16ML teaching note: Polynomial regression

This teaching note assumes that the student will have knowledge of linear regression and the kernel trick since these units are taught prior to this unit in weeks 1 and 4 respectively. We give a brief conceptual overview specific to polynomial kernel but do not go into depth with the mathematical mechanics. While units on cross-validation and bias-variance are taught prior to this course in week 2, we give a more in-depth conceptual review of these topics since our coding assignment heavily relies on students' understanding of them. Again, we do not go into the mathematical explanation in-depth and expect students to have gotten that understanding in the corresponding units.

1. **When to Use Polynomial over Linear Regression**

   As you have seen in previous assignments, linear regression is an easy way to generate a boundary when you have linearly separable data since we are modeling a linear combination of our raw features i.e. if our labels are y and inputs are x, we have $y = \Sigma_{i=1,\ldots,n} w_i x_i$. However, what do you do when the data isn't linearly separable? This is where polynomial regression can help. In the example below we see that the data in plot A can be classified using a linear boundary generated by linear regression. However, if we have circular data like in plot B, we can no longer use linear regression to accurately classify the data. Instead, we would like to generate a boundary that's a circle. Since the general formula for a circle is $x^2 + y^2 = r^2$, we could create such a boundary if we "lift" our features to a second-degree polynomial.

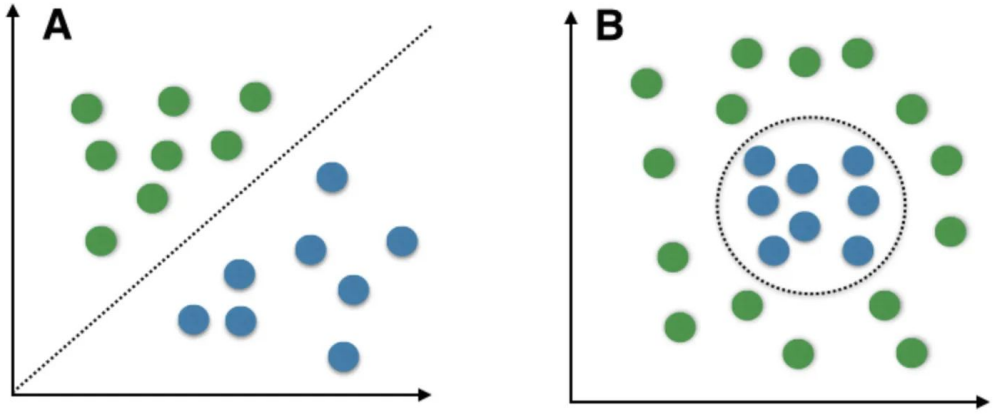


Figure 1: Comparing linear and polynomial (circular) boundaries

2. **What is "lifting"?**

   "Lifting" means taking our raw data and applying a function to them to get a different set of features. It helps us "lift" our data to a higher dimensional feature space. This new set of features is what we will feed into our regression model. An example is

shown in Figure 2. Suppose we have raw data x and y (shown on the left of Figure 2), and the underlying relation between them is a second-degree polynomial, $y = x^2 + 1$. Now suppose we know that the relation between x and y is a second-degree polynomial, and we want to determine what the polynomial is exactly. What we could do to have a generalized second-degree polynomial as a model: $y = ax^2 + bx + c$. Since y can be seen as a linear combination of x^2, x, and 1, we could compute x^2, x, and 1 from our initial data x (we would only need to compute x^2 in this case) and then run linear regression with x^2, x, and 1 being our features and y being our output. In our example, after running regression, the model that we find is $y = x^2 + 1$.

x	y
2	5
5	26

1	x	x^2	y
1	2	4	5
1	5	25	26

Figure 2: An example of "lifting"

3. **How to do polynomial regression via "lifting:"**

   Here is the algorithm to do polynomial regression via lifting features:

   (1) Pick a degree d for the polynomial you want to regress on

   (2) Lift raw features (X) to d-degree polynomial features. We will denote this matrix of features as **Φ**

      (a) You need to create all monomials from the raw features i.e. all the powers and cross-terms

   (3) Run OLS on these new features **Φ**

   Recall for linear regression, we want to solve y = w.T * X and the closed form OLS solution is w = (X.T @ X) ^ (-1) @ X.T @ y. For polynomial, we use our lifted features **Φ** instead of the raw data X

4. **Underfitting and Overfitting:**

   In our example of lifting, we supposed that we know y is a second-degree polynomial of x. Now what if we don't know the degree of the polynomial? How do we pick the degree of the polynomial? Figure 3 below shows an example where it might not be obvious what degree polynomial we should use.

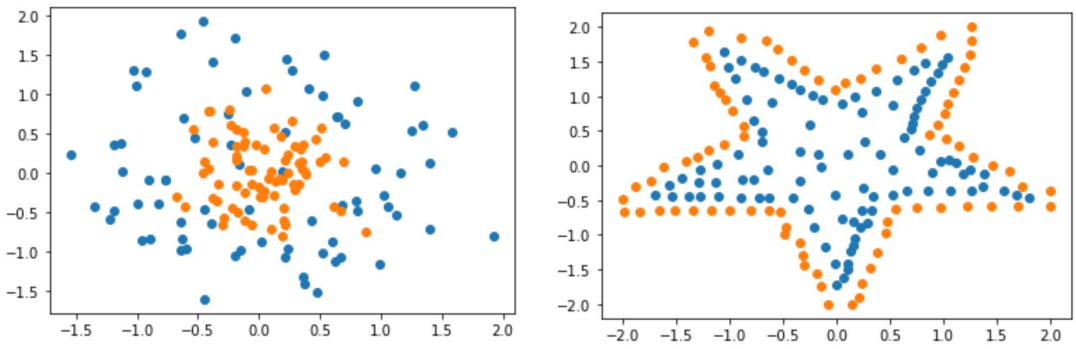


Figure 3: Left plot has obvious 2-degree polynomial boundary, left plot not so much

Would the number of polynomial features we use affect how well we can fit the training data? Will it affect how well our prediction is going to be? The answer is yes to both. If we have too few features, we would not be able to fully capture the underlying pattern. For example, linear functions can be thought of as first-degree polynomials. If we only use linear features to fit a second-degree polynomial, the error will be significant because linear features are fundamentally not enough to describe a parabolic function. Such a phenomenon is called underfitting. Examples of underfitting is shown in Figure 4.

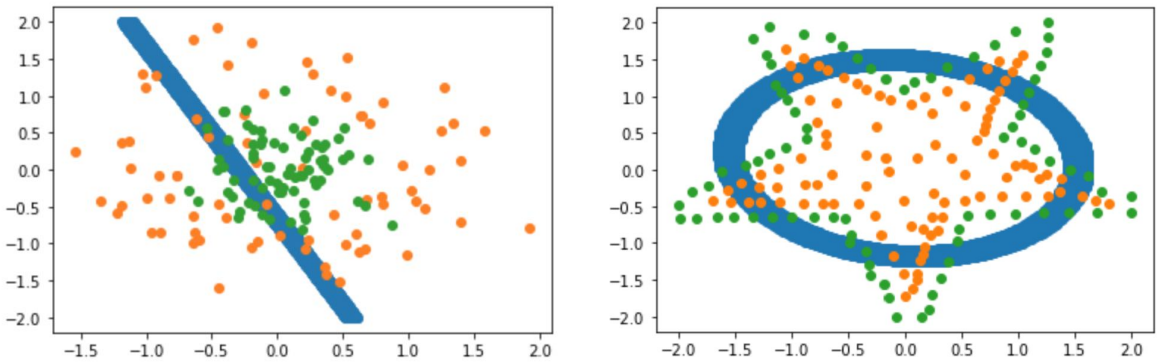


Figure #4: Underfit Graphs

However, If we have too many features, regression would try to fit the training data exactly even when there is noise since it has now more freedom to do so. Consequently, the model we learned may not generalize well with the test data, resulting in high test error even though the training error is small. Examples of overfitting is shown in Figure 5.

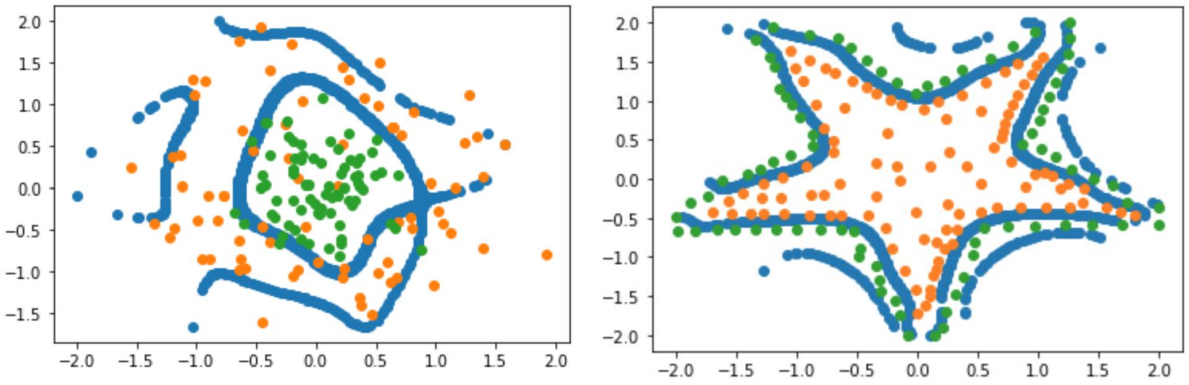


Figure #5: Overfit Graphs

Therefore, in our context of polynomial regression, if we decrease the degree of the polynomial by too much, our model would underfit and not be able generate a good fit for both the training and the test data. Conversely, increasing the degree of the polynomial by too much would make our model overfit. Since overfitting results from fitting the training data too exactly, the training error would be small. However, since the learned model is too specific to the training data, the test error would be large. However, if we pick the polynomial degree just right, we can get a good approximation for the boundary. Figure 6 shows examples with no underfitting or overfitting.

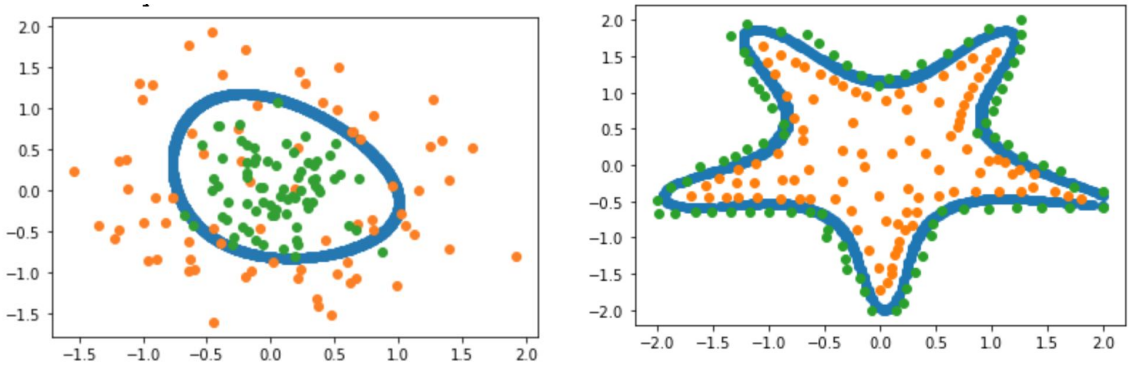


Figure #6: Graphs Without Underfitting or Overfitting

5. Hyperparameter tuning:

Now back to the question of "how do we pick the perfect degree polynomial?" We use a process called "validation" to solve this:

1. Split data into train, validation, and test sets. Set the test set aside.
2. Choose a range of hyperparameters to validate.
3. While in range of hyperparameters:
    a. Pick a hyperparameter and train model using train set
    b. Predict using validation set and calculate validation error

4. Compare validation errors across all hyperparameters tested. Pick hyperparameter with smallest error
5. Train model using train+validation set using picked hyperparameter
6. Predict using test set and calculate error to evaluate model

You might ask "how do we split the train, validation, and test sets?" There's a general rule of thumb that you should split 20% of your data for the test set, 80% for training and validation. From that 80%, 70% for training and 30% for validation. You should also randomly sample your data for each set so that they are coming from the same distribution. For example, if your data is ordered in any way, you don't want to just split it from there; you want to randomly shuffle it before splitting it.

Instead of splitting the combined train data into 70/30, another popular way is to use k-fold cross-validation. The algorithm is described below:

(1) Given your combined train set (e.g. the 80% of your total data mentioned above), split it into k equal sets
(2) For k iterations:
    (a) Leave one set out and combine the other k - 1 sets
    (b) Train your model with the chosen hyperparameter on the combined k - 1 sets
    (c) Validate your model using the one set that you left out
    (d) Calculate the validation error
(3) Average the k validation errors you calculated. This will be your validation error for the chosen hyperparameter

Using k-fold cross validation helps prevent concerns with the possibility that you randomly split the data to favor certain hyperparameters over others. By averaging over many validation errors, we are getting an average performance for the hyperparameter.

You might ask "why do we need to use a validation error?" The reason is the error on the training set will always go down as you add more features to the model. This is because adding more features gives us more freedom and flexibility to model to the training points. If we add infinite features, we could potentially model each training point perfectly (i.e. overfitting to the training data).

However, adding more features makes the model inflexible to handling "new" points and this is why we see validation error goes up. We care about how our model performs on new data and so we want to look at the validation error (and eventually test error) when we are evaluating the performance of our model. Hence when hyperparameter tuning, we want to be at the sweet spot where validation error is lowest. We will talk about this "inflexibility" concept in more detail in the next section.

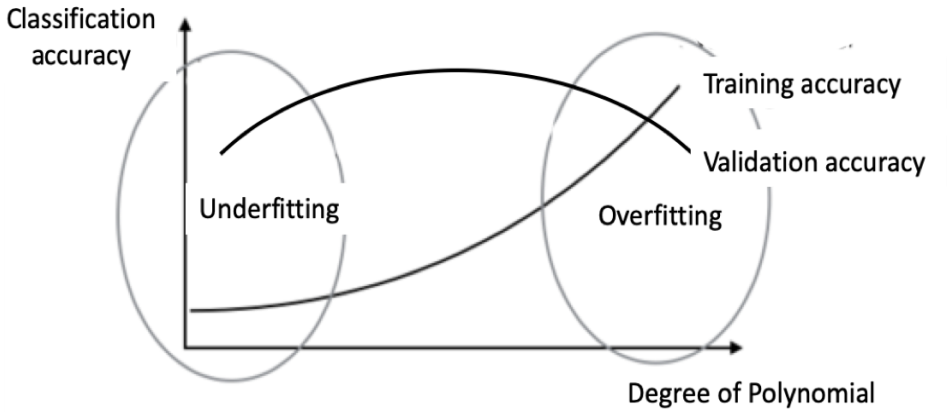


Figure #7: Graph showing relationship between training and validation error

6. Bias-Variance Tradeoff:

In this section we delve a little deeper into the theory behind underfitting and overfitting as well as this "inflexibility" when we add more features. To do that we will now introduce two new terms: bias and variance. Bias, in a loose sense, is error caused by wrong assumptions in our learning algorithm. Variance, on the other hand, is error caused by the algorithm's sensitivity to noise.

If our learning algorithm has high bias, it means that the assumptions we are making in our learning algorithm does not capture the true pattern of the data. For example, if we assume that y is linear to x and run linear regression when y is actually a second-degree polynomial of x, the error that we would get is categorized as bias. Therefore, high-bias implies underfitting. Conversely, if our learning algorithm has high variance, it means that little fluctuation in the data would introduce great error, which is contingent on the overfitting case we discussed above. More specifically, the reason that overfitting introduces low training error and high test error is that the test data is not the same as the training data. In the case of polynomial regression, since an overfitting model fits the training data exactly, little difference between the test and the training data will lead to large error. In other words, an overfitting model is sensitive to noise, and has high variance.

There will always be a bias-variance tradeoff when trying to fit models. In practice, we want a variance small enough to make our predicted pattern rigorous, but we also want a bias small enough to make sure that the predicted pattern does not oversimplify the underlying pattern. This is where hyperparameter tuning such as cross-validation and k-fold come in; it helps us achieve the best tradeoff between bias and variance for our hyperparameter. In summary, variance can be viewed as inversely proportional to bias. Finding the perfect balance is when test error is at its lowest, and when variance and bias intersect, as can be seen in Figure 8.

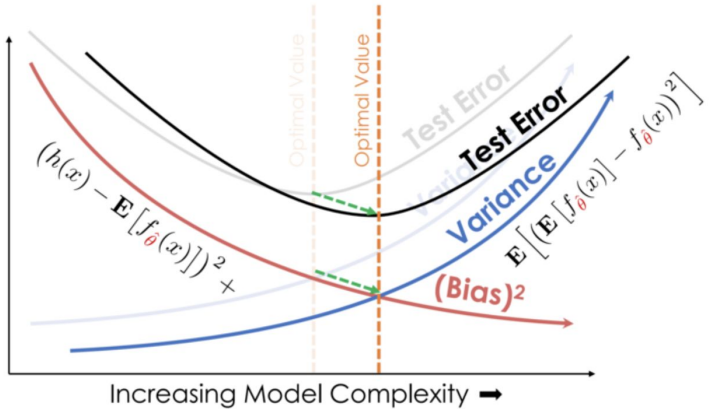


Figure #8: Graph showing relationship between variance, bias, and test error

7. Using the Kernel Trick to Potentially Improve Computing Speed.

It might seem to us now that polynomial regression is a great method. However, it does have its drawbacks; one of which is the amount of computations needed to perform it when the degree of the polynomial gets large. Let's take a simple third-degree polynomial, $(x+y)^3$ as an example. To perform polynomial regression, we need to first expand this expression to find out what are the needed features. The expanded expression is $x^3+3x^2y+3xy^2+y^3$. This would mean that for each data point, we need to lift it to x^3, x^2y, xy^2, and y^3. However, calculating each of the features takes three multiplications, and the number only grows with higher dimensional data points and higher degrees of polynomials.

However, we can also make an important observation point here: if we can calculate $(x+y)^3$ directly and use its result for regression, only one addition and three multiplication are needed. This is called the kernel trick. The kernel trick allows us to use the original feature space instead of having to compute the coordinates of the data in a higher dimensional space. This is crucial to increasing computation speed because by computing the inner products between the images of all pairs of data in the feature space, it stays in the lower-dimensional space and thus reduces computation power.

The polynomial kernel is given by the formula

$$K(x,y) = (x^T y + c)^d = (\Sigma_{i=1,\ldots,n} x_i y_i + c)^d$$

where x and y are raw data points (vectors of raw features) and c is a free parameter which weights the influence of higher degree monomials over lower degree monomials. d is the degree of the polynomial.

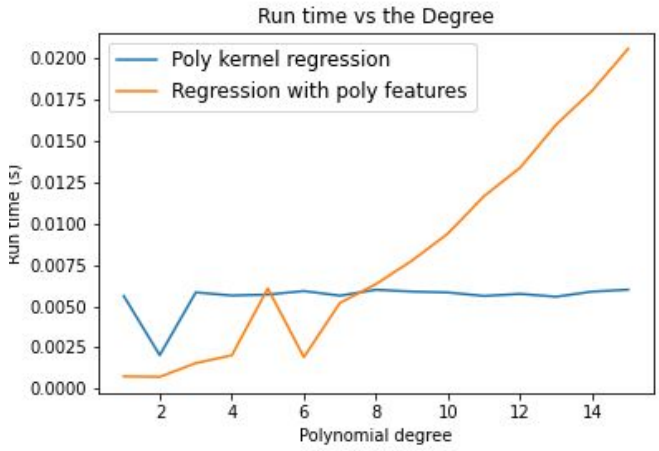


Figure #9: Graph showing runtime using polynomial features versus polynomial kernel

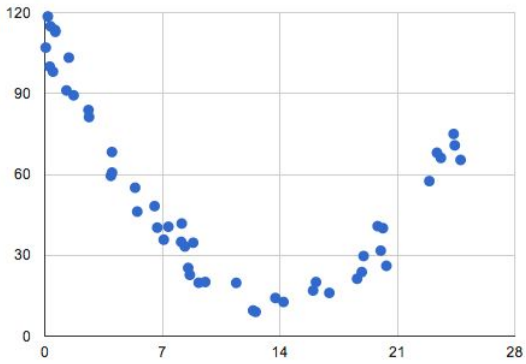
While the kernel trick will help when we have a lot of features, it doesn't necessarily help when we have few features as you can see in Figure #9 above. This is because calculating the gram matrix K(x,y) has a fixed cost dependent on the size of our dataset. Hence there is a tradeoff between lifting our raw data to a set of polynomial features and taking the inner product between each of the raw data points.

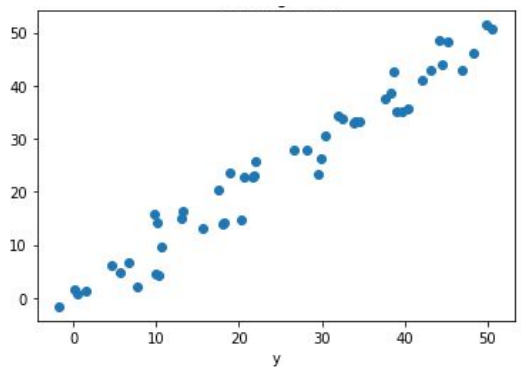
Quiz questions:

#1 True or false: The algorithm for linear regression and polynomial regression is fundamentally different.

Answer: False. Both algorithms use regression. The only difference is polynomial regression uses lifting to create polynomial features before using regression.

#2 In the previous slide we talked about the classification setting but the same holds true for the regression setting. Consider the two plots below:





Would you use linear regression or polynomial regression for the dataset on the left? How about the dataset on the right?

Answer: left: polynomial regression; right: linear regression

Sources for the graphics: https://www.statisticshowto.com/quadratic-regression/, https://www.geeksforgeeks.org/linear-regression-using-tensorflow/

#3 How should you split your data for hyperparameter tuning?

Answer: Three sets of data: train, validation, and test

#4 Which sets of data should you train your data when it comes test time?

Answer: Use both train and validation sets

#5 Write down the steps on how to do the validation process.

Answer:

7. Split data into train, validation, test sets. Set test set aside.
8. Choose a range of hyperparameters to validate.
9. While in range of hyperparameters:
    a. Pick a hyperparameter and train model using train set
    b. Predict using validation set and calculate validation error
10. Compare validation errors across all hyperparameters tested. Pick hyperparameter with smallest error
11. Train model using train+validation set using picked hyperparameter
12. Predict using test set and calculate error to evaluate model

#6 You trained a classifier to recognize hand-written digits. When training your model, you noticed that the training accuracy is very high. However, upon validation, you noticed that your model has much lower accuracy on the validation set. Please answer the following question:

(1) Which of the following is true?
    (a) Your model is underfitting
    (b) Your model is overfitting
(2) Utilizing the validation set, what should you do to address the issue above?

Answer: (1) (b); (2) hyperparameter tuning.

#7 If we have data with small enough noise, a high variance model will behave as if it were a low variance model.

Answer: False. Whether a model is high or low variance is determined by its hyperparameters and number of features, not the amount of noise in the data. Therefore, without changing the value of the hyperparameters/number of features in the model, a high variance model would still be high variance and prone to overfit.

#8 What is the meaning of bias and variance in a machine learning model?

Answer: The bias error is an error from erroneous assumptions in the learning algorithm. The variance is an error from sensitivity to small fluctuations in the training set.

#9 When doing hyperparameter tuning, if we increase the number of features for a polynomial function, what is the behavior of bias and variance?

Answer: The bias will decrease, the variance will increase.

#10 Assume that we have the number of features that achieves the highest test accuracy, what will happen to the training accuracy if we keep increasing the number of features?

Answer: The training accuracy will keep increasing.

#11 When using the kernel trick, can one get non-linear decision boundaries? Try to argue why.

Answer: True. Because the result of some kernel functions is effectively equal to the sum of each term in a higher degree polynomial.

#12 Comparing using polynomial features versus a polynomial kernel for polynomial regression, which method would be computationally faster when the number of features is a lot larger than

the number of data points?

Answer: Polynomial kernel

#13 True or false: it is always better to use the kernel trick when possible

Answer: False. In the context of polynomial regression, if the degree of the polynomial is small, it might be more expensive to compute to the kernel matrix.