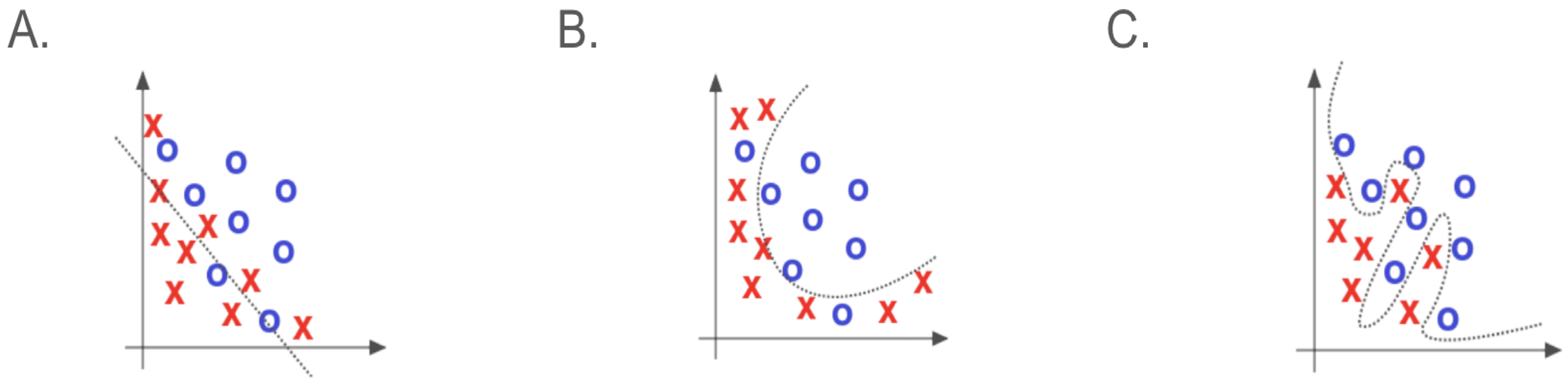
Intro to classification - Logistic regression - 3

One should look for what is and not what he thinks should be. (Albert Einstein)

Warm up: overfitting

- Based on the graphs below, which model is **overfitted**? Which is **underfitted**? Which appears to be a **good fit**?
- **Share your responses** in the chat or aloud



Source

Module completion checklist

Objectives	Complete
Analyze the model to determine if / when overfitting occurs	
Demonstrate how to tune the model using grid search cross-validation	

Accuracy on train vs. accuracy on test

- Our accuracy score for test data was:

```
0.9458577951728636
```

- Take a look at the accuracy score for the training data

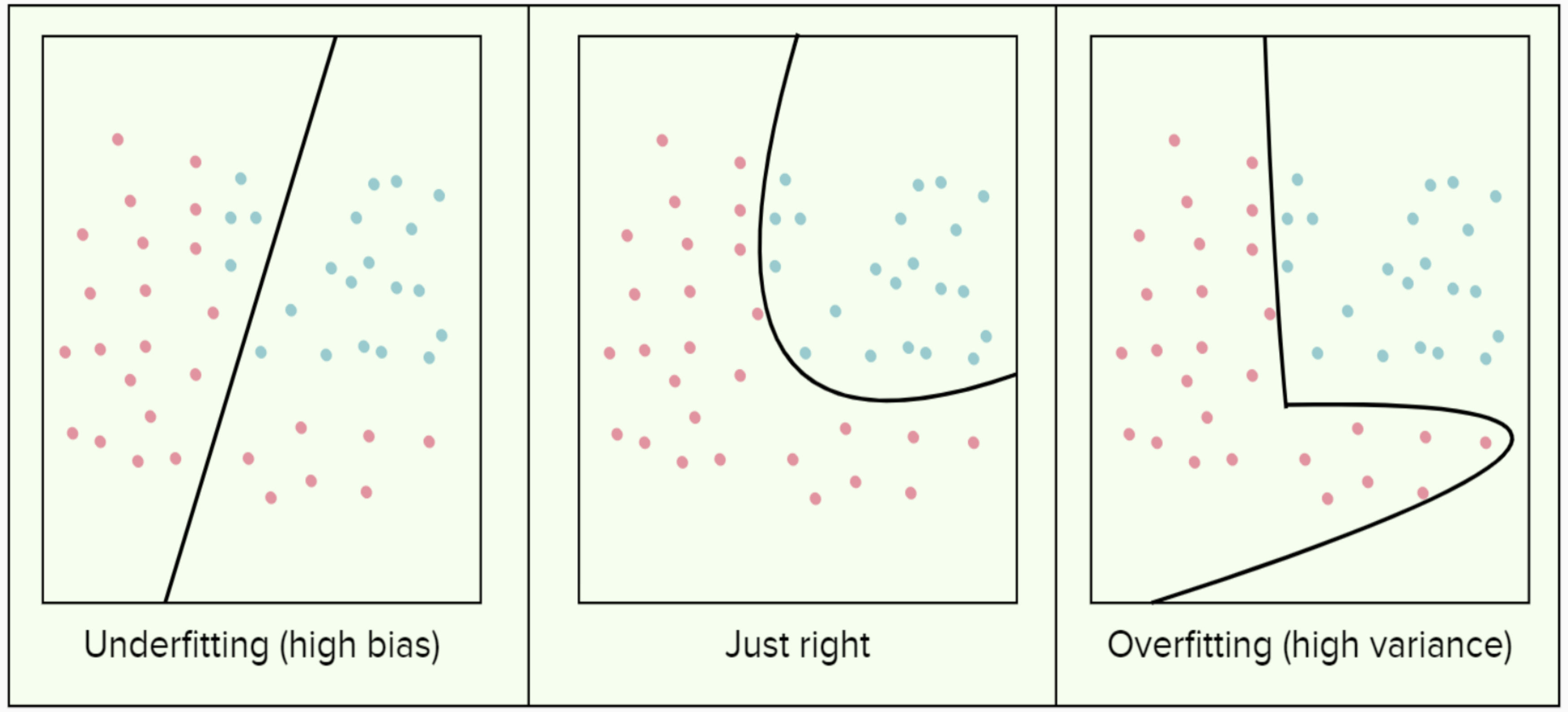
```
# Compute trained model accuracy score.
trained_accuracy_score = logistic_regression_model.score(X_train_scaled, y_train)
print("Accuracy on train data: " , trained_accuracy_score)
```

```
Accuracy on train data:  0.9535923958624546
```

- Did our model underperform?
- Is there a big difference in `train` and `test` accuracy?
- If there is a difference, the problem usually lies in **overfitting**

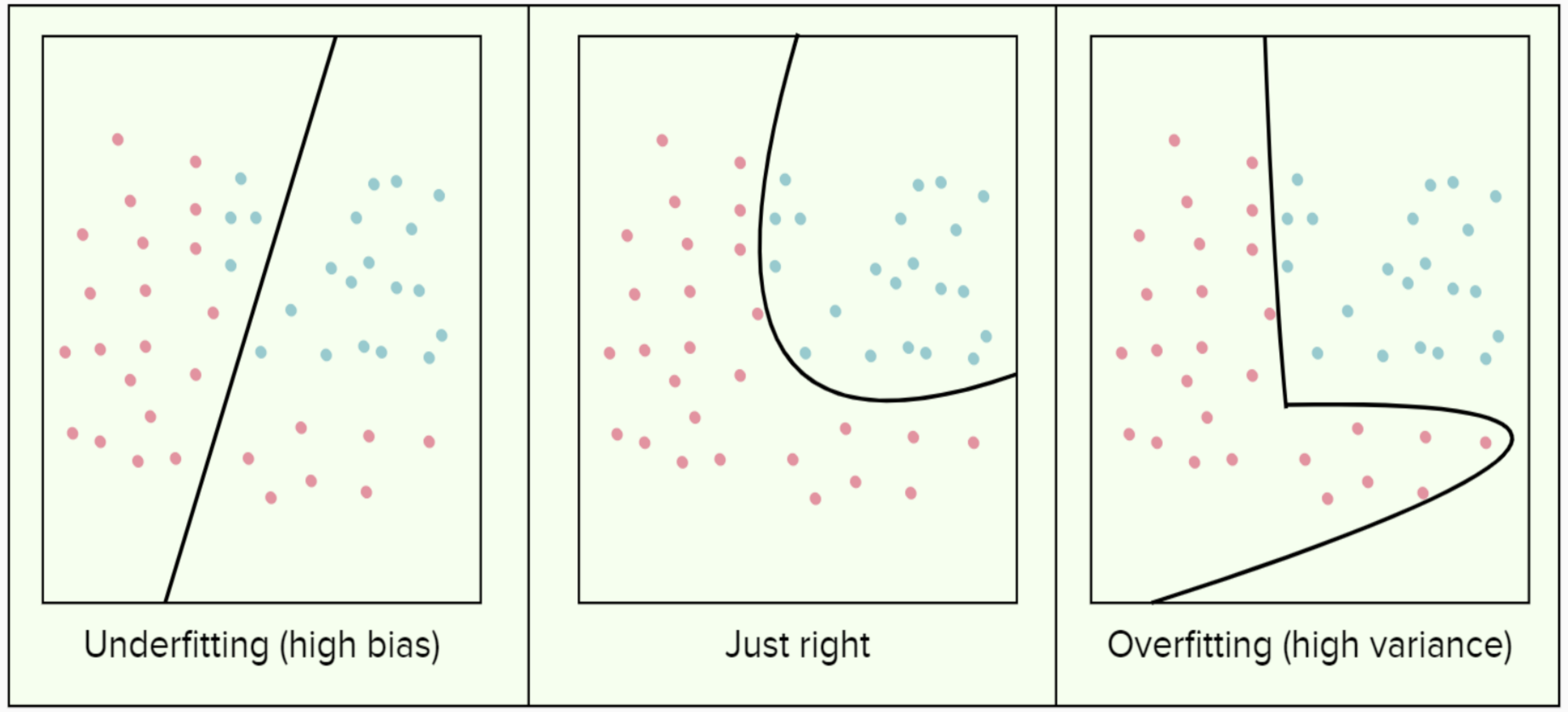
When overfitting occurs

- An overfitted model consists of high variance and usually shows a drastically higher accuracy in the training data because it doesn't generalize well to new data
- Creating a model that fits training data too well will lead to poor generalization and poor performance on new data



When overfitting occurs (cont'd)

- A model might treat **noise** as actual artifacts of the data, so when it encounters new data with new noise, it underperforms
- It might use **too many predictors** that only contribute tiny portions to variation in our data
- The train set might not be an **accurate representation** of the data, but a partial and inaccurate sample that doesn't translate



How to overcome overfitting

- Use so-called **soft-margin classifiers** to:
  - Utilize penalization constants and methods to make the model less prone to noise
  - Tune them to use the optimal parameters for best model performance
- Use **feature selection** and/ or **feature extraction** methods to:
  - Capture a few main features responsible for the most variation in the data
  - Discard the features that don't account for variation in the data
- **Gather more data**

Tuning logistic regression model

- Recall the two parameters that we mentioned before:
  - `penalty`: a regularization technique used to tune the model (either `l1`, a.k.a. *Lasso*, or `l2`, a.k.a. *Ridge*; default is `l2`)
  - `C`: a regularization constant used to amplify the effect of the regularization method (a value between $[0, \infty]$; default is `1`)
- These two parameters control a so-called **regularization term** that adds a penalty as the model complexity increases with added variables
- These two parameters play a key role in mitigating overfitting and feature pruning

Regularization techniques in logistic regression

- As you may know, any ML algorithm optimizes some *cost function* $f(x)$
- In logistic regression, `l1` (*Lasso*) adds a term to that function like so:

$$f(x) + C\sum_{j=1}^{n} |b_j|$$

- While `l2` (*Ridge*) adds a term like so:

$$f(x) + C\sum_{j=1}^{n} b_j^2$$

- You can see that *Lasso* uses the absolute value b_j, while *Ridge* uses a squared b_j
- That term, when added to the original *cost function*, **dampens** the margins of our classifier, making it more **forgiving** of the misclassification of some points that might be noise

Lasso vs. Ridge

Lasso (11)

$$C\sum_{j=1}^{n}|b_j|$$

- Stands for **L**east **A**bsolute **S**hrinkage and **S**election **O**perator
- It adds "absolute value of magnitude" of the coefficient as a penalty term to the loss function
- **Shrinks** (as the name suggests) the less important features' coefficients to zero, which leads to **removal** of some features

Ridge (12)

$$C\sum_{j=1}^{n}b_j^2$$

- Adds "squared magnitude" of coefficient as penalty term to the loss function
- **Dampens** the less important features' coefficients making them less significant, which leads to **weighting** of the features according to their importance

What is the role of C?

There are **4 scenarios** that might happen with a classifier with respect to C:

- Scenario 1: $C = 0$
  - The classifier becomes an **OLS** problem (i.e., Ordinary Least Squares, or just a strict regression without any penalization)
  - Since $0 \times anything = 0$, we are just left with optimizing $f(x)$, which is a definite **overfitting** problem
- Scenario 2: $C = small$
  - We still run into an **overfitting** problem
  - Since C will not "magnify" the effect of the penalty term enough

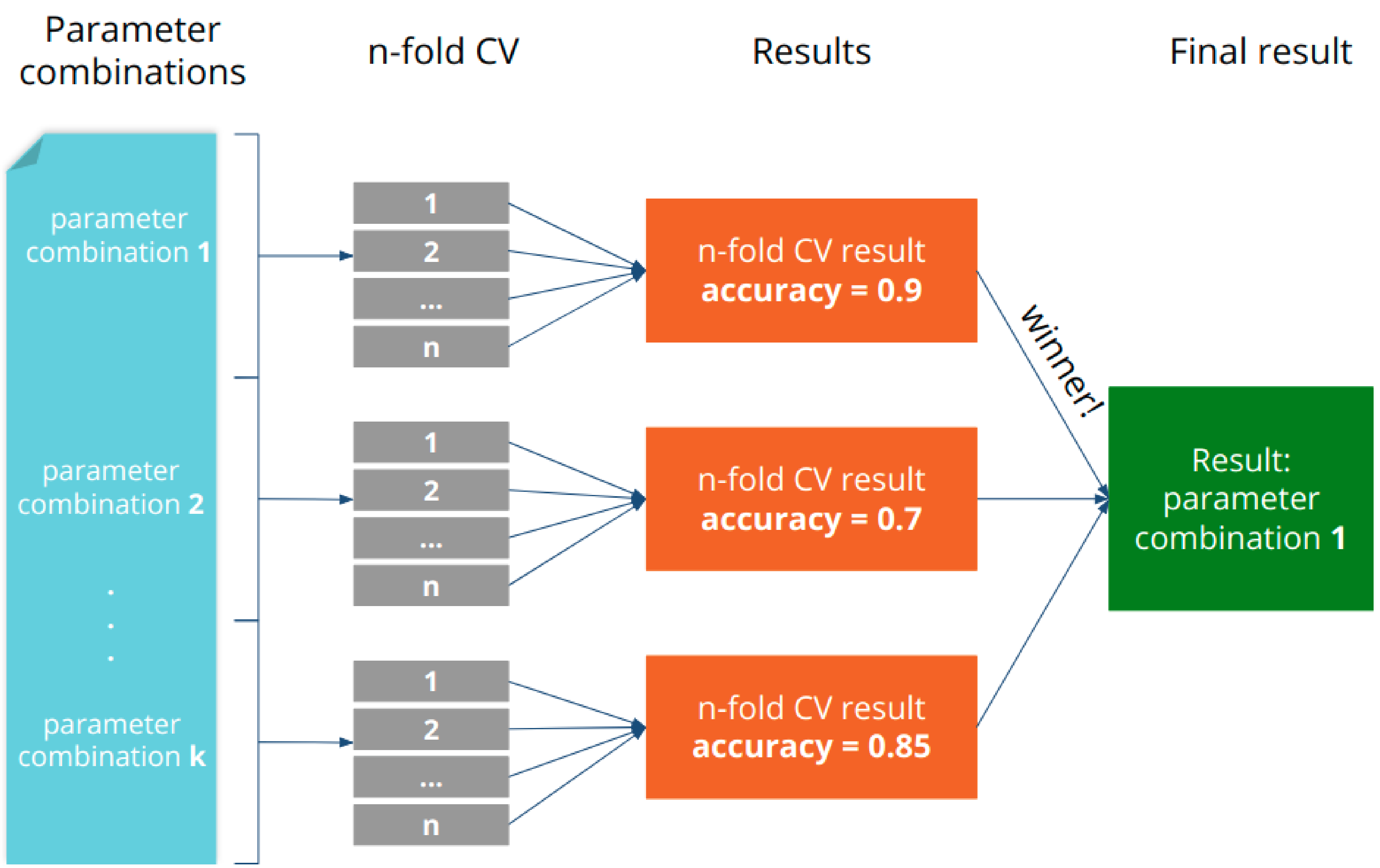
What is the role of C? (cont'd)

- Scenario 3: $C = large$
  - We run into an **underfitting** problem, where we've weighted and dampened the coefficients too much and we made the model too general
- Scenario 4: $C = optimal$
  - We have a **good, robust, and generalizable model** that works well with new data
  - The model ignores most of the noise while preserving the main pattern in data
- We can pick the right combination of parameters using a technique called **grid search cross-validation**

Module completion checklist

Objectives	Complete
Analyze the model to determine if / when overfitting occurs	✔
Demonstrate how to tune the model using grid search cross-validation	

What does grid search cross-validation do?



scikit-learn: model_selection.GridSearchCV

- `estimator` is the name of `sklearn` algorithm to optimize
- `param_grid` is a dictionary or list of parameters to optimize
- `cv` is an `int` of `n` for `n-fold` cross-validation
- `verbose` is an `int` of how much verbosity in messages you want to see as the function runs

sklearn.model_selection.GridSearchCV

class `sklearn.model_selection.` **GridSearchCV** (*estimator, param_grid, scoring=None, fit_params=None, n_jobs=1, iid=True, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score='raise', return_train_score='warn'*) [source]

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a “fit” and a “score” method. It also implements “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Click here for full documentation

Prepare parameters for optimization

```
# Create regularization penalty space.
penalty = ['l1', 'l2']
```

```
# Create regularization constant space.
C = np.logspace(0, 10, 10)
print("Regularization constant: ", C)
```

```
Regularization constant:  [1.00000000e+00 1.29154967e+01 1.66810054e+02 2.15443469e+03
 2.78255940e+04 3.59381366e+05 4.64158883e+06 5.99484250e+07
 7.74263683e+08 1.00000000e+10]
```

```
# Create hyperparameter options dictionary.
hyperparameters = dict(C = C, penalty = penalty)
print(hyperparameters)
```

```
{'C': array([1.00000000e+00, 1.29154967e+01, 1.66810054e+02, 2.15443469e+03,
       2.78255940e+04, 3.59381366e+05, 4.64158883e+06, 5.99484250e+07,
       7.74263683e+08, 1.00000000e+10]), 'penalty': ['l1', 'l2']}
```

Set up cross-validation logistic function

```
# Grid search 10-fold cross-validation with above parameters.
clf = GridSearchCV(linear_model.LogisticRegression(solver='liblinear'), #<- function to optimize
                   hyperparameters,                              #<- grid search parameters
                   cv = 10,                                      #<- 10-fold cv
                   verbose = 0)                                  #<- no messages to show
```

```
# Fit CV grid search.
best_model = clf.fit(X_train_scaled, y_train)
best_model
```

```
GridSearchCV(cv=10, estimator=LogisticRegression(solver='liblinear'),
             param_grid={'C': array([1.00000000e+00, 1.29154967e+01, 1.66810054e+02,
2.15443469e+03,
       2.78255940e+04, 3.59381366e+05, 4.64158883e+06, 5.99484250e+07,
       7.74263683e+08, 1.00000000e+10]),
                         'penalty': ['l1', 'l2']})
```

Check best parameters found by CV

```
# Get best penalty and constant parameters.
penalty = best_model.best_estimator_.get_params()['penalty']
constant = best_model.best_estimator_.get_params()['C']
print('Best penalty: ', penalty)
```

```
Best penalty:  l2
```

```
print('Best C: ', constant)
```

```
Best C:  1.0
```

- It seems like our grid search CV has found that `l1` (i.e. *Lasso* regularization method) works better than the default `l2` (i.e. *Ridge*)
- It also shows that the default `C`, which is `1`, creates a big enough soft margin for our classifier

Predict using the best model parameters

- Now let's use the tuned model to predict on our test data

```
# Predict on test data using best model.
best_predicted_values = best_model.predict(X_test_scaled)
print(best_predicted_values)
```

```
[False False False ... False False False]
```

```
# Compute best model accuracy score.
best_accuracy_score = metrics.accuracy_score(y_test, best_predicted_values)
print("Accuracy on test data (best model): ", best_accuracy_score)
```

```
Accuracy on test data (best model):  0.9458577951728636
```

Accuracy on train vs. accuracy on test

- Take a look at the accuracy score for the training data

```
# Compute trained model accuracy score.
trained_accuracy_score = best_model.score(X_train_scaled, y_train)
print("Accuracy on train data: " , trained_accuracy_score)
```

```
Accuracy on train data:  0.9535923958624546
```

- What do you notice by comparing the train and test accuracy?

Assessing the tuned model

- Now we can start to evaluate this model and compare how it works as compared to the previous version

```
# Compute confusion matrix for best model.
best_confusion_matrix = metrics.confusion_matrix(y_test, best_predicted_values)
print(best_confusion_matrix)
```

```
[[1450    0]
 [  83    0]]
```

```
# Create a list of target names to interpret class assignments.
target_names = ['Low value', 'High value']
```

```
# Compute classification report for best model.
best_class_report = metrics.classification_report(y_test, best_predicted_values,
                                                  target_names = target_names)
```

```
print(best_class_report)
```

```
              precision    recall  f1-score   support

   Low value       0.95      1.00      0.97      1450
  High value       0.00      0.00      0.00        83

    accuracy                           0.95      1533
   macro avg       0.47      0.50      0.49      1533
```

Save accuracy score

- Let's save our final model

```
model_final = {'metrics' : "accuracy",
                                'values' : round(best_accuracy_score, 4),
                                'model':'logistic_tuned' }
print(model_final)
```

```
{'metrics': 'accuracy', 'values': 0.9459, 'model': 'logistic_tuned'}
```

Get metrics for ROC curve

```
# Get probabilities instead of predicted values.
best_test_probabilities = best_model.predict_proba(X_test_scaled)
print(best_test_probabilities[0:5, ])
```

```
[[0.97142844 0.02857156]
 [0.83103003 0.16896997]
 [0.98592979 0.01407021]
 [0.85404288 0.14595712]
 [0.96287726 0.03712274]]
```

```
# Get probabilities of test predictions only.
best_test_predictions = best_test_probabilities[:, 1]
print(best_test_predictions[0:5])
```

```
[0.02857156 0.16896997 0.01407021 0.14595712 0.03712274]
```

Get metrics for ROC curve (cont'd)

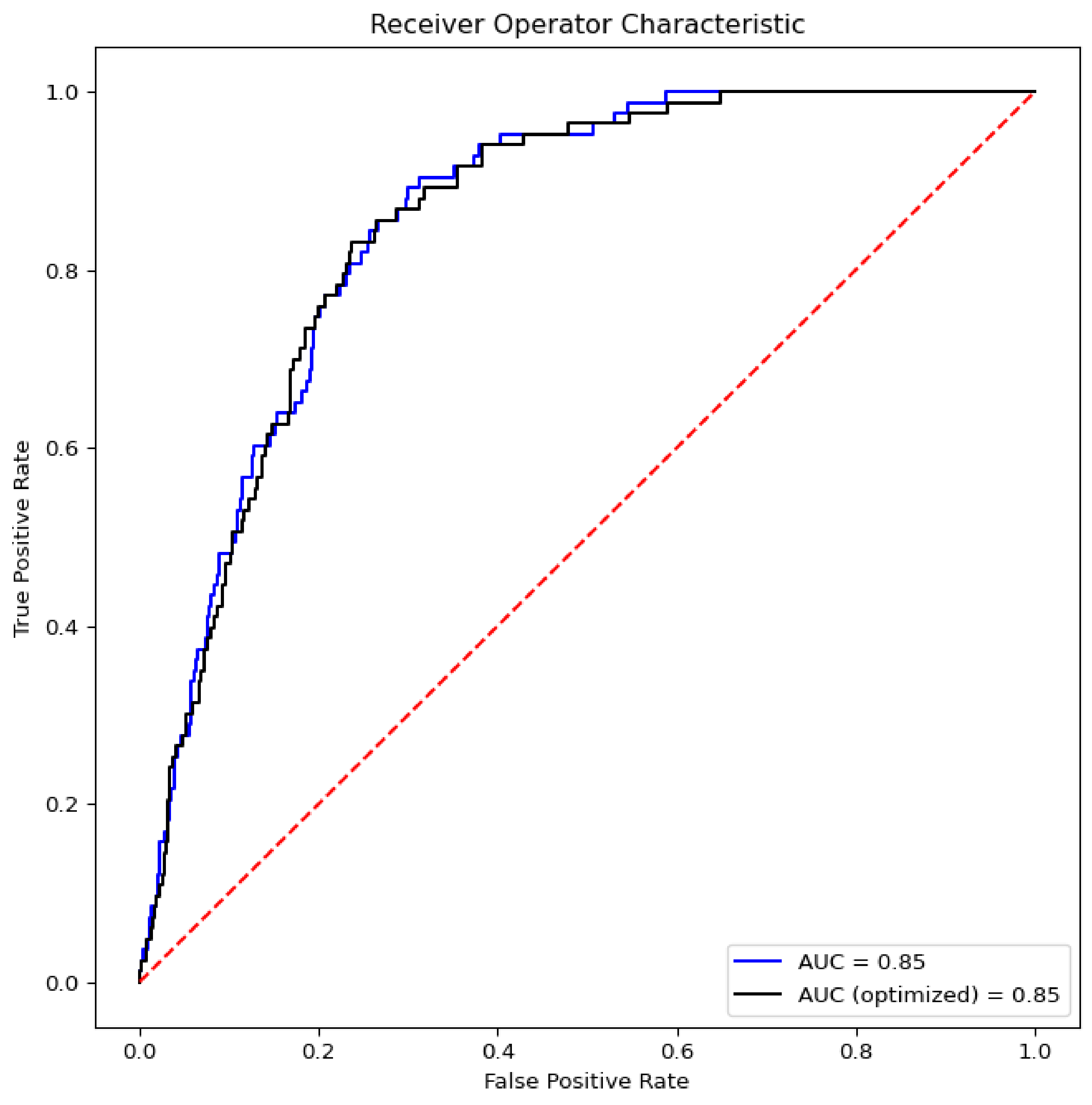
```
# Get ROC curve metrics.
best_fpr, best_tpr, best_threshold = metrics.roc_curve(y_test, best_test_predictions)
best_auc = metrics.auc(best_fpr, best_tpr)
print(best_auc)
```

```
0.8516078105525551
```

Plot ROC curve for both models

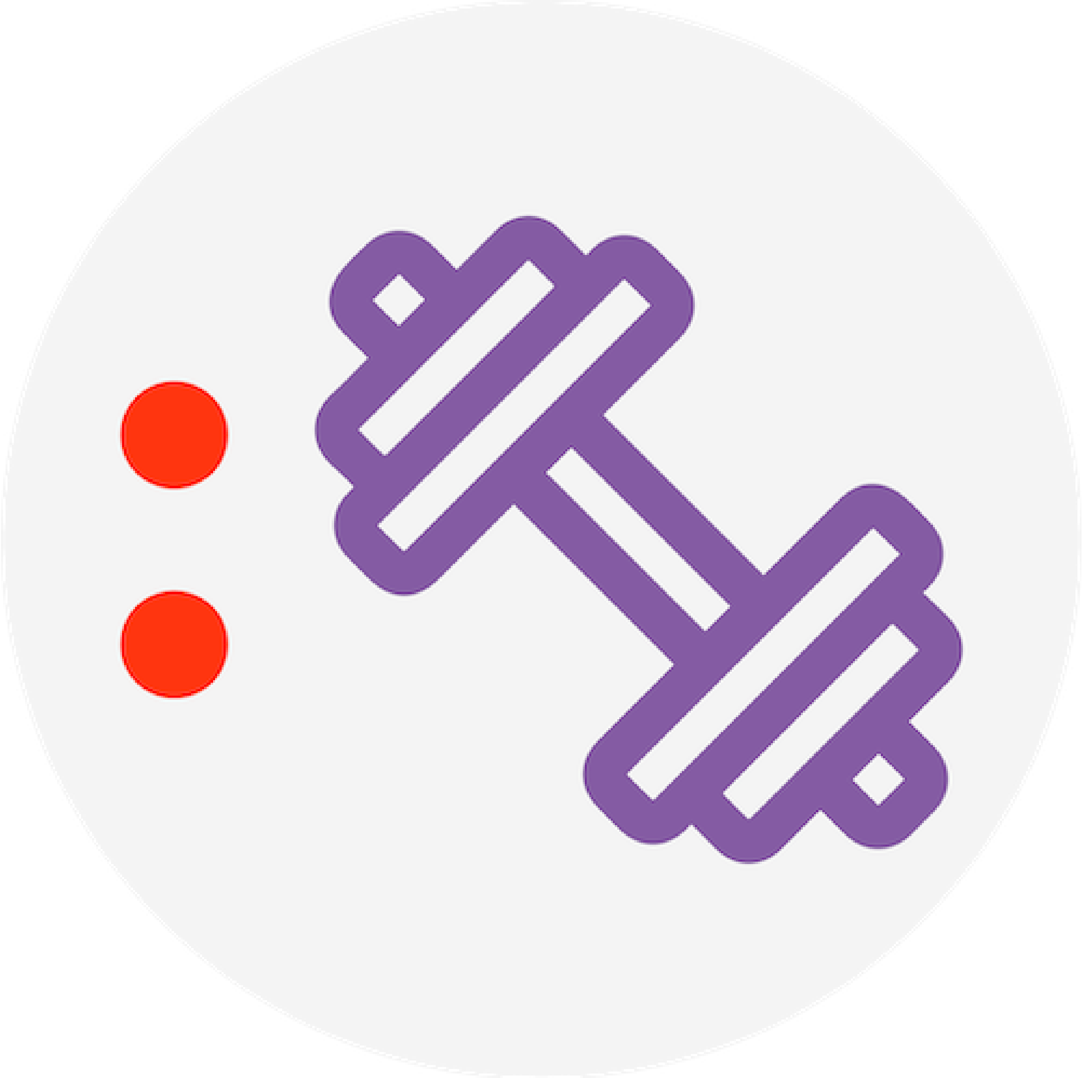
```
# Make an ROC curve plot.
plt.title('Receiver Operator Characteristic')
plt.plot(fpr, tpr, 'blue',
         label = 'AUC = %0.2f'%auc)
plt.plot(best_fpr, best_tpr, 'black',
         label = 'AUC (best) = %0.2f'%best_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.show()
```

- Does it look as though our model has improved?



Knowledge check

Exercise



You are now ready to try Tasks 9-13 in the Exercise for this topic

Module completion checklist

Objectives	Complete
Analyze the model to determine if / when overfitting occurs	✔
Demonstrate how to tune the model using grid search cross-validation	✔

Logistic regression: Topic summary

In this part of the course, we have covered:

- Logistic regression use cases and theory behind it
- Data transformation necessary for logistic regression
- Implementation of logistic regression on a dataset
- Model performance evaluation and tuning

Congratulations on completing this module!