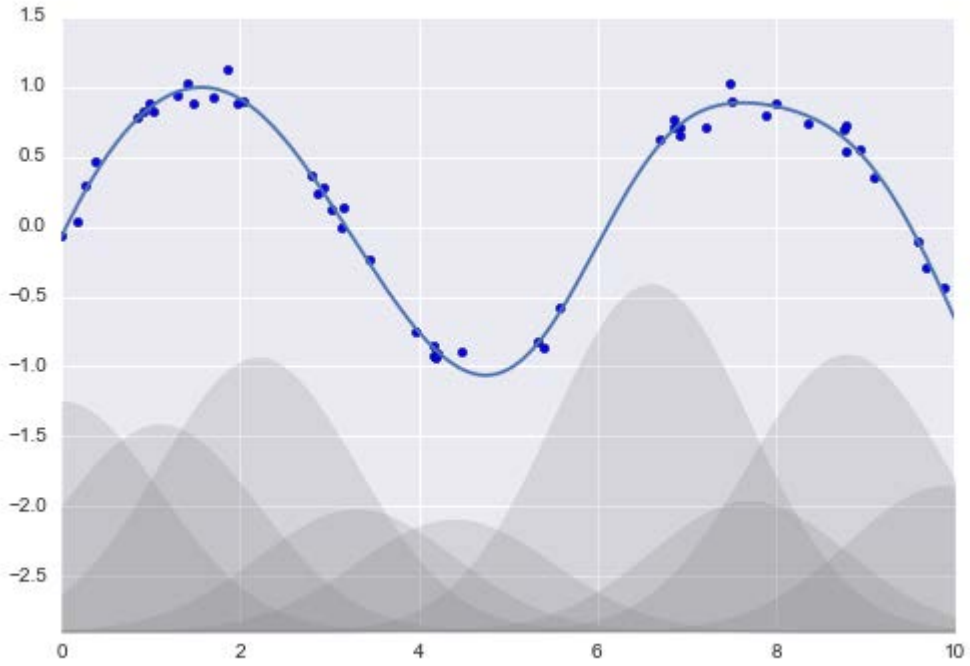
- The cleanest way to accomplish this is to use a pipeline; make a 7th-degree polynomial model in this way:
- With this transform in place, we can use the linear model to fit much more complicated relationships between x and y.

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

- For example, here is a sine wave with noise:

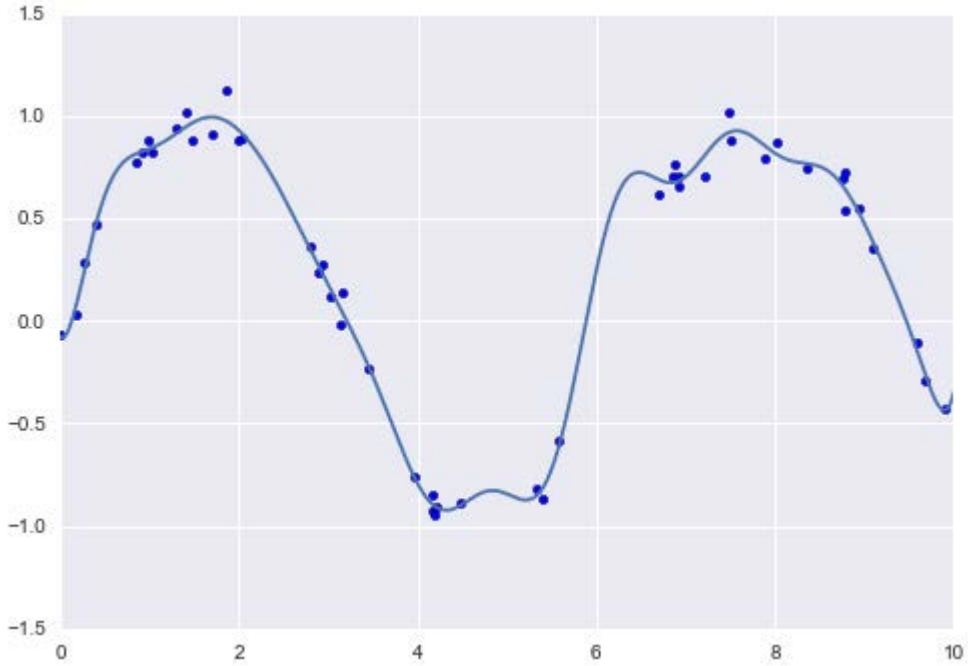
2.2.2 Gaussian basis functions

- 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 bases. The result might look something like the following figure:



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, as shown here and illustrated in the following figure (Scikit-Learn transformers are implemented as Python classes; reading Scikit-Learn's source is a good way to see how they can be created):

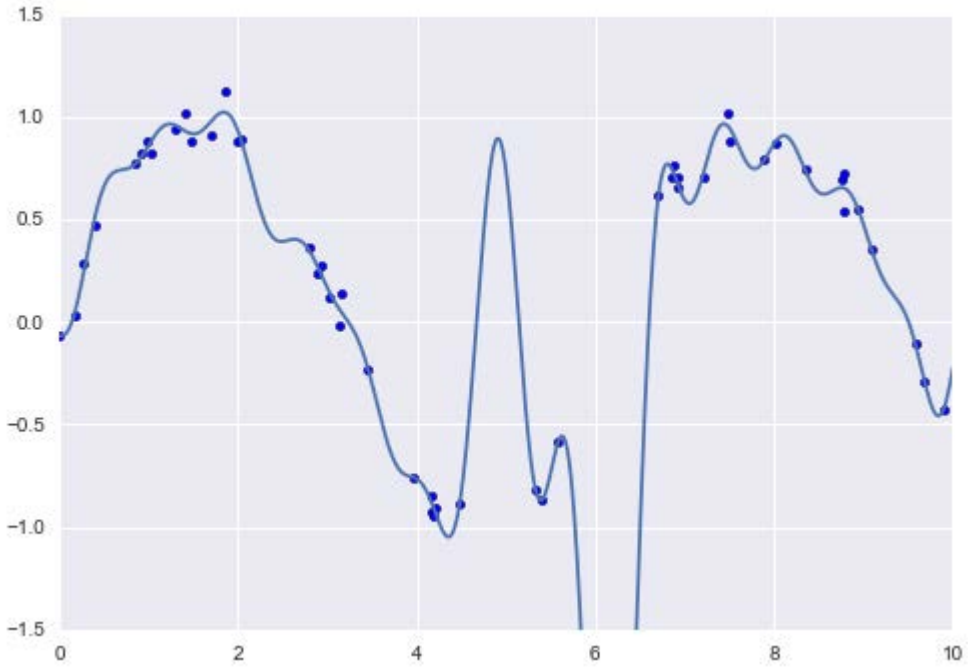


- We put this example here just to make clear that there is nothing magic about polynomial basis functions: if you have some sort of intuition into the generating process of your data that makes you think one basis or another might be appropriate, you can use them as well.

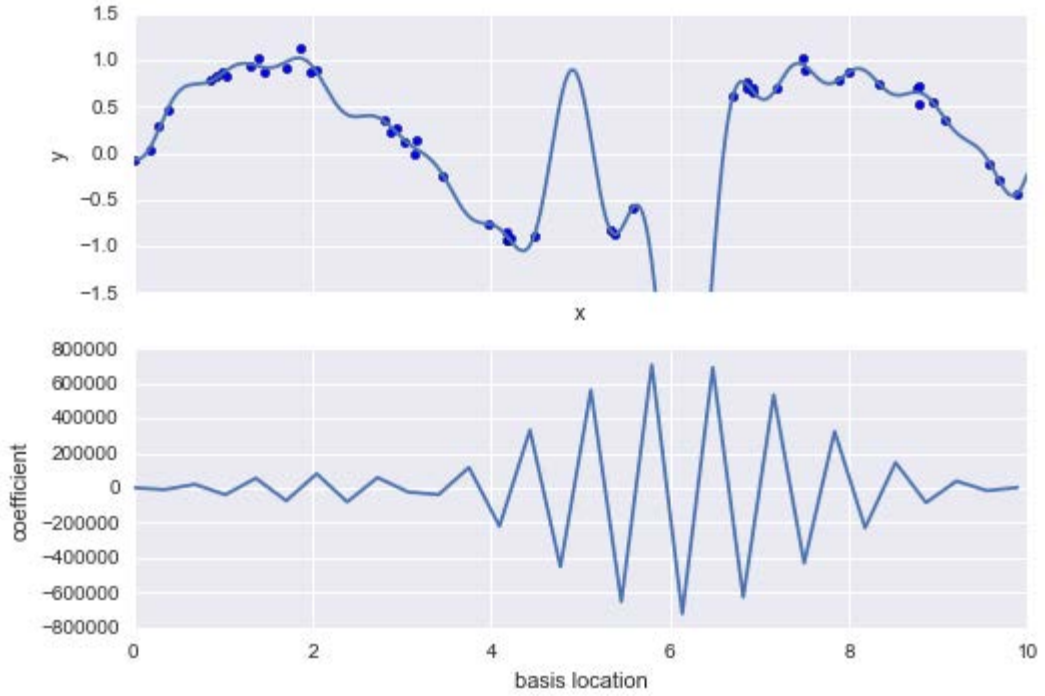
2.3 Regularization

- The introduction of basis functions into our linear regression makes the model much more flexible, but it also can very quickly lead to over-fitting (refer back to Hyperparameters and Model Validation for a discussion of this).

For example, if we choose too many Gaussian basis functions, we end up with results that don't look so good:



- With the data projected to the 30-dimensional basis, the model has far too much flexibility and goes to extreme values between locations where it is constrained by data.
- Plot the coefficients of the Gaussian bases with respect to their locations:



- The lower panel of this figure shows the amplitude of the basis function at each location. This is typical over-fitting behavior when basis functions overlap: the 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 comes in several forms.

2.3.1 Ridge regression (L_2 Regularization)

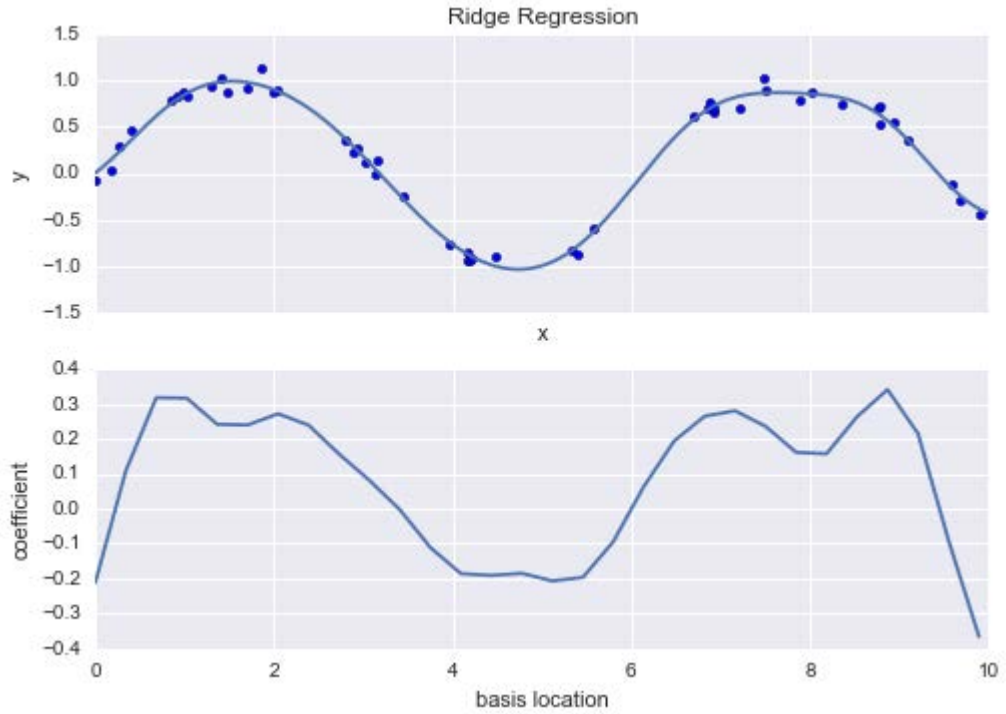
- Perhaps the most common form of regularization is known as *ridge regression* or L_2 *regularization*, sometimes also called *Tikhonov regularization*. This proceeds by penalizing the sum of squares (2-norms) of the model coefficients; in this case, the penalty on the model fit would be:

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

  where α is a free parameter that controls the strength of the penalty.

- This type of penalized model is built into Scikit-Learn with the Ridge estimator:

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



2.3.2 Lasso regression (L_1 regularization)

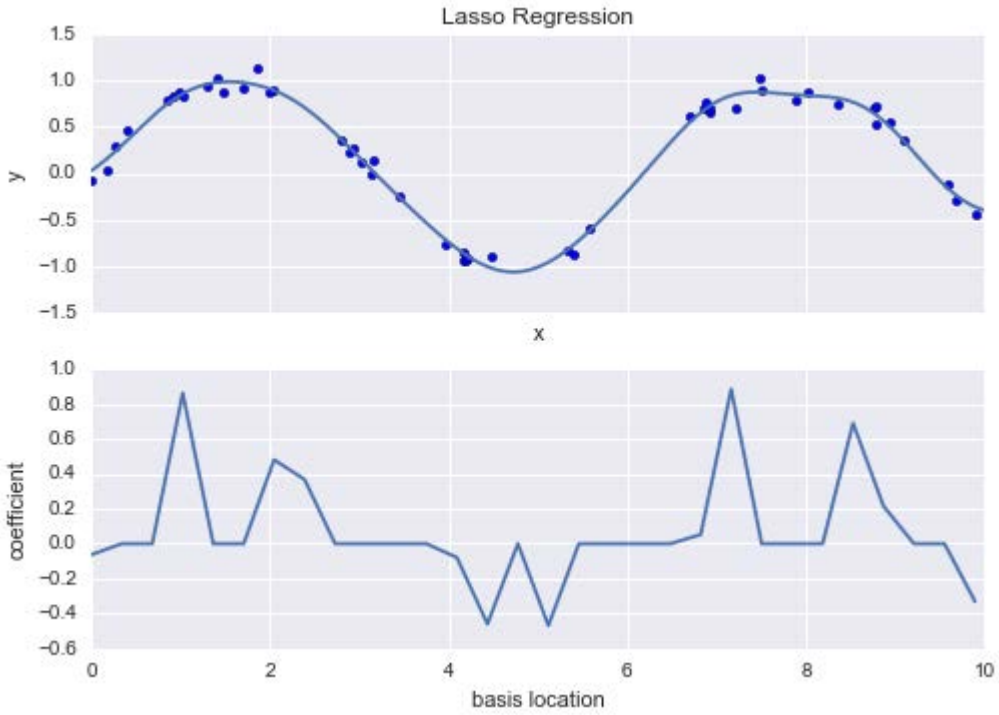
- Another very common type of regularization is known as lasso, and involves penalizing the sum of absolute values (1-norms) of regression coefficients:

$$P = \alpha \sum_{n=1}^{N} |\theta_n|$$

- Though this is conceptually very similar to ridge regression, the results can differ surprisingly: for example, due to geometric reasons lasso regression tends to favor sparse models where possible: that is, it preferentially sets model coefficients to exactly zero.

We can see this behavior in duplicating the ridge regression figure, but using L1-normalized coefficients:

- 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 (refer back to Hyperparameters and Model Validation for a discussion of this).



Example: Predicting Bicycle Traffic

- As an example, let's take a look at whether we can predict the number of bicycle trips across Seattle's Fremont Bridge based on weather, season, and other factors.
- The data is already in: Fremont Bridge Hourly Bicycle Counts by Month October 2012 to present and download here.
- In this example, we will join the bike data with another dataset, and try to determine the extent to which weather and seasonal factors—temperature, precipitation, and daylight hours—affect the volume of bicycle traffic through this corridor.
- NOAA makes available their daily weather station data.

- Easily use Pandas to join the two data sources.
- Perform a simple linear regression to relate weather and other information to bicycle counts, in order to estimate how a change in any one of these parameters affects the number of riders on a given day.
- In particular, this is an example of how the tools of Scikit-Learn can be used in a statistical modeling framework, in which the parameters of the model are assumed to have interpretable meaning.
- As discussed previously, this is not a standard approach within machine learning, but such interpretation is possible for some models.

1. Load the two datasets, indexing by date.
2. Compute the total daily bicycle traffic, and put this in its own dataframe; remove other columns.
3. The patterns of use generally vary from day to day; account for this (Monday to Sunday) in the data by adding binary columns that indicate the day of the week.
4. We might expect riders to behave differently on holidays; let's add an indicator of this as well.
5. We also might suspect that the hours of daylight would affect how many people ride; let's use the standard astronomical calculation to add this information:

6. We can also add the average temperature and total precipitation to the data. In addition to the inches of precipitation, let's add a flag that indicates whether a day is dry (has zero precipitation).
7. Finally, let's add a counter that increases from day 1, and measures how many years have passed. This will let us measure any observed annual increase or decrease in daily crossings:
8. Now our data is in order, and we can take a look at it:
9. With this in place, we can choose the columns to use, and fit a linear regression model to our data. We will set fit_intercept = False, because the daily flags essentially operate as their own day-specific intercepts:
10. Finally, we can compare the total and predicted bicycle traffic visually:

- It is evident that we have missed some key features, especially during the summer time. Either our features are not complete (i.e., people decide whether to ride to work based on more than just these) or there are some nonlinear relationships that we have failed to take into account (e.g., perhaps people ride less at both high and low temperatures).
- Nevertheless, our rough approximation is enough to give us some insights, and we can take a look at the coefficients of the linear model to estimate how much each feature contributes to the daily bicycle count:
  - These numbers are difficult to interpret without some measure of their uncertainty. We can compute these uncertainties quickly using bootstrap resamplings of the data:
- With these errors estimated, let's again look at the results:

The End!

Assignment:

1. Practice the example code I ran in this class.
2. Homework 2 will be assigned by the end of Sunday.