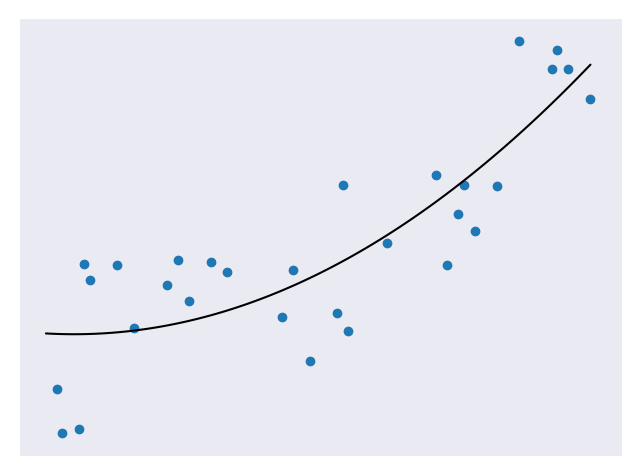
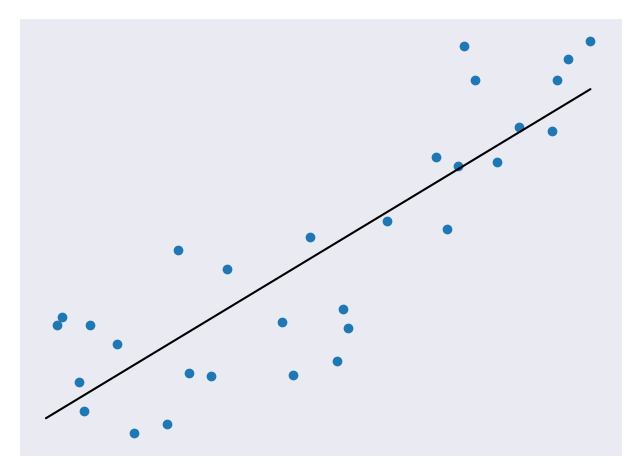
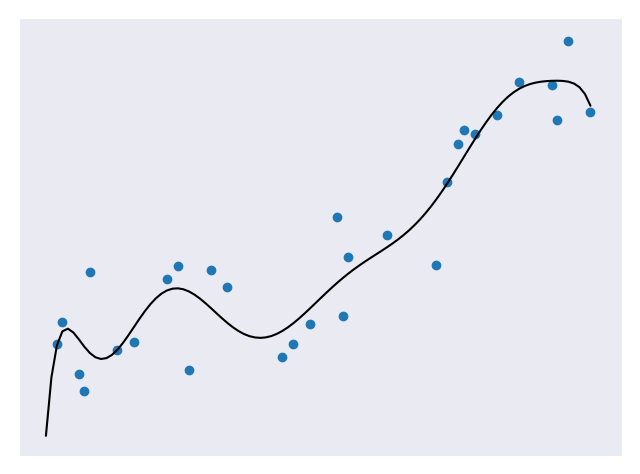
**Regularization**

Note: All code for data set creation and modeling is provided.

Consider the following scenario. You are making a peanut butter sandwich and are trying to adjust parameters so that it has the best taste. You might consider the type of bread, type of peanut butter, or peanut butter to bread ratio in your decision making process. But would you consider other factors like how warm it is in the room, what you had for breakfast, or what color socks you’re wearing? Probably not as these things don’t have as much impact on the taste of your sandwich. You would focus more on the first few features for whatever model you end up developing and avoid paying too much attention to the other ones.

In previous modules, we have seen prediction models trained on some sample set and scored against how close they are to a test set. We obviously want our models to make predictions that are accurate but can they be too accurate? When we look at a set of data, there are two main components: the underlying pattern and noise. We only want to match the pattern and not the noise. Consider the figures below that represent quadratic data.

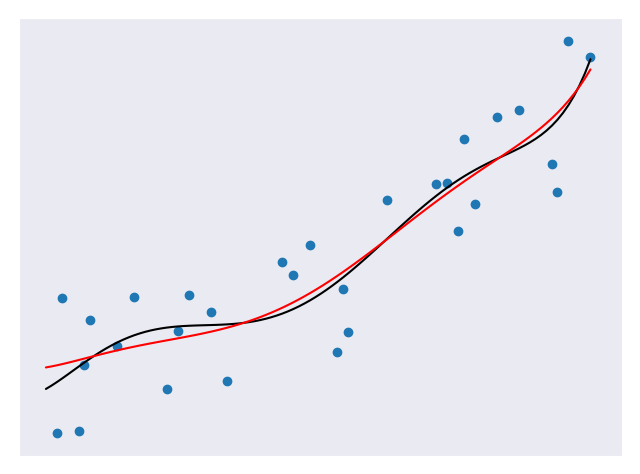
The first model underfits the data, the second model looks to be a good fit for the data, and the third model is a very close fit for the data. Of all the models above, the third is likely the most accurate against the test set. But this isn’t necessarily a good thing. If we add in some more test points, we’ll likely find that the third model is no longer as accurate at predicting them but the second model is still pretty good. This is because the third model suffers from overfitting. Overfitting means it does a really good job at fitting the test data (including noise) but is bad at generalizing to new data. The second model is a nice fit for the data and is not so complex that it won’t generalize.

The goal of **regularization** is to avoid overfitting by penalizing more complex models. The general form of regularization involves adding an extra term to our cost function. So if we were using a cost function CF, regularization might lead us to change it to CF + λ \* R where R is some function of our weights and λ is a **tuning parameter**. The result is that models with higher weights (more complex) get penalized more. The tuning parameter basically lets us adjust the regularization to get better results by changing its value. The higher the λ the less impact the weights have on the total cost. Below we’ll cover some methods of regularization and when they are good to use. When looking through the code, note the import statements used for each model.

**Ridge Regression**

**Ridge regression** is a type of regularization where the function R involves summing the squares of our weights. In statistics, this would be called an L2 norm.

The equation above is an example of the regularization with w representing our weights. Ridge regression forces weights to approach zero but will never cause them to be zero. This means that all the features will be represented in our model but overfitting will be minimized. Ridge regression is a good choice when we don’t have a very large number of features and just want to avoid overfitting.



In the figure above, the black line represents a model without Ridge regression applied and the red line represents a model with Ridge regression applied. Note how much smoother the red line is. It will probably do a better job against future data.

In the included file, regularization\_ridge.py, the code that adds ridge regression is:

regModel = Pipeline([('poly', PolynomialFeatures(degree=6)), \

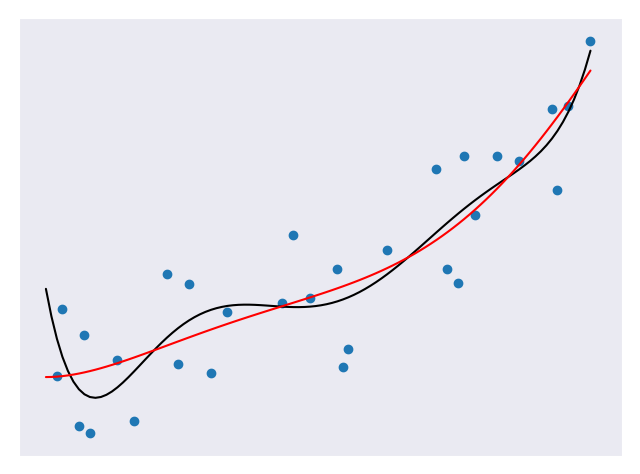
('ridge', Ridge(alpha=5.0))])

Adding the Ridge regression is as simple as adding an additional argument to our Pipeline call. Here, the parameter alpha represents our tuning variable. For additional information on Ridge regression in scikit-learn, consult https://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.Ridge.html.

**Lasso Regression**

**Lasso regression** is a type of regularization where the function R involves summing the absolute values of our weights. In statistics, this would be called an L1 norm.

The equation above is an example of the regularization with w representing our weights. Notice how similar ridge regression and lasso regression are. The only noticeable difference is that square on the weights. This happens to have a big impact on what they do. Unlike ridge regression, lasso regression can force weights to be zero. This means that our resulting model may not even consider some of the features! In the case we have a million features where only a small amount are important, this is an incredibly useful result. Lasso regression lets us avoid overfitting and focus on a small subset of all our features. In the original scenario, we would end up ignoring those factors that don’t have as much impact on our sandwich eating experience.



In the figure above, the black line represents a model without Lasso regression applied and the red line represents a model with Lasso regression applied. The red line is much smoother than the black line. The Lasso regression was applied to a model of degree 6 but the result looks like its degree 2! The Lasso model will probably do a better job against future data.

In the included file, regularization\_lasso.py, the code that adds Lasso regression is:

regModel = Pipeline([('poly', PolynomialFeatures(degree=6)), \

('lasso', Lasso(alpha=0.1, max\_iter=100000))])

Adding the Lasso regression is as simple as adding the Ridge regression. Here, the parameter alpha represents our tuning variable and max\_iter represents the max number of iterations to run for. For additional information on Lasso regression in scikit-learn, consult https://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.Lasso.html.

**Summary**

With regularization, we have found a good way to avoid overfitting our data. We also have some methods of regularization for different situations. Some of you may be wondering how to choose that tuning parameter to get the best results. That will be covered in another section.