

**Elements Of Data Science - F2025**

# **Week 7: Model Evaluation and Hyperparameter Tuning**

**10/28/2025**

# TODOs

- Readings:
  - PML 4.1 Dealing with Missing Data - 4.4 Bringing Features onto the Same Scale
  - Additional: [PDSH Chapter 5: Feature Engineering](#)
- Quiz 7 and Hw2, due Tue Nov 11th, 11:59pm ET
- No class next week, Nov 4

# Today

- Model Evaluation and Selection
- Hyperparameter Tuning
- Regularization

Questions?

# Environment Setup

# Model Evaluation and Hyperparameter Tuning

- How well are any of our models working?
- How can we compare different models?
- How do we decide on hyperparameter settings?
- How can we keep our models from "overfitting" (and what does that mean)?
- How do we do all this for both Regression and Classification?

# How well are our models performing?

## Regression

- Mean Squared Error (MSE) and Root Mean Squared Error (RMSE)
- $R^2$  or  $R^2$  or  $R^2$
- (Adjusted  $R^2$  - we'll talk about this during Feature Selection)

## Classification

- Accuracy
- Precision/Recall/F1
- ROC Area Under the Curve (AUC)

# Data Setup for Regression

# Data Setup for Regression

```
In [2]: 1 zscore = lambda x: (x - x.mean()) / x.std()
        2
        3 df_wine = pd.read_csv('../data/wine_dataset.csv',
        4                             usecols=['alcalinity_of_ash', 'magnesium', 'alcohol', 'ash', 'proline', 'hue', 'class'])
        5 numeric_cols = ['alcalinity_of_ash', 'magnesium', 'alcohol', 'ash', 'proline', 'hue']
        6
        7 df_wine[numeric_cols] = df_wine[numeric_cols].apply(zscore)  # standardize numeric feature cols
        8
        9 x = df_wine[['proline', 'hue', 'ash']]
       10
       11 y_r = df_wine['alcohol']  # regression target
```



# Data Setup for Regression

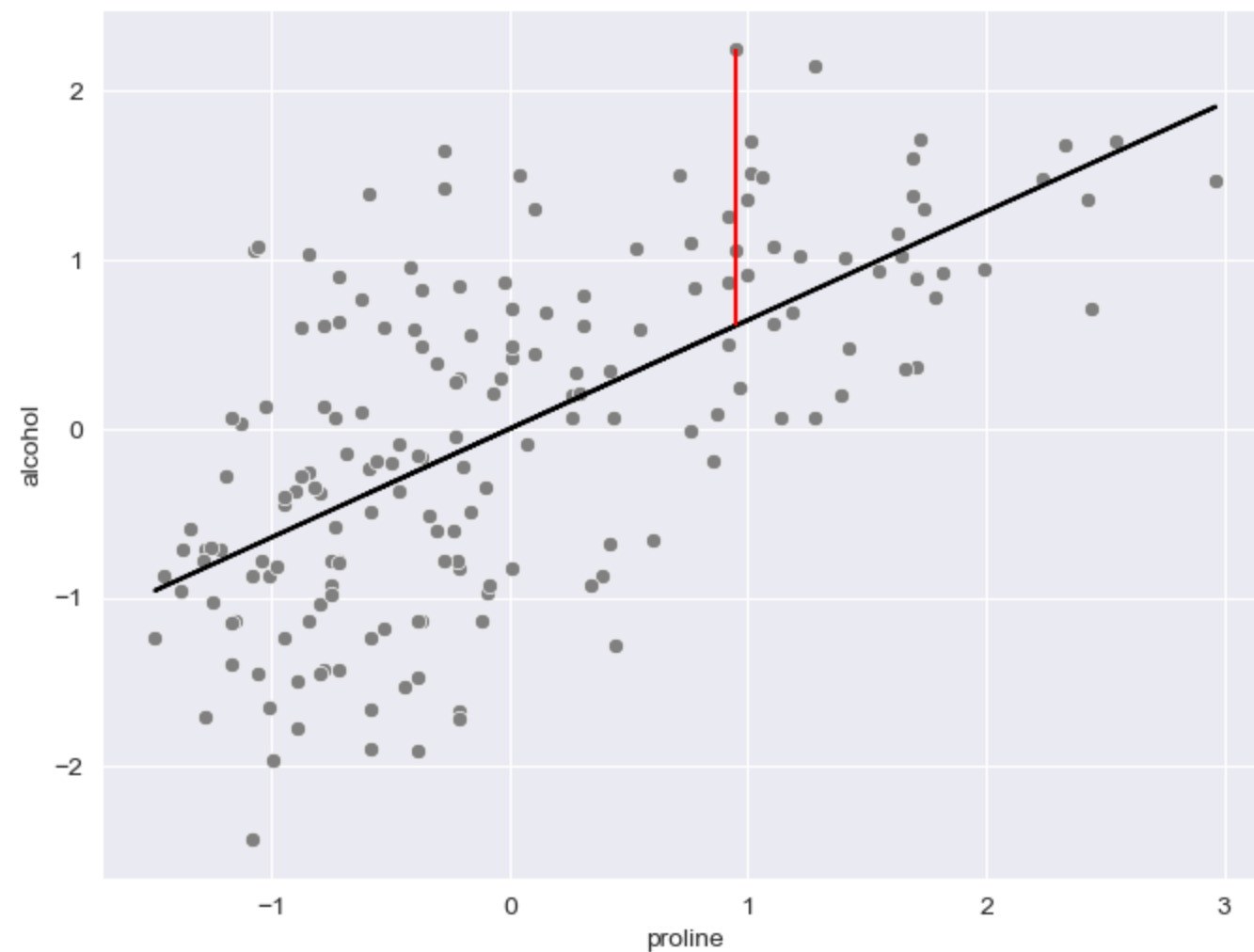
```
In [2]: 1 zscore = lambda x: (x - x.mean()) / x.std()
2
3 df_wine = pd.read_csv('../data/wine_dataset.csv',
4                       usecols=['alcalinity_of_ash', 'magnesium', 'alcohol', 'ash', 'proline', 'hue', 'class'])
5 numeric_cols = ['alcalinity_of_ash', 'magnesium', 'alcohol', 'ash', 'proline', 'hue']
6
7 df_wine[numeric_cols] = df_wine[numeric_cols].apply(zscore) # standardize numeric feature cols
8
9 x = df_wine[['proline', 'hue', 'ash']]
10
11 y_r = df_wine['alcohol'] # regression target
```

```
In [3]: 1 df_wine.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 178 entries, 0 to 177
Data columns (total 7 columns):
#   Column                Non-Null Count  Dtype
---  -
0   alcohol               178 non-null   float64
1   ash                   178 non-null   float64
2   alcalinity_of_ash     178 non-null   float64
3   magnesium             178 non-null   float64
4   hue                   178 non-null   float64
5   proline               178 non-null   float64
6   class                 178 non-null   int64
dtypes: float64(6), int64(1)
memory usage: 9.9 KB
```

# Regression with Simple Linear Model

```
In [4]: 1 from sklearn.linear_model import LinearRegression
2
3 lr = LinearRegression().fit(X[['proline']],y_r)
4 argmax_y_r = np.argmax(y_r)
5 y_lr_pred = lr.predict(X[['proline']])
6
7 fig,ax = plt.subplots(1,1,figsize=(8,6))
8 sns.scatterplot(x=X.proline, y=y_r, color='grey');
9 ax.plot(X.proline,y_lr_pred,color='k');
10 ax.vlines(X.proline.iloc[argmax_y_r],y_r.iloc[argmax_y_r],y_lr_pred[argmax_y_r],color='r');
```



# How Good is This Fit? MSE and RMSE

- Mean Squared Error (MSE) :  $\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$

# How Good is This Fit? MSE and RMSE

- Mean Squared Error (MSE):  $\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$

```
In [5]: 1 from sklearn.metrics import mean_squared_error
        2
        3 lr_mse = mean_squared_error(y_r, y_lr_pred)
        4 print(f'{lr_mse = :0.2f}')
```

```
lr_mse = 0.58
```

# How Good is This Fit? MSE and RMSE

- Mean Squared Error (MSE):  $\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$

```
In [5]: 1 from sklearn.metrics import mean_squared_error
        2
        3 lr_mse = mean_squared_error(y_r, y_lr_pred)
        4 print(f'{lr_mse = :0.2f}')
```

```
lr_mse = 0.58
```

- But this is the squared error! (alcohol<sup>2</sup>)

- Root Mean Squared Error (RMSE):  $\sqrt{\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2}$

**\*\*RMSE\*\*** has same unit as y

# How Good is This Fit? MSE and RMSE

- Mean Squared Error (MSE):  $\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2$

```
In [5]: 1 from sklearn.metrics import mean_squared_error
        2
        3 lr_mse = mean_squared_error(y_r, y_lr_pred)
        4 print(f'{lr_mse = :0.2f}')
```

```
lr_mse = 0.58
```

- But this is the squared error! (alcohol^2)

- Root Mean Squared Error (RMSE):  $\sqrt{\frac{1}{n} \sum_i (y_i - \hat{y}_i)^2}$

**\*\*RMSE\*\*** has same unit as y

```
In [6]: 1 lr_rmse = mean_squared_error(y_r, y_lr_pred, squared=False)
        2 print(f'{lr_rmse = :0.2f}')
```

```
lr_rmse = 0.76
```

# Is this good? Need a Baseline Comparison

- What's a baseline to compare against?
- Simple one for Regression: always predict the mean of the targets

# Is this good? Need a Baseline Comparison

- What's a baseline to compare against?
- Simple one for Regression: always predict the mean of the targets

```
In [1]: 1 from sklearn.dummy import DummyRegressor
2
3 dummyr = DummyRegressor(strategy='mean') # default strategy
4 dummyr.fit(X[['proline']],y_r)
5
6 dummy_rmse = mean_squared_error(y_r,dummyr.predict(X[['proline']] ),squared=False)
7
8 print(f'{dummy_rmse = :0.5f}')
```

-----

```
NameError                                Traceback (most recent call last)
Cell In[1], line 4
      1 from sklearn.dummy import DummyRegressor
      3 dummyr = DummyRegressor(strategy='mean') # default strategy
----> 4 dummyr.fit(X[['proline']],y_r)
      6 dummy_rmse = mean_squared_error(y_r,dummyr.predict(X[['proline']] ),squared=False)
      8 print(f'{dummy_rmse = :0.5f}')
```

NameError: name 'X' is not defined



# Comparing against the mean: $R^2$

- the proportion of variance explained by the model

$$R^2 = 1 - \frac{\sum (y_i - \hat{y}_i)^2}{\sum (y_i - \bar{y})^2}$$

- maximum value of 1
- a value below 0 means the model is predicting worse than just predicting the mean
- sklearn uses  $R^2$  as the default for regression scoring

```
In [8]: 1 r2_lr = lr.score(X[['proline']], y_r)
        2 r2_dummyr = dummyr.score(X[['proline']], y_r)
        3
        4 print(f'{r2_dummyr} = :0.2f'\n{r2_lr} = :0.2f')
```

```
r2_dummyr = 0.00
r2_lr      = 0.41
```

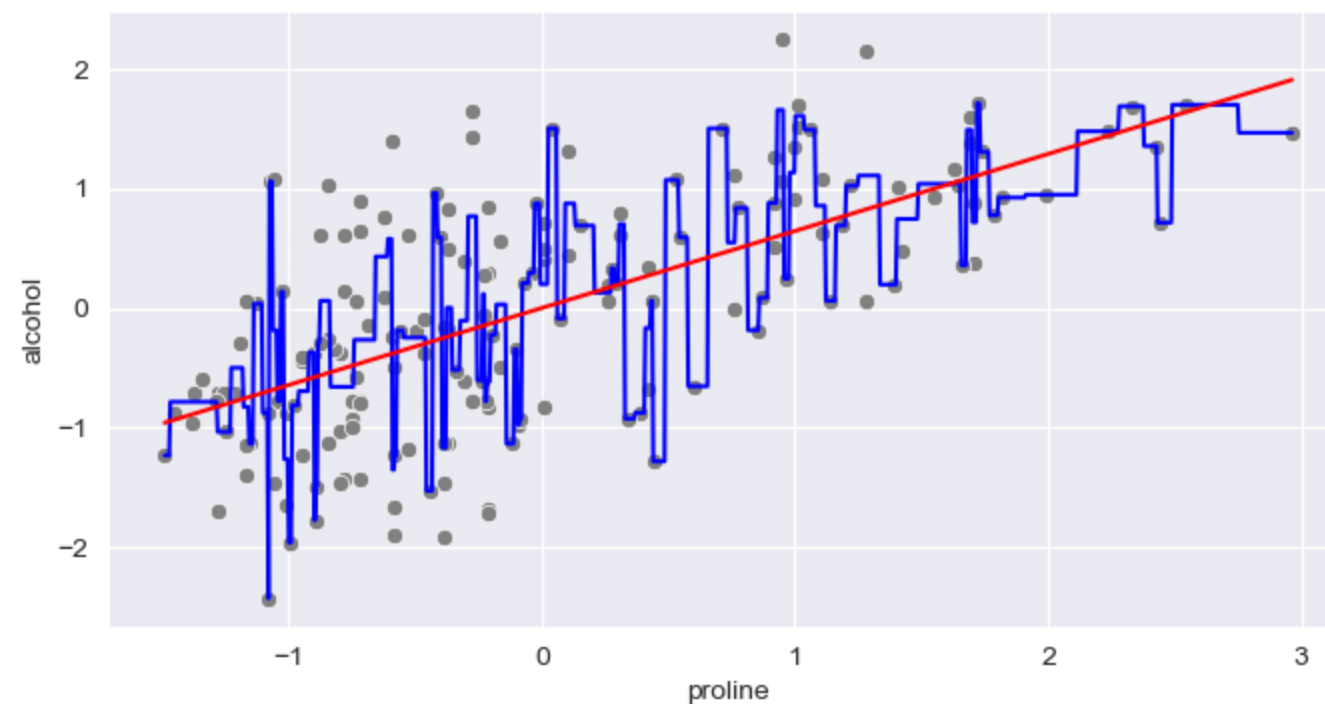
**Can we do better?**

# Can we do better?

```
In [9]: 1 from sklearn.tree import DecisionTreeRegressor
2
3 dtr = DecisionTreeRegressor(max_depth=10).fit(X[['proline']],y_r)
4 r2_dtr = dtr.score(X[['proline']],y_r)
5 print(f'{r2_lr = :0.2f}\n{r2_dtr = :0.2f}')
6
7 X_query = pd.DataFrame({'proline':np.linspace(X.proline.min(),X.proline.max(),1000)})
8 y_dtr_pred = dtr.predict(X_query)
9 y_lr_pred = lr.predict(X_query)
10 fig,ax = plt.subplots(1,1,figsize=(8,4))
11 sns.scatterplot(x=X.proline, y=y_r,color='gray')
12 ax.plot(X_query,y_dtr_pred,color='b')
13 ax.plot(X_query,y_lr_pred,color='r');
```

r2\_lr = 0.41

r2\_dtr = 0.76



# But is this what we want? Interpretation vs Prediction

Always good to ask:

- do we want our model to very closely fit our data for interpretation?
- do we want our model to predict well on new, unseen data (*generalize well*)?

**Generalization:**

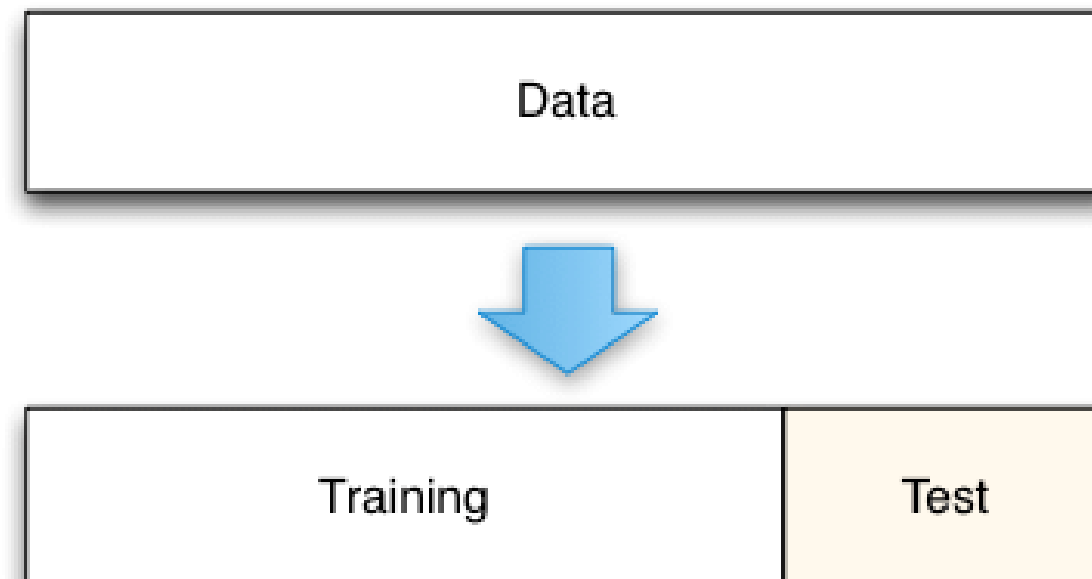
- how well will model predict on data that it hasn't seen yet?

But we used all of our data to train?

- Need to do a **Train/Test Split** to create a held-aside set

# Train/Test Split

- **Training Set:** portion of dataset used for training
- **Test/Held-Aside/Validation/Out of sample:** portion of dataset used for evaluation
- Want the test set to reflect the same distribution as training



# Train/Test split with Sklearn

# Train/Test split with Sklearn

```
In [10]: 1 from sklearn.model_selection import train_test_split
          2
          3 X_train_r,X_test_r,y_train_r,y_test_r = train_test_split(X[['proline']],
          4                                                         y_r,
          5                                                         train_size=.75,  #default (only need one of train/test)
          6                                                         test_size=.25,   #default
          7                                                         random_state=123)
          8 print(f"n_total = {X[['proline']].shape[0]}")
          9 print(f'n_train = {X_train_r.shape[0]}')
         10 print(f'n_test  = {X_test_r.shape[0]}')
```

n\_total = 178  
n\_train = 133  
n\_test = 45

# Train/Test split with Sklearn

```
In [10]: 1 from sklearn.model_selection import train_test_split
2
3 X_train_r,X_test_r,y_train_r,y_test_r = train_test_split(X[['proline']],
4                                                         y_r,
5                                                         train_size=.75, #default (only need one of train/test)
6                                                         test_size=.25, #default
7                                                         random_state=123)
8 print(f'n_total = {X[['proline']].shape[0]}')
9 print(f'n_train = {X_train_r.shape[0]}')
10 print(f'n_test  = {X_test_r.shape[0]}')

n_total = 178
n_train = 133
n_test  = 45
```

- How big should test be?
  - Large enough to capture variance of dataset.
  - Depends on the dataset and the models being trained



# Training and Evaluating on Different Data

# Training and Evaluating on Different Data

```
In [11]: 1 dummyr = DummyRegressor().fit(X_train_r,y_train_r)
          2 lr = LinearRegression().fit(X_train_r,y_train_r)
          3 dtr = DecisionTreeRegressor(max_depth=10).fit(X_train_r,y_train_r)
          4
          5 r2_dummyr = dummyr.score(X_test_r,y_test_r)
          6 r2_lr      = lr.score(X_test_r,y_test_r)
          7 r2_dtr     = dtr.score(X_test_r,y_test_r)
          8
          9 print(f'{r2_dummyr = : 0.2f}\n{r2_lr      = : 0.2f}\n{r2_dtr     = : 0.2f}')
```

```
r2_dummyr = -0.03
r2_lr      =  0.28
r2_dtr     = -0.31
```

# Training and Evaluating on Different Data

```
In [11]: 1 dummyr = DummyRegressor().fit(X_train_r,y_train_r)
          2 lr = LinearRegression().fit(X_train_r,y_train_r)
          3 dtr = DecisionTreeRegressor(max_depth=10).fit(X_train_r,y_train_r)
          4
          5 r2_dummyr = dummyr.score(X_test_r,y_test_r)
          6 r2_lr      = lr.score(X_test_r,y_test_r)
          7 r2_dtr     = dtr.score(X_test_r,y_test_r)
          8
          9 print(f'{r2_dummyr = : 0.2f}\n{r2_lr      = : 0.2f}\n{r2_dtr     = : 0.2f}')
```

r2\_dummyr = -0.03  
r2\_lr = 0.28  
r2\_dtr = -0.31

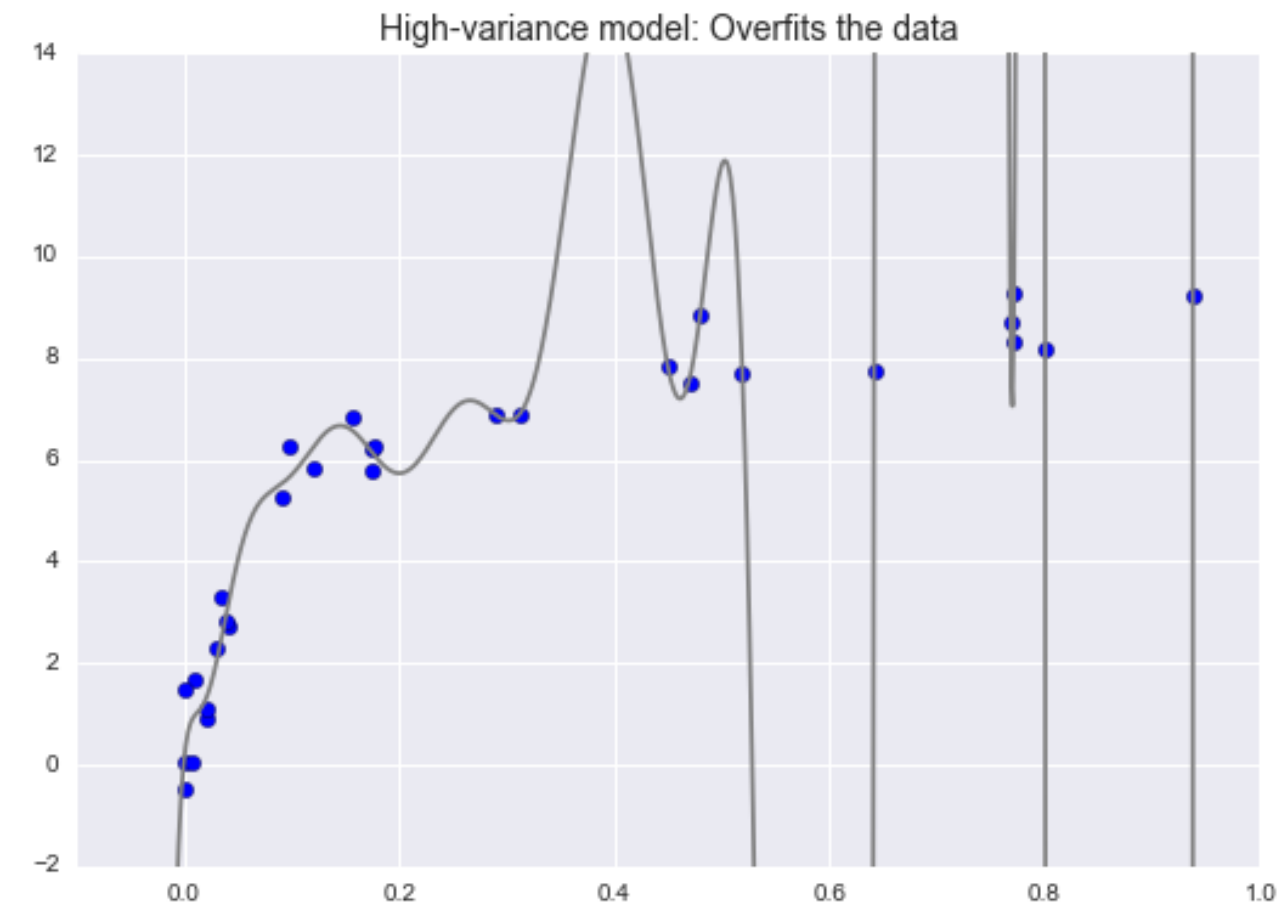
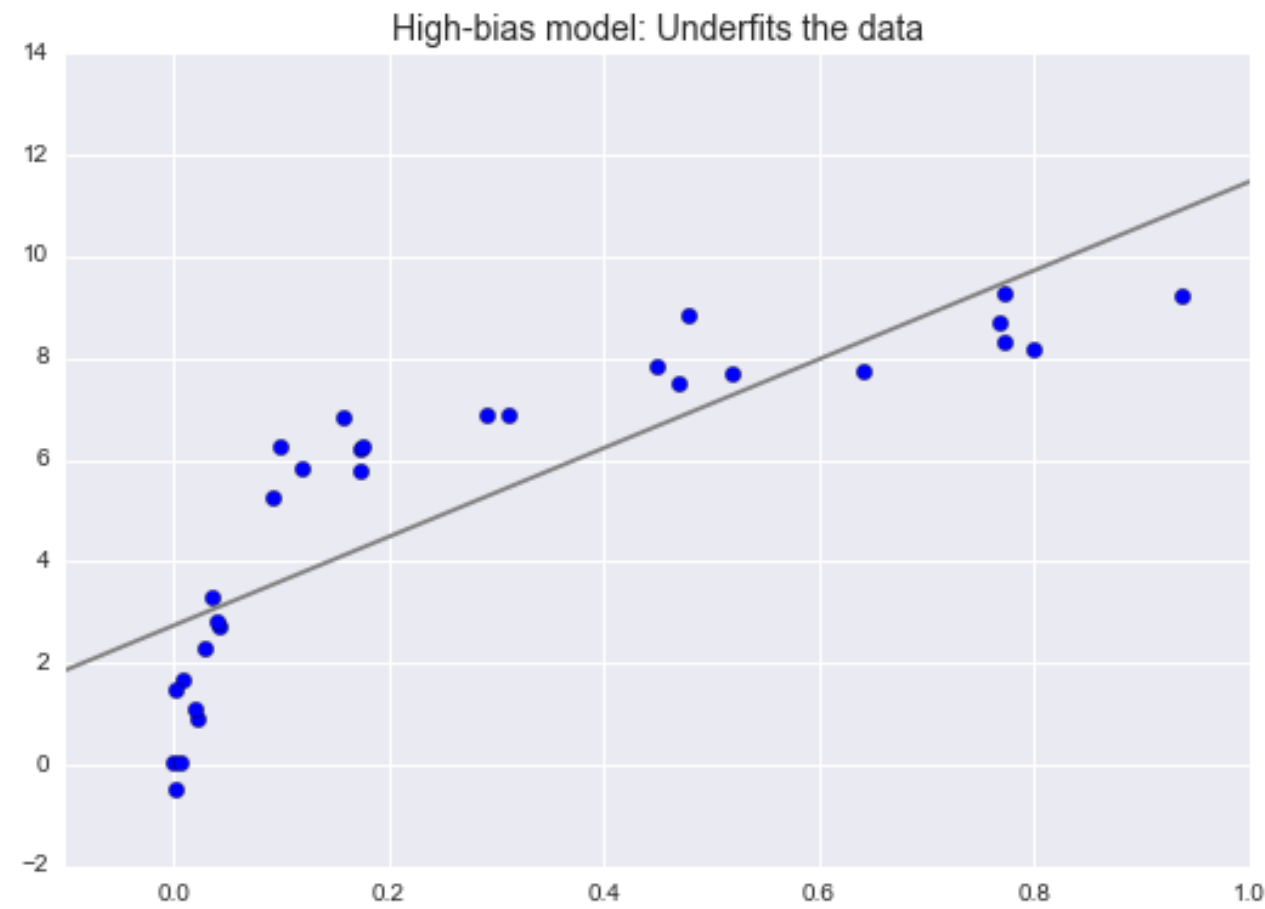
- DecisionTree model is doing worse than the Dummy model on the test set!

# Overfitting and Underfitting

# Overfitting and Underfitting

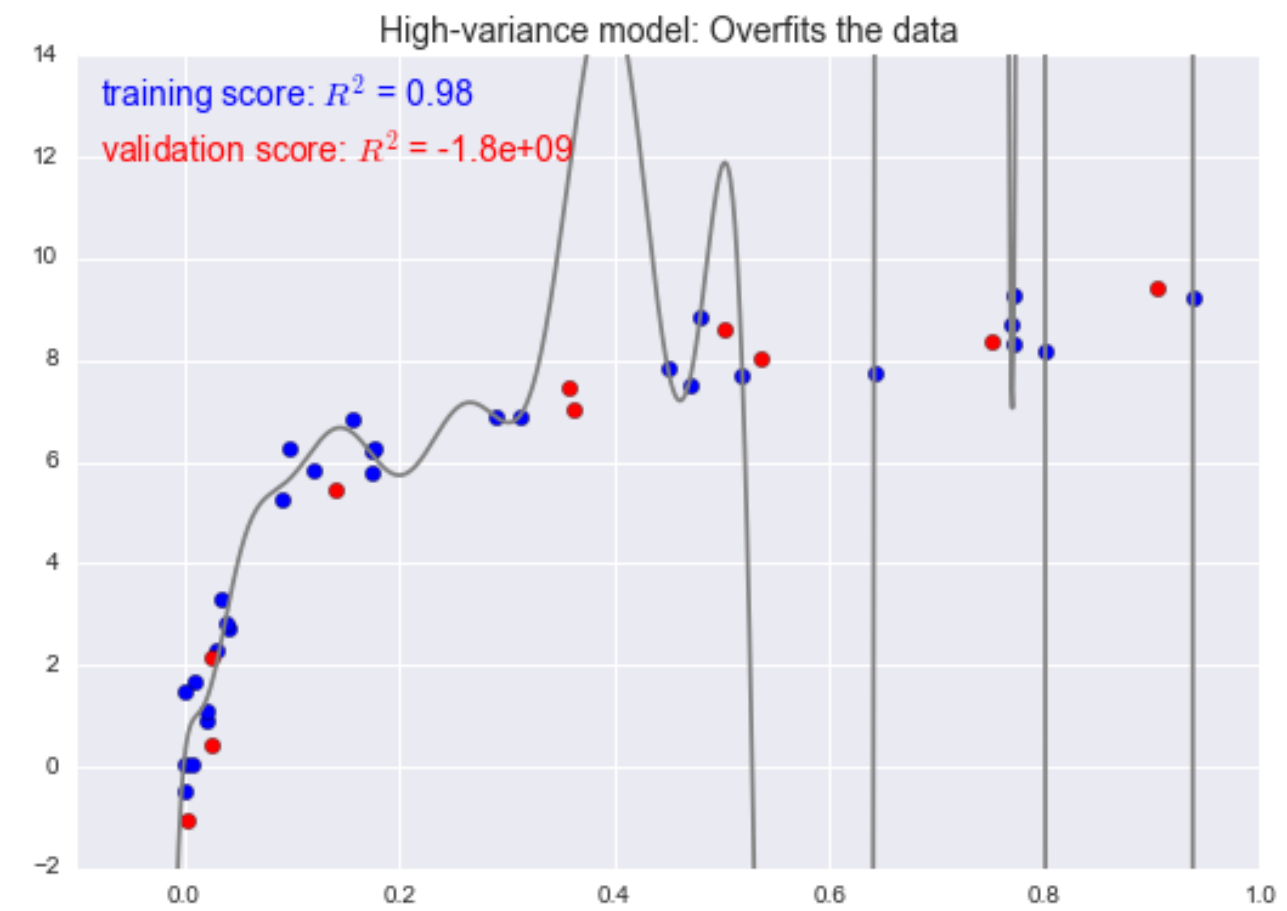
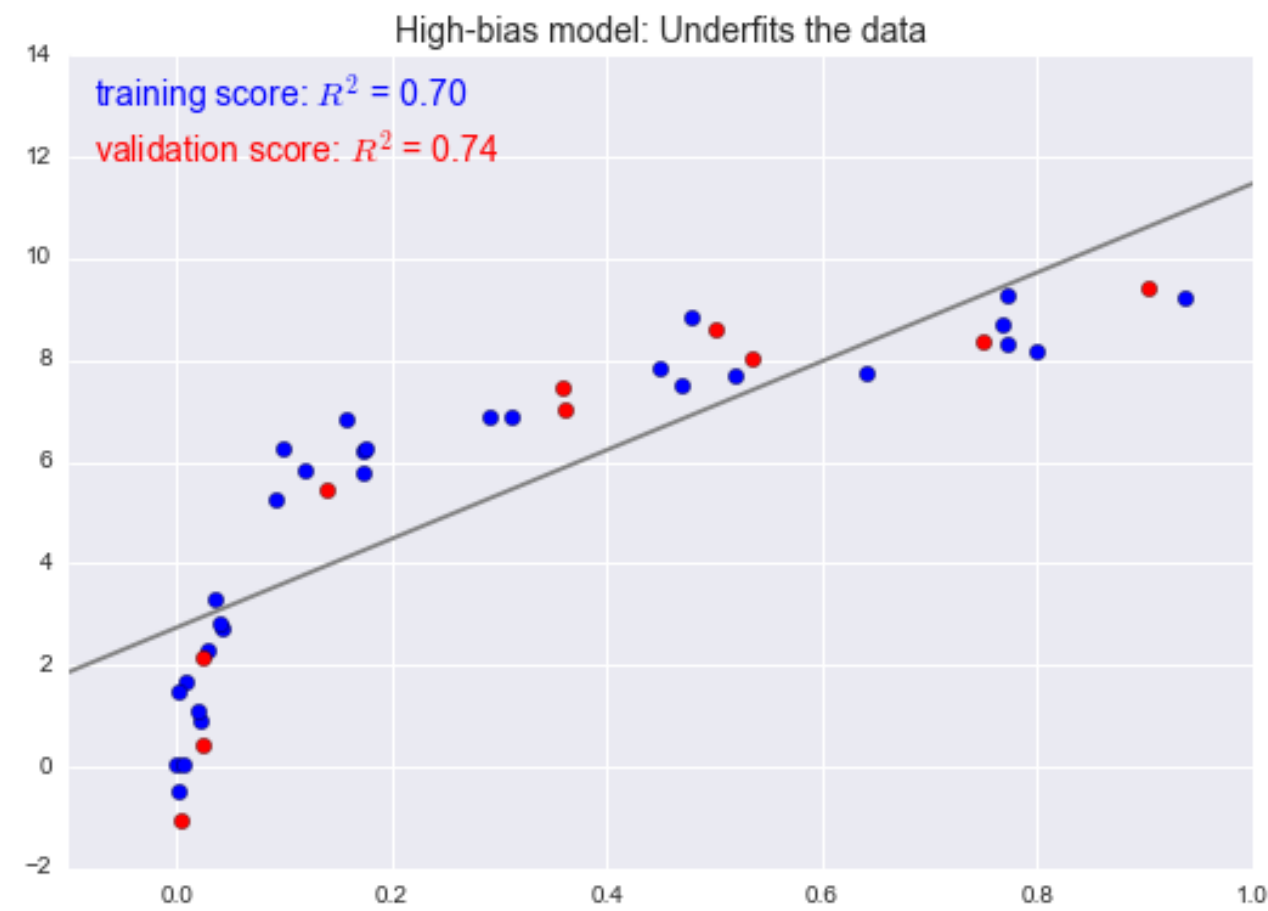
- **Overfitting:** poor generalization due to complexity
  - learning noise in training data
- **Underfitting:** poor generalization due to simplicity
  - not flexible enough to learn concept
- Need to find a balance between simplicity and complexity
- Need to find a balance between **bias** and **variance**

# Bias-Variance Tradeoff



From PDSH

# Bias-Variance Tradeoff



From PDSH

# Bias-Variance Tradeoff Continued

- How close is the model to the underlying concept?
- How sensitive is the model to the training set?

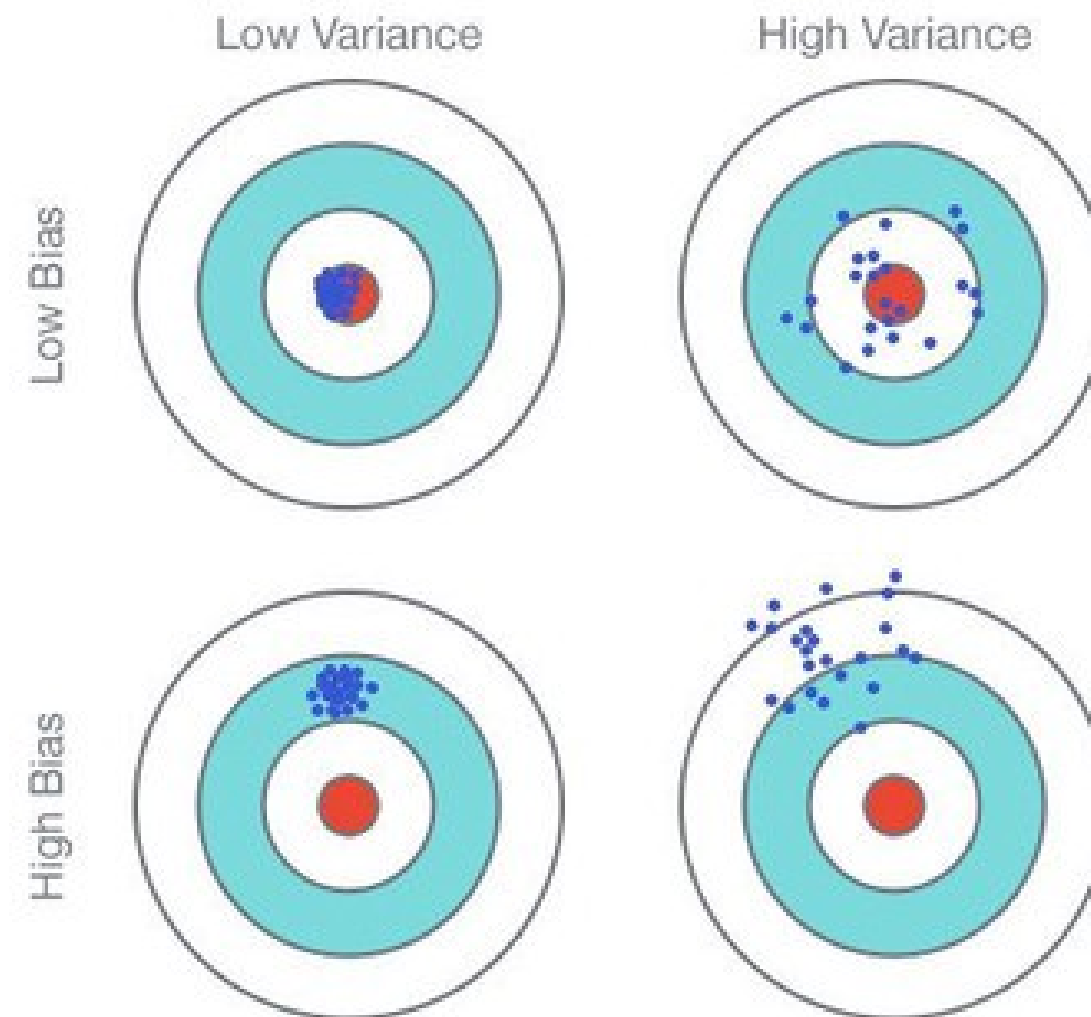
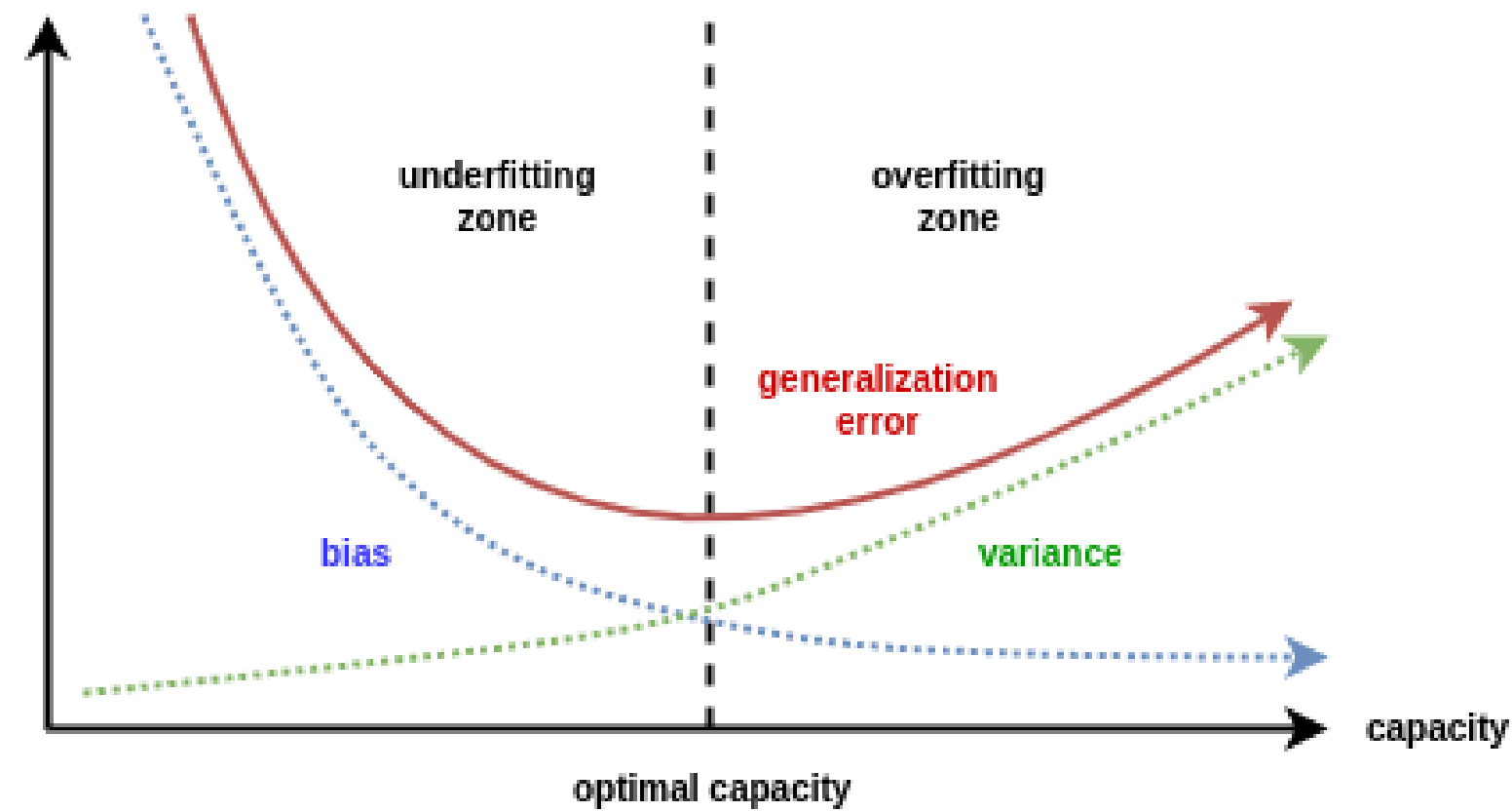


Fig. 1: Graphical Illustration of bias-variance trade-off , Source: Scott Fortmann-Roe., Understanding Bias-Variance Trade-off



# Bias-Variance Tradeoff Continued



- We'd like to:
  - reduce the Bias (use a model complex enough to capture the concept)
  - without introducing too much Variance (overfit the data)
  - all in order to minimize **Generalization Error**

# Overfitting/Underfitting Revisited

- **Overfitting:** poor generalization due to complexity
  - learning noise in training data
  - model has **high variance and low bias**
- **Underfitting:** poor generalization due to simplicity
  - not flexible enough to learn concept
  - model has **high bias and low variance**

# Avoiding Overfitting/Underfitting

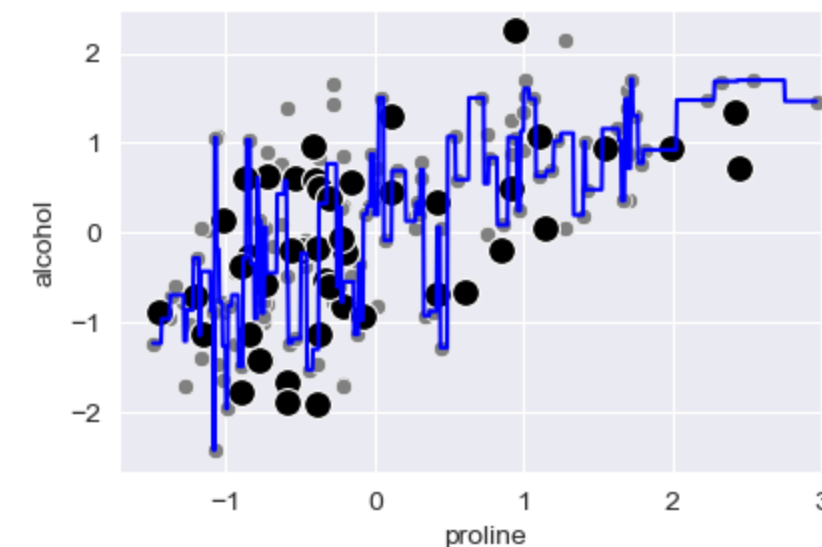
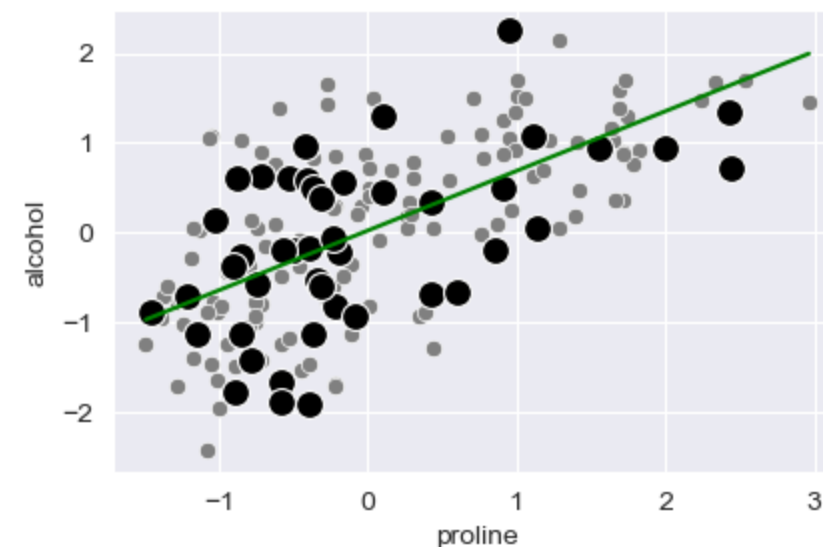
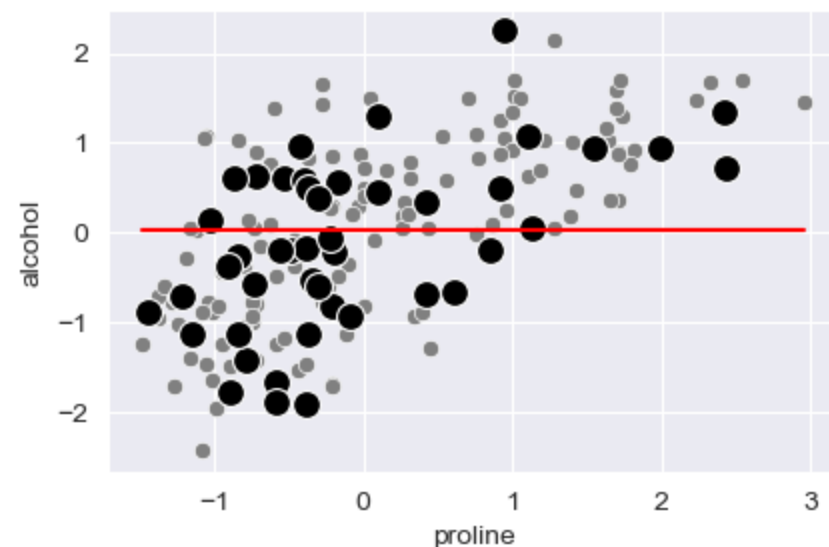
# Avoiding Overfitting/Underfitting

- Never train and evaluate on the same set of data!
  - train test split
  - **cross-validation**
- Rule of thumb: keep the model as simple as possible (Occom's Razor)

# Avoiding Overfitting/Underfitting

- Never train and evaluate on the same set of data!
  - train test split
  - **cross-validation**
- Rule of thumb: keep the model as simple as possible (Occom's Razor)

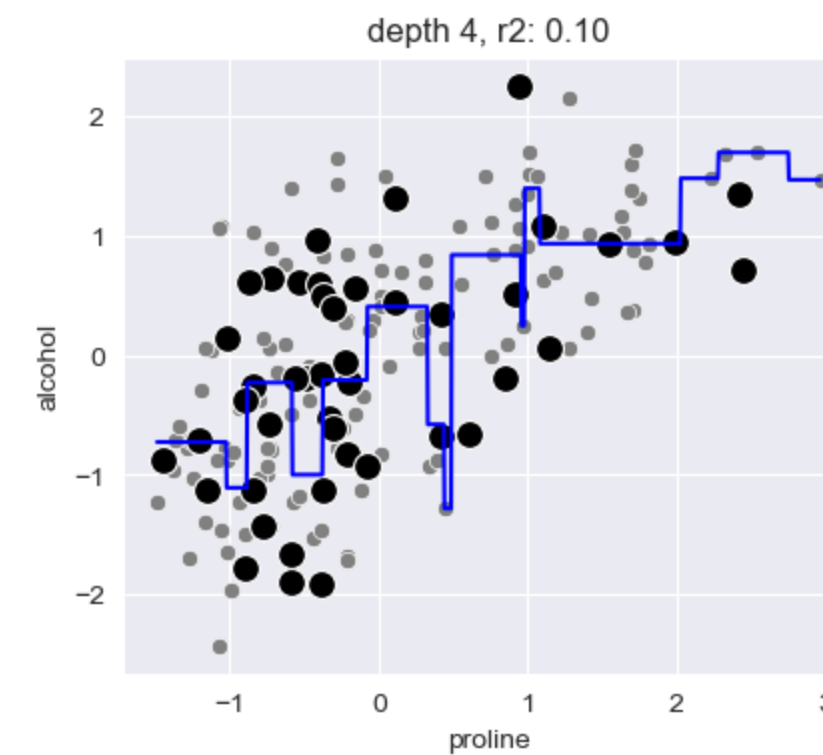
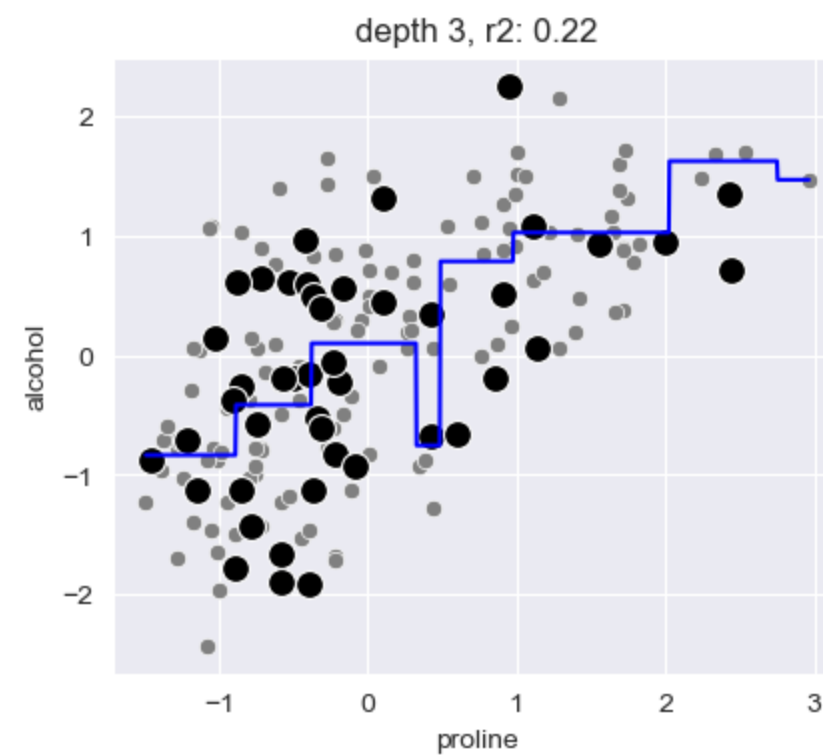
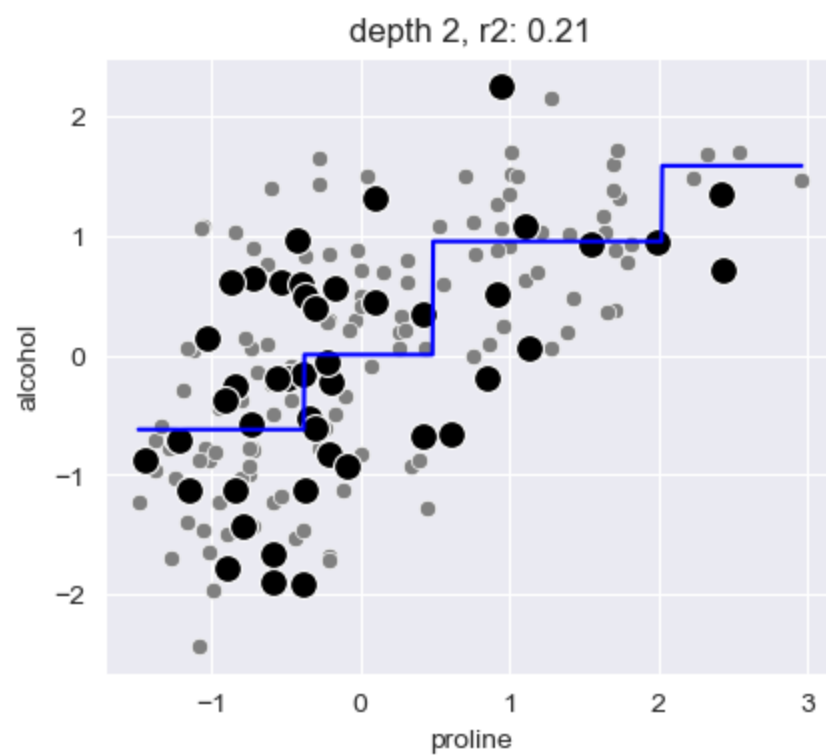
```
In [12]: 1 fig,ax = plt.subplots(1,3,figsize=(16,3))
2         for i in range(3):
3             sns.scatterplot(x=X_train_r.proline,y=y_train_r,color="gray",ax=ax[i])
4             sns.scatterplot(x=X_test_r.proline,y=y_test_r,color="black",s=100,ax=ax[i]);
5         ax[0].plot(X_query,dummys.predict(X_query),color='r');
6         ax[1].plot(X_query,lr.predict(X_query),color='g');
7         ax[2].plot(X_query,dtr.predict(X_query),color='b');
```



**Overfitting? Simplify the model**

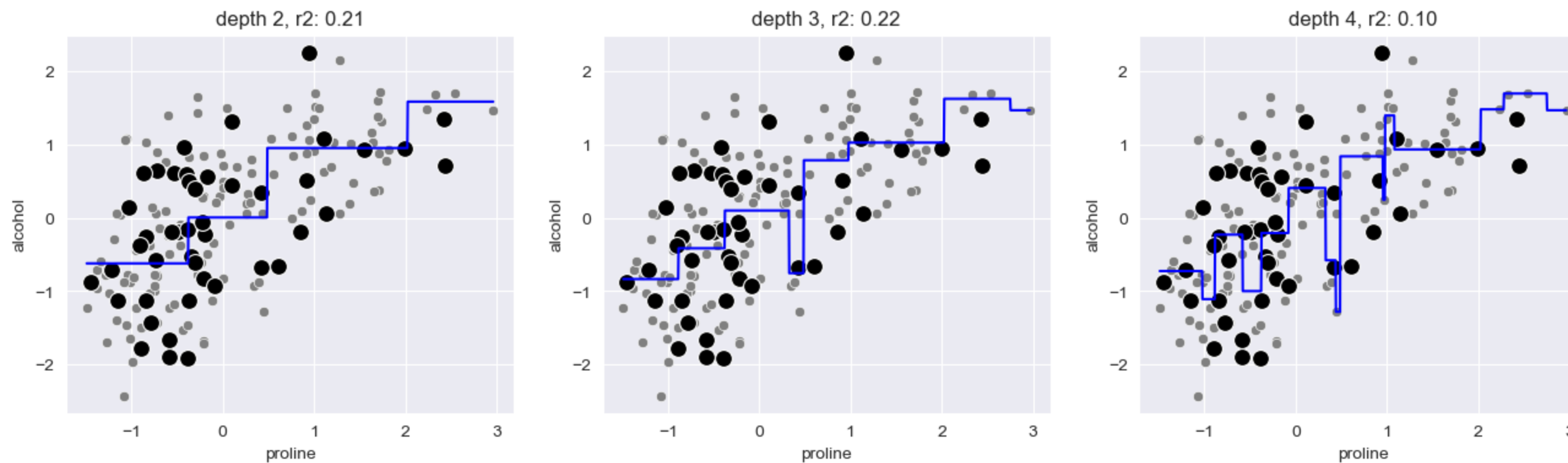
# Overfitting? Simplify the model

```
In [13]: 1 max_depths = [2,3,4]
2 fig,ax = plt.subplots(1,3,figsize=(16,4))
3 for i in range(3):
4     dtr_tmp = DecisionTreeRegressor(max_depth=max_depths[i]).fit(X_train_r,y_train_r)
5     sns.scatterplot(x=X_train_r.proline,y=y_train_r,color="gray",ax=ax[i])
6     sns.scatterplot(x=X_test_r.proline,y=y_test_r,color="black",s=100,ax=ax[i]);
7     ax[i].plot(X_query,dtr_tmp.predict(X_query),color='b');
8     ax[i].set_title(f'depth {max_depths[i]}, r2: {dtr_tmp.score(X_test_r,y_test_r):0.2f}')
```



# Overfitting? Simplify the model

```
In [13]: 1 max_depths = [2,3,4]
2 fig,ax = plt.subplots(1,3,figsize=(16,4))
3 for i in range(3):
4     dtr_tmp = DecisionTreeRegressor(max_depth=max_depths[i]).fit(X_train_r,y_train_r)
5     sns.scatterplot(x=X_train_r.proline,y=y_train_r,color="gray",ax=ax[i])
6     sns.scatterplot(x=X_test_r.proline,y=y_test_r,color="black",s=100,ax=ax[i]);
7     ax[i].plot(X_query,dtr_tmp.predict(X_query),color='b');
8     ax[i].set_title(f'depth {max_depths[i]}, r2: {dtr_tmp.score(X_test_r,y_test_r):0.2f}')
```



- But now we might be overfitting on the test set!
- How to choose hyperparameters: **Cross-Validation**



## Aside: Hyperparameters

- **parameter:** something learned by the model itself (eg. coefficient in linear model)
- **hyperparameter:** something we set by hand (eg. decision tree max depth)

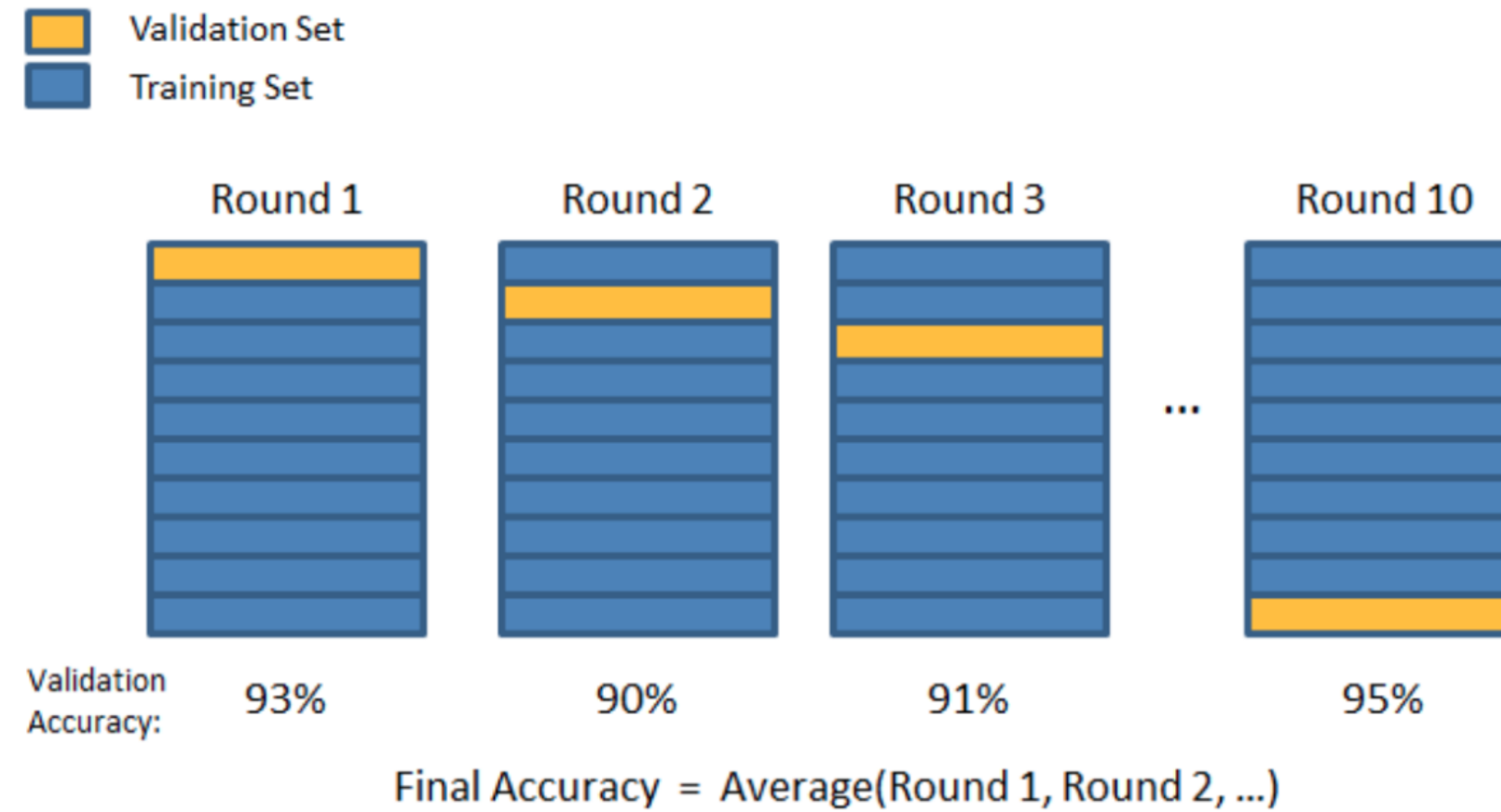
# $k$ -Fold Cross-Validation

1. split dataset into  $k$  equal sized subsets (folds)
2. for each subset (fold)
  - train on the other  $k - 1$  subsets combined
  - test on this subset to get a score
3. average across all scores

# $k$ -Fold Cross-Validation

1. split dataset into  $k$  equal sized subsets (folds)
  2. for each subset (fold)
    - train on the other  $k - 1$  subsets combined
    - test on this subset to get a score
  3. average across all scores
- Result is a set of samples of model performance
  - Can use to set hyperparameters without overfitting on train or test
  - Can also use to estimate range of generalization performance

# Example: 10-Fold Cross-Validation



# k-Fold Cross-Validation Continued

- Can be used for:
  - tuning hyperparameters
  - model selection
  - any time we need estimate of model performance
- **Issue:** each fold requires training the model
  - Training time can be an issue for large  $k$  or models with long training time
- What values can  $k$  take?
  - min: 2
  - max:  $n$ , the size of the dataset (aka Leave-One-Out CV)

# k-Fold Cross-Validation in sklearn

# k-Fold Cross-Validation in sklearn

```
In [14]: 1 from sklearn.model_selection import cross_val_score
          2
          3 scores = cross_val_score(DecisionTreeRegressor(max_depth=2),
          4                             X_train_r,
          5                             y_train_r,
          6                             n_jobs=-1, # default None == 1
          7                             cv=5, # default
          8                             )
          9 scores.round(2)
```

```
Out[14]: array([0.42, 0.37, 0.45, 0.26, 0.23])
```

# k-Fold Cross-Validation in sklearn

```
In [14]: 1 from sklearn.model_selection import cross_val_score
          2
          3 scores = cross_val_score(DecisionTreeRegressor(max_depth=2),
          4                             X_train_r,
          5                             y_train_r,
          6                             n_jobs=-1, # default None == 1
          7                             cv=5, # default
          8                             )
          9 scores.round(2)
```

```
Out[14]: array([0.42, 0.37, 0.45, 0.26, 0.23])
```

```
In [15]: 1 print(f'{np.mean(scores).round(2)} +- {2*np.std(scores).round(2)} ' , ' (mean +- std)')

          0.35 +- 0.18    (mean +- std)
```



# Tuning Hyperparameters with CV

# Tuning Hyperparameters with CV

```
In [16]: 1 mean_scores = []
          2
          3 for depth in [1, 2, 3, 5, 10]:
          4     dtr_tmp = DecisionTreeRegressor(max_depth=depth)
          5     scores = cross_val_score(dtr_tmp,X_train_r,y_train_r,cv=5)
          6     mean_scores.append( (depth, scores.mean().round(3)) )
          7
          8 for depth,mean_score in mean_scores:
          9     print(f'{depth = :2d} : {mean_score: .3f}')
```

```
depth = 1 : 0.289
depth = 2 : 0.346
depth = 3 : 0.341
depth = 5 : 0.050
depth = 10 : -0.142
```

# Tuning Hyperparameters with CV

```
In [16]: 1 mean_scores = []
2
3 for depth in [1, 2, 3, 5, 10]:
4     dtr_tmp = DecisionTreeRegressor(max_depth=depth)
5     scores = cross_val_score(dtr_tmp,X_train_r,y_train_r,cv=5)
6     mean_scores.append( (depth, scores.mean().round(3)) )
7
8 for depth,mean_score in mean_scores:
9     print(f'{depth = :2d} : {mean_score: .3f}')
```

```
depth = 1 : 0.289
depth = 2 : 0.346
depth = 3 : 0.341
depth = 5 : 0.050
depth = 10 : -0.142
```

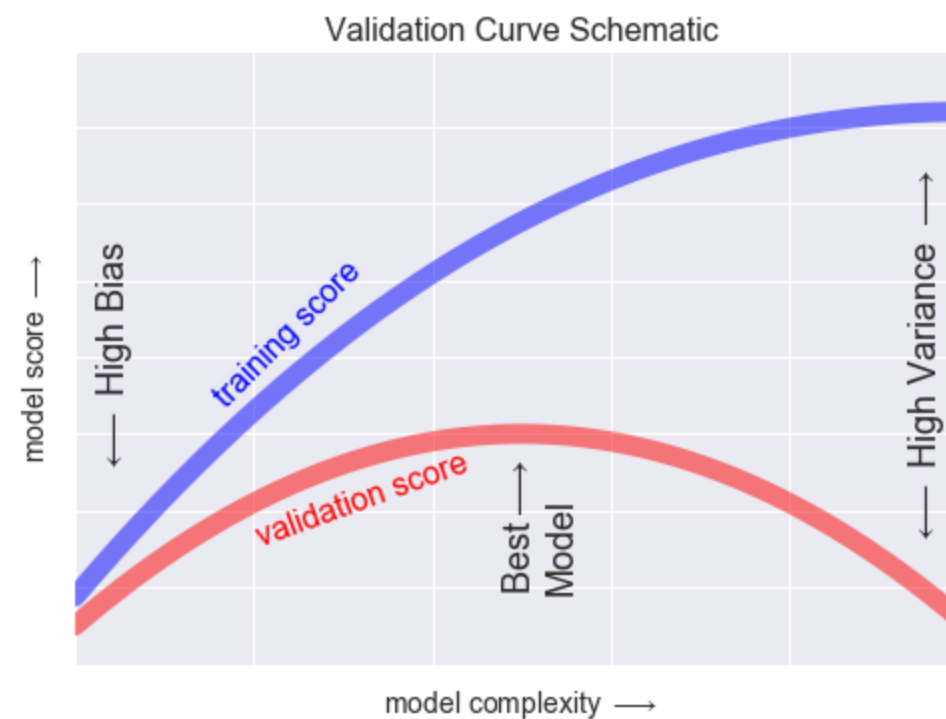
```
In [17]: 1 # find the depth that gives best score (highest R^2)
2 best_depth,best_score = sorted(mean_scores, key=lambda x:x[1],reverse=True)[0] # sorted is ascending by default
3 print(f'{best_depth = :2d} : {best_score = : .3f}')
```

```
best_depth = 2 : best_score = 0.346
```

# Visualize Tuning: Validation Curve

## Validation Curve

- Show model complexity vs model performance on both train and test/validation
- Want to find point where performance on validation set begins to decline (*elbow rule*)



From PDSH

# Validation Curve in sklearn

# Validation Curve in sklearn

```
In [18]: 1 from sklearn.model_selection import validation_curve
          2
          3 depth = [1,2,3,5,8,10]
          4 train_scores, test_scores = validation_curve(DecisionTreeRegressor(),
          5                                           X_train_r, y_train_r,
          6                                           param_name='max_depth',
          7                                           param_range=depth,
          8                                           cv=3)
          9 train_scores.round(2)
```

```
Out[18]: array([[0.35, 0.42, 0.45],
                [0.48, 0.49, 0.51],
                [0.52, 0.53, 0.56],
                [0.68, 0.69, 0.65],
                [0.79, 0.83, 0.76],
                [0.82, 0.84, 0.81]])
```

# Validation Curve in sklearn

```
In [18]: 1 from sklearn.model_selection import validation_curve
2
3 depth = [1,2,3,5,8,10]
4 train_scores, test_scores = validation_curve(DecisionTreeRegressor(),
5                                             X_train_r, y_train_r,
6                                             param_name='max_depth',
7                                             param_range=depth,
8                                             cv=3)
9 train_scores.round(2)
```

```
Out[18]: array([[0.35, 0.42, 0.45],
                [0.48, 0.49, 0.51],
                [0.52, 0.53, 0.56],
                [0.68, 0.69, 0.65],
                [0.79, 0.83, 0.76],
                [0.82, 0.84, 0.81]])
```

```
In [19]: 1 test_scores.round(2)
```

```
Out[19]: array([[0.41, 0.31, 0.17],
                [0.33, 0.42, 0.16],
                [0.29, 0.4 , 0.22],
                [0.2 , 0.25, 0.24],
                [0.17, 0.04, 0.12],
                [0.24, 0.04, 0.03]])
```

# Validation Curve in sklearn

```
In [18]: 1 from sklearn.model_selection import validation_curve
2
3 depth = [1,2,3,5,8,10]
4 train_scores,test_scores = validation_curve(DecisionTreeRegressor(),
5                                           X_train_r, y_train_r,
6                                           param_name='max_depth',
7                                           param_range=depth,
8                                           cv=3)
9 train_scores.round(2)
```

```
Out[18]: array([[0.35, 0.42, 0.45],
                [0.48, 0.49, 0.51],
                [0.52, 0.53, 0.56],
                [0.68, 0.69, 0.65],
                [0.79, 0.83, 0.76],
                [0.82, 0.84, 0.81]])
```

```
In [19]: 1 test_scores.round(2)
```

```
Out[19]: array([[0.41, 0.31, 0.17],
                [0.33, 0.42, 0.16],
                [0.29, 0.4 , 0.22],
                [0.2 , 0.25, 0.24],
                [0.17, 0.04, 0.12],
                [0.24, 0.04, 0.03]])
```

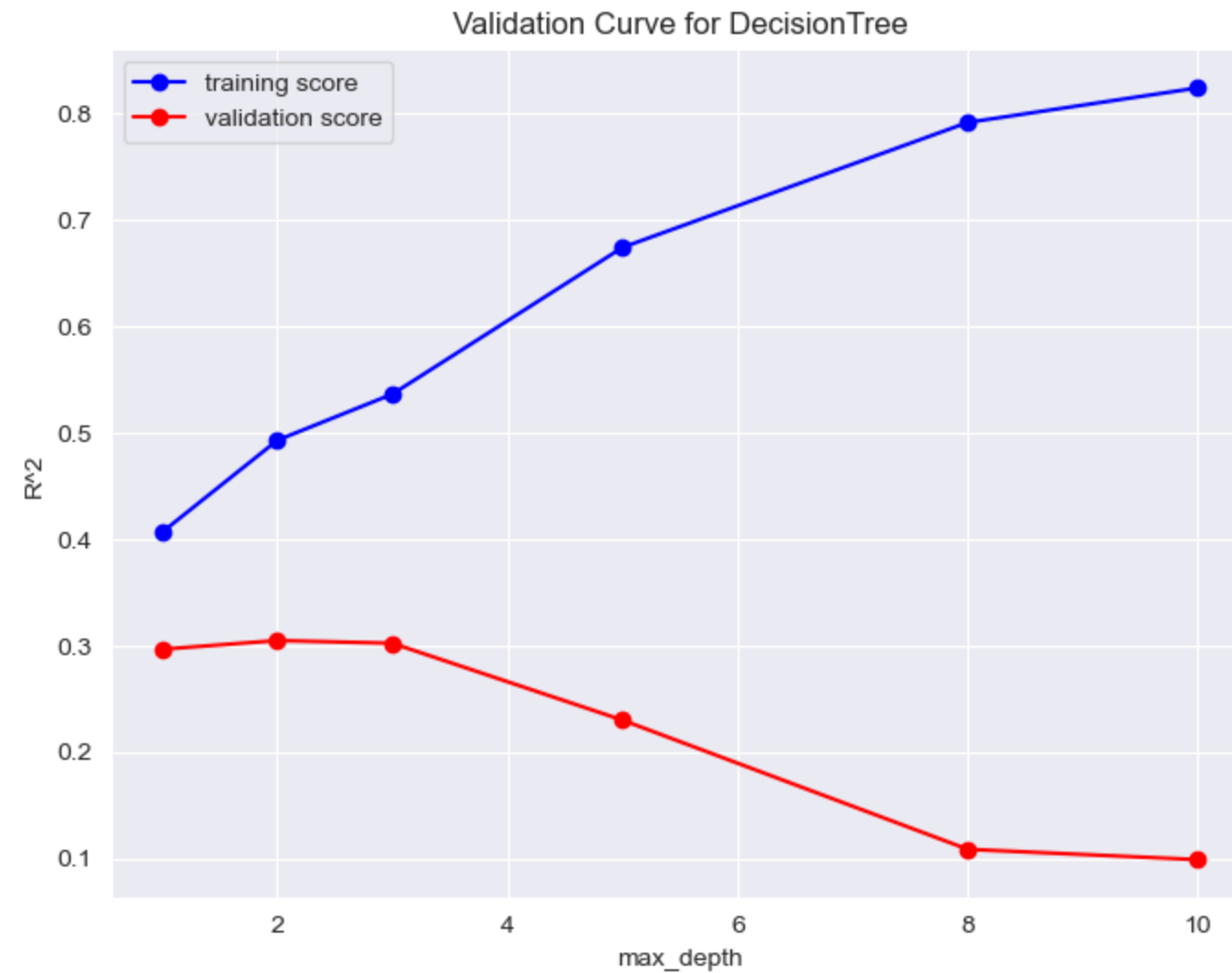
```
In [20]: 1 mean_train_scores = np.mean(train_scores,axis=1) # take the mean across columns
2 mean_test_scores = np.mean(test_scores,axis=1)
```



# Validation Curve in sklearn Continued

# Validation Curve in sklearn Continued

```
In [21]: 1 fig,ax = plt.subplots(1,1,figsize=(8,6))
2 ax.plot(depth, mean_train_scores, 'o-', color='b',label='training score');
3 ax.plot(depth, mean_test_scores, 'o-', color='r', label='validation score');
4 ax.set_xlabel('max_depth'), ax.set_ylabel('R^2'); ax.set_title('Validation Curve for DecisionTree');
5 ax.legend();
```



# More Than One HyperParameter? Grid Search

**Grid Search:** Search over a 'grid' of hyperparameter settings

Example: KNN "number of neighbors" and "distance metric"

# More Than One HyperParameter? Grid Search

**Grid Search:** Search over a 'grid' of hyperparameter settings

Example: KNN "number of neighbors" and "distance metric"

```
In [22]: 1 distance_metrics = ['euclidean', 'manhattan']
          2 n_neighbors = [1, 3, 5]
          3
          4 grid = []
          5 for d in distance_metrics:
          6     for k in n_neighbors:
          7         print([d, k])
```

```
['euclidean', 1]
['euclidean', 3]
['euclidean', 5]
['manhattan', 1]
['manhattan', 3]
['manhattan', 5]
```

# Grid Search in sklearn

# Grid Search in sklearn

```
In [23]: 1 from sklearn.model_selection import GridSearchCV
2 from sklearn.neighbors import KNeighborsRegressor
3
4 params = {'n_neighbors':[1,2,3,5,10],
5           'metric':['euclidean','manhattan']}
6 gscv = GridSearchCV(KNeighborsRegressor(),
7                     param_grid=params,      # grid of size 10
8                     cv=3,                  # do 3-fold CV at every grid point
9                     refit=True)           # refit True trains one more time on the entire training set
10
11 gscv.fit(X_train_r,y_train_r)    # Q: How many times are we training a model here? (2*5*3 + 1 = 31)
12
13 print(gscv.best_params_)
```

```
{'metric': 'euclidean', 'n_neighbors': 5}
```

# Grid Search in sklearn

```
In [23]: 1 from sklearn.model_selection import GridSearchCV
2 from sklearn.neighbors import KNeighborsRegressor
3
4 params = {'n_neighbors':[1,2,3,5,10],
5           'metric':['euclidean','manhattan']}
6 gscv = GridSearchCV(KNeighborsRegressor(),
7                     param_grid=params,      # grid of size 10
8                     cv=3,                  # do 3-fold CV at every grid point
9                     refit=True)           # refit True trains one more time on the entire training set
10
11 gscv.fit(X_train_r,y_train_r)    # Q: How many times are we training a model here? (2*5*3 + 1 = 31)
12
13 print(gscv.best_params_)
```

```
{'metric': 'euclidean', 'n_neighbors': 5}
```

```
In [24]: 1 scores = cross_val_score(gscv.best_estimator_,X_train_r,y_train_r,cv=5)
2
3 print(f'{np.mean(scores).round(2):0.2f} +- {2*np.std(scores).round(2):0.2f}')
```

```
0.34 +- 0.26
```

# Review So Far

- Regression Metrics
  - MSE and RMSE
  - $R^2$
- Model Selection
  - Comparison to Baseline Model
  - Underfitting/Overfitting and Bias/Variance
  - Train/Test Split
- Hyperparameter Tuning
  - Cross-Validation
  - Validation Curve
  - Grid Search



# Data Setup for Classification

# Data Setup for Classification

```
In [25]: 1 idx_binary = df_wine['class'].isin([0,1])           # reduce to binary classification
          2
          3 X_bc = df_wine.loc[idx_binary,['alcalinity_of_ash','magnesium']]           # only 2 features for ease of plotting
          4 y_bc = df_wine.loc[idx_binary,'class']           # pull out classification target [0,1]
          5
          6 X_train_bc,X_test_bc,y_train_bc,y_test_bc = train_test_split(X_bc,
          7                                                                y_bc,
          8                                                                stratify=y_bc, # maintain label proportions
          9                                                                random_state=0
          10                                                                )
          11
          12 pd.DataFrame({'train':y_train_bc.value_counts(),'test':y_test_bc.value_counts()}).sort_index()
```

Out[25]:

	train	test
0	44	15
1	53	18

# Data Setup for Classification

```
In [25]: 1 idx_binary = df_wine['class'].isin([0,1])           # reduce to binary classification
2
3 X_bc = df_wine.loc[idx_binary,['alcalinity_of_ash','magnesium']]      # only 2 features for ease of plotting
4 y_bc = df_wine.loc[idx_binary,'class']           # pull out classification target [0,1]
5
6 X_train_bc,X_test_bc,y_train_bc,y_test_bc = train_test_split(X_bc,
7                                                                y_bc,
8                                                                stratify=y_bc, # maintain label proportions
9                                                                random_state=0
10                                                             )
11
12 pd.DataFrame({'train':y_train_bc.value_counts(),'test':y_test_bc.value_counts()}).sort_index()
```

Out[25]:

	train	test
0	44	15
1	53	18

```
In [26]: 1 X_mc = df_wine.loc[:,['alcalinity_of_ash', 'magnesium']]      # multiple features for multiclass classification task
2 y_mc = df_wine.loc[:, 'class']           # pull out classification target [0,1,2]
3 X_train_mc,X_test_mc,y_train_mc,y_test_mc = train_test_split(X_mc,
4                                                                y_mc,
5                                                                stratify=y_mc, # maintain label proportions
6                                                                random_state=123
7                                                             )
8 pd.DataFrame({'train':y_train_mc.value_counts(),'test':y_test_mc.value_counts()}).sort_values(by="train")
```

Out[26]:

	train	test
2	36	12
0	44	15
1	53	18

# Default Metric in Classification: Accuracy

- **Accuracy:** out of all the observations, how many did I get right?

# Default Metric in Classification: Accuracy

- **Accuracy:** out of all the observations, how many did I get right?

```
In [27]: 1 from sklearn.dummy import DummyClassifier
          2 from sklearn.tree import DecisionTreeClassifier
          3 dummyc = DummyClassifier(strategy='prior').fit(X_train_bc,y_train_bc) # works like 'most-frequent'
          4 dtc = DecisionTreeClassifier(max_depth=2).fit(X_train_bc,y_train_bc)
          5
          6 print(f'{dummyc.score(X_test_bc,y_test_bc) = :0.2f}') # default classification score is accuracy
          7 print(f'{dtc.score(X_test_bc,y_test_bc) = :0.2f}')
```

```
dummyc.score(X_test_bc,y_test_bc) = 0.55
dtc.score(X_test_bc,y_test_bc) = 0.79
```

# Default Metric in Classification: Accuracy

- **Accuracy:** out of all the observations, how many did I get right?

```
In [27]: 1 from sklearn.dummy import DummyClassifier
          2 from sklearn.tree import DecisionTreeClassifier
          3 dummyc = DummyClassifier(strategy='prior').fit(X_train_bc,y_train_bc) # works like 'most-frequent'
          4 dtc = DecisionTreeClassifier(max_depth=2).fit(X_train_bc,y_train_bc)
          5
          6 print(f'{dummyc.score(X_test_bc,y_test_bc) = :0.2f}') # default classification score is accuracy
          7 print(f'{dtc.score(X_test_bc,y_test_bc) = :0.2f}')
```

```
dummyc.score(X_test_bc,y_test_bc) = 0.55
dtc.score(X_test_bc,y_test_bc) = 0.79
```

- But what if the cost of calling a negative a positive is different from calling a positive a negative?
- Examples:
  - disease testing
  - medical product failures
  - incarceration

# Errors in Classification

- Just like hypothesis testing, there are different kinds of error in classification

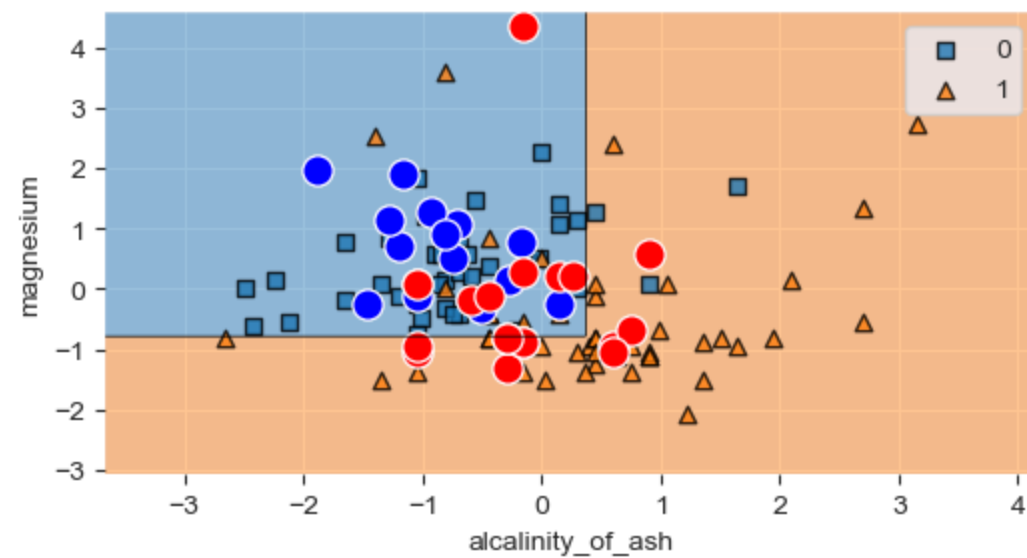
		Predicted class	
		P	N
Actual class	P	True positives (TP)	False negatives (FN)
	N	False positives (FP)	True negatives (TN)

# Visualizing Errors with a Confusion Matrix



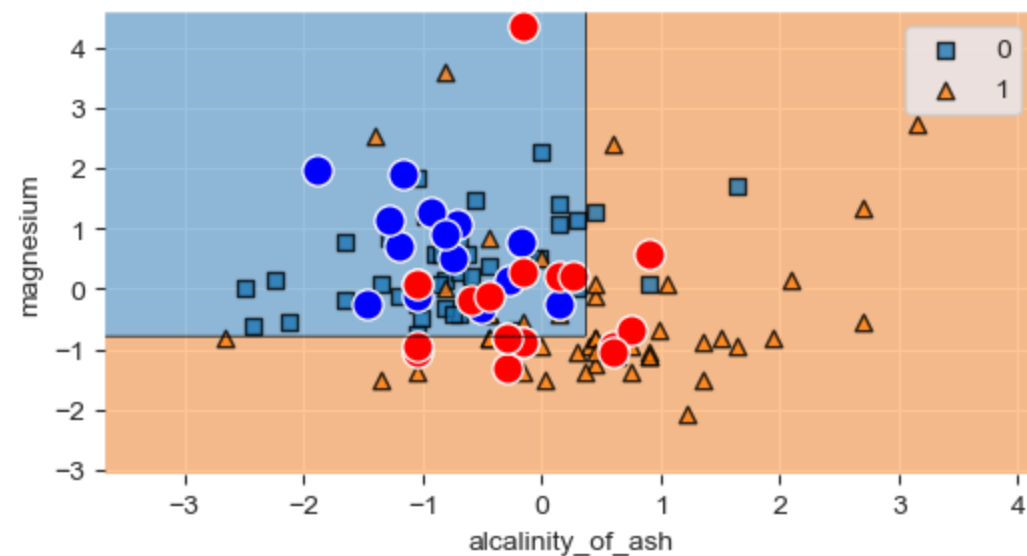
# Visualizing Errors with a Confusion Matrix

```
In [28]: 1 fig,ax = plt.subplots(1,1,figsize=(6,3));  
2 plot_decision_regions(X_train_bc.values,y_train_bc.values,dtc,ax=ax);  
3 sns.scatterplot(x=X_bc.columns[0],y=X_bc.columns[1],data=X_test_bc[y_test_bc == 0],color="blue",s=120);  
4 sns.scatterplot(x=X_bc.columns[0],y=X_bc.columns[1],data=X_test_bc[y_test_bc == 1],color="red",s=120);
```



# Visualizing Errors with a Confusion Matrix

```
In [28]: 1 fig,ax = plt.subplots(1,1,figsize=(6,3));
2 plot_decision_regions(X_train_bc.values,y_train_bc.values,dtc,ax=ax);
3 sns.scatterplot(x=X_bc.columns[0],y=X_bc.columns[1],data=X_test_bc[y_test_bc == 0],color="blue",s=120);
4 sns.scatterplot(x=X_bc.columns[0],y=X_bc.columns[1],data=X_test_bc[y_test_bc == 1],color="red",s=120);
```



```
In [29]: 1 from sklearn.metrics import confusion_matrix
2
3 print('training set error\n', confusion_matrix(y_train_bc,dtc.predict(X_train_bc)))
4 print()
5 print('test set error\n', confusion_matrix(y_test_bc,dtc.predict(X_test_bc)))
```

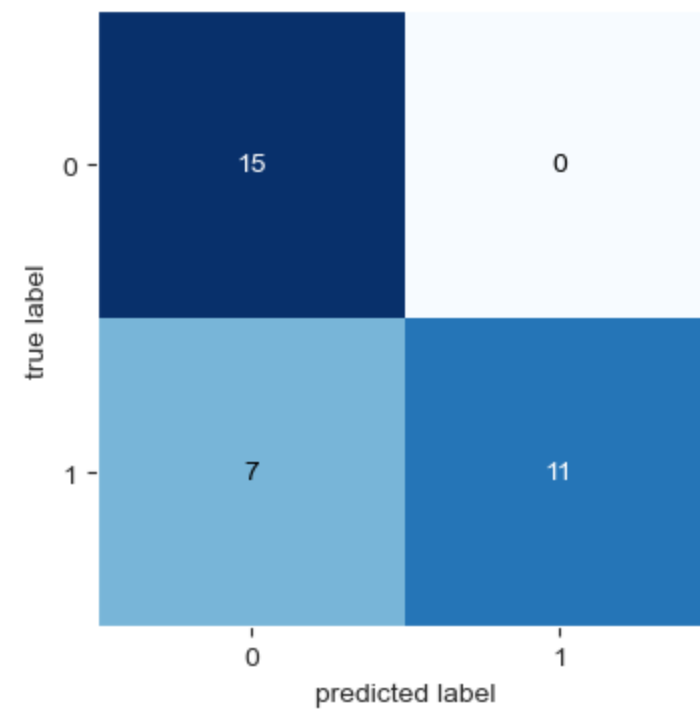
```
training set error
[[41  3]
 [14 39]]
```

```
test set error
[[15  0]
 [ 7 11]]
```

# Plot Confusion Matrix with mlxtend

# Plot Confusion Matrix with mlxtend

```
In [30]: 1 from mlxtend.plotting import plot_confusion_matrix  
2  
3 fig, ax = plt.subplots(1, 1, figsize=(4, 4))  
4 plot_confusion_matrix(confusion_matrix(y_test_bc, dtc.predict(X_test_bc)), axis=ax);
```



# Weighing Errors: Precision vs. Recall

## Precision

- Out of the observations I predicted positive (TP+FP), how many are truly positive (TP)?

$$precision = \frac{TP}{TP+FP}$$

## Recall

- Out of the truly positive (TP+FN), how many observations did I predict positive (TP)?

$$recall = \frac{TP}{TP+FN}$$

# Using Other Measures in sklearn

# Using Other Measures in sklearn

```
In [31]: 1 dummyc_precision_scores = cross_val_score(dummyc,X_train_bc,y_train_bc,cv=5,scoring='precision')
2 dummyc_recall_scores      = cross_val_score(dummyc,X_train_bc,y_train_bc,cv=5,scoring='recall')
3
4 print(f'dummy precision: {np.mean(dummyc_precision_scores):0.2f} +- {2*np.std(dummyc_precision_scores):0.2f}')
5 print(f'dummy recall    : {np.mean(dummyc_recall_scores):0.2f} +- {2*np.std(dummyc_recall_scores):0.2f}')
6 print()
7
8 dtc_precision_scores = cross_val_score(dtc,X_train_bc,y_train_bc,cv=5,scoring='precision')
9 dtc_recall_scores    = cross_val_score(dtc,X_train_bc,y_train_bc,cv=5,scoring='recall')
10
11 print(f'dtc precision   : {np.mean(dtc_precision_scores):0.2f} +- {2*np.std(dtc_precision_scores):0.2f}')
12 print(f'dtc recall     : {np.mean(dtc_recall_scores):0.2f} +- {2*np.std(dtc_recall_scores):0.2f}')
```

```
dummy precision: 0.55 +- 0.04
dummy recall    : 1.00 +- 0.00
```

```
dtc precision   : 0.85 +- 0.36
dtc recall      : 0.66 +- 0.38
```

# How do we decide if something is positive or negative?

Usually set a threshold :

$$\hat{y}_i = \begin{cases} 1 & \text{if } P(y_i = 1 | x_i) > \text{threshold,} \\ 0 & \text{o.w.} \end{cases}$$

Usually, threshold = .5, but it doesn't have to be.

What happens if we change it?

- High threshold → High Precision, Low Recall
- Low threshold → High Recall, Low Precision



# Combining Precision and Recall: $F_1$ -score

Usually, we just want one number to optimize

$F_1$  -score: harmonic mean of precision and recall

- eg. weighted average of the precision and recall

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Note that  $F_1$  ignores True Negatives!

# Combining Precision and Recall: $F_1$ -score

Usually, we just want one number to optimize

$F_1$ -score: harmonic mean of precision and recall

- eg. weighted average of the precision and recall

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

Note that  $F_1$  ignores True Negatives!

```
In [32]: 1 dummyc_f1_scores = cross_val_score(dummyc,X_train_bc,y_train_bc,cv=5,scoring='f1')
          2 dtc_f1_scores = cross_val_score(dtc,X_train_bc,y_train_bc,cv=5,scoring='f1')
          3 print(f'dummyc f1 = {np.mean(dummyc_f1_scores):0.2f} +- {2*np.std(dummyc_f1_scores):0.2f}')
          4 print(f'    dtc f1 = {np.mean(dtc_f1_scores):0.2f} +- {2*np.std(dtc_f1_scores):0.2f}')
```

```
dummyc f1 = 0.71 +- 0.03
    dtc f1 = 0.72 +- 0.31
```

# Paying attention to True Negatives: ROC

## Receiver Operating Characteristic

- displays FPR vs TPR

$$\text{False Positive Rate (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}} = \frac{\text{negatives we got wrong}}{\text{all negatives}}$$

$$\text{True Positive Rate (TPR)} = \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{positives we got right}}{\text{all positives}}$$

How do these change as we move our threshold?

# Paying attention to True Negatives: ROC

## Receiver Operating Characteristic

- displays FPR vs TPR

$$\text{False Positive Rate (FPR)} = \frac{\text{FP}}{\text{FP} + \text{TN}} = \frac{\text{negatives we got wrong}}{\text{all negatives}}$$

$$\text{True Positive Rate (TPR)} = \text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{positives we got right}}{\text{all positives}}$$

How do these change as we move our threshold?

```
In [33]: 1 from sklearn.metrics import roc_curve
          2 fpr_dtc, tpr_dtc, thresholds = roc_curve(y_train_bc, dtc.predict_proba(X_train_bc)[:,-1])
          3
          4 print('thresholds: ', thresholds.round(2))
          5 print('False Positive Rates: ', fpr_dtc.round(2))
          6 print('True Positive Rates: ', tpr_dtc.round(2))
```

```
thresholds: [2.  1.  0.75 0.25]
False Positive Rates: [0.  0.  0.07 1.  ]
True Positive Rates: [0.  0.57 0.74 1.  ]
```

# Plotting ROC Curves

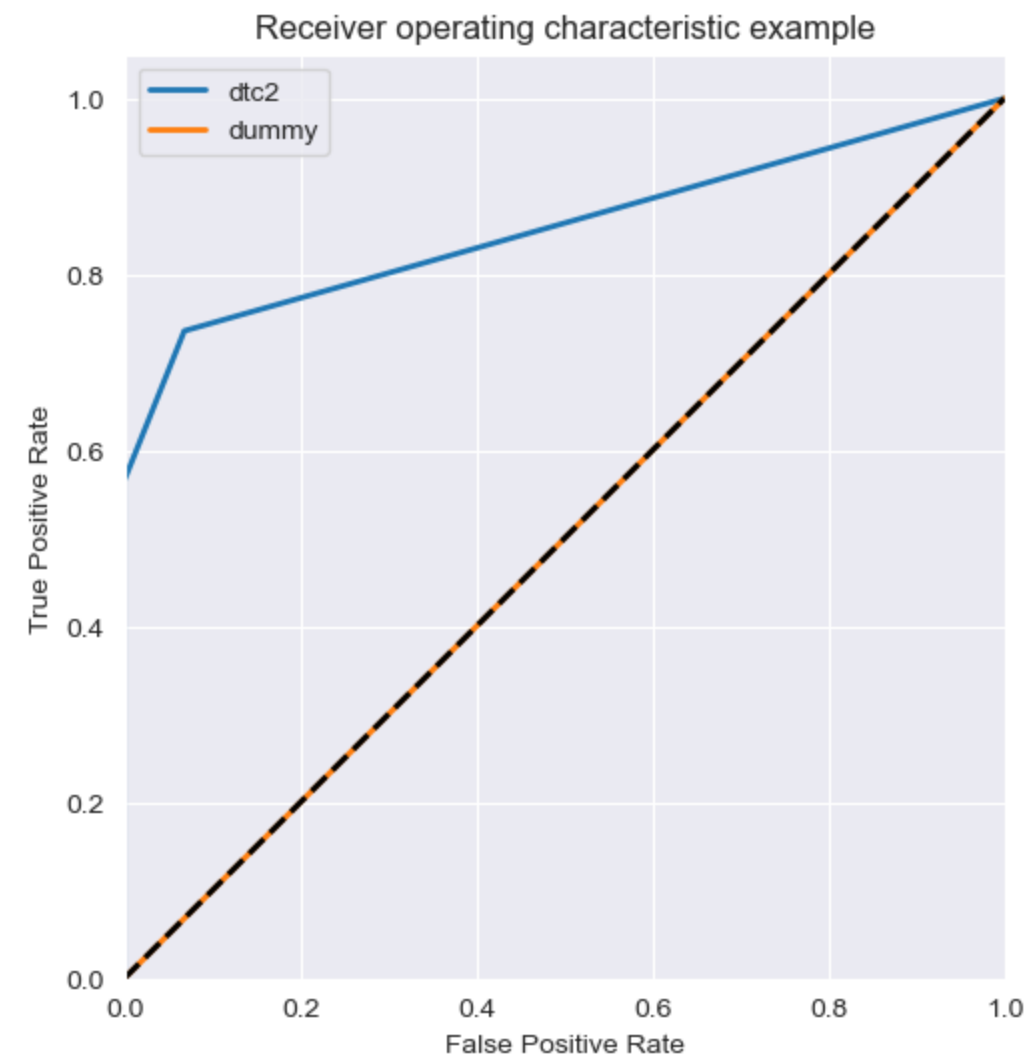
# Plotting ROC Curves

```
In [34]: 1 def plot_roc(curves):
2         fig, ax = plt.subplots(1, 1, figsize=(6, 6))
3         lw = 2
4         for fpr, tpr, model_name in curves:
5             ll, = ax.plot(fpr, tpr, lw=lw, label=model_name)
6         ax.plot([0, 1], [0, 1], color='k', lw=lw, linestyle='--')
7         ax.set_xlim([0.0, 1.0])
8         ax.set_ylim([0.0, 1.05])
9         ax.set_xlabel('False Positive Rate')
10        ax.set_ylabel('True Positive Rate')
11        ax.set_aspect('equal', 'box')
12        ax.set_title('Receiver operating characteristic example')
13        ax.legend()
```

# Plotting ROC Curves

# Plotting ROC Curves

```
In [35]: 1 curves = [(fpr_dtc, tpr_dtc, 'dtc2')]
2 fpr_dummyc, tpr_dummyc, _ = roc_curve(y_train_bc, dummyc.predict_proba(X_train_bc)[: , 1]) # Compare dummy
3 curves.append((fpr_dummyc, tpr_dummyc, 'dummy'));
4 plot_roc(curves);
```

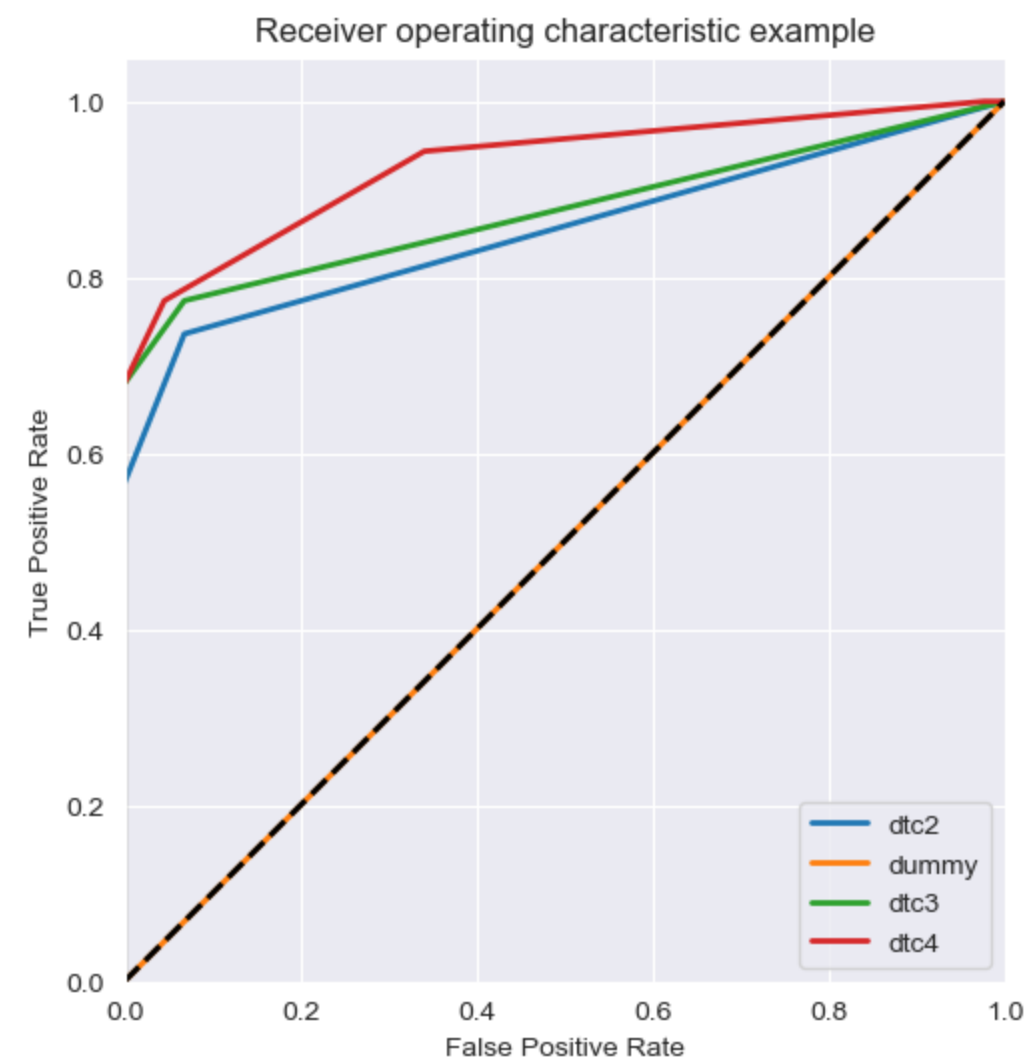




# Plotting ROC Curves

# Plotting ROC Curves

```
In [36]: 1 for depth in [3,4]:
2         fpr, tpr, _ = roc_curve(y_train_bc,
3                                 DecisionTreeClassifier(max_depth=depth).fit(X_train_bc,y_train_bc).predict_proba(X_train_bc)[: ,1])
4         curves.append((fpr,tpr,'dtc'+str(depth)))
5 plot_roc(curves);
```



# ROC AUC

- But again, we'd like one number to optimize
- ROC **A**rea **U**nder the **C**urve
  - How much area falls under the ROC curve?

# ROC AUC

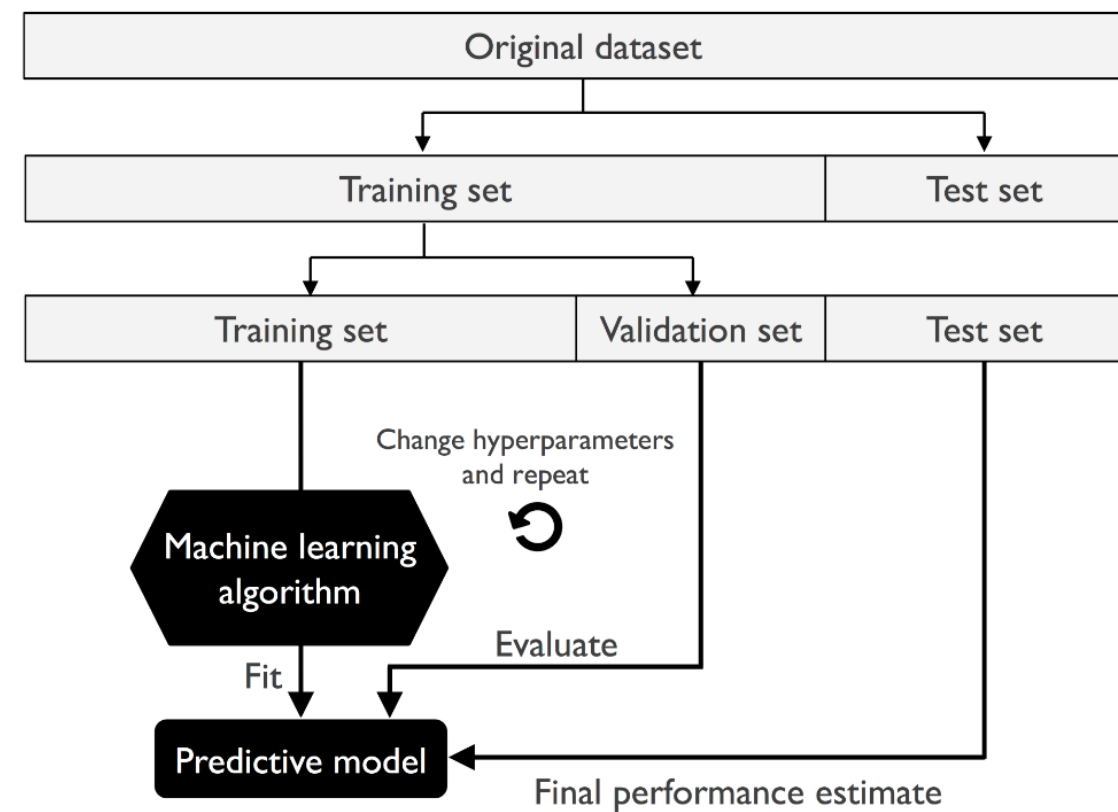
- But again, we'd like one number to optimize
- ROC Area **U**nder the **C**urve
  - How much area falls under the ROC curve?

```
In [37]: 1 dummyc_rocauc_scores = cross_val_score(dummyc,X_train_bc,y_train_bc,cv=5,scoring='roc_auc')
          2 dtc_rocauc_scores = cross_val_score(dtc,X_train_bc,y_train_bc,cv=5,scoring='roc_auc')
          3
          4 print(f'dummyc rocauc = {np.mean(dummyc_rocauc_scores).round(2):0.2f} +- {2*np.std(dummyc_rocauc_scores).round(2):0.2f}')
          5 print(f'dtc rocauc      = {np.mean(dtc_rocauc_scores).round(2):0.2f} +- {2*np.std(dtc_rocauc_scores).round(2):0.2f}')
```

```
dummyc rocauc = 0.50 +- 0.00
dtc rocauc     = 0.78 +- 0.22
```

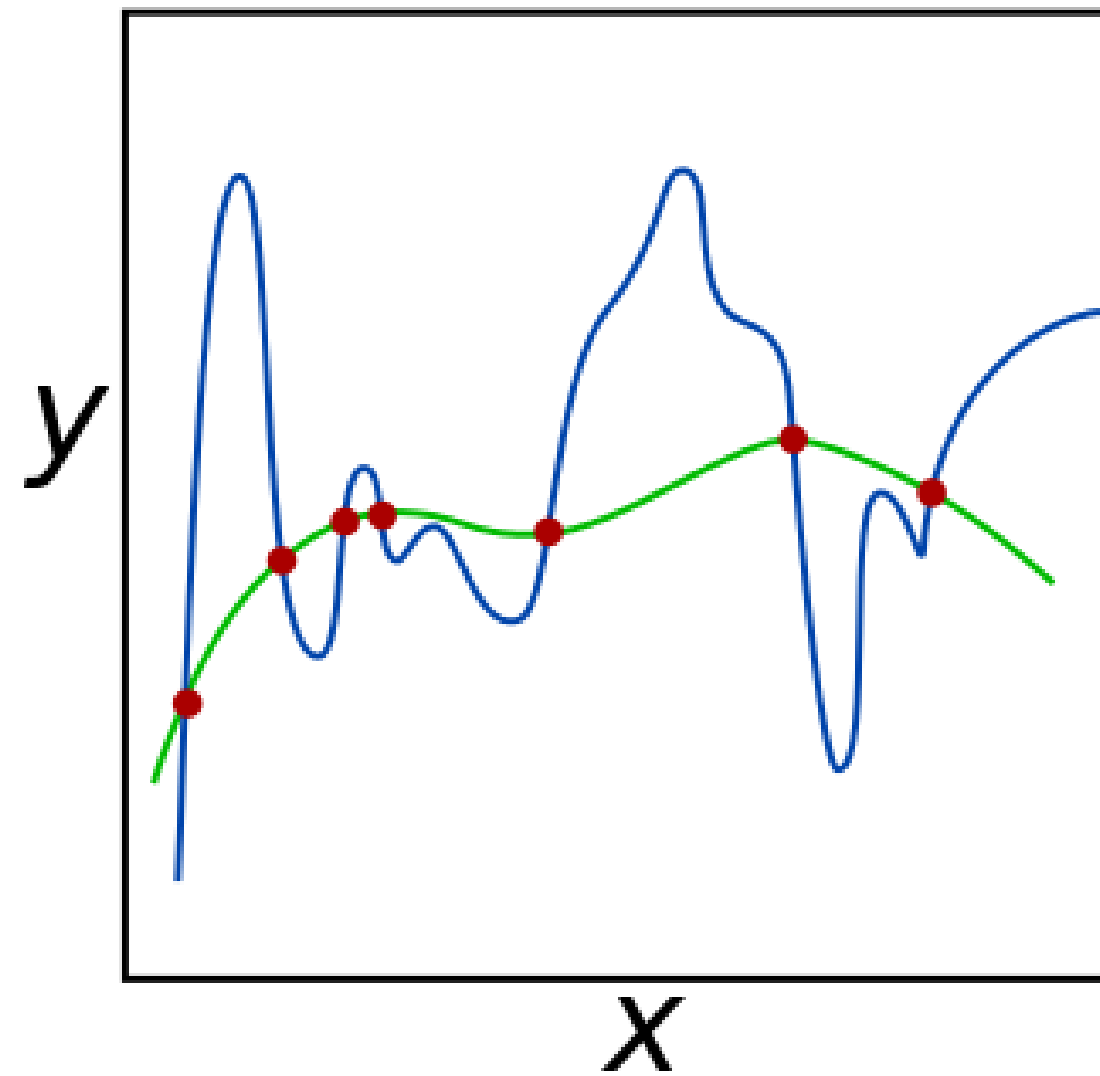
# Review: Steps to Choosing a Model

1. Create Held-Aside Set (Train/Test Split)
2. Determine Metric to use (or combination of metrics)
3. Get a Baseline for comparison
4. Use Cross-Validation to fit Hyperparameters and Choose Model
5. Evaluate Chosen Model on Held-Aside Set



# Avoiding Overfitting in Linear Models: Regularization

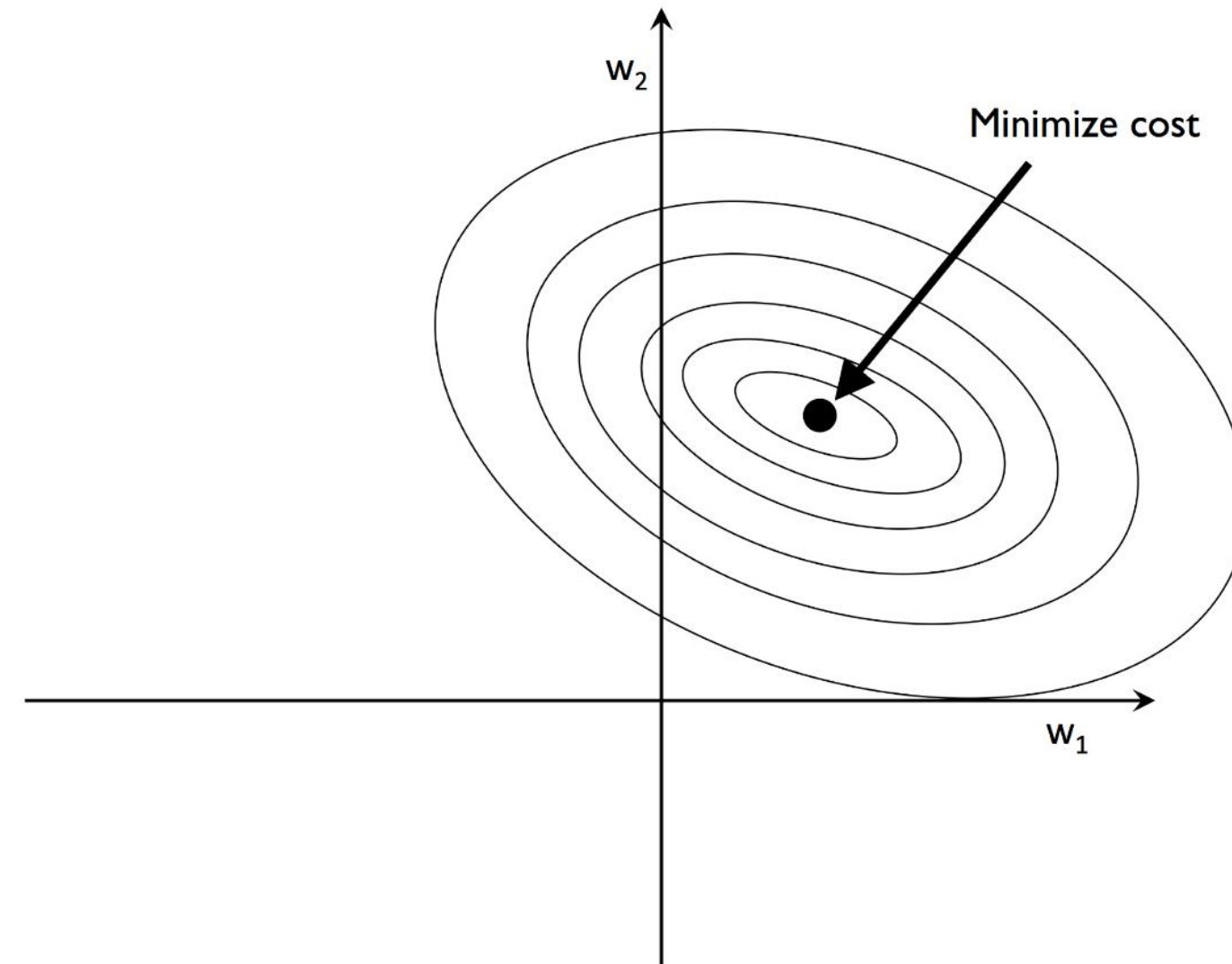
- Use to avoid overfitting in linear models
- Idea: can we reduce complexity of our linear model by minimizing weights?



From [https://www.wikiwand.com/en/Regularization\\_\(mathematics\)](https://www.wikiwand.com/en/Regularization_(mathematics)).

# Regression: Finding the Weights

- Linear models learn by finding weights that minimize a cost.
- Can we get close to the solution while still keeping weights small (simpler model)?



# Regularization: Add a cost for large weights

Goal: Penalize extreme weights ( $w$ )

If the original cost function looks like:

$$\arg \min_w C(f(w, x), y)$$

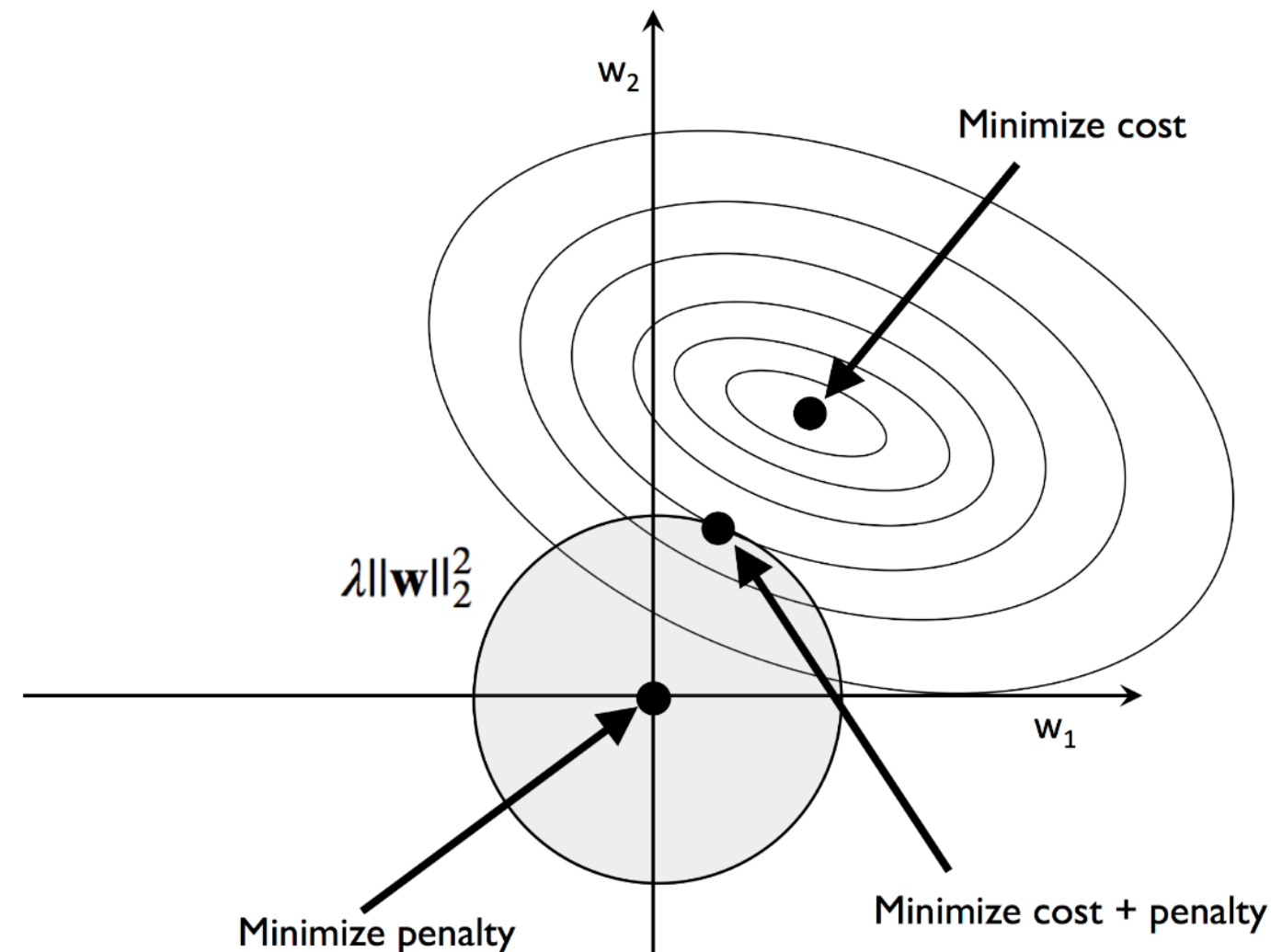
we can add a **regularization term**:

$$\arg \min_w C(f(w, x), y) + \lambda g(w)$$



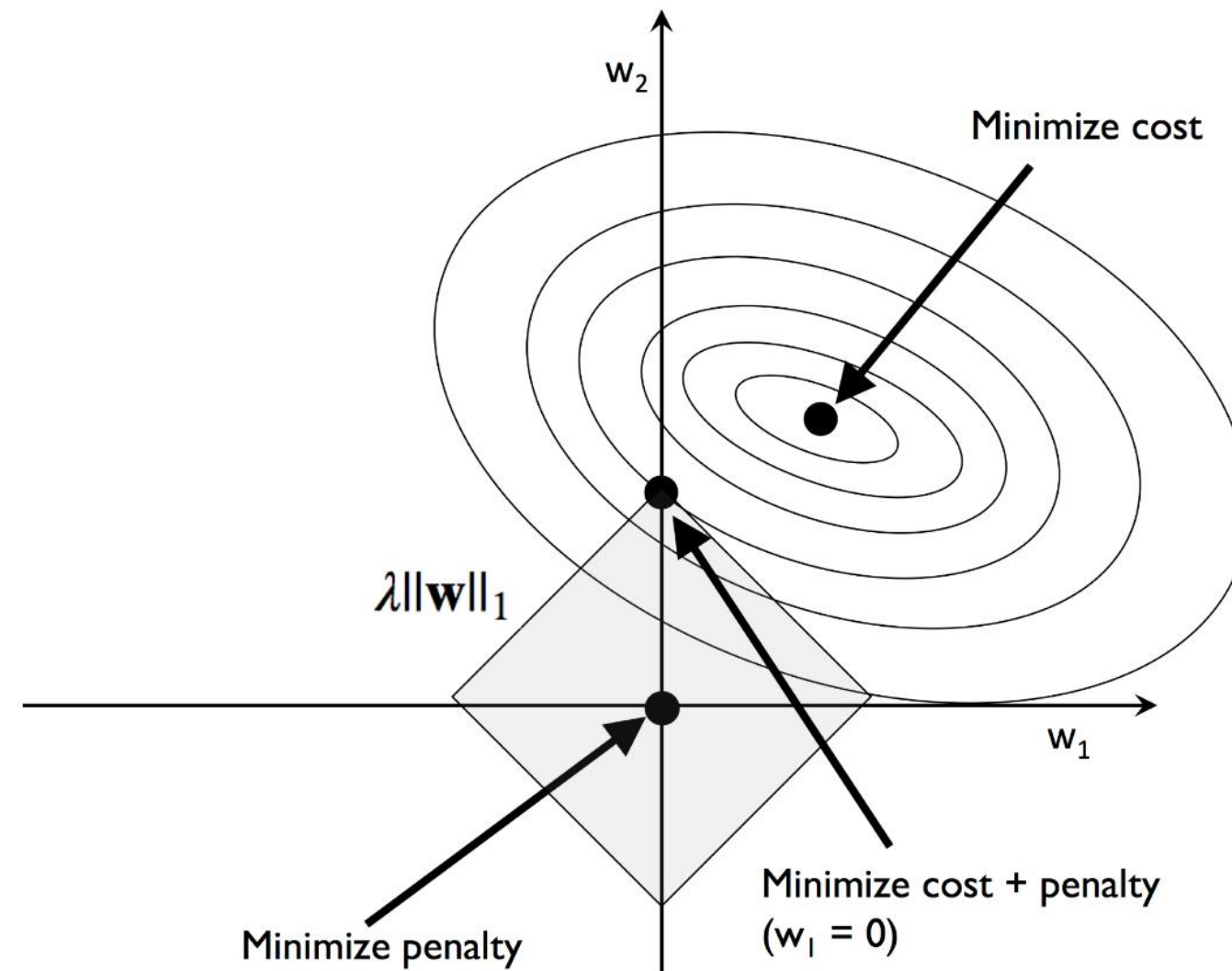
# Regularization: Ridge

- Goal:
  - Keep all coefficients relatively small
  - Drive coefficients of uninformative features to be smaller
- Uses  $L_2$  norm:  $\|w\|_2 = \sqrt{\sum_{j=1}^m w_j^2}$



# Regularization: LASSO

- Goal:
  - Keep all coefficients relatively small
  - Drive coefficients of unhelpful features to zero
- Uses  $L_1$  norm:  $\|w\|_1 = \sum_{j=1}^m |w_j|$



# Regularization: ElasticNet

- Mixture of  $L_1$  and  $L_2$
- $\alpha L_1 + (1 - \alpha)L_2$
- introduces a new hyperparameter  $\alpha$  or `l1_ratio`
- `l1_ratio` = 1 is LASSO ( $L_1$ )
- `l1_ratio` = 0 is Ridge ( $L_2$ )

# Regularization in sklearn

# Regularization in sklearn

```
In [38]: 1 from sklearn.linear_model import LogisticRegression
          2
          3 logr = LogisticRegression(penalty='l2', # default
          4                             C=1.0,      # weight on regularization, 1/lambda above
          5                             l1_ratio=None # only used when penalty is 'elasticnet'
          6                             )
```

# Regularization in sklearn

```
In [38]: 1 from sklearn.linear_model import LogisticRegression
2
3 logr = LogisticRegression(penalty='l2', # default
4                             C=1.0,      # weight on regularization, 1/lambda above
5                             l1_ratio=None # only used when penalty is 'elasticnet'
6                             )
```

```
In [39]: 1 for C in [.001,.1,1,10,1000]:
2         logr = LogisticRegression(penalty='l2', # default
3                                     C=C,          # weight on regularization, 1/lambda above
4                                     ).fit(X_train_bc,y_train_bc)
5         print(f'{str(C):5s} : {logr.coef_[0].round(2)}')
```

```
0.001 : [ 0.02 -0.02]
0.1    : [ 0.69 -0.57]
1      : [ 1.09 -0.93]
10     : [ 1.19 -1.02]
1000   : [ 1.2  -1.03]
```

# Regularization in sklearn

```
In [38]: 1 from sklearn.linear_model import LogisticRegression
2
3 logr = LogisticRegression(penalty='l2', # default
4                             C=1.0,      # weight on regularization, 1/lambda above
5                             l1_ratio=None # only used when penalty is 'elasticnet'
6                             )
```

```
In [39]: 1 for C in [.001,.1,1,10,1000]:
2     logr = LogisticRegression(penalty='l2', # default
3                               C=C,          # weight on regularization, 1/lambda above
4                               ).fit(X_train_bc,y_train_bc)
5     print(f'{str(C):5s} : {logr.coef_[0].round(2)}')
```

```
0.001 : [ 0.02 -0.02]
0.1    : [ 0.69 -0.57]
1      : [ 1.09 -0.93]
10     : [ 1.19 -1.02]
1000   : [ 1.2  -1.03]
```

```
In [40]: 1 for C in [.001,.1,1,10,1000]:
2     logr = LogisticRegression(penalty='l1',
3                               C=C,          # weight on regularization, 1/lambda above
4                               solver='liblinear'
5                               ).fit(X_train_bc,y_train_bc)
6     print(f'{str(C):5s} : {logr.coef_[0].round(2)}')
```

```
0.001 : [0. 0.]
0.1    : [ 0.5  -0.37]
1      : [ 1.08 -0.92]
10     : [ 1.19 -1.02]
1000   : [ 1.2  -1.03]
```

# GridSearchCV with Regularization



# GridSearchCV with Regularization

```
In [41]: 1 param_grid = {'l1_ratio':[0,.5,1],
2               'C': [.001,.01,1,10]}
3 logr_gscv = GridSearchCV(estimator=LogisticRegression(penalty='elasticnet',solver='saga'),
4                           param_grid=param_grid,
5                           cv=3,
6                           n_jobs=-1).fit(X_train_bc,y_train_bc)
7
8 print(f'best parameter setting found: {logr_gscv.best_params_}')
9 print(f'best coefficients found      : {logr_gscv.best_estimator_.coef_[0].round(2)}')
10 print(f'best training score found   : {logr_gscv.best_score_.round(3)}')
11
12 logr_gscv_test_score = logr_gscv.score(X_test_bc,y_test_bc)
13 logr_noreg_test_score = (LogisticRegression(penalty='none')
14                           .fit(X_train_bc,y_train_bc)
15                           .score(X_test_bc,y_test_bc)
16                           )
17 print()
18 print(f'logr_gscv test score  : {logr_gscv_test_score.round(3)}')
19 print(f'logr_noreg test score : {logr_noreg_test_score.round(3)}')
20
```

```
best parameter setting found: {'C': 1, 'l1_ratio': 1}
best coefficients found      : [ 1.1 -0.93]
best training score found   : 0.825
```

```
logr_gscv test score  : 0.818
logr_noreg test score : 0.818
```

# ElasticNetCV

# ElasticNetCV

```
In [42]: 1 from sklearn.datasets import make_regression
2 from sklearn.linear_model import ElasticNetCV
3
4 X_synth,y_synth = make_regression(n_samples=100,
5                                   n_features=200,
6                                   n_informative=10,
7                                   random_state=123
8                                   )
9 X_synth_train,X_synth_test,y_synth_train,y_synth_test = train_test_split(X_synth, y_synth, random_state=123)
10
11 dummy_synth = DummyRegressor(strategy='mean').fit(X_synth_train,y_synth_train)
12 lr_synth = LinearRegression().fit(X_synth_train,y_synth_train)
13 en_synth = ElasticNetCV(alphas=[.01,.1,1,100]).fit(X_synth_train,y_synth_train)
14
15 print(f'found alpha: {en_synth.alpha_}, found l1_ratio: {en_synth.l1_ratio_}\n')
16 print(f'dummy_synth train: {dummy_synth.score(X_synth_train,y_synth_train).round(2)} : 0.2f}')
17 print(f'lr_synth train    : {lr_synth.score(X_synth_train,y_synth_train).round(2)} : 0.2f}')
18 print(f'en_synth train    : {en_synth.score(X_synth_train,y_synth_train).round(2)} : 0.2f'\n')
19 print(f'dummy_synth test  : {dummy_synth.score(X_synth_test,y_synth_test).round(2)} : 0.2f}')
20 print(f'lr_synth test     : {lr_synth.score(X_synth_test,y_synth_test).round(2)} : 0.2f}')
21 print(f'en_synth test     : {en_synth.score(X_synth_test,y_synth_test).round(2)} : 0.2f}')
```

found alpha: 1.0, found l1\_ratio: 0.5

dummy\_synth train: 0.00  
lr\_synth train : 1.00  
en\_synth train : 0.95

dummy\_synth test : -0.00  
lr\_synth test : 0.13  
en\_synth test : 0.24

**Questions?**