CS4619: Artificial Intelligence 2

Underfitting and Overfitting

Derek Bridge
School of Computer Science and Information Technology
University College Cork

Initialization

```python
In [1]: %load_ext autoreload
    %autoreload 2
    %matplotlib inline

In [2]: import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
```

Underfitting and Overfitting

Your estimator is underperforming. What should you do? The options include:

- Add more training data
- Add more features
- Change the complexity of the model
- Change the class of model, e.g., to a more complex one

Surprisingly, adding more data may not help; and, surprisingly, moving to a more complex model may in some cases worsen the results. This lecture shows you how to diagnose the problem and choose remedies that suit the diagnosis.

Defining Underfitting and Overfitting

To illustrate the concepts, we will use an artificial dataset. So that we can plot things in 2D, the dataset will have just one feature: a numeric-valued feature whose values range from 0 to 1. The dependent variable will be a non-linear function of the feature, and we'll add a bit of noise.
Let's create a dataset

```python
In [4]: X_train, y_train = make_dataset(50, f, 1.0)
X_test = np.linspace(-0.1, 1.1, 500).reshape(500, 1)
```

And let's plot the training set:

```python
In [5]: fig = plt.figure()
plt.xlabel("Feature")
plt.ylabel("MSE")
plt.ylim(-4, 14)
plt.scatter(X_train.ravel(), y_train, color = 'green')
plt.show()
```

### Fitting a Linear Model to the Data

We'll use OLS Linear Regression to fit a linear model, and we'll plot that model:
It's easy to see that a linear model is a poor choice. It **underfits** the data.

**Fitting a Quadratic Model to the Data**

What happens if we try to fit a more complex model such as a quadratic function?

Here's a class for fitting polynomials on a single feature — the details do not matter.

```python
In [7]: class PolynomialRegression(LinearRegression):
    def __init__(self, degree = 1):
        self.degree = degree
        LinearRegression.__init__(self)

    def fit(self, X, y):
        return LinearRegression.fit(self, X ** (1 + np.arange(self.degree)), y)

    def predict(self, X):
        return LinearRegression.predict(self, X ** (1 + np.arange(self.degree)))
```
So now we can fit a quadratic model:

```python
In [8]: estimator = PolynomialRegression(degree = 2)
estimator.fit(X_train, y_train)
y_predicted = estimator.predict(X_test)

fig = plt.figure()
plt.xlabel("Feature")
plt.ylabel("MSE")
plt.ylim(-4, 14)
plt.scatter(X_train.ravel(), y_train, color = 'green')
plt.plot(X_test.ravel(), y_predicted, color = 'blue')
plt.show()
```

This fits the training data much better.

**Fitting a Much Higher Degree Polynomial to the Data**

So what happens if we fit a higher degree polynomial? Here we’ll try 30:
While a model of this complexity fits the training set really well, it seems clear that this model is a poor choice. It is not capturing the target function; it is fitting to the noise in the training set. It overfits the data.

**Underfitting and Overfitting Summary**

In summary,

- A hypothesis underfits the training set if there is a more complex hypothesis with lower test error.
- A hypothesis overfits the training set if there is a less complex hypothesis with lower test error.

**The Bias-Variance Trade-Off**
The test error of a hypothesis is made up of three parts:

- the bias
- the variance, and
- the irreducible error

The irreducible error is...well..irreducible. It is caused by noise in the training set.

Bias refers to the error we will obtain when we try to capture a target function by a model that is too simple, e.g. trying to capture a non-linear target function using linear regression. It is therefore independent of the training set size. Generally, models of low complexity result in high bias. If your learning algorithm learns models that are not complex enough, bias may be high. Adding more training data will not help: the models are simply too restrictive to capture the target function.

Variance measures the stability of the learning algorithm in the face of different training sets, i.e. how much the function that gets learned changes when the examples in the training set change. Ideally, the function that gets learned will not vary too much between training sets. Generally, models of high complexity result in high variance. The models may fit the training data 'too well'; they can even fit to the noise in the training set; but they do not generalize well to unseen data. Adding more data may help: it can reinforce the true 'patterns' in the data and swamp the noise.

Ideally, we want a learning method that results in models with low bias and low variance. In general, as we increase complexity, bias will decrease and variance will increase. So we have a bias-variance trade-off.

**Visualizing the Bias-Variance Trade-Off**

We can explore the trade-off by plotting training error and test error for models of different complexity — in this case, polynomials of different degrees.

Showing this kind of thing, by the way, is why we sometimes want the training error as well as the test error.

We'd like to use $k$-fold cross-validation for this but you might remember that scikit-learn's implementation for $k$-fold cross-validation only computes test error. So we'll use my implementation of Repeated Holdout, which does compute both training error and test error:
We'll create a larger dataset than we were using earlier in this lecture.

In [11]:  

And finally here's the plot of training error and test error against complexity (the degree of the polynomial):
Because of the unreliability of the repeated holdout method, we may have to run the above code several times until we get a 'typical' graph. In a 'typical' graph, the test error will be higher than the training error, as you'd expect.

But, additionally, the graph should show the bias-variance trade-off:

- Where the complexity is low (low degree polynomials in this case), we have high bias: both the training error and the test error will be high, because the hypothesis underfits the data.
- Where the complexity is high (high degree polynomials in this case), we have high variance: the training error is low and the test error will be high, because the hypothesis overfits the data.

Between the two extremes, there's an optimum complexity where test error is at its lowest.

**Learning Curves**
Learning curves plot training error and test error against the number of examples in the training set. When an estimator is underperforming, these curves give some insight into whether the problem is high bias or high variance, and hence what to do about it.

In [13]:

def plot_learning_curve(estimator, X, y, num_iterations = 10, test_size = 0.3):
    max_training_set_size = int((1 - test_size) * len(y))
    training_set_sizes = range(1, max_training_set_size)
    mses_incremental_train = np.zeros((max_training_set_size, num_iterations))
    mses_incremental_test = np.zeros((max_training_set_size, num_iterations))
    ss = ShuffleSplit(n = len(y), n_iter = num_iterations, test_size = test_size,
                      random_state = np.random)
    for i, (train_indexes, test_indexes) in zip(range(num_iterations), ss):
        X_train = X[train_indexes]
        y_train = y[train_indexes]
        X_test = X[test_indexes]
        y_test = y[test_indexes]
        for j in training_set_sizes:
            estimator.fit(X_train[:j], y_train[:j])
            y_predicted = estimator.predict(X_train[:j])
            mses_incremental_train[j][i] = mean_squared_error(y_train[:j], y_predicted)
            y_predicted = estimator.predict(X_test)
            mses_incremental_test[j][i] = mean_squared_error(y_test, y_predicted)
        mses_train = []
        mses_test = []
        for j in training_set_sizes:
            mses_train.append(np.mean(mses_incremental_train[j]))
            mses_test.append(np.mean(mses_incremental_test[j]))
        fig = plt.figure()
        plt.xlabel('num. training examples')
        plt.ylabel('MSE')
        plt.ylim(0, 5)
        plt.plot(training_set_sizes, mses_train, label = 'training error', color = 'purple')
        plt.plot(training_set_sizes, mses_test, label = 'test error', color = 'orange')
        plt.plot(training_set_sizes, np.ones_like(training_set_sizes), linestyle = 'dotted', color = 'gray')
        plt.legend()
        plt.show()

Learning Curve When Complexity is About Right

We'll plot a learning curve for what we know to be a reasonable level of complexity for this dataset, namely polynomials of degree 5:
When the complexity of the model is about right, the 'typical' learning curve will show a large gap at the start: training error is very low (there are few examples, so they're easy to fit) but test error is high. As the number of training examples grows, the gap narrows and should converge. If the model complexity is appropriate then, after enough examples have been seen, we should converge on the irreducible error in the dataset. In our case this is approximately 1, because that's how much error we added to the artificial dataset.

**Learning Curve When Complexity is Too Low**

Let's look at the learning curve when the model is not complex enough. Here, the polynomials are of degree 1:

```
In [23]: estimator = PolynomialRegression(1)
plot_learning_curve(estimator, X, y)
```
We see that error is higher than the irreducible error.

Also in a ‘typical’ learning curve for this situation, the training error and test error will probably not converge. Adding more examples doesn't help much. This is because we have high bias: no amount of examples will compensate for the fact that the set of hypotheses is just too restrictive.

Learning Curve When Complexity is Too High

Let's look at a curve where the model is too complex. Here, the polynomials are of degree 16:

```python
In [16]:
   estimator = PolynomialRegression(16)
   plot_learning_curve(estimator, X, y)
```

Here we have high variance: adding more examples is helping. But it's not converging as quickly as it did for degree 5. This is because the set of hypotheses is larger, so we need more examples to discriminate between them.

Remedial Actions
If the curves aren't converging:

- Increase the complexity — if you think you're underfitting due to high bias
  - Move to a more complex model (e.g. a polynomial of a higher degree, or a different kind of model, such as a nonparametric one that makes fewer assumptions about the target function)
  - Move to a more complex variant of your current model
- Add more features — ones that are predictive of the dependent variable

If the curves are converging but too slowly:

- Decrease the complexity — if you think you're overfitting due to high variance
  - Move to a less complex model (e.g. a polynomial of a lower degree or a different kind of model that makes more assumptions about the target function)
  - Move to a less complex variant of your current model (e.g. regularization, smoothing, ...)
- Add more examples

This isn't an exhaustive set of remedial actions. For example, if the curves aren't converging, maybe you have lots of irrelevant features: removing some might help the learner fit a good hypothesis. And if the curves are converging too slowly, then maybe using a method to detect and remove noisy examples will help.

Credits. I based this notebook on these:

- [http://www.astroml.org/sklearn_tutorial/practical.html](http://www.astroml.org/sklearn_tutorial/practical.html)