Python for Machine Learning and Data Analysis

Dr. Handan Liu

h.liu@northeastern.edu

Northeastern University

Spring 2019

Machine Learning 02

Lecture 8

Content

- Section 1: Hyperparameters and Model Validation

- Section 2: Feature Engineering

Section 1: Hyperparameters and Model Validation

- Basic 4 steps for applying a supervised machine learning model:
  - Choose a class of model
  - Choose model hyperparameters
  - Fit the model to the training data
  - Use the model to predict labels for new data
- In order to make an informed choice, we need a way to validate that our model and our hyperparameters are a good fit to the data.
- While this may sound simple, there are some pitfalls that you must avoid to do this effectively.

1. Thinking about Model Validation

- In principle, model validation is very simple:
  - after choosing a model and its hyperparameters, estimate how effective it is by applying it to some of the training data and comparing the prediction to the known value.
- The following sections first show a naive approach to model validation and why it fails;
- Then exploring the use of holdout sets and cross-validation for more robust model evaluation.

1.1 Model validation the wrong way

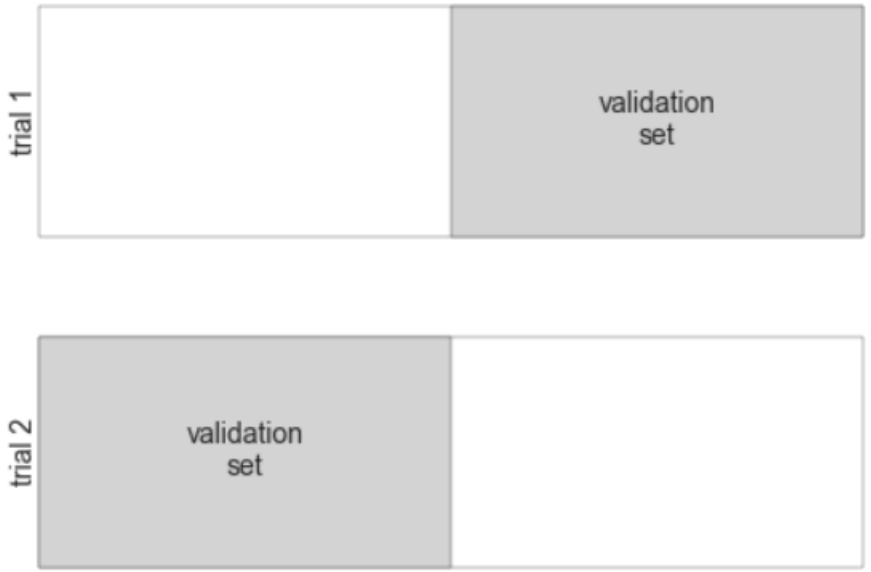
- Use the regular approach to choose a model, hyperparameters; train and predict; finally compute the fraction of the correctly labeled points.
- In this case, the accuracy score is 1.0. → Is this truly measuring the expected accuracy? Have we really come upon a model that we expect to be correct 100% of the time?
- The answer is NO.
  - The model used here is the *k-neighbors* classifier with *n_neighbors=1*, which says "the label of an unknown point is the same as the label of its closest training point".
  - It contains a fundamental flaw : *it trains and evaluates the model on the same data*. Furthermore, the nearest neighbor model is an *instance-based* estimator that simply stores the training data, and predicts labels by comparing new data to these stored points: except in contrived cases, it will get 100% accuracy *every time!*

1.2 Model validation the right way: Holdout sets

- Hold back some subset of the data from the training of the model, and then
- Use this *holdout set* to check the model performance.
- This splitting can be done using the train_test_split utility in Scikit-Learn:
  - A more reasonable result is about 0.9: the nearest-neighbor classifier is about 90% accurate on this hold-out set.
- The hold-out set is similar to unknown data, because the model has not "seen" it before.
- One disadvantage of using a holdout set for model validation is that we have lost a portion of our data to the model training.
  - This is not optimal, and can cause problems – especially if the initial set of training data is small.

1.3 Model validation via cross-validation

- Use *cross-validation*: to do a sequence of fits where each subset of the data is used both as a training set and as a validation set.
- Here we do two validation trials, alternately using each half of the data as a holdout set. Using the split data from before, we could implement it like this:



- What comes out are two accuracy scores, which could be combined (taking the mean) to get a better measure of the global model performance.
- This particular form of cross-validation is a *two-fold cross-validation*

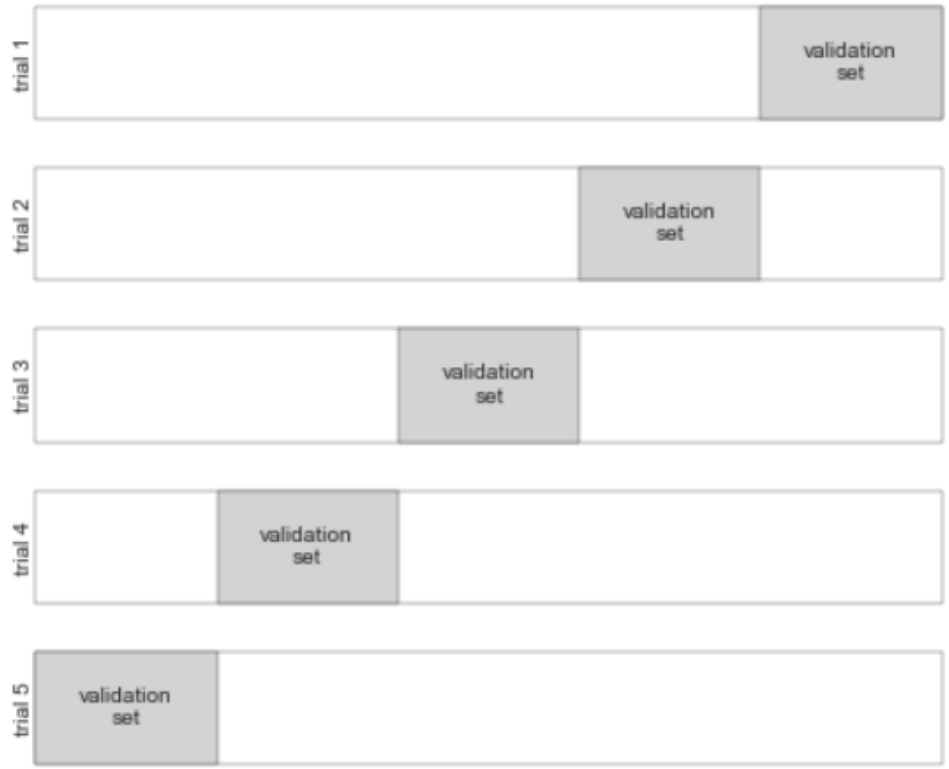
  —that is, one in which we have split the data into two sets and used each in turn as a validation set.

- We could expand on this idea to use even more trials, and more folds in the data—for example, here is a visual depiction of five-fold cross-validation:

- Here we split the data into five groups, and use each of them in turn to evaluate the model fit on the other 4/5 of the data.

- This would be rather tedious to do by hand, and so we can use Scikit-Learn's cross_val_score convenient routine to do it succinctly:

```
from sklearn.model_selection import cross_val_score
cross_val_score(model, X, y, cv=5)
```



- Scikit-Learn implements a number of useful cross-validation schemes that are useful in particular situations;
- these are implemented via iterators in the cross_validation module. For details, see cross-validation: evaluating estimator performance
- For example, a type of cross-validation is known as leave-one-out cross validation:
  - go to the extreme case in which the number of folds is equal to the number of data points: that is, we train on all points but one in each trial.
  - Because of having 150 samples, the LeaveOneOut cross-validation yields scores for 150 trials, and the score indicates either successful (1.0) or unsuccessful (0.0) prediction. Taking the mean of these gives an estimate of the error rate:

2. Selecting the Best Model

- The model selection and selection of hyperparameters are some of the most important aspects of the practice of machine learning.

- IMPORTANCE: : *if our estimator is underperforming, how should we move forward?* There are several possible answers:
  - Use a more complicated/more flexible model **will give worse results**
  - Use a less complicated/less flexible model
  - Gather more training samples **may not improve your results**
  - Gather more data to add features to each sample
- The answer to this question is often counter-intuitive.

2. Selecting the Best Mode

I find that this information is often glossed over in introductory machine learning tutorials.

- The model selection and selecti… of the most important aspects of th… learning.

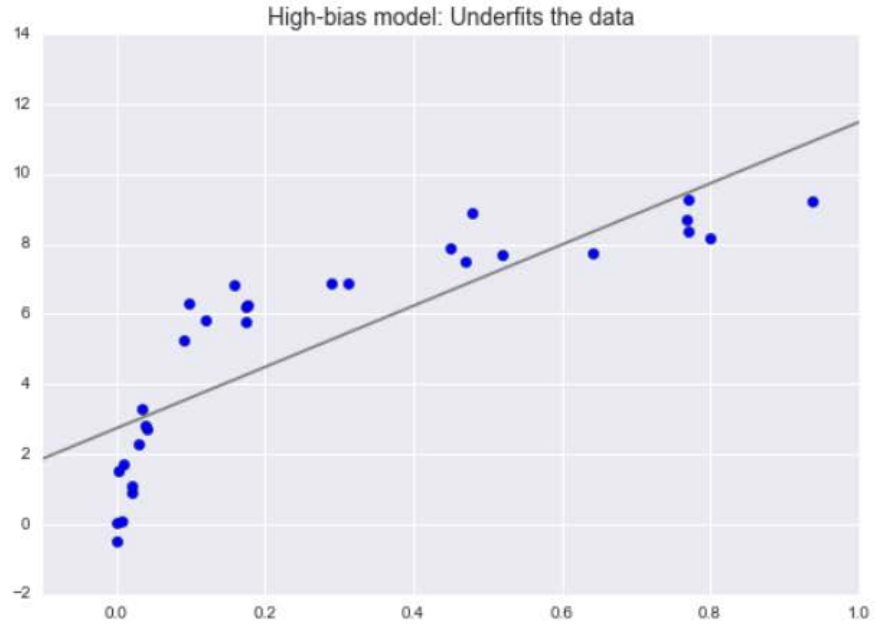
- IMPORTANCE: : *if our estimator is u… should we move forward?* There are s…
  - Use a more complicated… esults
  - Use a less complicated/le…
  - Gather more training samp…
  - Gather more data to add features to ch… t improve your results

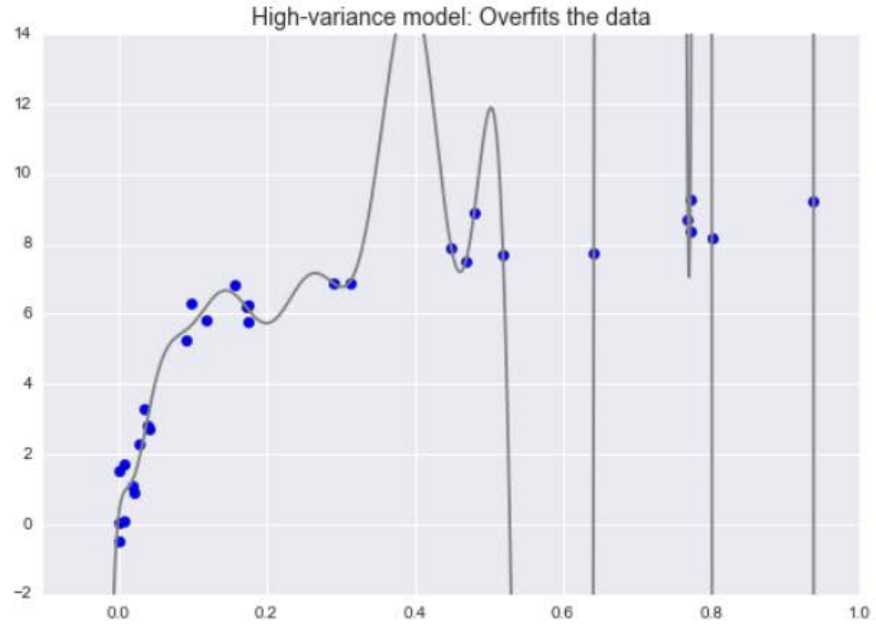
The ability to determine what steps will improve your model is what separates the successful machine learning practitioners from the unsuccessful.

- The answer to this question is often counter-intuitive.

2.1 The Bias-variance trade-off

- Fundamentally, the question of "the best model" is about finding a sweet spot in the tradeoff between *bias* and *variance*. Consider the following figure, which presents two regression fits to the same dataset:

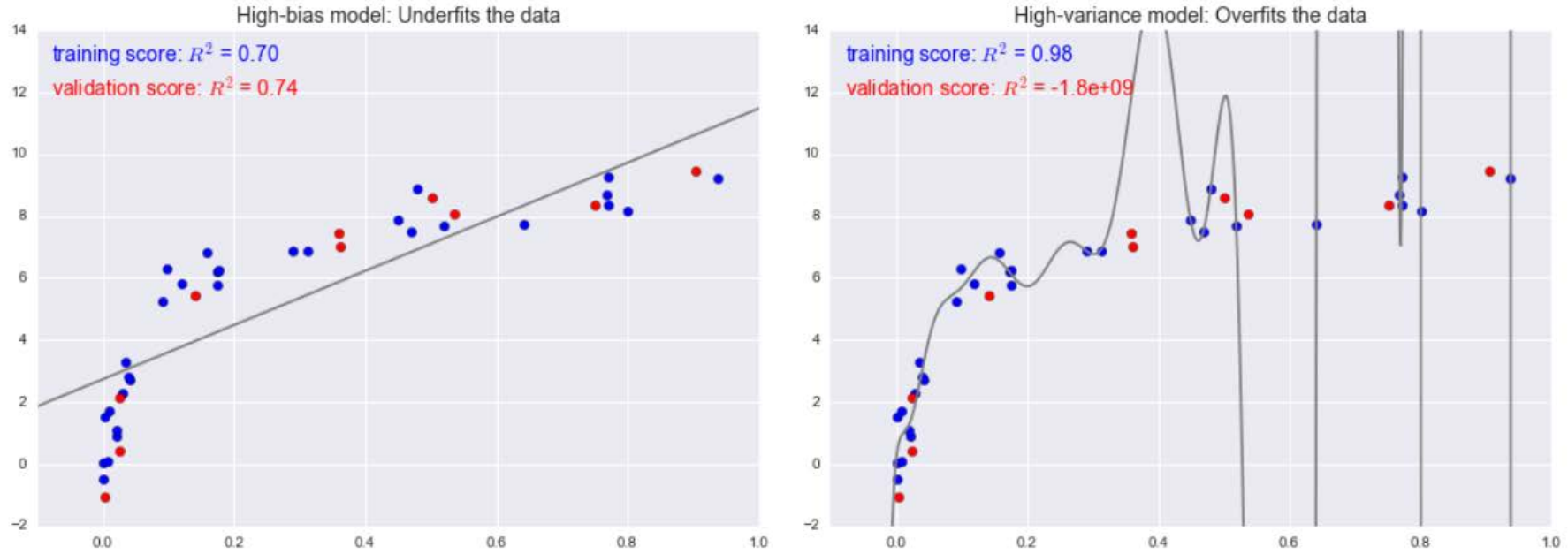




It is clear that neither of these models is a particularly good fit to the data, but they fail in different ways.

- The model on the left attempts to find a straight-line fit through the data. Because the data are intrinsically more complicated than a straight line, the straight-line model will never be able to describe this dataset well. Such a model is said to *underfit* the data: that is, it does not have enough model flexibility to suitably account for all the features in the data; another way of saying this is that the model has high *bias*.
- The model on the right attempts to fit a high-order polynomial through the data. Here the model fit has enough flexibility to nearly perfectly account for the fine features in the data, but even though it very accurately describes the training data, its precise form seems to be more reflective of the particular noise properties of the data rather than the intrinsic properties of whatever process generated that data. Such a model is said to *overfit* the data: that is, it has so much model flexibility that the model ends up accounting for random errors as well as the underlying data distribution; another way of saying this is that the model has high *variance*.

To look at this in another light, consider what happens if we use these two models to predict the y-value for some new data. In the following diagrams, the red/lighter points indicate data that is omitted from the training set:



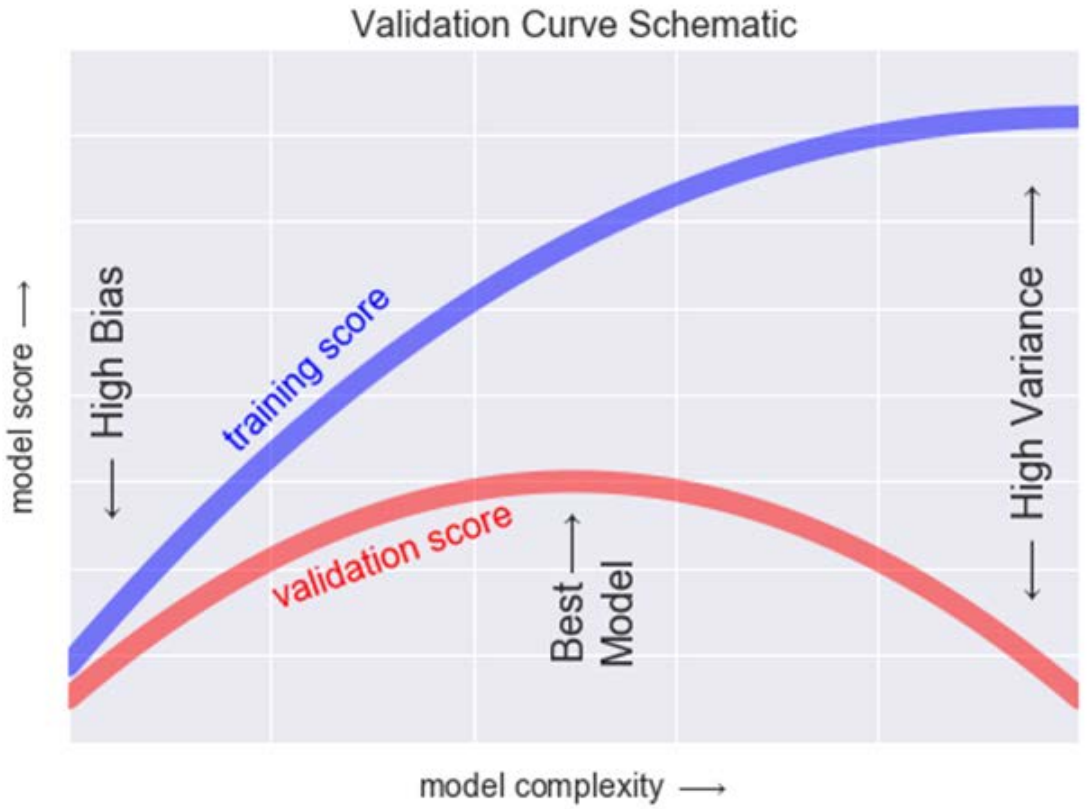
The score here is the ***R²*** score, or coefficient of determination, which measures how well a model performs relative to a simple mean of the target values. ***R²*** =1 indicates a perfect match, ***R²*** =0 indicates the model does no better than simply taking the mean of the data, and negative values mean even worse models. From the scores associated with these two models, we can make an observation that holds more generally:

- For high-bias models, the performance of the model on the validation set is similar to the performance on the training set.
- For high-variance models, the performance of the model on the validation set is far worse than the performance on the training set.

If we imagine that we have some ability to tune the model complexity, we would expect the training score and validation score to behave as illustrated in the following figure:

The diagram shown here is often called a *validation curve*, and we see the following essential features:

- The training score is everywhere higher than the validation score. This is generally the case: the model will be a better fit to data it has seen than to data it has not seen.
- For very low model complexity (a high-bias model), the training data is under-fit, which means that the model is a poor predictor both for the training data and for any previously unseen data.
- For very high model complexity (a high-variance model), the training data is over-fit, which means that the model predicts the training data very well, but fails for any previously unseen data.
- For some intermediate value, the validation curve has a maximum. This level of complexity indicates a suitable trade-off between bias and variance.



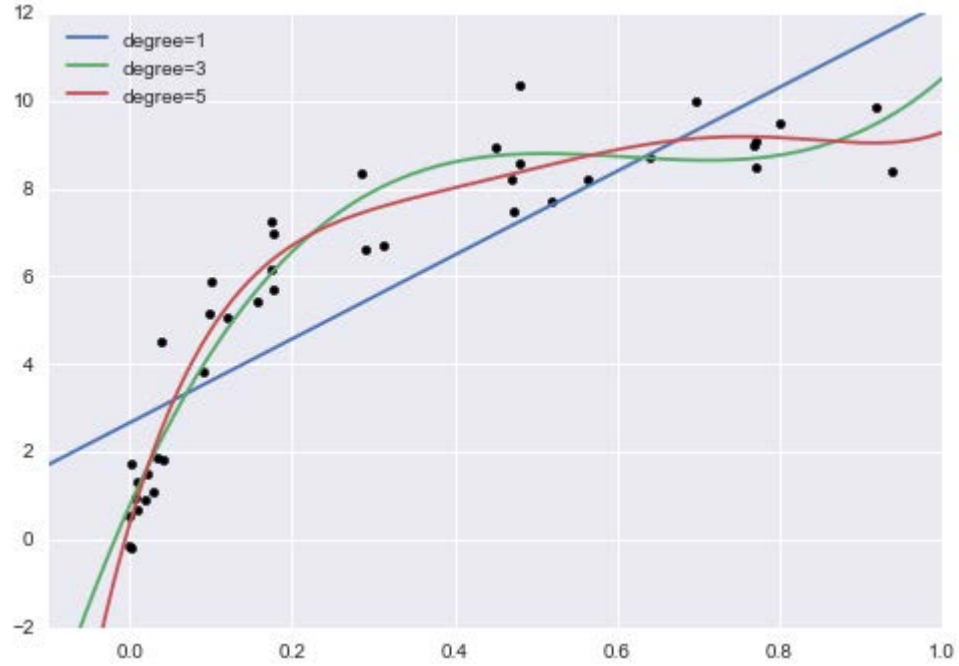
The means of tuning the model complexity varies from model to model; when we discuss individual models in depth in later classes, we will see how each model allows for such tuning.

2.2 Validation curves in Scikit-Learn

- Let's look at an example of using cross-validation to compute the validation curve for a class of models.
- Here we will use a polynomial regression model: this is a generalized linear model in which the degree of the polynomial is a tunable parameter.
- For example, a degree-1 polynomial fits a straight line to the data; for model parameters a and b: $y = ax + b$
- A degree-3 polynomial fits a cubic curve to the data; for model parameters: $y = ax^3 + bx^2 + cx + d$

- We can generalize this to any number of polynomial features.
- In Scikit-Learn, we can implement this with a simple linear regression combined with the polynomial preprocessor. We will use a pipeline to string these operations together: see the example code.

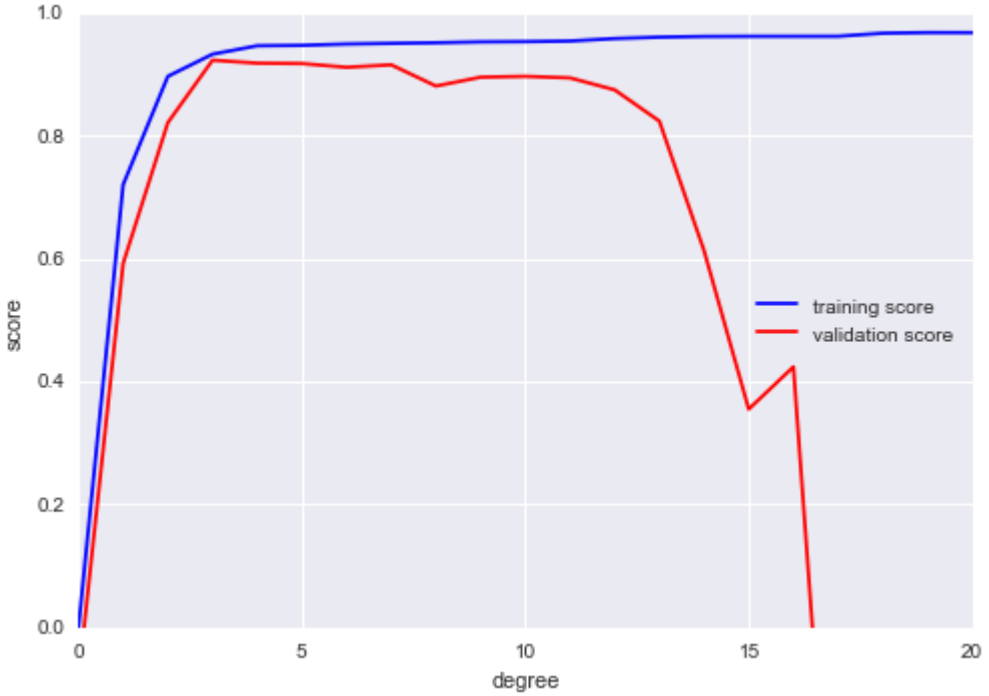
- The knob controlling model complexity in this case is the *degree of the polynomial*, which can be any non-negative integer.
- A useful question to answer is this: what degree of polynomial provides a *suitable trade-off* between *bias (under-fitting)* and *variance (over-fitting)*?



- Make progress in this by visualizing the validation curve for this particular data and model.
- This can be done straightforwardly using the learning_curve and validation_curve convenience routine provided by Scikit-Learn.
- Given a model, data, parameter name, and a range to explore, this function will automatically compute both the training score and validation score across the range:

This shows precisely the qualitative behavior we expect:

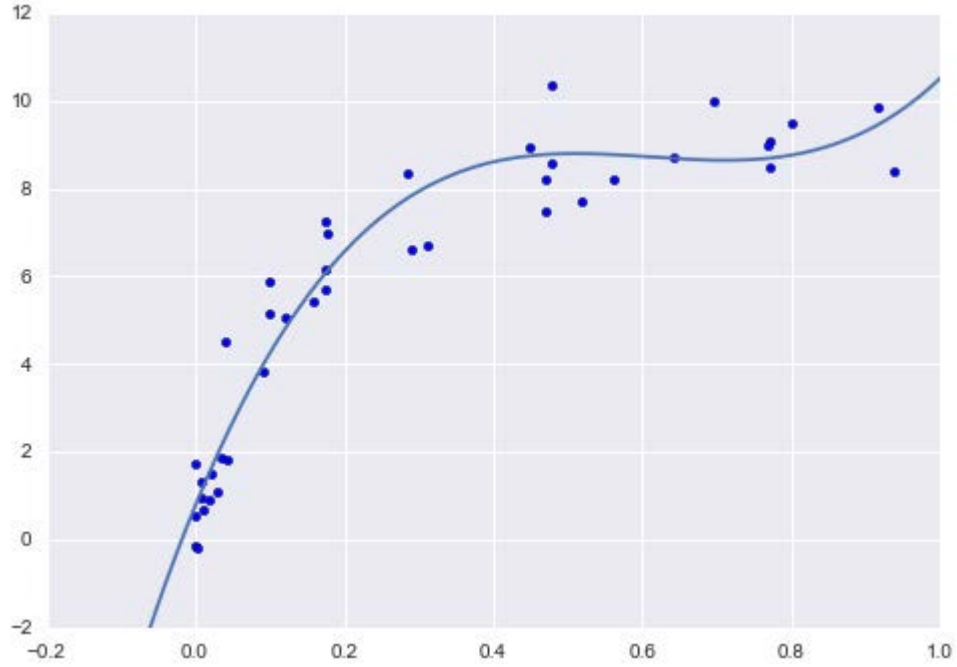
- the training score is everywhere higher than the validation score;
- the training score is monotonically improving with increased model complexity; and
- the validation score reaches a maximum before dropping off as the model becomes over-fit.



- From the validation curve, we can read-off that the optimal trade-off between bias and variance is found for a third-order polynomial; we can compute and display this fit over the original data as follows:

<u>Notice that:</u>

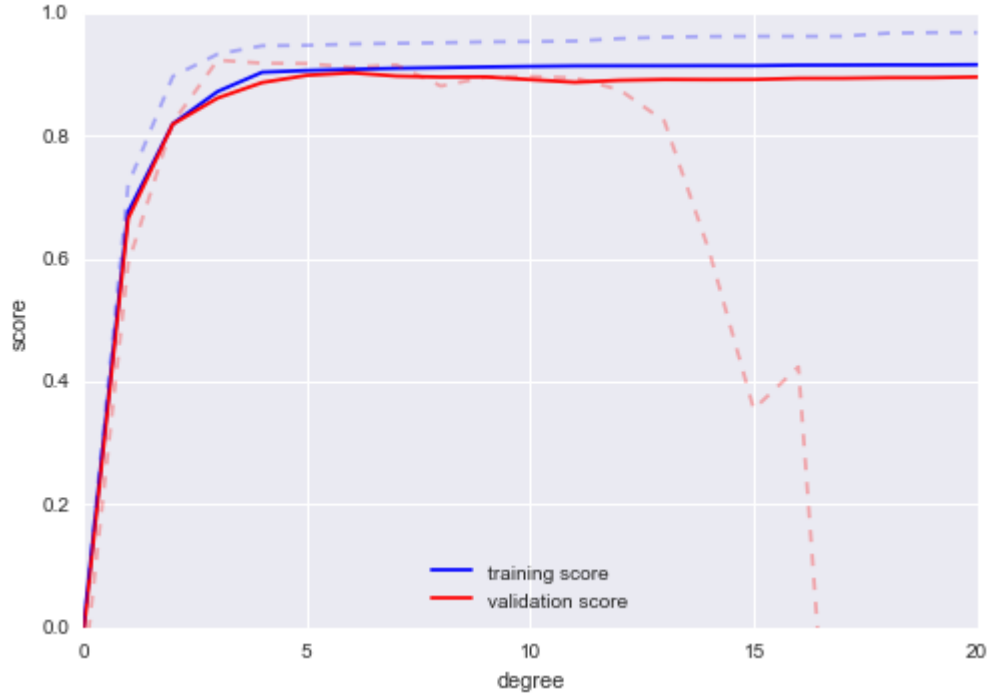
- finding this optimal model did not actually require us to compute the training score, but examining the relationship between the training score and validation score can give us useful insight into the performance of the model.



3. Learning Curves

- One important aspect of model complexity is that the optimal model will generally depend on the size of your training data.
- For example, let's generate a new dataset:

- The solid lines show the new results, while the fainter dashed lines show the results of the previous smaller dataset.
- It is clear from the validation curve that the larger dataset can support a much more complicated model: the peak here is probably around a degree of 6, but even a degree-20 model is not seriously over-fitting the data—the validation and training scores remain very close.



- Thus we see that the behavior of the validation curve has not one but two important inputs: the model complexity and the number of training points.
- It is often useful to explore the behavior of the model as a function of the number of training points, which we can do by using increasingly larger subsets of the data to fit our model.
- A plot of the training/validation score with respect to the size of the training set is known as a learning curve.

- The general behavior we would expect from a learning curve is this:
  - A model of a given complexity will overfit a small dataset: this means the training score will be relatively high, while the validation score will be relatively low.
  - A model of a given complexity will underfit a large dataset: this means that the training score will decrease, but the validation score will increase.
  - A model will never, except by chance, give a better score to the validation set than the training set: this means the curves should keep getting closer together but never cross.

- With these features in mind, we would expect a learning curve to look qualitatively like that shown in the following figure:

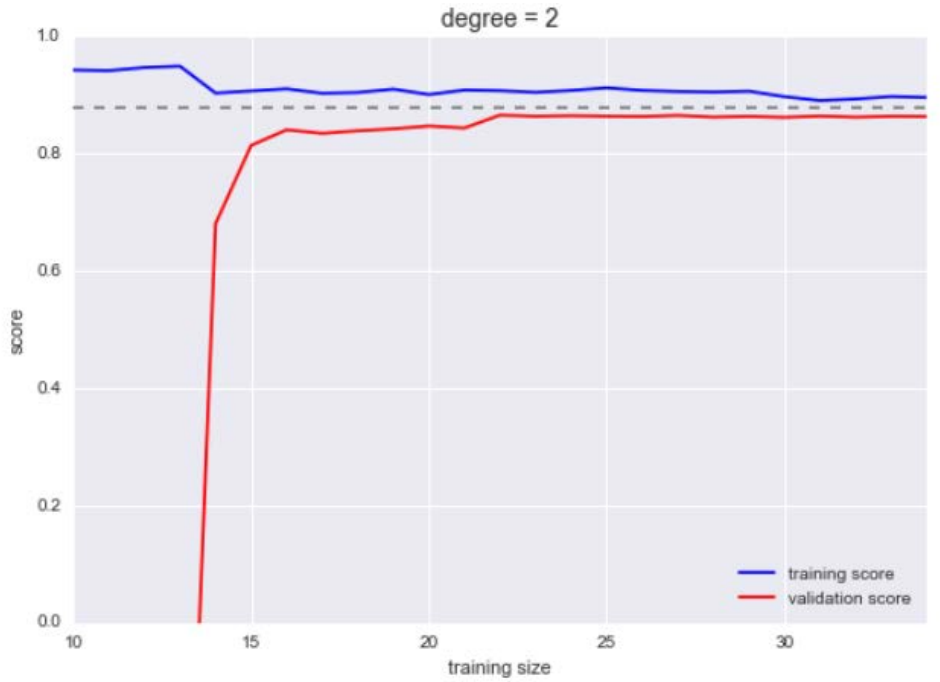
- The notable feature of the learning curve is the convergence to a particular score as the number of training samples grows.
- In particular, once you have enough points that a particular model has converged, adding more training data will not help you!
- The only way to increase model performance in this case is to use another (often more complex) model.

3.1 Learning curves in Scikit-Learn

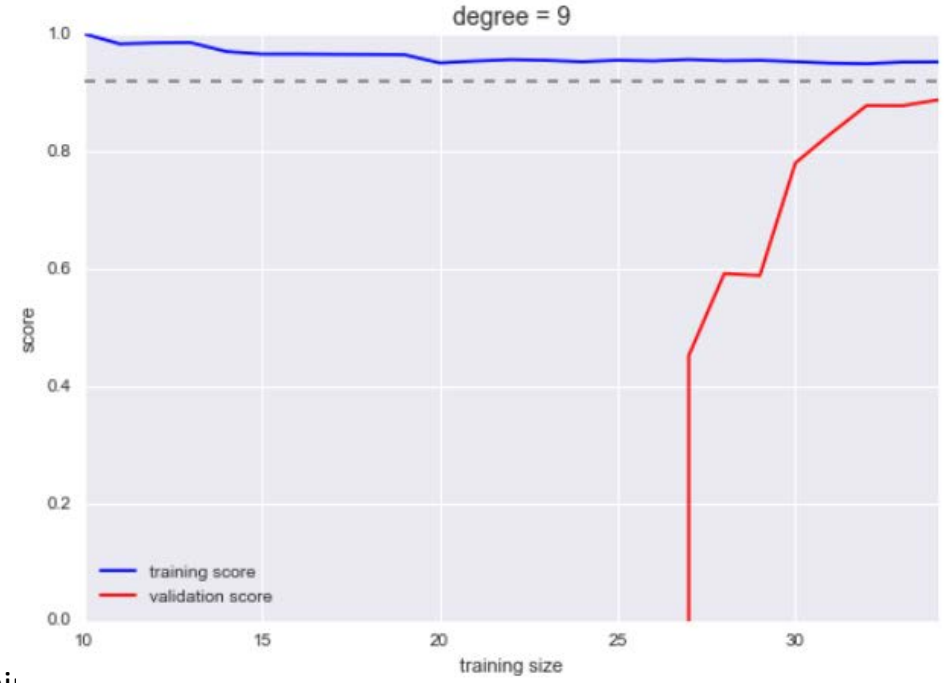
- Scikit-Learn offers a convenient utility for computing such learning curves from your models;
- here we will compute a learning curve for our original dataset with a second-order polynomial model and a ninth-order polynomial:

- This is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing training data.
- In particular, when your learning curve has already converged (i.e., when the training and validation curves are already close to each other) adding more training data will not significantly improve the fit! This situation is seen in the left panel, with the learning curve for the degree-2 model.

- This is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing training data.
- In particular, when your learning curve has already converged, adding more training data will not significantly improve the fit!
- This situation is seen in the upper figure, with the learning curve for the degree-2 model.

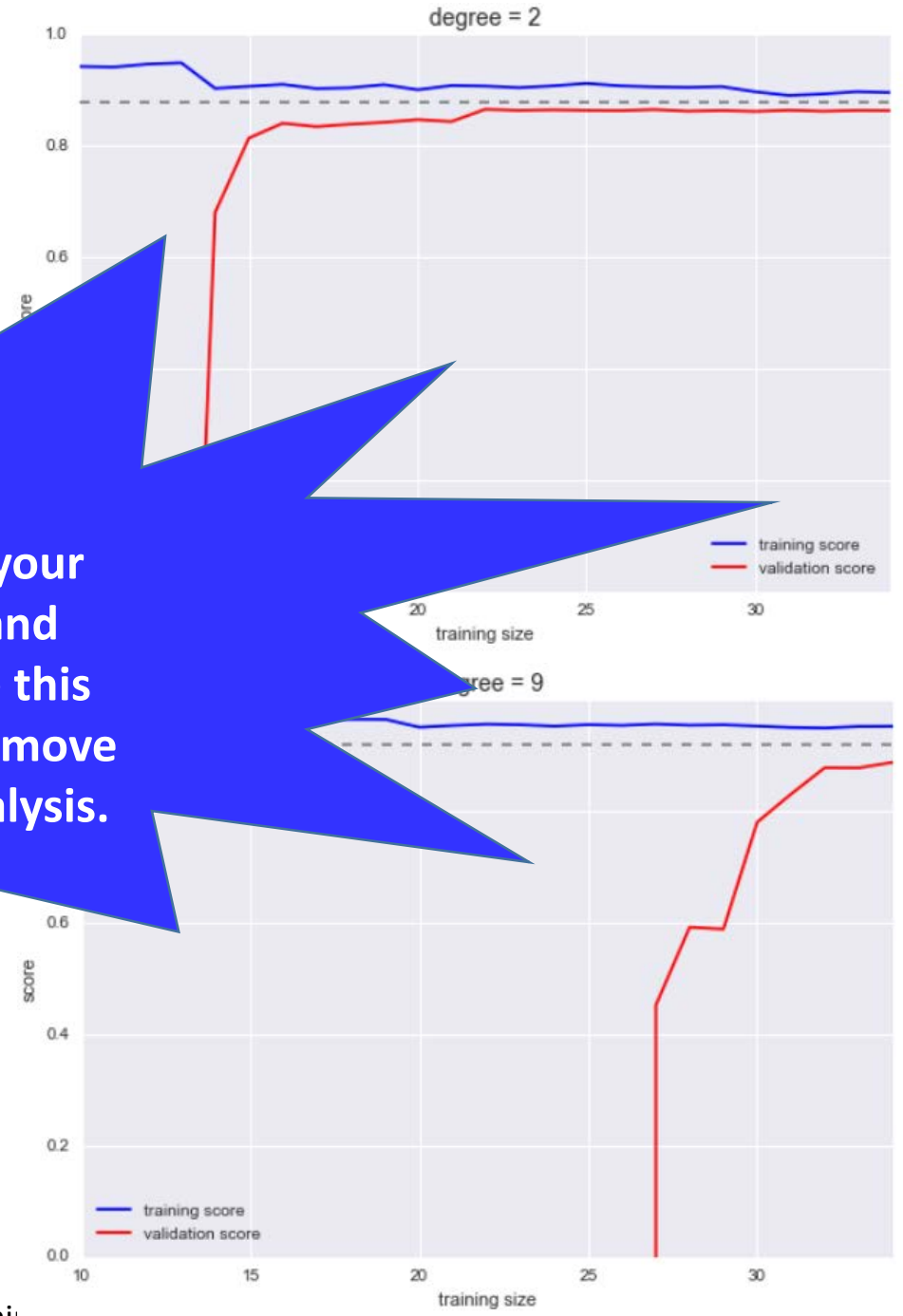


- The only way to increase the converged score is to use a different (usually more complicated) model.
- In the lower figure: by moving to a much more complicated model, we increase the score of convergence, but at the expense of higher model variance.
- If we were to add even more data points, the learning curve for the more complicated model would eventually converge.



- This is a valuable diagnostic, because it gives us a visual depiction of how our model responds to increasing training data.
- In particular, when your learning curve has already converged, adding more training data will not signif[illegible]ntly improve the fit!
- This situation is seen in the upp[illegible] curve for the de[illegible]
- The only way to i[illegible] different (usuall[illegible]
- In the lower figure: b[illegible] model, we increas[illegible] expense of higher model varia[illegible]
- If we were to add even more data points, the learning curve for the more complicated model would eventually converge.

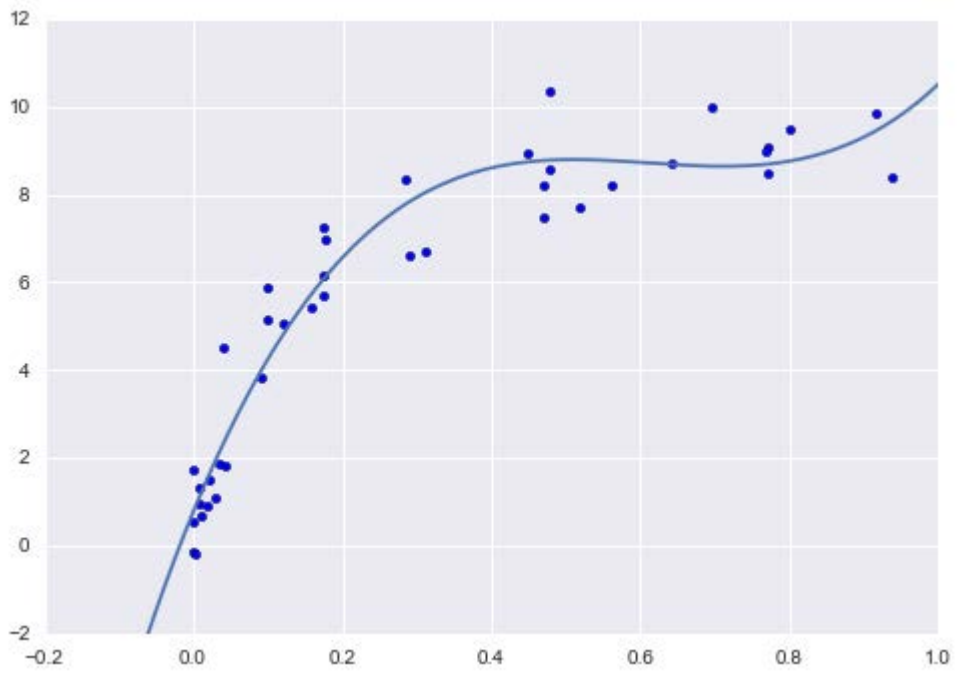
Plotting a learning curve for your particular choice of model and dataset can help you to make this type of decision about how to move forward in improving your analysis.



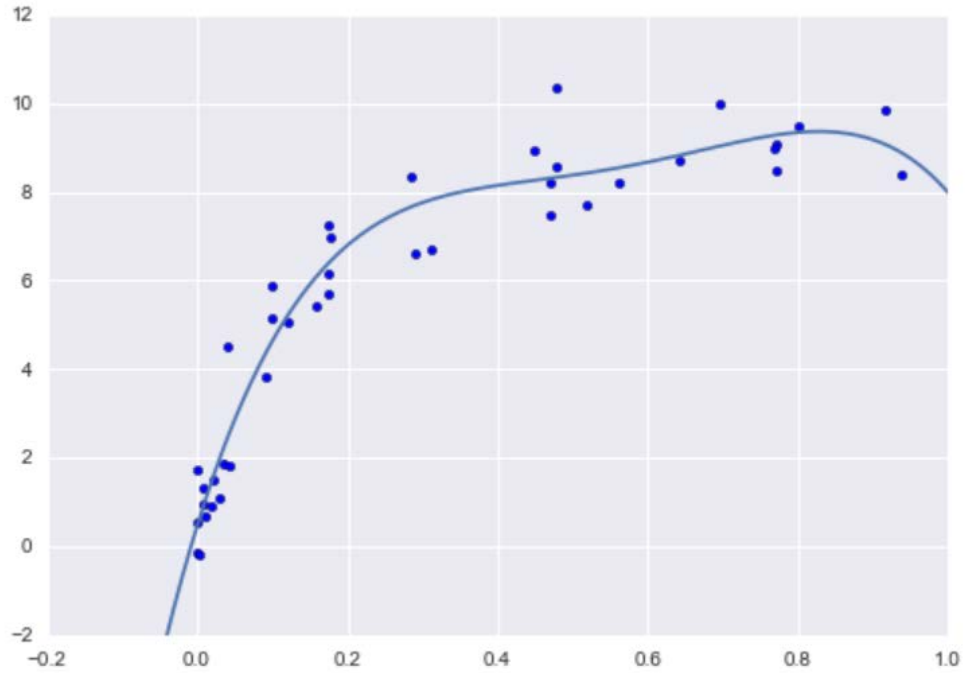
4. Validation in Practice: Grid Search

- The preceding discussion is meant to give you some intuition into the trade-off between bias and variance, and its dependence on model complexity and training set size.
- In practice, models generally have more than one knob to turn, and thus plots of validation and learning curves change from lines to multi-dimensional surfaces.
- In these cases, such visualizations are difficult and we would rather simply find the particular model that maximizes the validation score.

- Scikit-Learn provides automated tools to do this in the grid search module.
- Here is an example of using grid search to find the optimal polynomial model.
- We will explore a three-dimensional grid of model features; namely the polynomial degree,
  - the flag telling us whether to fit the intercept, and
  - the flag telling us whether to normalize the problem.
- This can be set up using Scikit-Learn's GridSearchCV meta-estimator:




```
plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = PolynomialRegression(3).fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test);
plt.axis(lim);
```




```
model = grid.best_estimator_

plt.scatter(X.ravel(), y)
lim = plt.axis()
y_test = model.fit(X, y).predict(X_test)
plt.plot(X_test.ravel(), y_test, hold=True);
plt.axis(lim);
```

Summary

- In this section, we have begun to explore the concept of model validation and hyperparameter optimization, focusing on intuitive aspects of the bias–variance trade-off and how it comes into play when fitting models to data.
- In particular, we found that the use of a validation set or cross-validation approach is vital when tuning parameters in order to avoid over-fitting for more complex/flexible models.
- In later sections, we will discuss the details of particularly useful models, and throughout will talk about what tuning is available for these models and how these free parameters affect model complexity. Keep this lesson in mind as you read on and learn about these machine learning approaches!

Section 2: Feature Engineering

- The section 1 outlines the fundamental ideas of machine learning, but all of the examples assume that you have numerical data in a tidy, [n_samples, n_features] format.
  - In the real world, data rarely comes in such a form. With this in mind, one of the more important steps in using machine learning in practice is feature engineering:
  - that is, taking whatever information you have about your problem and turning it into numbers that you can use to build your feature matrix.
- In section 2, we will cover a few common examples of feature engineering tasks:
  - features for representing categorical data,
  - features for representing text, and
  - features for representing images.
- Additionally, we will discuss derived features for increasing model complexity and imputation of missing data. Often this process is known as vectorization, as it involves converting arbitrary data into well-behaved vectors.

1. Categorical Features

- One common type of non-numerical data is *categorical* data.
- For example, imagine you are exploring some data on housing prices:

```
data = [
    {'price': 850000, 'rooms': 4, 'neighborhood': 'Queen Anne'},
    {'price': 700000, 'rooms': 3, 'neighborhood': 'Fremont'},
    {'price': 650000, 'rooms': 3, 'neighborhood': 'Wallingford'},
    {'price': 600000, 'rooms': 2, 'neighborhood': 'Fremont'}
]
```

- You might be tempted to encode this data with a straightforward numerical mapping: {'Queen Anne': 1, 'Fremont': 2, 'Wallingford': 3};

- It turns out that this is not generally a useful approach in Scikit-Learn:
  - the package's models make the fundamental assumption that numerical features reflect algebraic quantities.
  - Thus such a mapping would imply, for example, that Queen Anne < Fremont < Wallingford, or even that Wallingford - Queen Anne = Fremont, which does not make much sense.
- In this case, one proven technique is to use *one-hot encoding*, which effectively creates extra columns indicating the presence or absence of a category with a value of 1 or 0, respectively. When your data comes as a list of dictionaries, Scikit-Learn's DictVectorizer will do this for you:

```
array([[    0,     1,     0, 850000,     4],
       [    1,     0,     0, 700000,     3],
       [    0,     0,     1, 650000,     3],
       [    1,     0,     0, 600000,     2]],
```

Notice that

- The 'neighborhood' column has been expanded into three separate columns, representing the three neighborhood labels, and that each row has a 1 in the column associated with its neighborhood.
- With these categorical features thus encoded, you can proceed as normal with fitting a Scikit-Learn model.

There is one clear **disadvantage** of this approach:

- if your category has many possible values, this can greatly increase the size of your dataset.
- However, because the encoded data contains mostly zeros, a sparse output can be a very efficient solution:

```
vec = DictVectorizer(sparse=True, dtype=int)
vec.fit_transform(data)
Out[5]:
<4x5 sparse matrix of type '<class 'numpy.int64'>'
        with 12 stored elements in Compressed Sparse
Row format>
```

2. Text Features

- Another common need in feature engineering is to convert text to a set of representative numerical values.
- For example, most automatic mining of social media data relies on some form of encoding the text as numbers. One of the simplest methods of encoding data is by word counts:
  - To take each snippet of text, count the occurrences of each word within it, and put the results in a table.

Example: considering the three phases:

```
sample = ['problem of evil',  'evil queen',  'horizon problem']
```

- For a vectorization of this data based on word count, we could construct a column representing the word "problem," the word "evil," the word "horizon," and so on. While doing this by hand would be possible, the tedium can be avoided by using Scikit-Learn's CountVectorizer:

- The result is a sparse matrix recording the number of times each word appears; it is easier to inspect if we convert this to a DataFrame with labeled columns:
- There are some issues with this approach, however: the raw word counts lead to features which put too much weight on words that appear very frequently, and this can be sub-optimal in some classification algorithms.
- One approach to fix this is known as *term frequency-inverse document frequency* (TF–IDF) which weights the word counts by a measure of how often they appear in the documents. The syntax for computing these features is similar to the previous example:

3. Image Features

- Another common need is to suitably encode images for machine learning analysis. The simplest approach is what we used for the digits data in Introducing Scikit-Learn:
  - simply using the pixel values themselves. But depending on the application, such approaches may not be optimal.

- A comprehensive summary of feature extraction techniques for images is well beyond the scope of this course, which is a specialized subject.
- But you can find excellent implementations of many of the standard approaches in the Scikit-Image project. For one example of using Scikit-Learn and Scikit-Image together, see a later lesson.

4. Derived Features

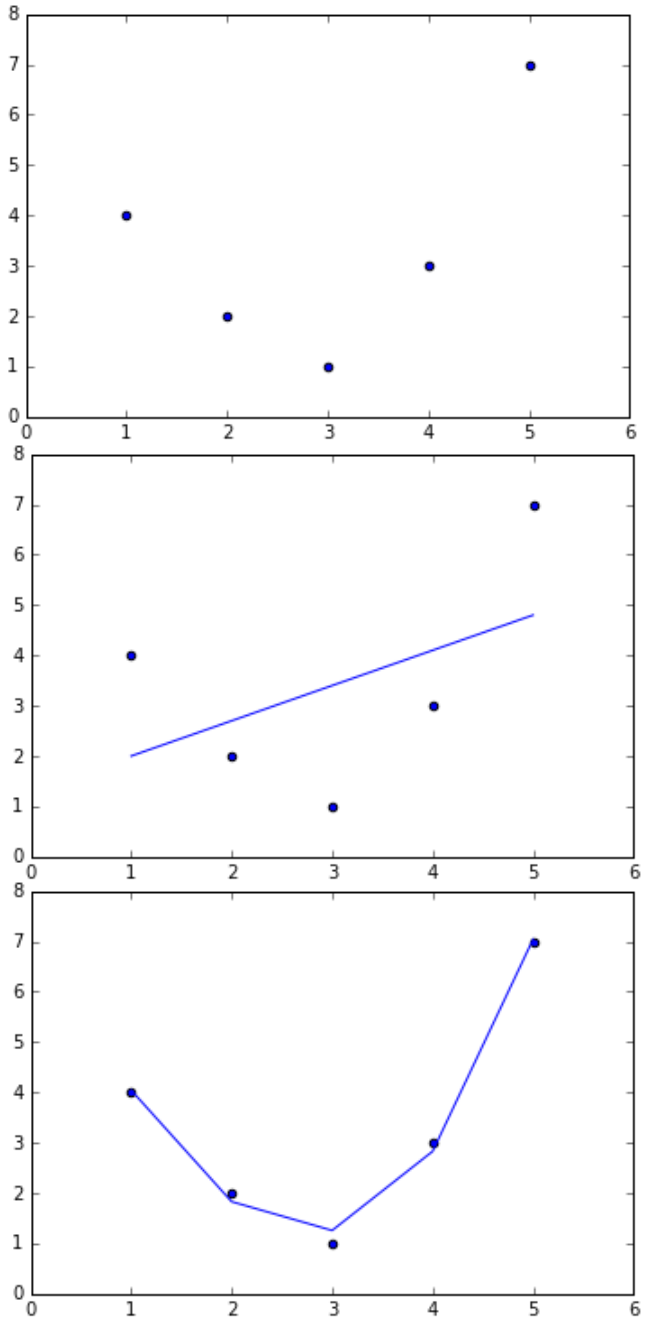
- Another useful type of feature is one that is mathematically derived from some input features.
- An example of this is in Hyperparameters and Model Validation when we constructed polynomial features from our input data.
  - Convert a linear regression into a polynomial regression not by changing the model, but by transforming the input!
  - This is sometimes known as basis function regression.

- For example, this data clearly cannot be well described by a straight line:
- Still, we can fit a line to the data using LinearRegression and get the optimal result:
- It's clear that we need a more sophisticated model to describe the relationship between x and y.
- One approach to this is to transform the data, adding extra columns of features to drive more flexibility in the model. For example, we can add polynomial features to the data this way:

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=3, include_bias=False)
X2 = poly.fit_transform(X)
print(X2)
```

```
[[  1.   1.   1.]
 [  2.   4.   8.]
 [  3.   9.  27.]
 [  4.  16.  64.]
 [  5.  25. 125.]]
```

- The derived feature matrix has one column representing , and a second column representing x^2, and a third column representing x^3. Computing a linear regression on this expanded input gives a much closer fit to our data:
- This idea of improving a model not by changing the model, but by transforming the inputs, is fundamental to many of the more powerful machine learning methods. We explore this idea further in later classes.



5. Imputation of Missing Data

- Another common need in feature engineering is handling of missing data.
- Handling of missing data in DataFrames in Pandas section, and often the NaN value is used to mark missing values.
- When applying a typical machine learning model to such data,
  - first, replace such missing data with some appropriate fill value. This is known as imputation of missing values, and
  - strategies range from simple (e.g., replacing missing values with the mean of the column) to sophisticated (e.g., using matrix completion or a robust model to handle such data).
- The sophisticated approaches tend to be very application-specific, and we won't dive into them here. For a baseline imputation approach, using the mean, median, or most frequent value, Scikit-Learn provides the SimpleImputer class:

- In the resulting data, the two missing values have been replaced with the mean of the remaining values in the column.
- This imputed data can then be fed directly into, for example, a LinearRegression estimator:

```
from sklearn.preprocessing import
Imputer
imp = Imputer(strategy='mean')
X2 = imp.fit_transform(X)
X2
Out[15]:
array([[ 4.5,  0. ,  3. ],
       [ 3. ,  7. ,  9. ],
       [ 3. ,  5. ,  2. ],
       [ 4. ,  5. ,  6. ],
       [ 8. ,  8. ,  1. ]])
```

5. Feature Pipelines

- With any of the preceding examples, it can quickly become tedious to do the transformations by hand, especially if you wish to string together multiple steps. For example, we might want a processing pipeline that looks something like this:
  - Impute missing values using the mean
  - Transform features to quadratic
  - Fit a linear regression
- To streamline this type of processing pipeline, Scikit-Learn provides a **Pipeline** object, which can be used as follows:
- This pipeline looks and acts like a standard Scikit-Learn object, and will apply all the specified steps to any input data.
- All the steps of the model are applied automatically. Notice that for the simplicity of this demonstration, we've applied the model to the data it was trained on; this is why it was able to perfectly predict the result.

The End!

Assignment: Practice the example code I ran in this class.

Course Review:

- Thank you for your high praise for my teaching in the mid-semester course review.
- If you have any suggestion or idea, please feel free to email me.