5.4 Validation and Test Errors

May 9, 2019

1 5.4 Test and Validation Errors

In the previous section, we saw that training error is not a great measure of a model's quality. For example, a 1-nearest neighbor model will have a training error of 0.0 (or close to it), but it is not necessarily the best prediction model, especially if there are outliers in the training data.

In order to come up with a better measure of model quality, we need to formalize what it is we want to measure.

In [1]:
```
%matplotlib inline
import numpy as np
import pandas as pd
pd.options.display.max_rows = 5

housing = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/maste
                      sep="\t")
housing
```

Out[1]:
```
      Order        PID  MS SubClass MS Zoning  Lot Frontage  Lot Area Street  \
0         1  526301100           20        RL         141.0     31770   Pave
1         2  526350040           20        RH          80.0     11622   Pave
...     ...        ...          ...       ...           ...       ...    ...
2928   2929  924100070           20        RL          77.0     10010   Pave
2929   2930  924151050           60        RL          74.0      9627   Pave

     Alley Lot Shape Land Contour    ...     Pool Area Pool QC  Fence  \
0      NaN       IR1          Lvl    ...             0     NaN    NaN
1      NaN       Reg          Lvl    ...             0     NaN  MnPrv
...    ...       ...          ...    ...           ...     ...    ...
2928   NaN       Reg          Lvl    ...             0     NaN    NaN
2929   NaN       Reg          Lvl    ...             0     NaN    NaN

     Misc Feature Misc Val Mo Sold Yr Sold Sale Type  Sale Condition  \
0             NaN        0       5    2010       WD           Normal
1             NaN        0       6    2010       WD           Normal
...           ...      ...     ...     ...      ...              ...
2928          NaN        0       4    2006       WD           Normal
2929          NaN        0      11    2006       WD           Normal
```

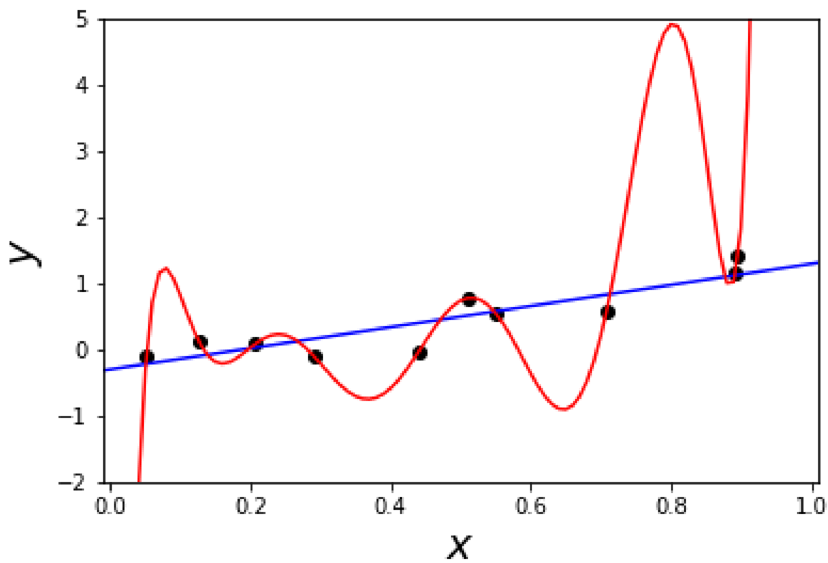
```
      SalePrice
0        215000
1        105000
...         ...
2928     170000
2929     188000

[2930 rows x 82 columns]
```

1.1 Overfitting and Test Error

Ultimately, the goal of any prediction model is to make predictions on *future* data. Therein lies the problem with training error. Training error measures how well a model predicts on the current data. It is possible to build a model that **overfits** to the training data—that is, a model that fits so well to the current data that it does poorly on future data.

For example, consider fitting two different models to the 10 training observations shown below. The model represented by the red line actually passes through every observation (that is, its training error is zero). However, most people would prefer the model represented by the blue line. If one had to make a prediction for y when $x = 0.8$, the value of the blue line is intuitively more plausible than the value of the red line, which is out of step with the nearby points.



The argument for the blue model depends on *future* data because the blue model is actually worse than the red model on the current data. The red model tries so hard to get the predictions on the training data right that it ends up *overfitting*.

If the goal is to build a model that performs well on future data, then we ought to evaluate it (i.e., by calculating MSE, MAE, etc.) on future data. The prediction error on future data is also known as **test error**, in contrast to training error, which is the prediction error on current data. To calculate the test error, we need *labeled* future data. In many applications, future data is expensive to collect and *labeled* future data is even more expensive. How can we approximate the test error, using just the data that we have?

1.2 Validation Error

The solution is to split the training data into a **training set** and a **validation set**. The model will only be fit on the observations of the training set. Then, the model will be evaluated on the validation set. Because the validation set has not been seen by the model already, it essentially plays the role of "future" data, even though it was carved out of the current data.

The prediction error on the validation set is known as the **validation error**. The validation error is an approximation to the test error.

To split our data into training and validation sets, we use the `.sample()` function in `pandas`. Let's use this to split our data into two equal halves, which we will call `train` and `val`.

```
In [2]: train = housing.sample(frac=.5)
        val = housing.drop(train.index)

        train
```

```
Out[2]:       Order        PID  MS SubClass MS Zoning  Lot Frontage  Lot Area Street  \
        129     130  534450180           20        RL          50.0      7207   Pave
        1964   1965  535453150           20        RL          70.0      7315   Pave
        ...     ...        ...          ...       ...           ...       ...    ...
        1319   1320  902401010           50        RM           NaN      5700   Pave
        261     262  907200340           20        RL          75.0     10650   Pave

             Alley Lot Shape Land Contour   ...     Pool Area Pool QC  Fence  \
        129    NaN       IR1          Lvl   ...             0     NaN    NaN
        1964   NaN       Reg          Lvl   ...             0     NaN    NaN
        ...    ...       ...          ...   ...           ...     ...    ...
        1319   NaN       Reg          Lvl   ...             0     NaN    NaN
        261    NaN       Reg          Lvl   ...             0     NaN  MnPrv

             Misc Feature Misc Val Mo Sold Yr Sold Sale Type  Sale Condition  \
        129           NaN        0       2    2010       WD           Normal
        1964          NaN        0       3    2007       WD           Normal
        ...           ...      ...     ...     ...      ...              ...
        1319          NaN        0       8    2008       WD           Normal
        261           NaN        0       2    2010       WD           Normal

              SalePrice
        129      116500
        1964     140000
        ...         ...
        1319     116900
        261      128200

        [1465 rows x 82 columns]
```

Now let's use this training/validation split to approximate the test error of a 10-nearest neighbors model.

First, we extract the variables we need.

```
In [3]: # Features in our model. All quantitative, except Neighborhood.
        features = ["Lot Area", "Gr Liv Area",
                    "Full Bath", "Half Bath",
                    "Bedroom AbvGr",
                    "Year Built", "Yr Sold",
                    "Neighborhood"]

        X_train_dict = train[features].to_dict(orient="records")
        X_val_dict = val[features].to_dict(orient="records")

        y_train = train["SalePrice"]
        y_val = val["SalePrice"]
```

Next, we use Scikit-Learn to preprocess the training and the validation data. Note that the vectorizer and the scaler are both fit to the training data, so we learn the categories, the mean, and standard deviation from the training set—and use these to transform both the training and validation sets.

```
In [4]: from sklearn.feature_extraction import DictVectorizer
        from sklearn.preprocessing import StandardScaler

        # convert categorical variables to dummy variables
        vec = DictVectorizer(sparse=False)
        vec.fit(X_train_dict)
        X_train = vec.transform(X_train_dict)
        X_val = vec.transform(X_val_dict)

        # standardize the data
        scaler = StandardScaler()
        scaler.fit(X_train)
        X_train_sc = scaler.transform(X_train)
        X_val_sc = scaler.transform(X_val)
```

We are now ready to fit a k-nearest neighbors model to the training data.

```
In [5]: from sklearn.neighbors import KNeighborsRegressor

        # Fit a 10-nearest neighbors model.
        model = KNeighborsRegressor(n_neighbors=10)
        model.fit(X_train_sc, y_train)

Out[5]: KNeighborsRegressor(algorithm='auto', leaf_size=30, metric='minkowski',
                  metric_params=None, n_jobs=1, n_neighbors=10, p=2,
                  weights='uniform')
```

We make predictions on the validation set and calculate the validation RMSE:

```
In [6]: y_val_pred = model.predict(X_val_sc)
        rmse = np.sqrt(((y_val - y_val_pred) ** 2).mean())
        rmse
```

Out[6]: 41669.803996811032

Notice that the test error is higher than the training error that we calculated in the previous section. In general, this will be true. It is harder for a model to predict for new observations it has not seen, than for observations it has seen!

1.3 Cross Validation

One downside of the validation error above is that it was calculated using only 50% of the data. As a result, the estimate is noisy.

There is a cheap way to obtain a second opinion of how well our model will do on future data. Previously, we split our data at random into two halves, training the model on the first half and evaluating it using the second half. Because the model has not already seen the second half of the data, this approximates how well the model would perform on future data.

But the way we split our data was arbitrary. We might as well swap the roles of the two halves, training the model on the *second* half and evaluating it using the *first* half. As long as the model is always evaluated on data that is different from the data that was used to train it, we have a valid measure of how well our model would perform on future data. A schematic of this approach, known as **cross-validation**, is shown below.

Because we will be doing all computations twice, just with different data, let's wrap the k-nearest neighbors algorithm above into a function called `get_val_error()`, that computes the validation error given training and validation data.

In [7]:
```
def get_val_error(X_train_dict, y_train, X_val_dict, y_val):

    # convert categorical variables to dummy variables
    vec = DictVectorizer(sparse=False)
    vec.fit(X_train_dict)
    X_train = vec.transform(X_train_dict)
    X_val = vec.transform(X_val_dict)

    # standardize the data
    scaler = StandardScaler()
    scaler.fit(X_train)
    X_train_sc = scaler.transform(X_train)
    X_val_sc = scaler.transform(X_val)

    # Fit a 10-nearest neighbors model.
    model = KNeighborsRegressor(n_neighbors=10)
    model.fit(X_train_sc, y_train)

    # Make predictions on the validation set.
    y_val_pred = model.predict(X_val_sc)
    rmse = np.sqrt(((y_val - y_val_pred) ** 2).mean())

    return rmse
```

If we apply this function to the training and test sets from earlier, we get the same estimate of the test error.

```
In [8]: get_val_error(X_train_dict, y_train, X_val_dict, y_val)
```

```
Out[8]: 41669.803996811032
```

But if we reverse the roles of the training and test sets, we get another estimate of the test error.

```
In [9]: get_val_error(X_val_dict, y_val, X_train_dict, y_train)
```

```
Out[9]: 41248.325516531797
```

Now we have two, somewhat independent estimates of the test error. It is common to average the two to obtain an overall estimate of the test error, called the **cross-validation error**. Notice that the cross-validation error uses each observation in the data exactly once. We make a prediction for each observation, but always using a model that was trained on data that does not include that observation.

2 Exercises

Exercise 1. Use cross-validation to estimate the test error of a 1-nearest neighbor classifier on the Ames housing price data. How does a 1-nearest neighbor classifier compare to a 10-nearest neighbor classifier in terms of its ability to predict on *future* data?

```
In [10]: def get_val_error(X_train_dict, y_train, X_val_dict, y_val, k):

             # convert categorical variables to dummy variables
             vec = DictVectorizer(sparse=False)
             vec.fit(X_train_dict)
             X_train = vec.transform(X_train_dict)
             X_val = vec.transform(X_val_dict)

             # standardize the data
             scaler = StandardScaler()
             scaler.fit(X_train)
             X_train_sc = scaler.transform(X_train)
             X_val_sc = scaler.transform(X_val)

             # Fit a 1-nearest neighbors model.
             model = KNeighborsRegressor(n_neighbors=k)
             model.fit(X_train_sc, y_train)

             # Make predictions on the validation set.
             y_val_pred = model.predict(X_val_sc)
             rmse = np.sqrt(((y_val - y_val_pred) ** 2).mean())

             return rmse

         def get_crossval_error(X_train_dict, y_train, X_val_dict, y_val, k=10):
             er1 = get_val_error(X_train_dict, y_train, X_val_dict, y_val, k)
```

```
        er2 = get_val_error(X_val_dict, y_val, X_train_dict, y_train, k)

        return (er1 + er2)/2
```

```
In [11]: print("k=1: ", get_crossval_error(X_train_dict, y_train, X_val_dict, y_val, k=1),
              "\nk=10:", get_crossval_error(X_train_dict, y_train, X_val_dict, y_val, k=10))
```

```
k=1:  43775.7029009
k=10: 41459.0647567
```

Exercise 2. Using the Tips data set (https://raw.githubusercontent.com/dlsun/data-science-book/maste train k-nearest neighbors regression models to predict the tip for different values of k. Calculate the training and validation MAE of each model, and make a plot showing these errors as a function of k.

```
In [12]: tips = pd.read_csv("https://raw.githubusercontent.com/dlsun/data-science-book/master/c

         train = tips.sample(frac=.5)
         val = tips.drop(train.index)
```

```
In [13]: features = ["total_bill", "sex", "smoker", "day", "time", "size"]

         X_train_dict = train[features].to_dict(orient="records")
         X_val_dict = val[features].to_dict(orient="records")
         y_train = train["tip"]
         y_val = val["tip"]

         def container(X_train_dict, y_train, X_val_dict, y_val):
             def get_val_error(X_train_dict, y_train, X_val_dict, y_val, k):

                 # convert categorical variables to dummy variables
                 vec = DictVectorizer(sparse=False)
                 vec.fit(X_train_dict)
                 X_train = vec.transform(X_train_dict)
                 X_val = vec.transform(X_val_dict)

                 # standardize the data
                 scaler = StandardScaler()
                 scaler.fit(X_train)
                 X_train_sc = scaler.transform(X_train)
                 X_val_sc = scaler.transform(X_val)

                 # Fit a 1-nearest neighbors model.
                 model = KNeighborsRegressor(n_neighbors=k)
                 model.fit(X_train_sc, y_train)

                 # Make predictions on the validation set.
                 y_val_pred = model.predict(X_val_sc)
```

```
        mae = (y_val - y_val_pred).abs().mean()

        return mae

    def get_crossval_error(k):
        er1 = get_val_error(X_train_dict, y_train, X_val_dict, y_val, k)
        er2 = get_val_error(X_val_dict, y_val, X_train_dict, y_train, k)

        return (er1 + er2)/2

    return get_crossval_error

ks = pd.Series(range(1,51,1))
ks.index = range(1,51,1)
ks.apply(container(X_train_dict, y_train, X_val_dict, y_val)).plot.line(),
ks.apply(container(X_train_dict, y_train, X_train_dict, y_train)).plot.line()
```

```
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x7f960ac556a0>
```