

Elements Of Data Science - F2020

Week 7: Model Evaluation and Hyperparameter Tuning

10/26/2020

TODOs

- Readings:
 - PDSH Chapter 5: Feature Engineering
 - (Recommended) PML Chapter 4
- HW2, Out this week?
- Answer and submit Quiz 7, **Sunday Nov 9th, 11:59pm ET**
- Midterm
 - Release **Monday Oct 26th, 11:59pm ET**
 - Due **Saturday Oct 31st, 11:59pm ET**
 - Have 24hrs after starting exam to finish
 - 30 questions (fill in the blank/multiple choice/short answer)
 - Online via Gradescope
 - Questions asked/answered **privately** via Piazza
 - Open-book, open-note, open-python

Generating Predictions for a Single Value

```
In [2]: from sklearn.linear_model import LogisticRegression
```

```
In [3]: X = np.random.rand(5,2)
y = (np.random.rand(5) > .5).astype(int)

lr = LogisticRegression().fit(X,y)

# use reshape(1,-1) when generating a single prediction
# recall that sklearn expects X to be two dimensional
lr.predict(np.array([.2, .3]).reshape(1, -1))
```

```
Out[3]: array([0])
```

```
In [4]: # Note: this is different from when training on a single feature: .reshape(-1,1)
lr.fit(X[:,0].reshape(-1,1),y)

lr.predict(np.array(.2).reshape(1, -1))
```

```
Out[4]: array([0])
```

Today

- Model Evaluation and Selection
- Hyperparameter Tuning
- Regularization

Questions?

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"?
- How do we do all this both for Regression and Classification?

How well are our models performing?

Regression

- Mean Squared Error (MSE) and Root Mean Squared Error (RMSE)
- R^2
- Adjusted R^2

Classification

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

Data Setup for Regression

```
In [5]: zscore = lambda x: (x - x.mean()) / x.std()

df_wine = pd.read_csv('../data/wine_dataset.csv',
                      usecols=['alcalinity_of_ash', 'magnesium', 'alcohol', 'ash', 'proline', 'hue', 'class'])
numeric_cols = ['alcalinity_of_ash', 'magnesium', 'alcohol', 'ash', 'proline', 'hue']

df_wine[numeric_cols] = df_wine[numeric_cols].apply(zscore) # standardize numeric feature cols

X_r = df_wine[['proline', 'hue', 'ash']] # multiple features for regression task

X_1d = X_r.proline.values.reshape(-1,1) # create single feature vector using reshape

y_r = df_wine['alcohol'] # pull out regression target
```

In [6]: df_wine

Out[6]:

	alcohol	ash	alcalinity_of_ash	magnesium	hue	proline	class
0	1.514341	0.231400	-1.166303	1.908522	0.361158	1.010159	0
1	0.245597	-0.825667	-2.483841	0.018094	0.404908	0.962526	0
2	0.196325	1.106214	-0.267982	0.088110	0.317409	1.391224	0
3	1.686791	0.486554	-0.806975	0.928300	-0.426341	2.328007	0
4	0.294868	1.835226	0.450674	1.278379	0.361158	-0.037767	0
...
173	0.873810	0.304301	0.300954	-0.331985	-1.388840	-0.021890	2
174	0.491955	0.413653	1.049555	0.158126	-1.126341	0.009866	2
175	0.331822	-0.388260	0.151234	1.418411	-1.607590	0.279786	2
176	0.208643	0.012696	0.151234	1.418411	-1.563840	0.295664	2
177	1.391162	1.361368	1.498716	-0.261969	-1.520090	-0.593486	2

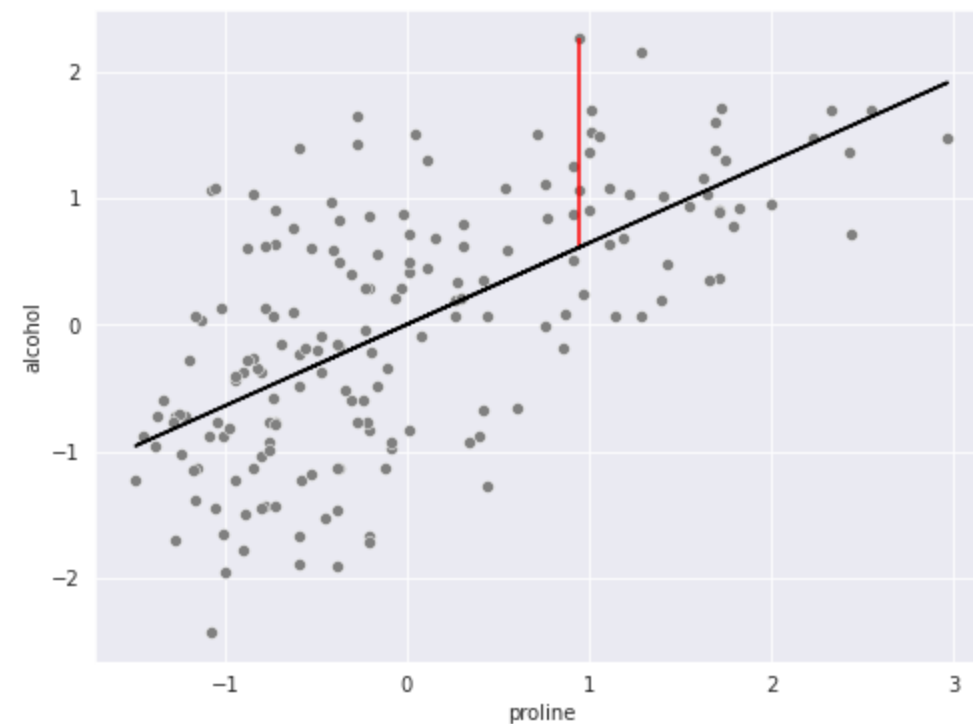
178 rows × 7 columns

Regression with Simple Linear Model

```
In [7]: from sklearn.linear_model import LinearRegression

lr = LinearRegression().fit(X_1d,y_r)
argmax_y_r = np.argmax(y_r)
y_pred = lr.predict(X_1d)

fig,ax = plt.subplots(1,1,figsize=(8,6))
sns.scatterplot(x=X_r.proline, y=y_r, color='grey');
ax.plot(X_1d,y_pred,color='k');
ax.vlines(X_1d[argmax_y_r],y_r.iloc[argmax_y_r],y_pred[argmax_y_r],color='r');
```



How Good is This Fit? MSE and RMSE

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

```
In [8]: from sklearn.metrics import mean_squared_error

lr_mse = mean_squared_error(y_r, y_pred)
f'{lr_mse = :0.2f}'
```

```
Out[8]: 'lr_mse = 0.58'
```

- But this is the squared error! (alcohol²)

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

```
In [9]: def root_mean_squared_error(y_true, y_pred):
        return np.sqrt(mean_squared_error(y_true, y_pred))

lr_rmse = root_mean_squared_error(y_r, y_pred)
f'{lr_rmse = :0.2f}'
```

```
Out[9]: '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

```
In [10]: from sklearn.dummy import DummyRegressor

dummyr = DummyRegressor(strategy='mean') # default strategy
dummyr.fit(X_1d,y_r)

dummy_rmse = root_mean_squared_error(y_r,dummyr.predict(X_1d))

f'{dummy_rmse = :0.2f}'
```

```
Out[10]: 'dummy_rmse = 1.00'
```

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 [11]: r2_lr = lr.score(X_1d, y_r)
         r2_dummyr = dummyr.score(X_1d, y_r)

         print(f'{r2_dummyr} = :0.2f'\n{r2_lr} = :0.2f}')

r2_dummyr = 0.00
r2_lr      = 0.41
```

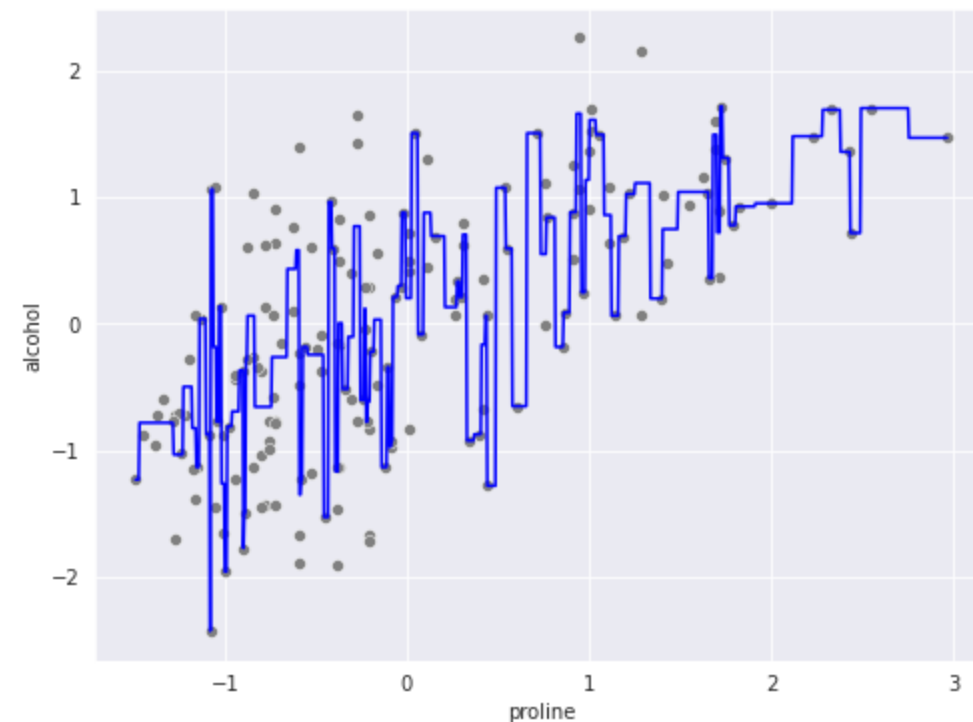
Can we do better?

```
In [12]: from sklearn.tree import DecisionTreeRegressor
```

```
dtr = DecisionTreeRegressor(max_depth=10)
dtr.fit(X_1d,y_r)
r2_dtr = dtr.score(X_1d,y_r)
print(f'{r2_lr  = :0.2f}\n{r2_dtr = :0.2f}')
```

```
r2_lr  = 0.41
r2_dtr = 0.76
```

```
In [13]: X_1d_sorted = np.sort(X_1d.flatten()).reshape(-1,1)
X_query = np.linspace(X_1d.min(),X_1d.max(),1000).reshape(-1,1)
y_pred = dtr.predict(X_query)
fig,ax = plt.subplots(1,1,figsize=(8,6))
sns.scatterplot(x=X_r.proline, y=y_r,color='gray');
ax.plot(X_query,y_pred,color='b');
```



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?

Generalization:

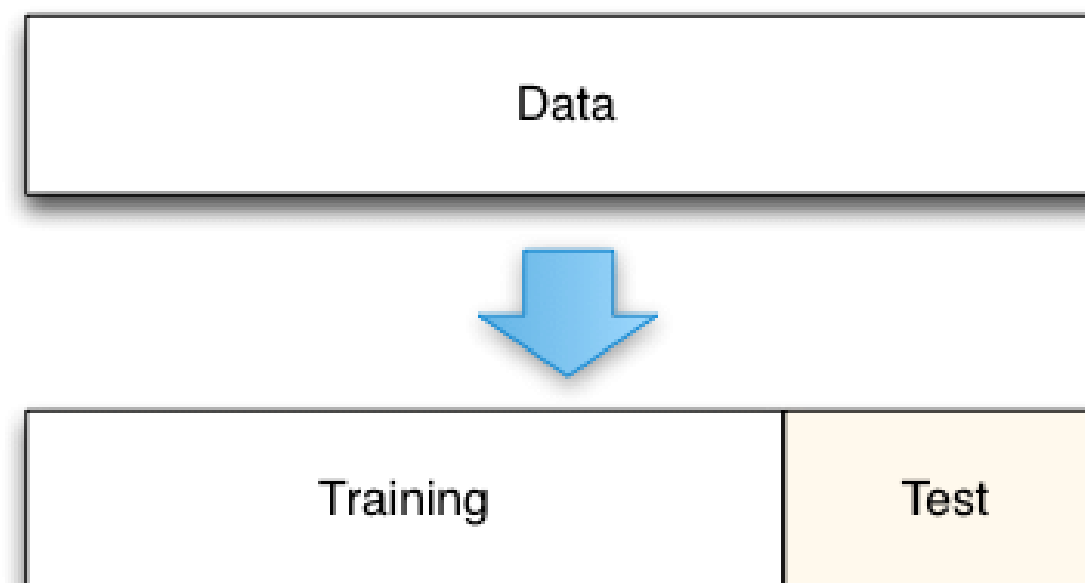
- how well will model predict on data that it hasn't seen yet?

But we used all of our data to train?

- Need to do a **Train/Test Split**

Train/Test Split

- **Training Set:** portion of dataset used for training
- **Test/Held-Aside/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

```
In [14]: from sklearn.model_selection import train_test_split

X_train_r, X_test_r, y_train_r, y_test_r = train_test_split(X_1d,
                                                            y_r,
                                                            test_size=.25, #default
                                                            random_state=123)

print(X_1d.shape[0])
print(X_train_r.shape[0])
print(X_test_r.shape[0])
```

```
178
133
45
```

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

Training and Evaluate on Different Data

```
In [15]: dummyr = DummyRegressor().fit(X_train_r,y_train_r)
lr = LinearRegression().fit(X_train_r,y_train_r)
dtr = DecisionTreeRegressor(max_depth=10).fit(X_train_r,y_train_r)

r2_dummyr = dummyr.score(X_test_r,y_test_r)
r2_lr      = lr.score(X_test_r,y_test_r)
r2_dtr     = dtr.score(X_test_r,y_test_r)

print(f'{r2_dummyr} = : 0.2f'\n{r2_lr      = : 0.2f}\n{r2_dtr     = : 0.2f}')
```

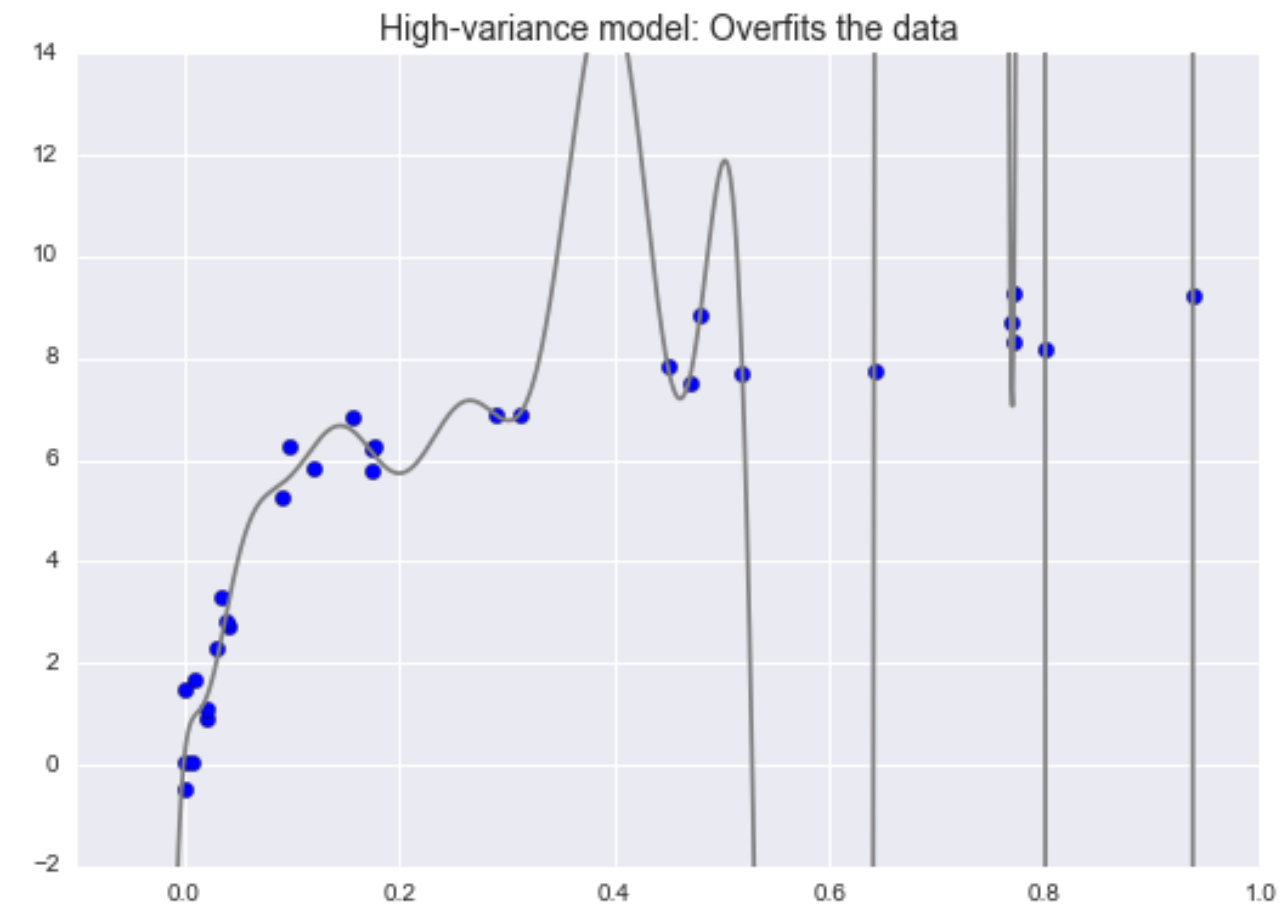
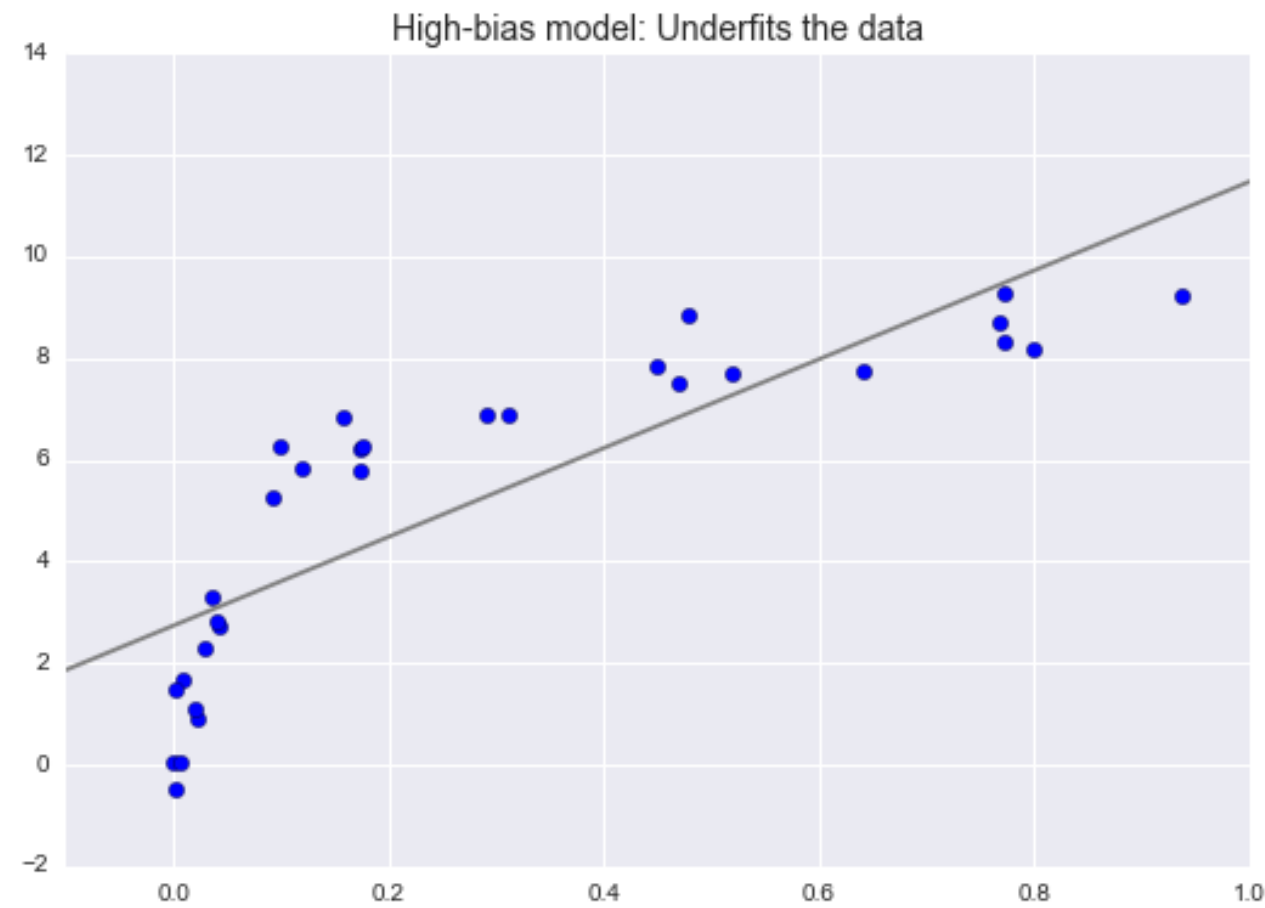
```
r2_dummyr = -0.03
r2_lr      =  0.28
r2_dtr     = -0.31
```

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

Overfitting and Underfitting

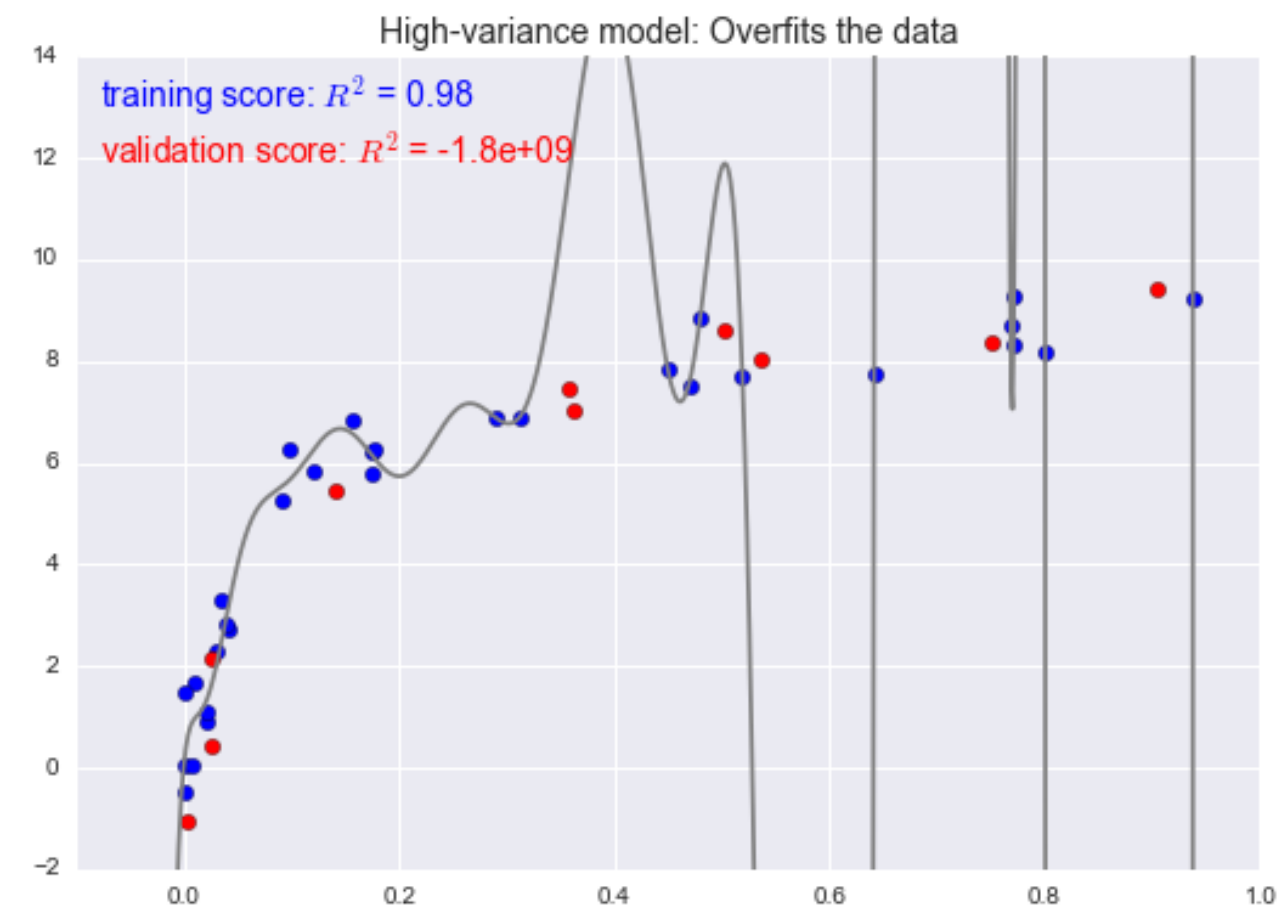
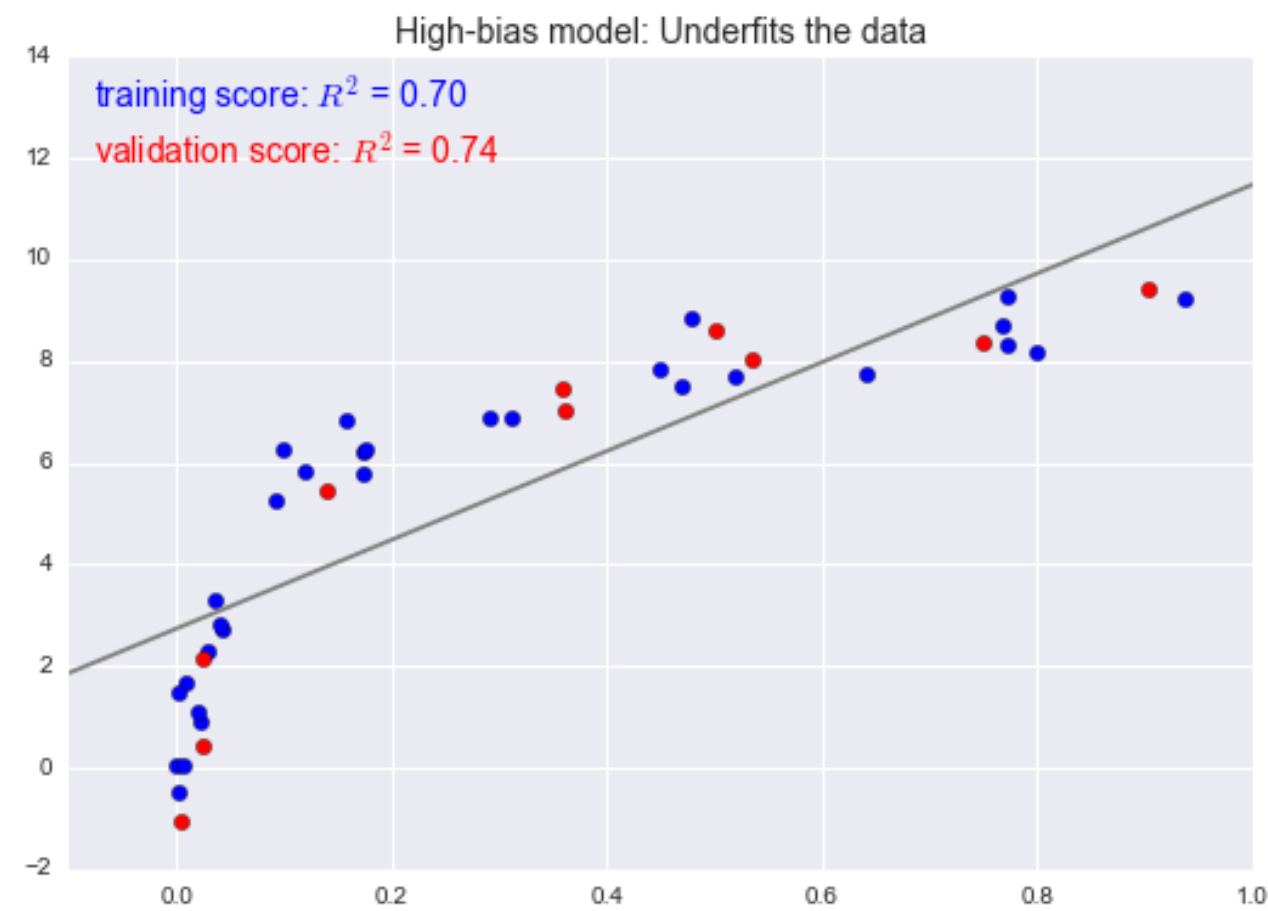
- **Overfitting:** poor generalization due to complexity
 - learning noise in training data
- **Underfitting:** poor generalization due to simplicity
 - not flexible enough to learn concept
- Need to find a balance between simplicity and complexity
- Need to find a balance between **bias** and **variance**

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?

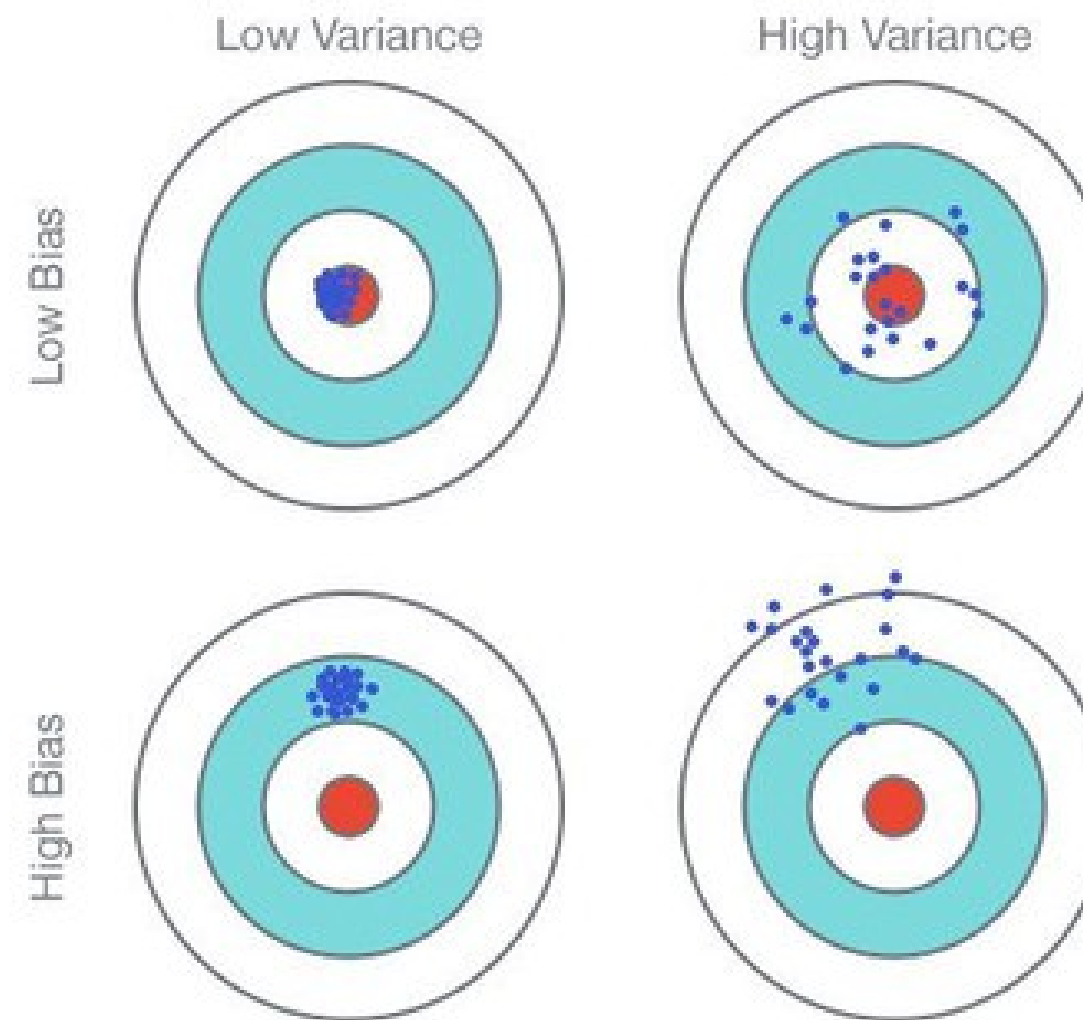
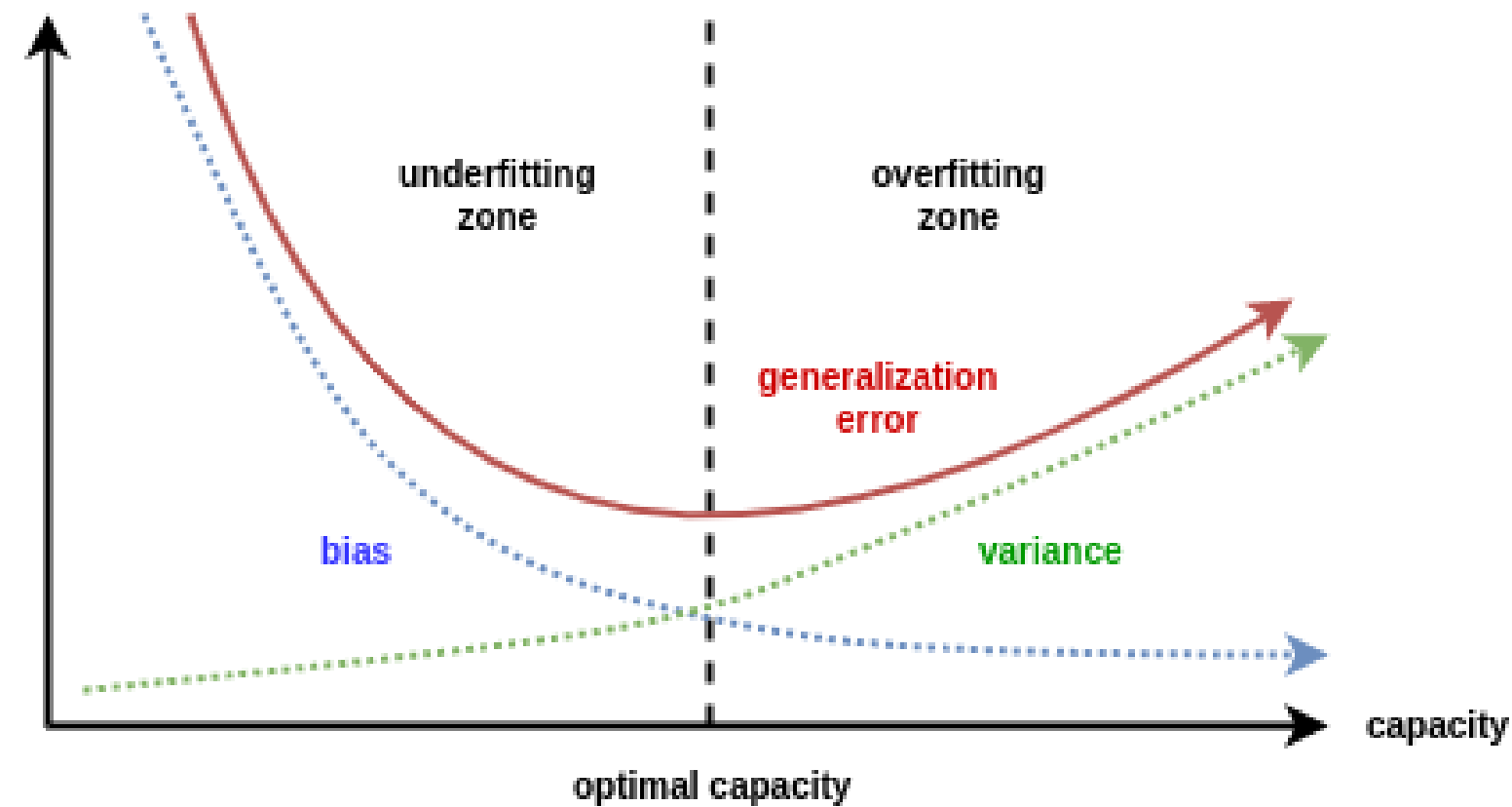


Fig. 1: Graphical Illustration of bias-variance trade-off , Source: Scott Fortmann-Roe., Understanding Bias-Variance Trade-off

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**

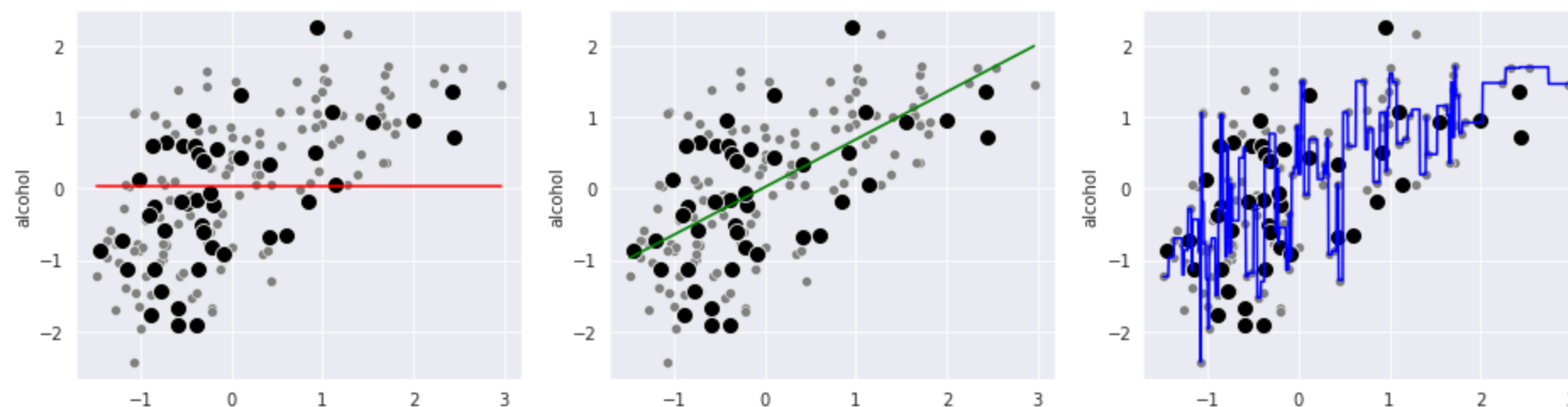
Avoiding Overfitting/Underfitting

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

Avoiding Overfitting/Underfitting

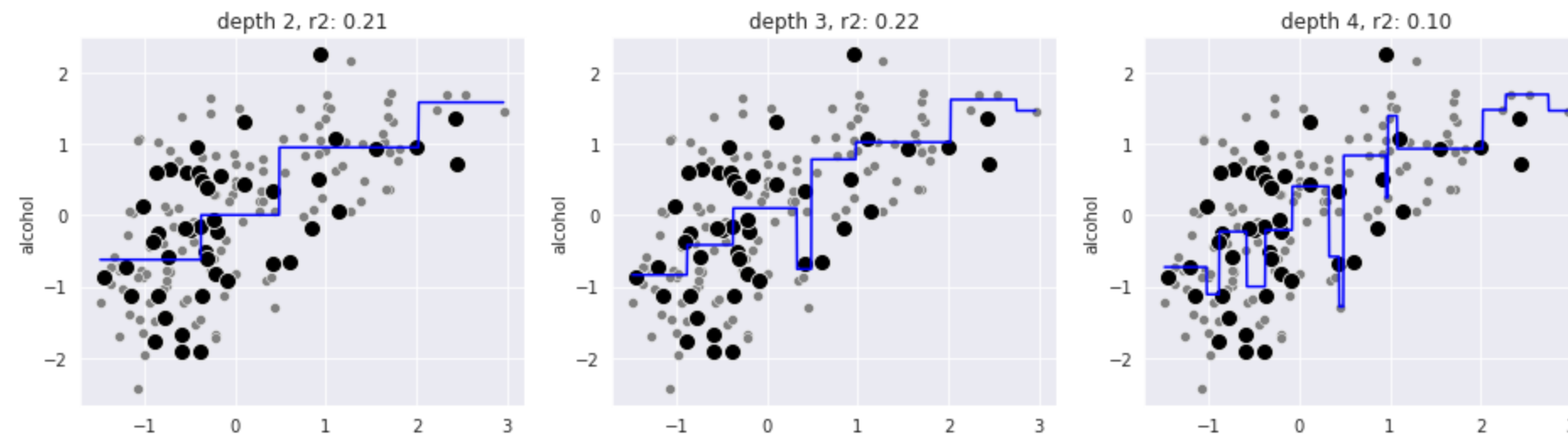
- Never train and evaluate on the same set of data!
 - train test split
 - **cross-validation**
- Keep the model as simple as possible (Occom's Razor)

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



Overfitting? Simplify the model

```
In [17]: max_depths = [2,3,4]
fig,ax = plt.subplots(1,3,figsize=(16,4))
for i in range(3):
    dtr = DecisionTreeRegressor(max_depth=max_depths[i]).fit(X_train_r,y_train_r)
    sns.scatterplot(x=X_train_r.flatten(),y=y_train_r,color="gray",ax=ax[i])
    sns.scatterplot(x=X_test_r.flatten(),y=y_test_r,color="black",s=100,ax=ax[i]);
    ax[i].plot(X_query,dtr.predict(X_query),color='b');
    ax[i].set_title(f'depth {max_depths[i]}, r2: {dtr.score(X_test_r,y_test_r):0.2f}')
```

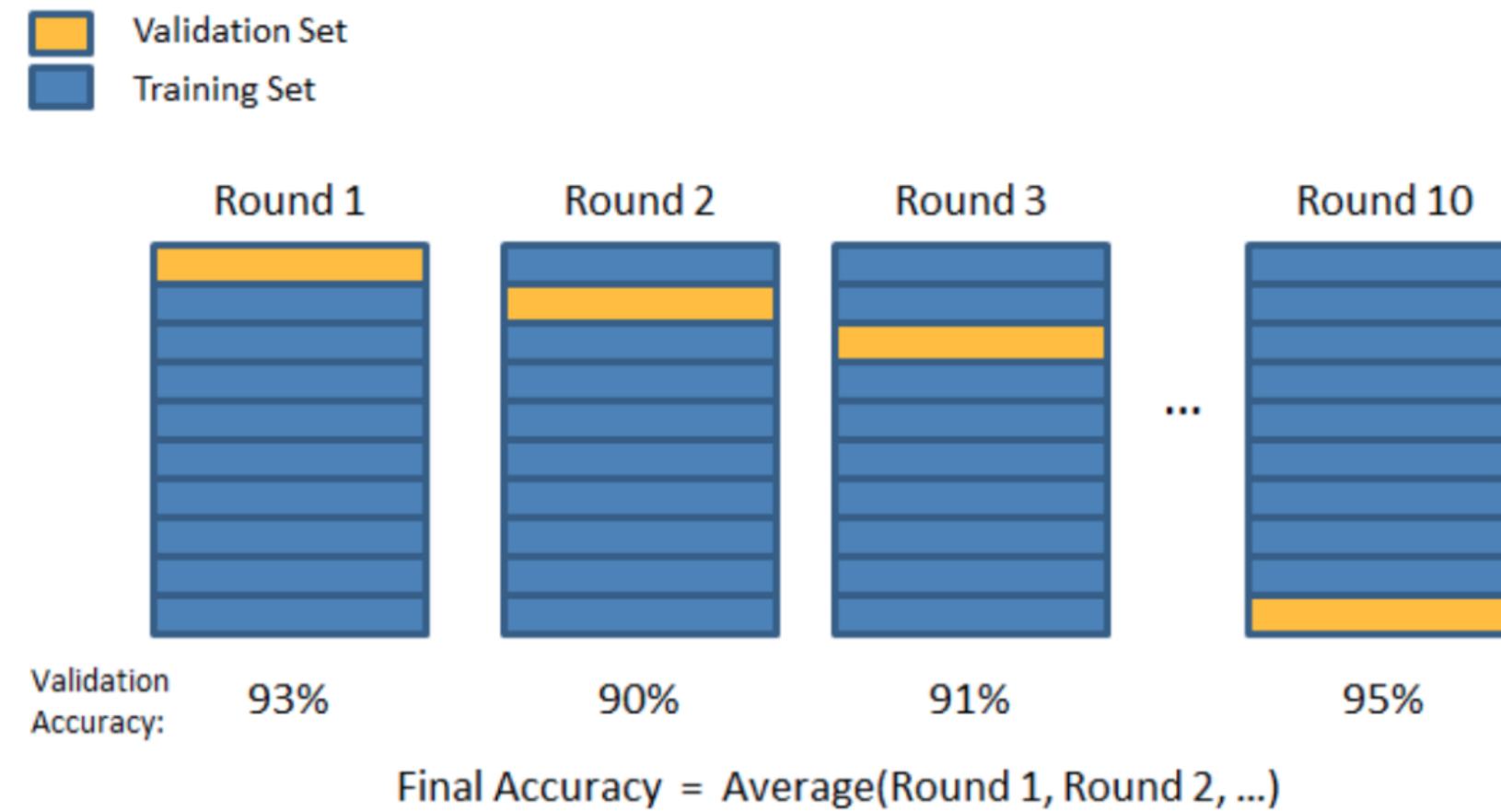


- But now we might be overfitting on the test set!
- How to choose hyperparameters: **Cross-Validation**

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
- 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

```
In [18]: from sklearn.model_selection import cross_val_score

scores = cross_val_score(DecisionTreeRegressor(max_depth=2),
                        X_train_r,
                        y_train_r,
                        cv=5) #default

scores
```

```
Out[18]: array([0.41853032, 0.37075208, 0.45405394, 0.2588269 , 0.22799476])
```

```
In [19]: print(f'{np.mean(scores) :0.2f} +- {2*np.std(scores) :0.2f}')
```

```
0.35 +- 0.18
```

Tuning Hyperparameters with CV

```
In [20]: mean_scores = []

for depth in [1, 2, 3, 5, 10]:
    dtr = DecisionTreeRegressor(max_depth=depth)
    scores = cross_val_score(dtr, X_train_r, y_train_r, cv=5)
    mean_scores.append( (depth, scores.mean()) )

for depth, mean_score in mean_scores:
    print(f'{depth = :2d} : {mean_score: 0.3f}')
```

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

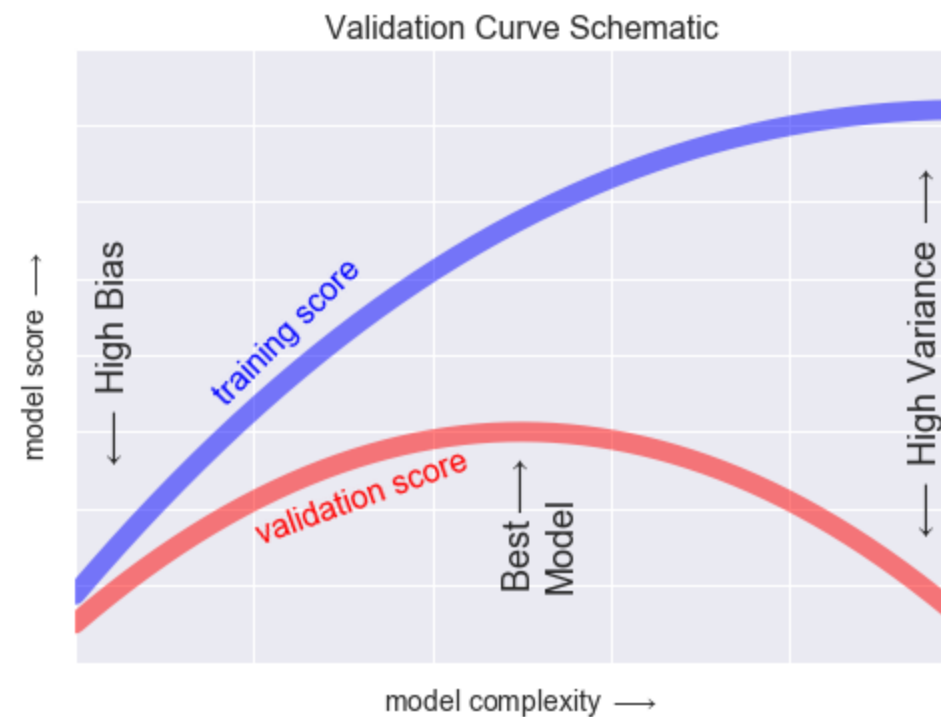
```
In [21]: # find the depth that gives best score (highest R^2)
sorted(mean_scores, key=lambda x:x[1], reverse=True)[0] # sorted is ascending by default
```

```
Out[21]: (2, 0.3460316005269502)
```

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 (overfitting)



From PDSH

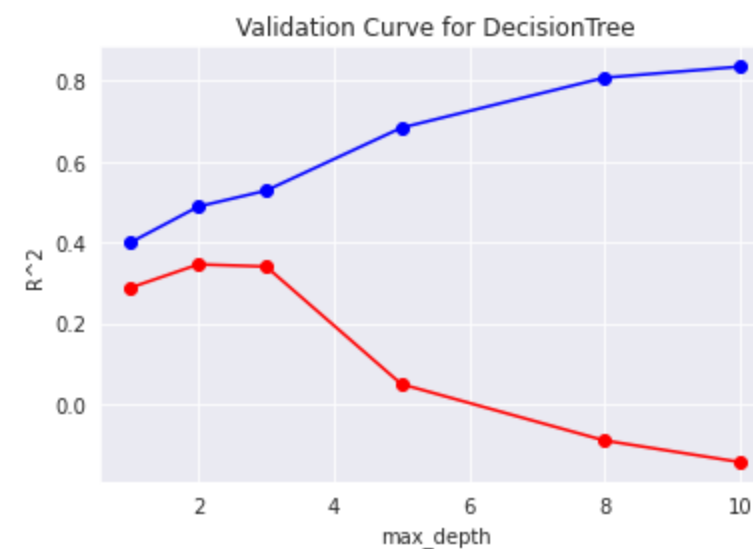
Validation Curve in sklearn

```
In [22]: from sklearn.model_selection import validation_curve

depth = [1,2,3,5,8,10]
train_scores, test_scores = validation_curve(DecisionTreeRegressor(),
                                             X_train_r, y_train_r,
                                             param_name='max_depth',
                                             param_range=depth,
                                             cv=5)

mean_train_scores = np.mean(train_scores,axis=1) # take the mean across rows
mean_test_scores = np.mean(test_scores,axis=1)
```

```
In [23]: plt.plot(depth, mean_train_scores, 'o-', color='b', label='training score');
plt.plot(depth, mean_test_scores, 'o-', color='r', label='validation score');
plt.xlabel('max_depth'), plt.ylabel('R^2'); plt.title('Validation Curve for DecisionTree');
```



More Than One HyperParameter? Grid Search

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

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

```
In [24]: distance_metrics = ['euclidean', 'manhattan']  
n_neighbors = [1, 3, 5]  
  
grid = []  
for d in distance_metrics:  
    for k in n_neighbors:  
        print([d, k])
```

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

Grid Search in sklearn

```
In [25]: from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsRegressor

params = {'n_neighbors':[1,2,3,5,10],
          'metric':['euclidean','manhattan']}
gscv = GridSearchCV(KNeighborsRegressor(),
                    params,          # grid of size 10
                    cv=3,           # do 3-fold CV at every grid point
                    refit=True)     # refit True trains one more time on the entire training set

gscv.fit(X_train_r,y_train_r)      # How many times are we training a model here? (2*5*3 + 1 = 31)

print(gscv.best_params_)

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

```
In [26]: scores = cross_val_score(gscv.best_estimator_,X_train_r,y_train_r,cv=5)

print(f'{np.mean(scores):0.2f} +- {2*np.std(scores):0.2f}')

0.35 +- 0.21
```

Review So Far

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

Data Setup for Classification

```
In [27]: idx_binary = df_wine['class'].isin([0,1])           # reduce to binary classification

X_bc = df_wine.loc[idx_binary,['alcalinity_of_ash','magnesium']]      # only 2 features for ease of plotting

y_bc = df_wine.loc[idx_binary,'class']                                # pull out classification target [0,1]

X_train_bc,X_test_bc,y_train_bc,y_test_bc = train_test_split(X_bc,
                                                             y_bc,
                                                             stratify=y_bc, # maintain label proportions
                                                             random_state=0
                                                             )

pd.DataFrame({'train':y_train_bc.value_counts(),'test':y_test_bc.value_counts()}).sort_index()
```

```
Out[27]:
```

	train	test
0	44	15
1	53	18

```
In [28]: X_mc = df_wine.loc[:,['alcalinity_of_ash','magnesium']]      # multiple features for multiclass classification task
y_mc = df_wine.loc[:, 'class']                                       # pull out classification target [0,1,2]

X_train_mc,X_test_mc,y_train_mc,y_test_mc = train_test_split(X_mc,
                                                             y_mc,
                                                             stratify=y_mc, # maintain label proportions
                                                             random_state=123
                                                             )

pd.DataFrame({'train':y_train_mc.value_counts(),'test':y_test_mc.value_counts()}).sort_values(by="train")
```

```
Out[28]:
```

	train	test
2	36	12
0	44	15

Default Metric in Classification: Accuracy

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

```
In [29]: from sklearn.dummy import DummyClassifier
from sklearn.tree import DecisionTreeClassifier
dummyc = DummyClassifier(strategy='prior').fit(X_train_bc,y_train_bc) # works like 'most-frequent'
dtc = DecisionTreeClassifier(max_depth=2).fit(X_train_bc,y_train_bc)

print(f'{dummyc.score(X_test_bc,y_test_bc) = :0.2f}') # default classification score is accuracy
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

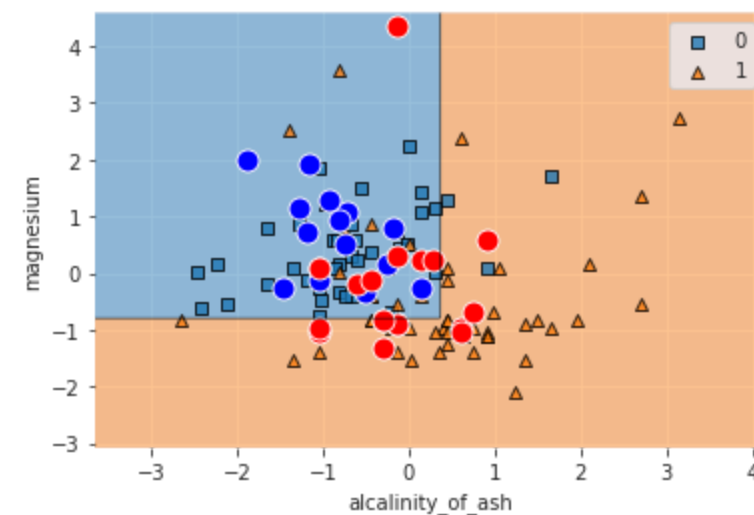
- There are different kinds of error in classification

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

From PML

Visualizing Errors with a Confusion Matrix

```
In [30]: fig, ax = plt.subplots(1, 1, figsize=(6, 4));
plot_decision_regions(X_train_bc.values, y_train_bc.values, dtc, ax=ax);
sns.scatterplot(x=X_bc.columns[0], y=X_bc.columns[1], data=X_test_bc[y_test_bc == 0], color="blue", s=120);
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 [31]: from sklearn.metrics import confusion_matrix

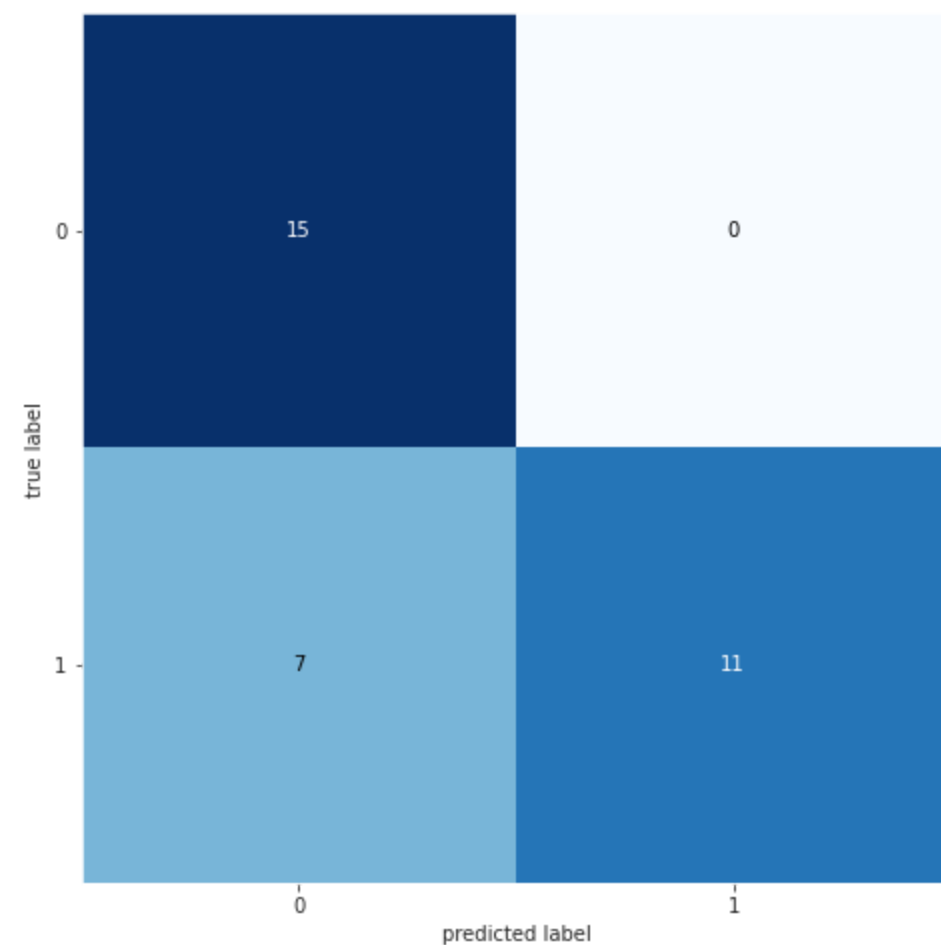
print('training set error\n', confusion_matrix(y_train_bc, dtc.predict(X_train_bc)))
print()
print('test set error\n', confusion_matrix(y_test_bc, dtc.predict(X_test_bc)))
```

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

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


Plot Confusion Matrix with mlxtend

```
In [32]: from mlxtend.plotting import plot_confusion_matrix  
  
fig, ax = plt.subplots(1, 1, figsize=(8, 8))  
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

```
In [33]: dummyc_precision_scores = cross_val_score(dummyc,X_train_bc,y_train_bc,cv=5,scoring='precision')
dummyc_recall_scores = cross_val_score(dummyc,X_train_bc,y_train_bc,cv=5,scoring='recall')

print(f'dtc precision: {np.mean(dummyc_precision_scores):0.2f} +- {2*np.std(dummyc_precision_scores):0.2f}')
print(f'dtc recall : {np.mean(dummyc_recall_scores):0.2f} +- {2*np.std(dummyc_recall_scores):0.2f}')
print()

dtc_precision_scores = cross_val_score(dtc,X_train_bc,y_train_bc,cv=5,scoring='precision')
dtc_recall_scores = cross_val_score(dtc,X_train_bc,y_train_bc,cv=5,scoring='recall')

print(f'dtc precision: {np.mean(dtc_precision_scores):0.2f} +- {2*np.std(dtc_precision_scores):0.2f}')
print(f'dtc recall : {np.mean(dtc_recall_scores):0.2f} +- {2*np.std(dtc_recall_scores):0.2f}')
```

dtc precision: 0.55 +- 0.04
dtc 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 False 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 False Negatives!

```
In [34]: dummyc_f1_scores = cross_val_score(dummyc, X_train_bc, y_train_bc, cv=5, scoring='f1')
dttc_f1_scores = cross_val_score(dttc, X_train_bc, y_train_bc, cv=5, scoring='f1')
print(f'dummyc f1 = {np.mean(dummyc_f1_scores):0.2f} +- {2*np.std(dummyc_f1_scores):0.2f}')
print(f'dttc f1 = {np.mean(dttc_f1_scores):0.2f} +- {2*np.std(dttc_f1_scores):0.2f}')
```

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

Paying attention to False 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 False 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 [35]: from sklearn.metrics import roc_curve
fpr_dtc, tpr_dtc, _ = roc_curve(y_train_bc, dtc.predict_proba(X_train_bc)[:,-1])

fpr_dtc, tpr_dtc
```

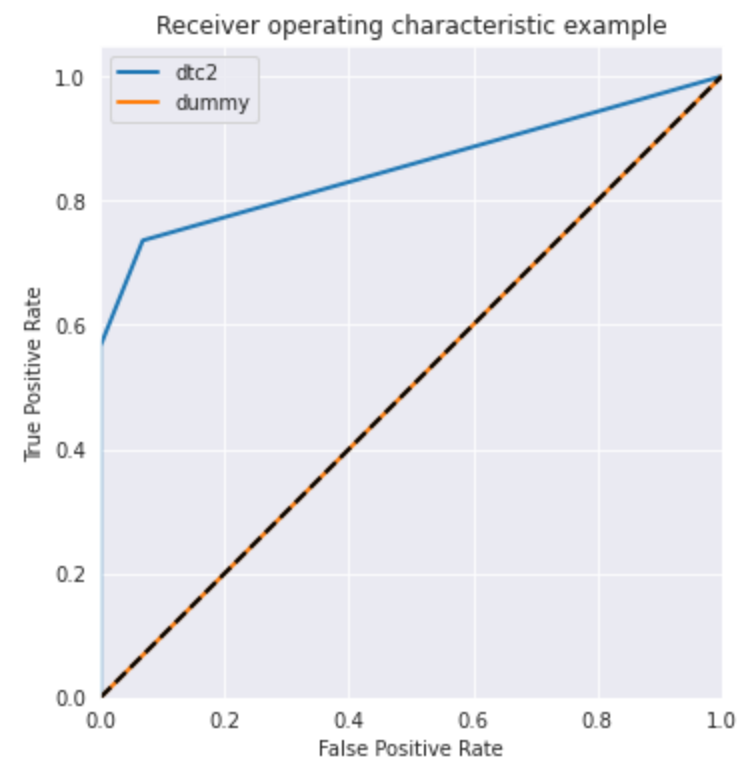
```
Out[35]: (array([0.          , 0.          , 0.06818182, 1.          ]),
          array([0.          , 0.56603774, 0.73584906, 1.          ]))
```


Plotting ROC Curves

```
In [36]: def plot_roc(curves):
          fig, ax = plt.subplots(1, 1, figsize=(6, 6))
          lw = 2
          for fpr, tpr, model_name in curves:
              l1, = ax.plot(fpr, tpr, lw=lw, label=model_name)
          ax.plot([0, 1], [0, 1], color='k', lw=lw, linestyle='--')
          ax.set_xlim([0.0, 1.0])
          ax.set_ylim([0.0, 1.05])
          ax.set_xlabel('False Positive Rate')
          ax.set_ylabel('True Positive Rate')
          ax.set_aspect('equal', 'box')
          ax.set_title('Receiver operating characteristic example')
          ax.legend()
```

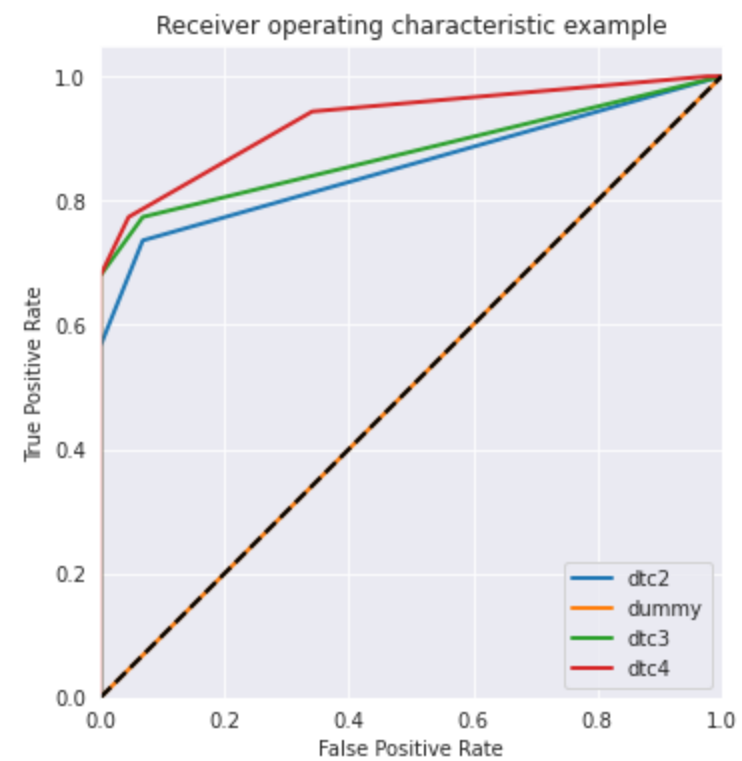
Plotting ROC Curves

```
In [37]: curves = [(fpr_dtc, tpr_dtc, 'dtc2')]
fpr_dummys, tpr_dummys, _ = roc_curve(y_train_bc, dummys.predict_proba(X_train_bc)[: , 1]) # Compare dummy
curves.append((fpr_dummys, tpr_dummys, 'dummy'));
plot_roc(curves);
```



Plotting ROC Curves

```
In [38]: for depth in [3,4]:  
    fpr, tpr, _ = roc_curve(y_train_bc, DecisionTreeClassifier(max_depth=depth).fit(X_train_bc, y_train_bc).predict_proba(X_train_bc))  
    curves.append((fpr, tpr, 'dtc'+str(depth)))  
plot_roc(curves);
```



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?

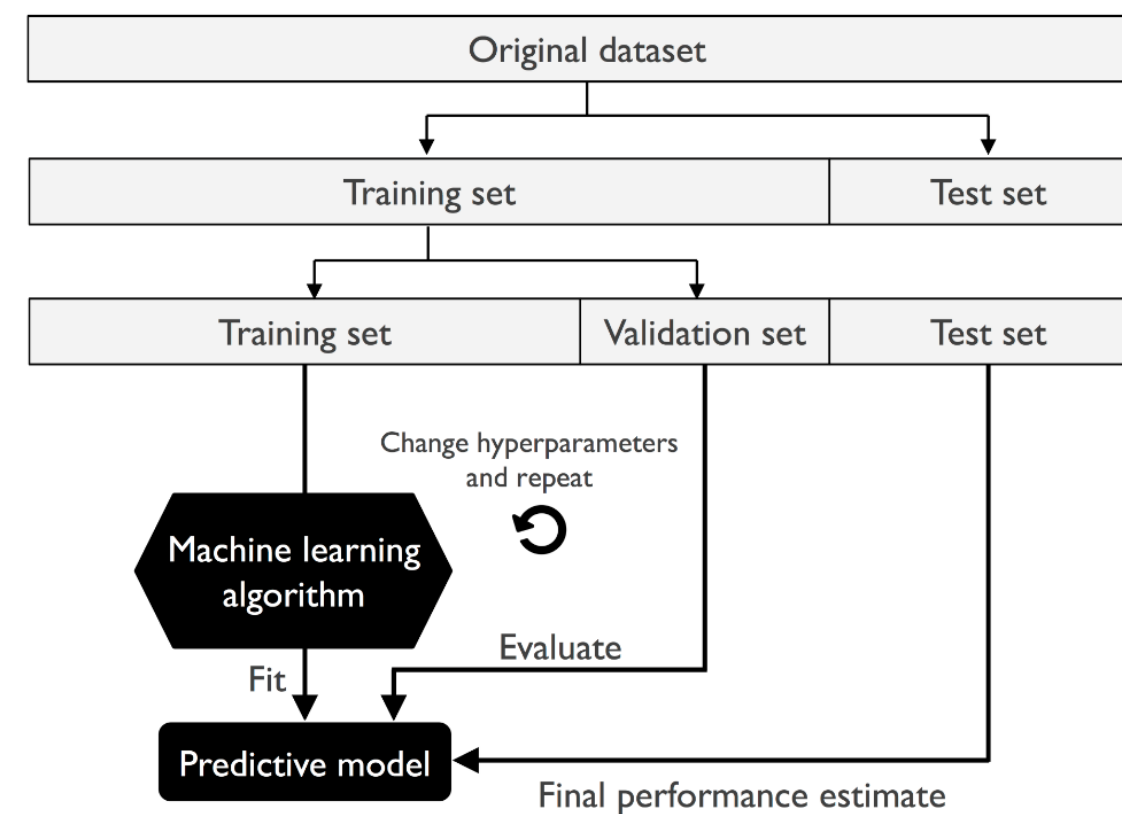
```
In [39]: dummyc_rocauc_scores = cross_val_score(dummyc, X_train_bc, y_train_bc, cv=5, scoring='roc_auc')
         dtc_rocauc_scores = cross_val_score(dtc, X_train_bc, y_train_bc, cv=5, scoring='roc_auc')

         print(f'dummyc rocauc = {np.mean(dummyc_rocauc_scores):0.2f} +- {2*np.std(dummyc_rocauc_scores):0.2f}')
         print(f'dtc rocauc    = {np.mean(dtc_rocauc_scores):0.2f} +- {2*np.std(dtc_rocauc_scores):0.2f}')
```

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

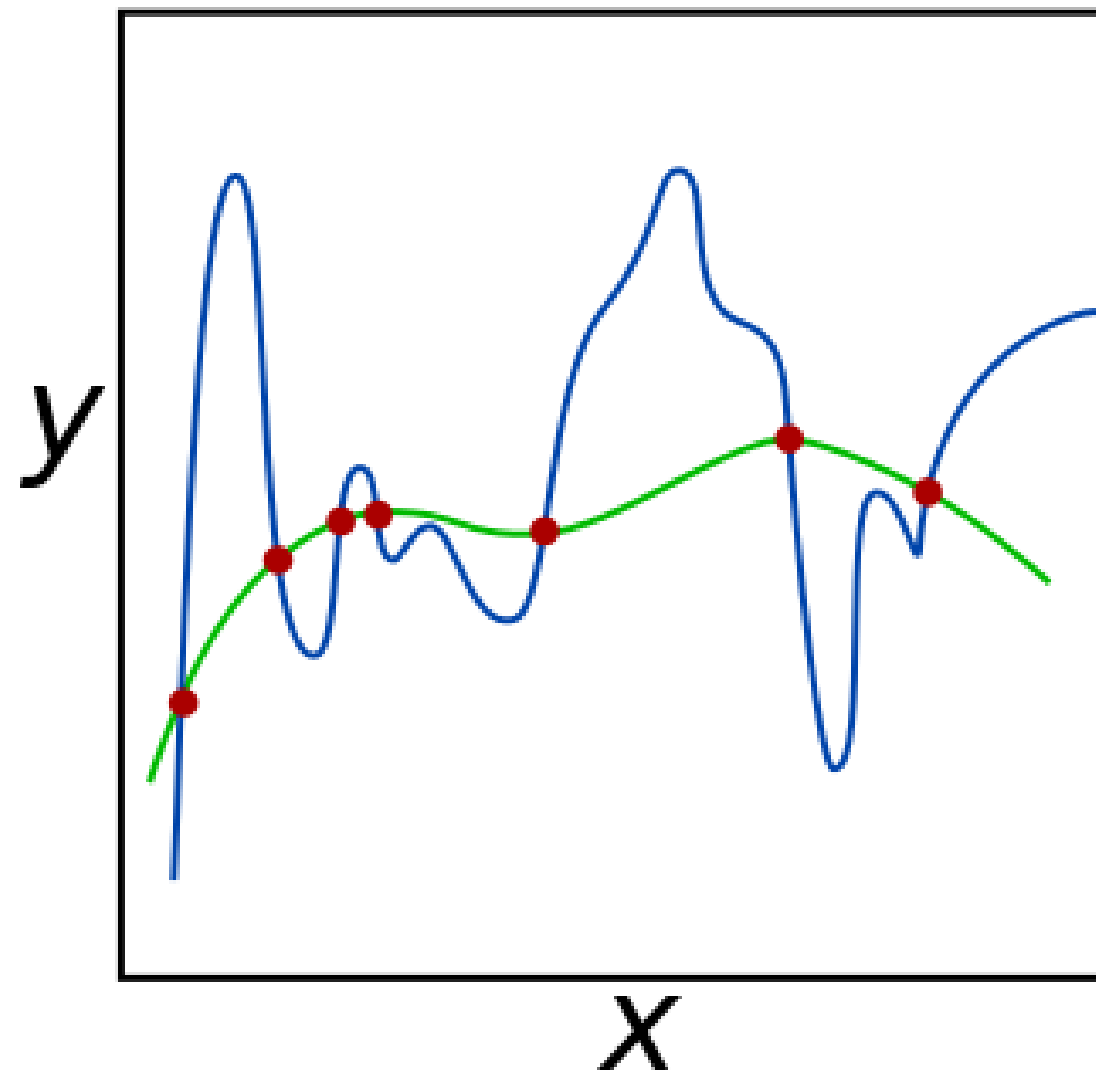
Review: Steps to Choosing a Model

1. Create Held-Aside Set (Train/Test Split)
2. Determine Metric (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

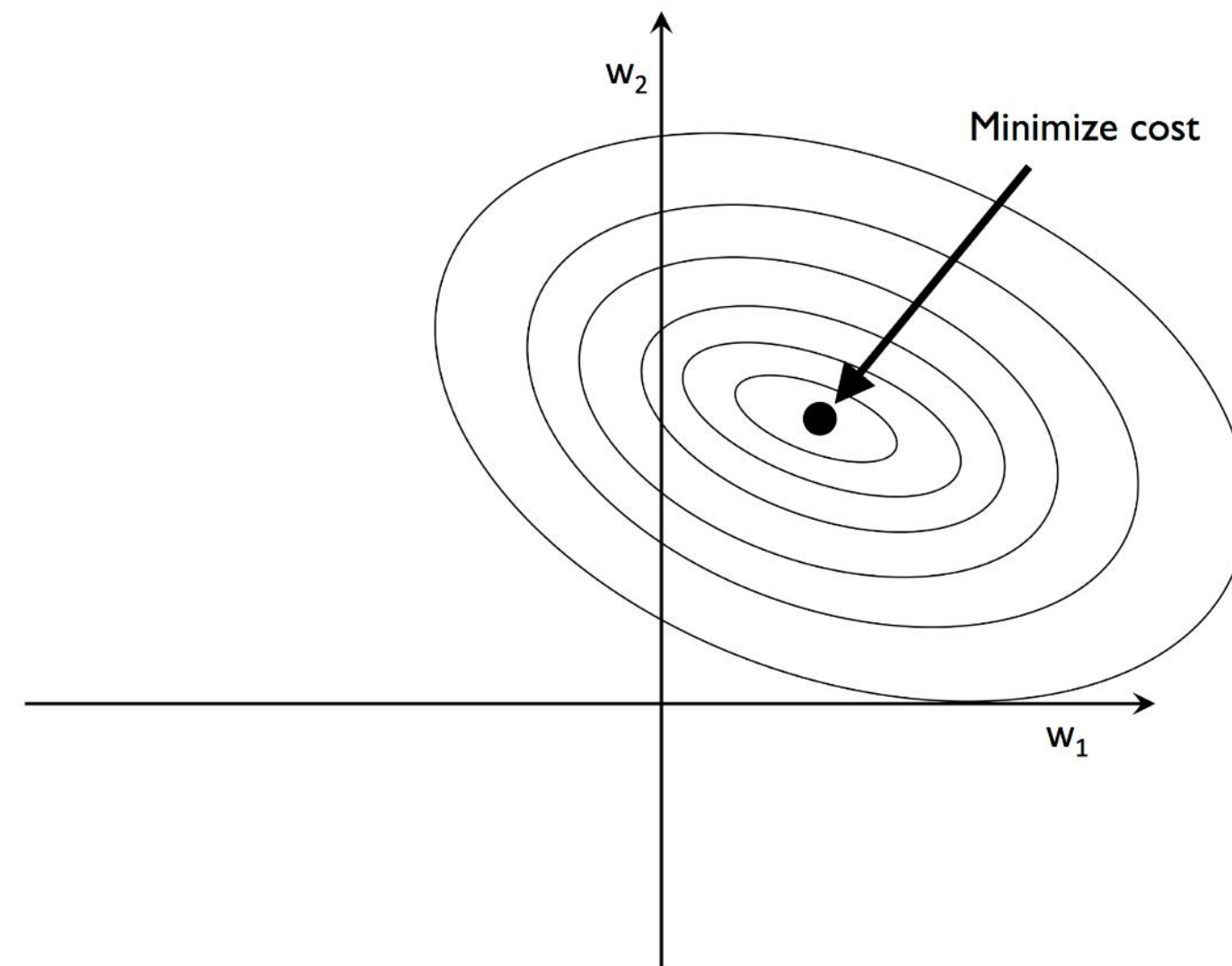
- Use to avoid overfitting in linear models
- Idea: can we reduce complexity of our linear model by minimizing weights?



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

Penalizing extreme weights (w)

If the original cost function looks like:

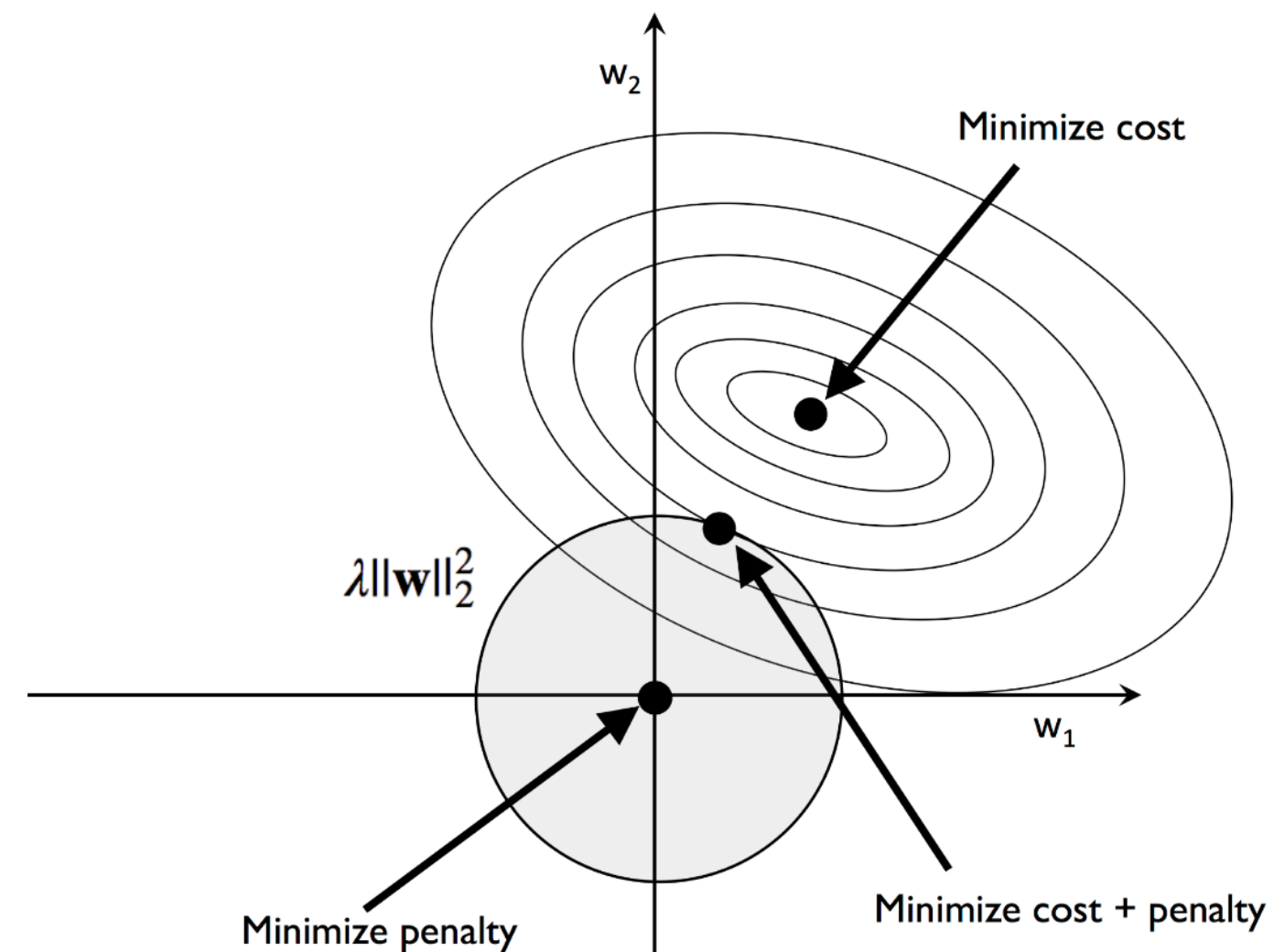
$$\arg \min_w C(f(w, x), y)$$

we can add a **regularization term**:

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

Regularization: Ridge

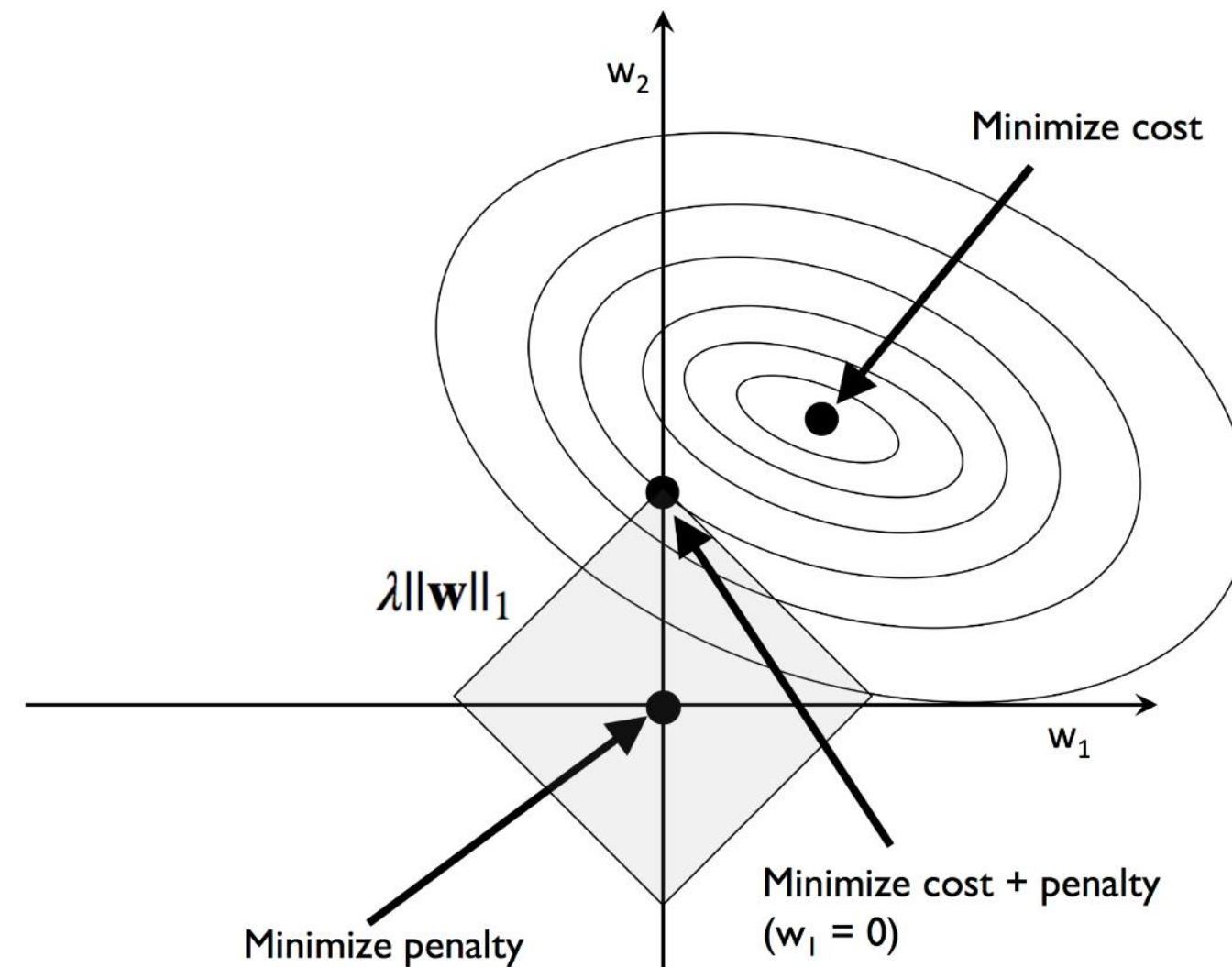
- Coefficients are encouraged to be "small"
- Uses L_2 norm: $\|w\|_2 = \sqrt{\sum_{j=1}^m w_j^2}$



From PML

Regularization: LASSO

- Coefficients are (likely) driven to zero
- Uses L_1 norm: $\|w\|_1 = \sum_{j=1}^m |w_j|$



From PML

Regularization: ElasticNet

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

Regularization: ElasticNet

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

```
In [40]: from sklearn.linear_model import LogisticRegression

lr = LogisticRegression(penalty='l2', # default
                        C=1.0,        # weight on regularization, 1/lambda above
                        l1_ratio=None # only used when penalty is 'elasticnet'
                        )
```