1 Module 8: Feature Engineering

Contents

1.1 Parabolic/Nonlinear Model Fitting

- 2nd-order model with d features

$$\hat{y} = \sum_{j=1}^{d} \theta_{j2}\phi_j^2 + \sum_{j=1}^{d} \theta_{j1}\phi_j + \alpha$$

- In order to fit a squared regression, we can just square the feature and fit as linear (.e.g. horsepower^2 $\propto$ mpg)

```
LinearRegression().fit(data[x, x^2], data[y])
```

- Sklearn Transformers:

```
PolynomialFeatures(degree=d,
↪ include_bias=False).fit_transform([[x]])
return array([[x^1, x^2, ..., x^d]])
```

where include_bias=True includes x^0
So a 1D array of length n will return a 2D array of size $d + 1 \times n$

- Reconstruct the dataframe with

```
pd.DataFrame(Fitted_PolyFtObj, columns =
↪ Fitted_PolyFtObj.get_feature_names_out())
```

- To make a sequence of operations more convenient:

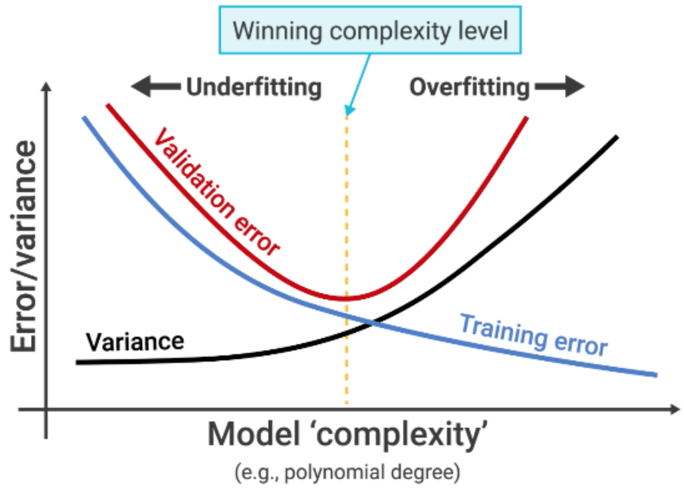
```
model = sklearn.pipeline.Pipeline([
('transform', PolynomialFeatures()),
('regression',LinearRegression())
])

model.fit(x,y)
model.predict([[x_i]])
```

- Access individual components e.g.

```
model.named_steps['regression'].coef_
```

- Model variance encapsulates the model's sensitivity to the training data (ten-



dency to overfit)

- Given data with n points, we can find an nth-order model which fits it perfectly

$$\hat{Y} = \Phi\Theta \text{ solving for } \Theta = \Phi^{-1}\hat{Y}$$

- Cross-validate model:
  Split into train & test data to evaluate MSE independently

- Shuffle data so that location of split is randomized

```
data = sklearn.utils.shuffle(data)
train, val = numpy.split(data, [n])
```

- **Hyperparameters** are the params whih decide between models (e.g. degree) and uses the validation set of data only

- The test set of data is a third set which evaluates the performance of the model, separate from the parameter selection bias of the training set & the hyperparameter bias of the validation set

- Rule of thumb is Train/Val/Test data proportion is 60/20/20

1.2 Prediction vs. Inference

- Prediction: Fitting a model and sampling predictions from the fitted domain only. (i.e. a parabolic model predicts that high HP will give lower fuel efficiency, which is incorrect)
- Inference: Using the model to understand the true relationship of the features.

1.3 Encoding methods

- Variations of one-hot encoding:

```
pd.get_dummies(data, dummy_na=False)
# Or
ohe = OneHotEncoder(sparse = False, drop='if_binary')
data_train = ohe.fit_transform(data_train)
data_test = ohe.transform(data_test)
```

  Additionally, one can avoid having to take out a column & then re-add it to a DataFrame by using: (e.g., 'CentralAir' needs one-hot; 'OverallQual' does not.

```
col_transformer = make_column_transformer(
    (OneHotEncoder(drop = 'if_binary'), ['CentralAir']),
    remainder='passthrough')

col_transformer.fit_transform(X_train[['OverallQual', 'CentralAir']])
```

- Ordinal encoding (e.g. column with values 'Poor', 'Fair', 'Good', 'Excellent' which we want to correspond to 1, 2, 3, 4 respectively)

```
oe = OrdinalEncoder(categories = [['Poor', 'Fair', 'Good',
↪  'Excellent']])
data = oe.fit_transform(X_train[['Quality']])
```

- Column transformer can handle multiple encoders just like a pipeline can. The arguments are tuples which correspond the encoder with a set of input columns.

```
multi = make_column_transformer(
    (OrdinalEncoder(categories = [['Po', 'Fa', 'TA', 'Gd', 'Ex']]),
    ↪  ['HeatingQC']),
    (OneHotEncoder(drop = 'if_binary'), ['CentralAir']),
    (PolynomialFeatures(include_bias = False, degree = 2),
    ↪  ['OverallQual']))

multi.fit_transform(X_train[['OverallQual', 'CentralAir', 'HeatingQC']])
```