Machine learning with scikit-learn

SUPERVISED LEARNING WITH SCIKIT-LEARN

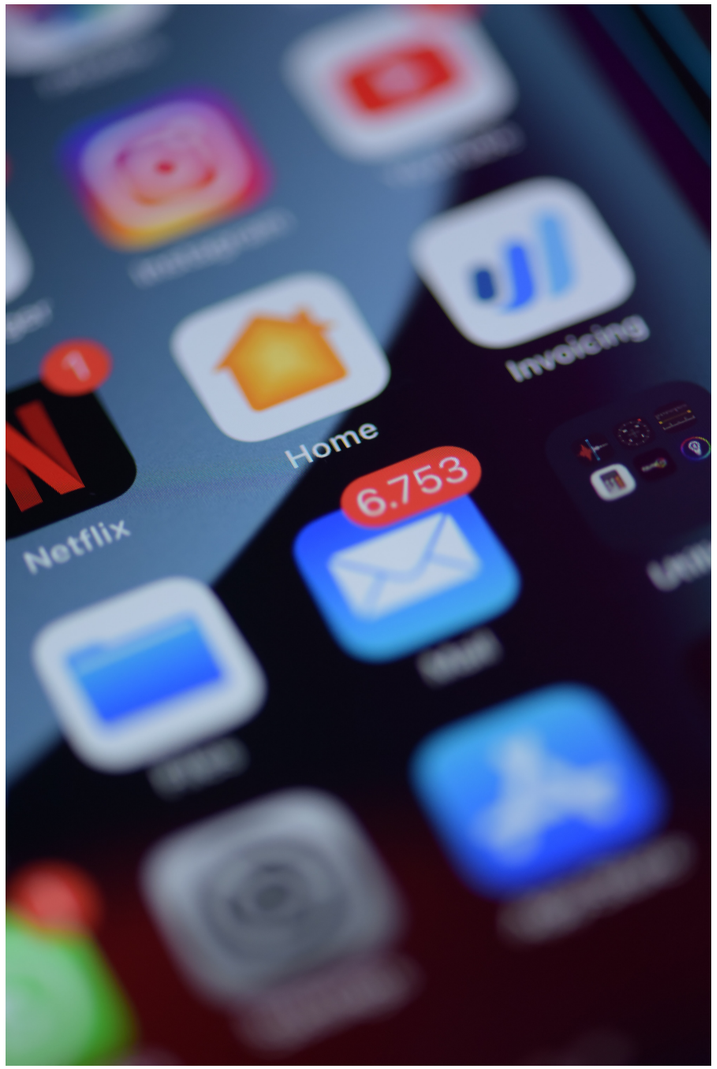
George Boorman
Core Curriculum Manager, DataCamp

datacamp

What is machine learning?

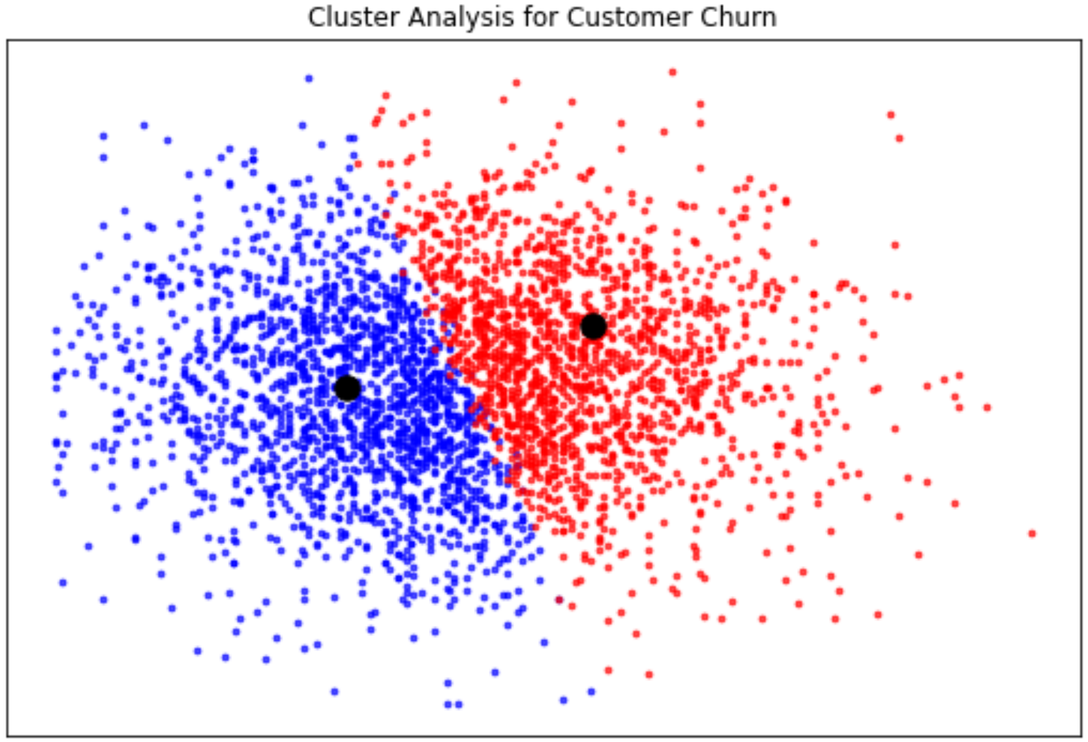
- Machine learning is the process whereby:
  - Computers are given the ability to learn to make decisions from data
  - without being explicitly programmed!

Examples of machine learning



Unsupervised learning

- Uncovering hidden patterns from unlabeled data
- Example:
  - Grouping customers into distinct categories (Clustering)



Supervised learning

- The predicted values are known
- Aim: Predict the target values of unseen data, given the features

	Features					Target variable
	points_per_game	assists_per_game	rebounds_per_game	steals_per_game	blocks_per_game	position
0	26.9	6.6	4.5	1.1	0.4	Point Guard
1	13	1.7	4	0.4	1.3	Center
2	17.6	2.3	7.9	1.00	0.8	Power Forward
3	22.6	4.5	4.4	1.2	0.4	Shooting Guard

Types of supervised learning

- Classification: Target variable consists of categories

- Regression: Target variable is continuous

Naming conventions

- Feature = predictor variable = independent variable
- Target variable = dependent variable = response variable

Before you use supervised learning

- Requirements:
  - No missing values
  - Data in numeric format
  - Data stored in pandas DataFrame or NumPy array

- Perform Exploratory Data Analysis (EDA) first

scikit-learn syntax

```
from sklearn.module import Model
model = Model()
model.fit(X, y)
predictions = model.predict(X_new)
print(predictions)
```

```
array([0, 0, 0, 0, 1, 0])
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

The classification challenge

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Classifying labels of unseen data

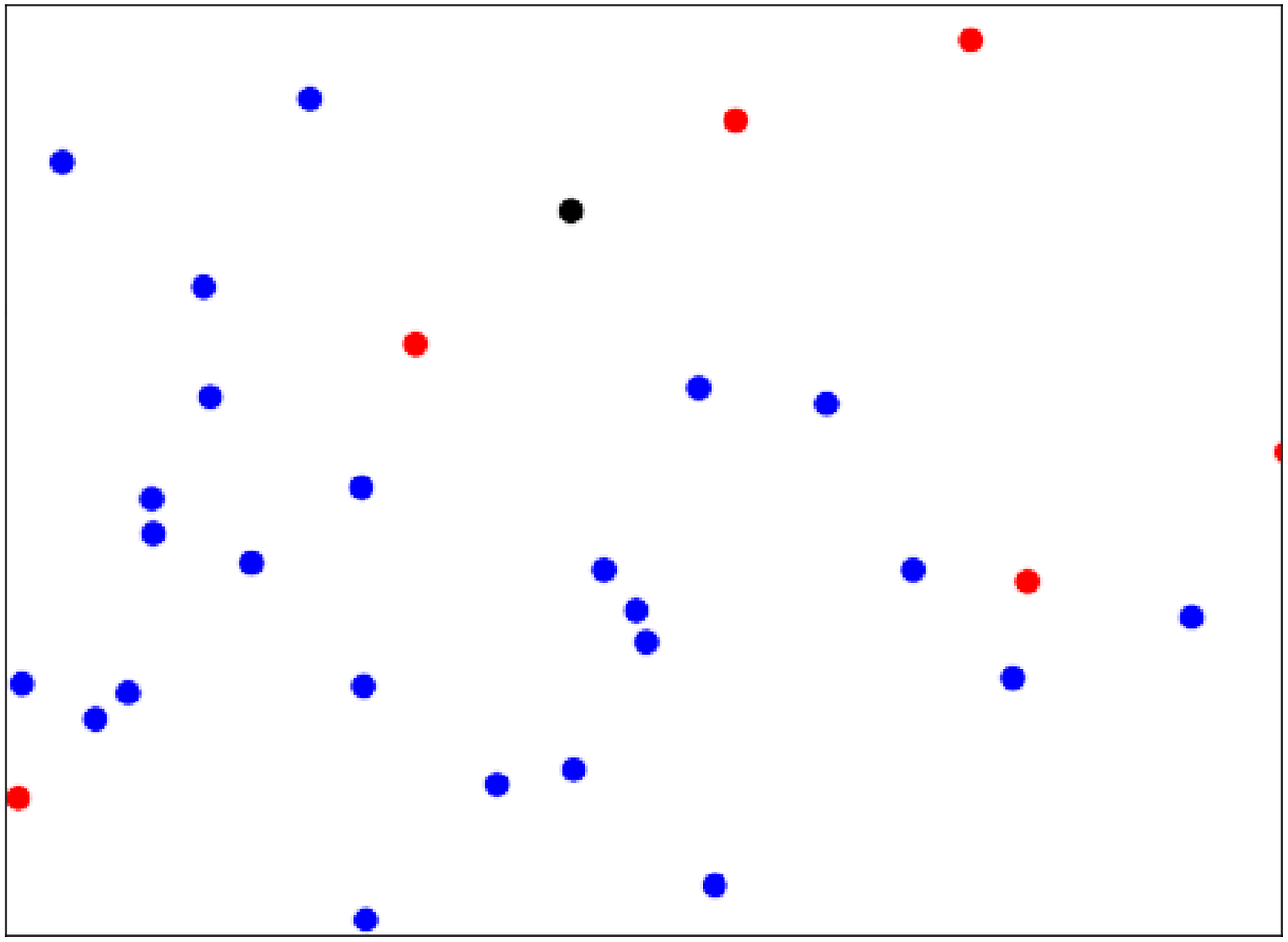
1. Build a model
2. Model learns from the labeled data we pass to it
3. Pass unlabeled data to the model as input
4. Model predicts the labels of the unseen data

- Labeled data = training data

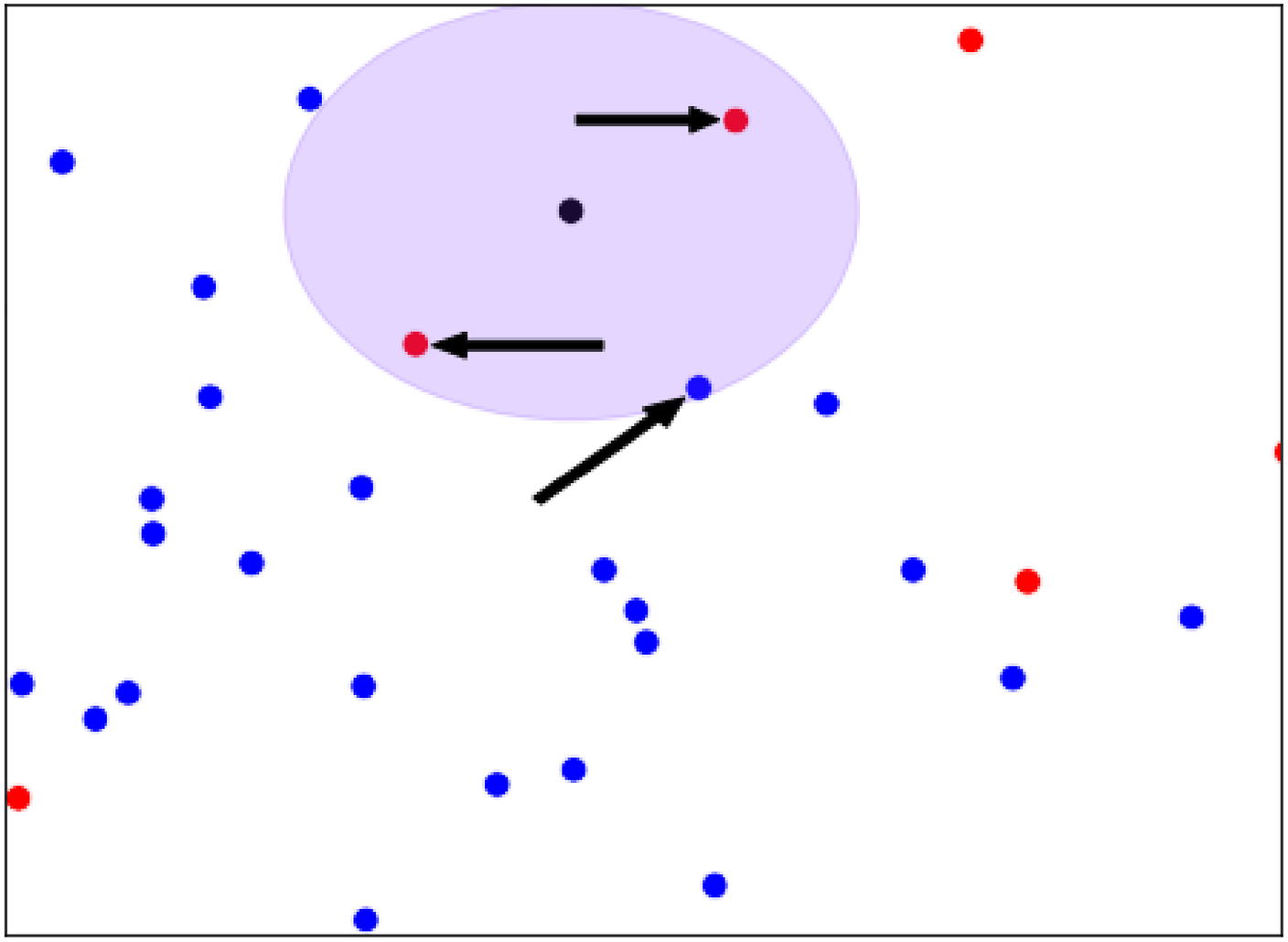
k-Nearest Neighbors

- Predict the label of a data point by
  - Looking at the `k` closest labeled data points
  - Taking a majority vote

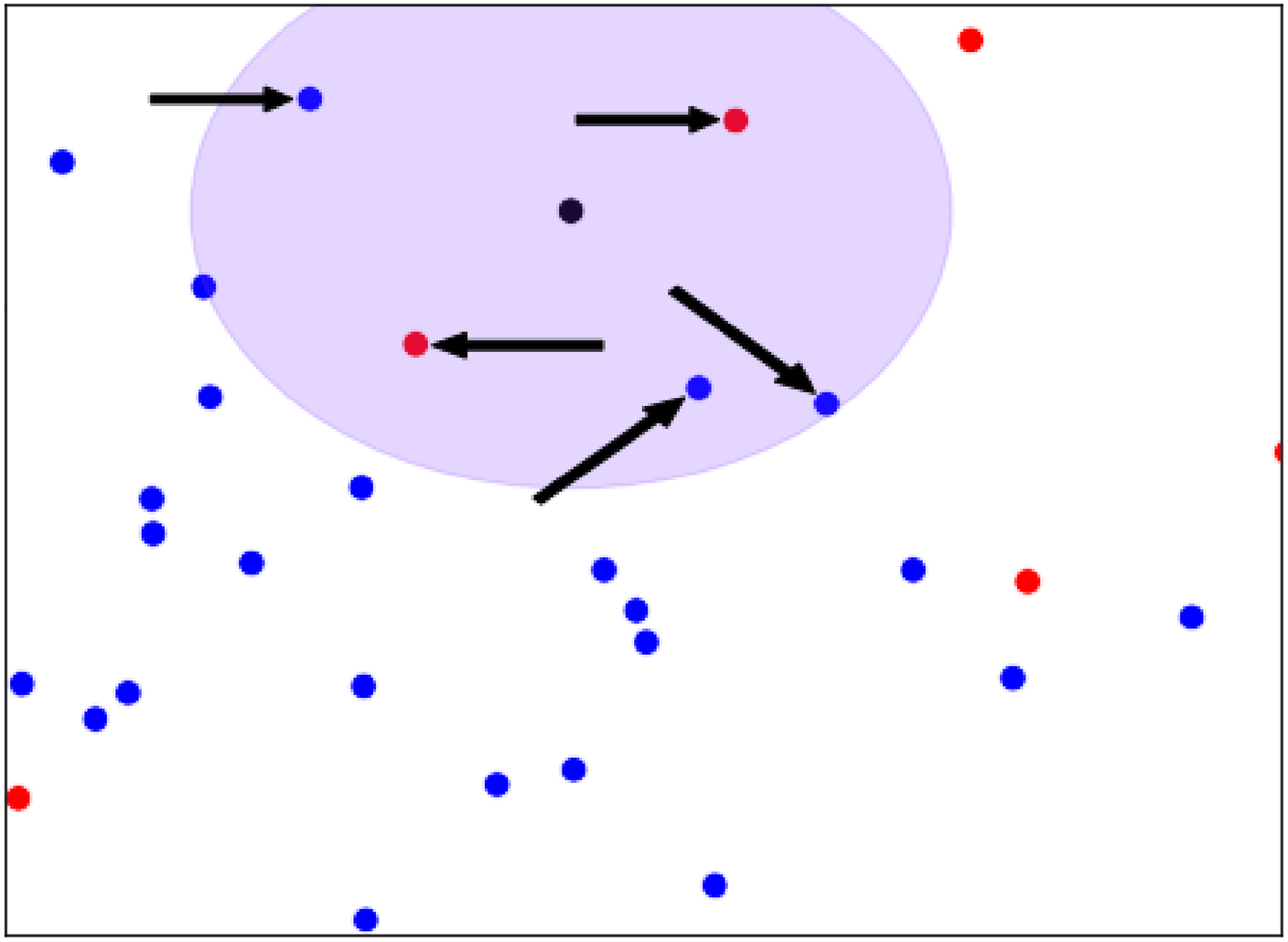
k-Nearest Neighbors



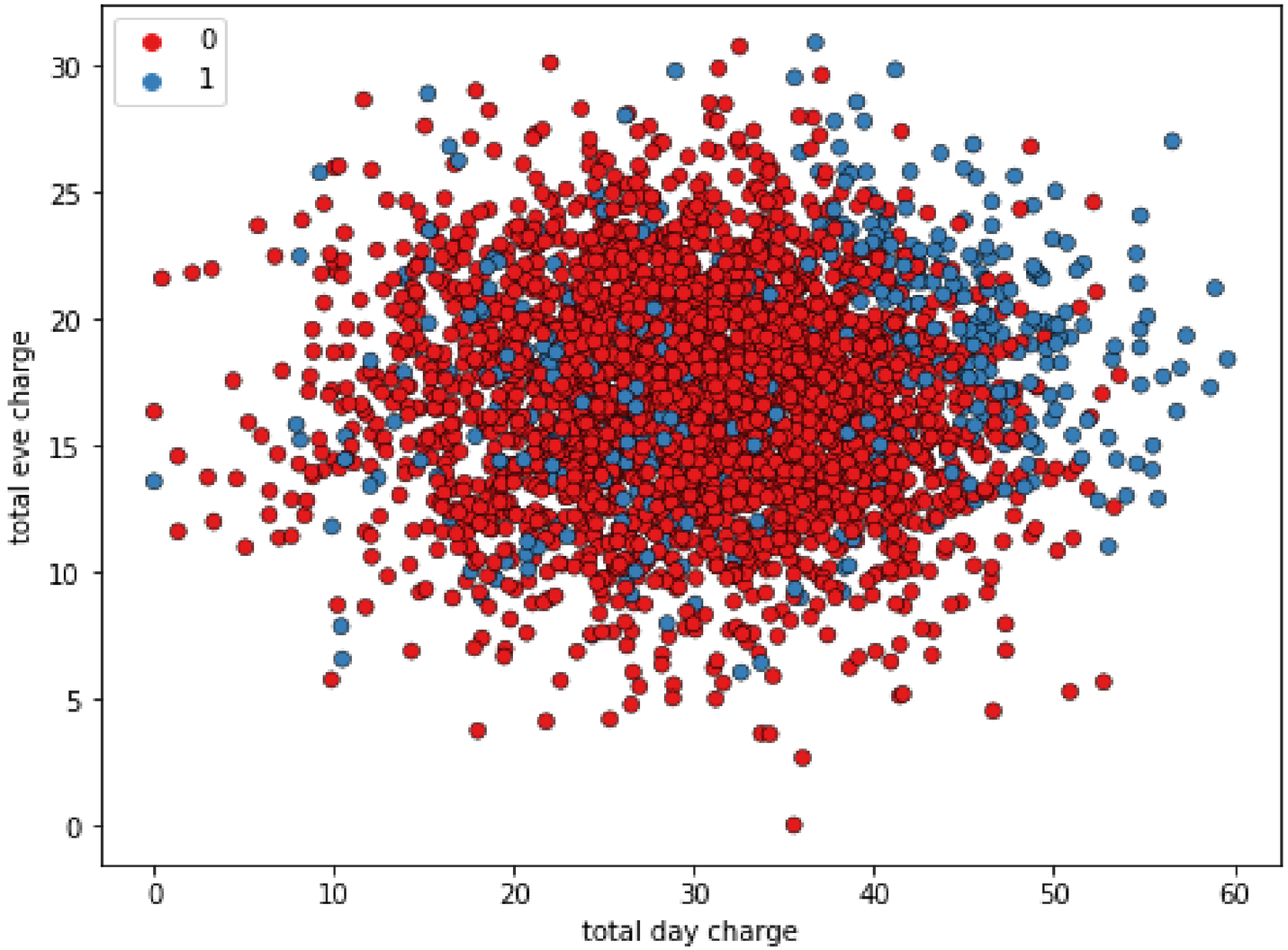
k-Nearest Neighbors



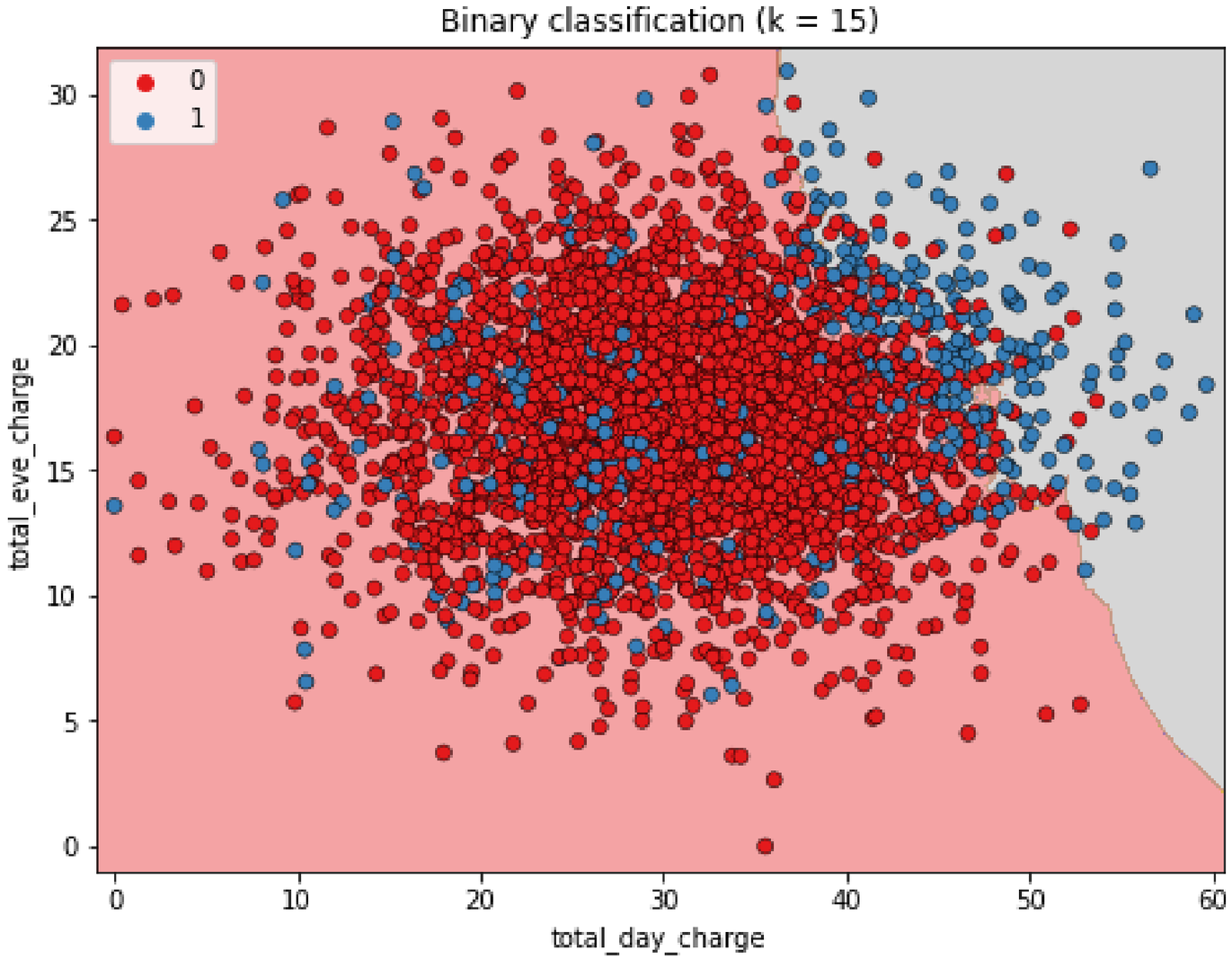
k-Nearest Neighbors



KNN Intuition



KNN Intuition



Using scikit-learn to fit a classifier

```
from sklearn.neighbors import KNeighborsClassifier
X = churn_df[["total_day_charge", "total_eve_charge"]].values
y = churn_df["churn"].values
print(X.shape, y.shape)
```

```
(3333, 2), (3333,)
```

```
knn = KNeighborsClassifier(n_neighbors=15)
knn.fit(X, y)
```

Predicting on unlabeled data

```
X_new = np.array([[56.8, 17.5],
                  [24.4, 24.1],
                  [50.1, 10.9]])
print(X_new.shape)
```

```
(3, 2)
```

```
predictions = knn.predict(X_new)
print('Predictions: {}'.format(predictions))
```

```
Predictions: [1 0 0]
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Measuring model performance

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Measuring model performance

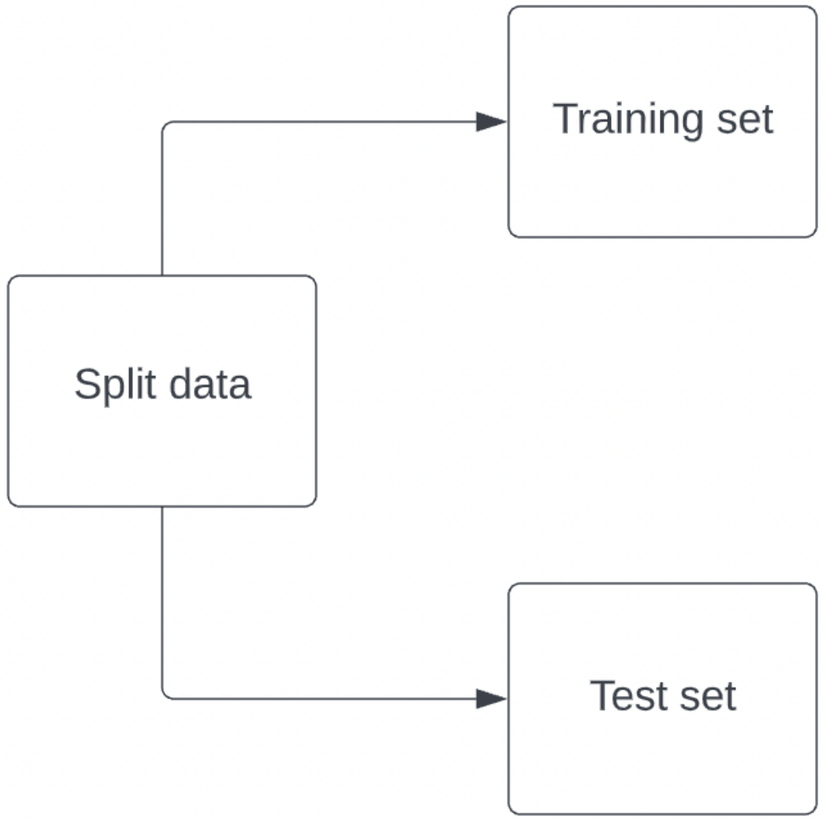
- In classification, accuracy is a commonly used metric
- Accuracy:

$$\frac{correct\ predictions}{total\ observations}$$

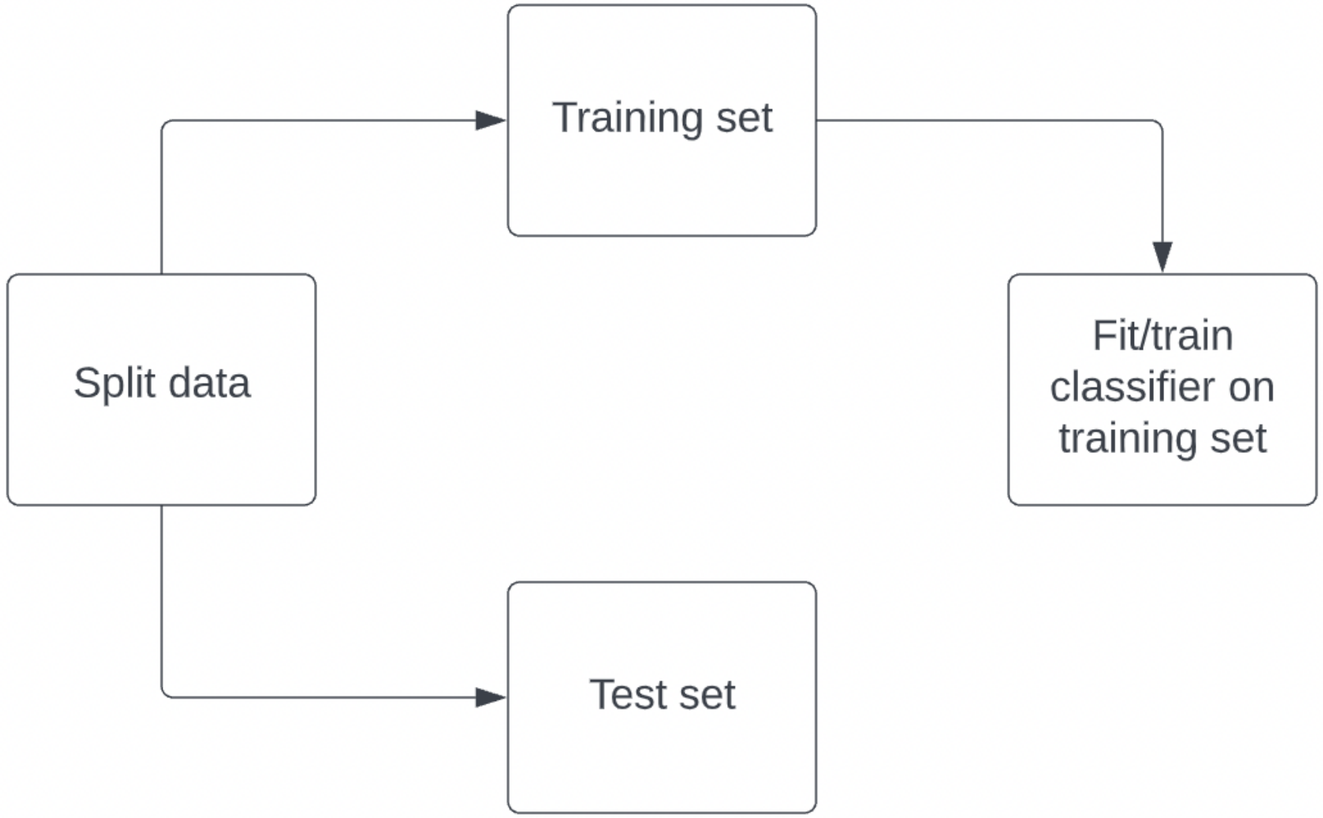
Measuring model performance

- How do we measure accuracy?
- Could compute accuracy on the data used to fit the classifier
- NOT indicative of ability to generalize

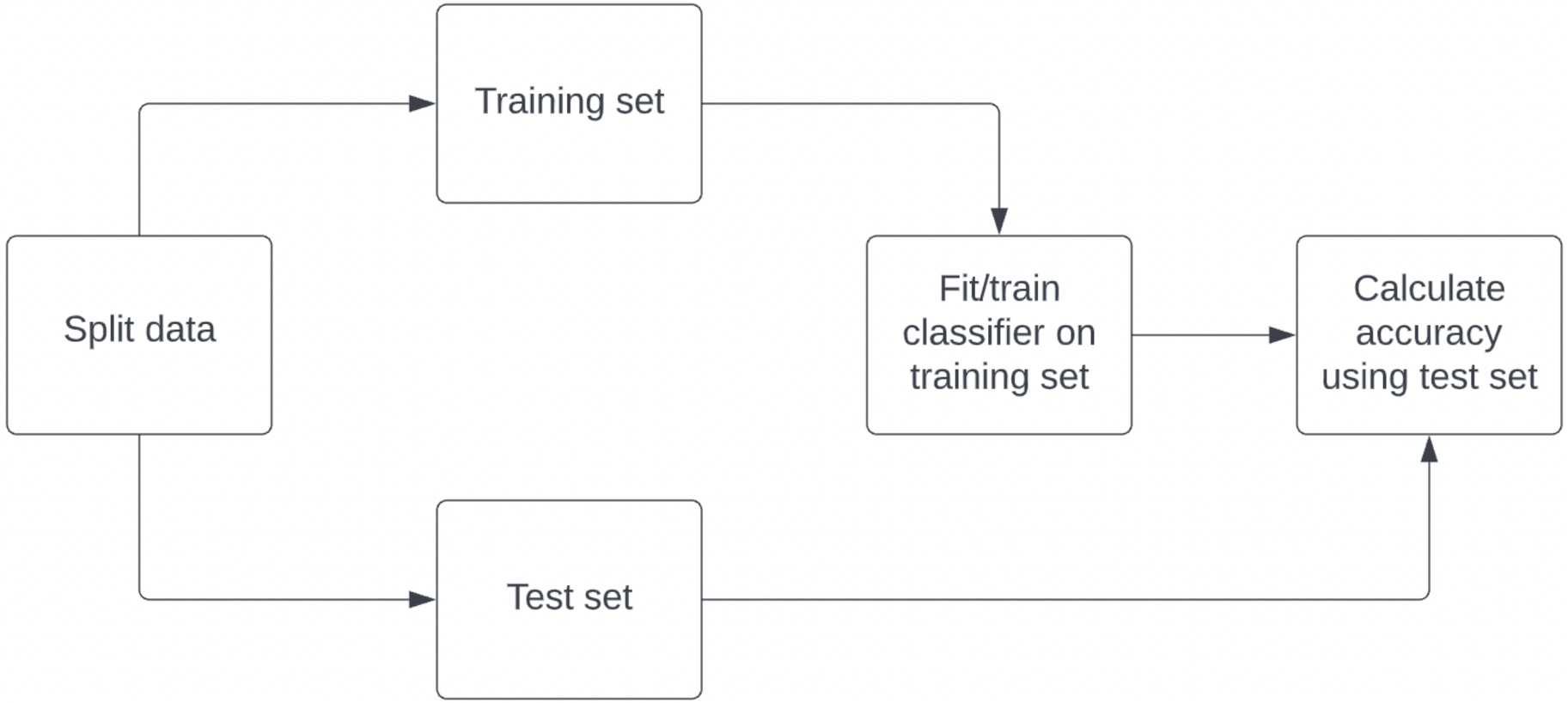
Computing accuracy



Computing accuracy



Computing accuracy



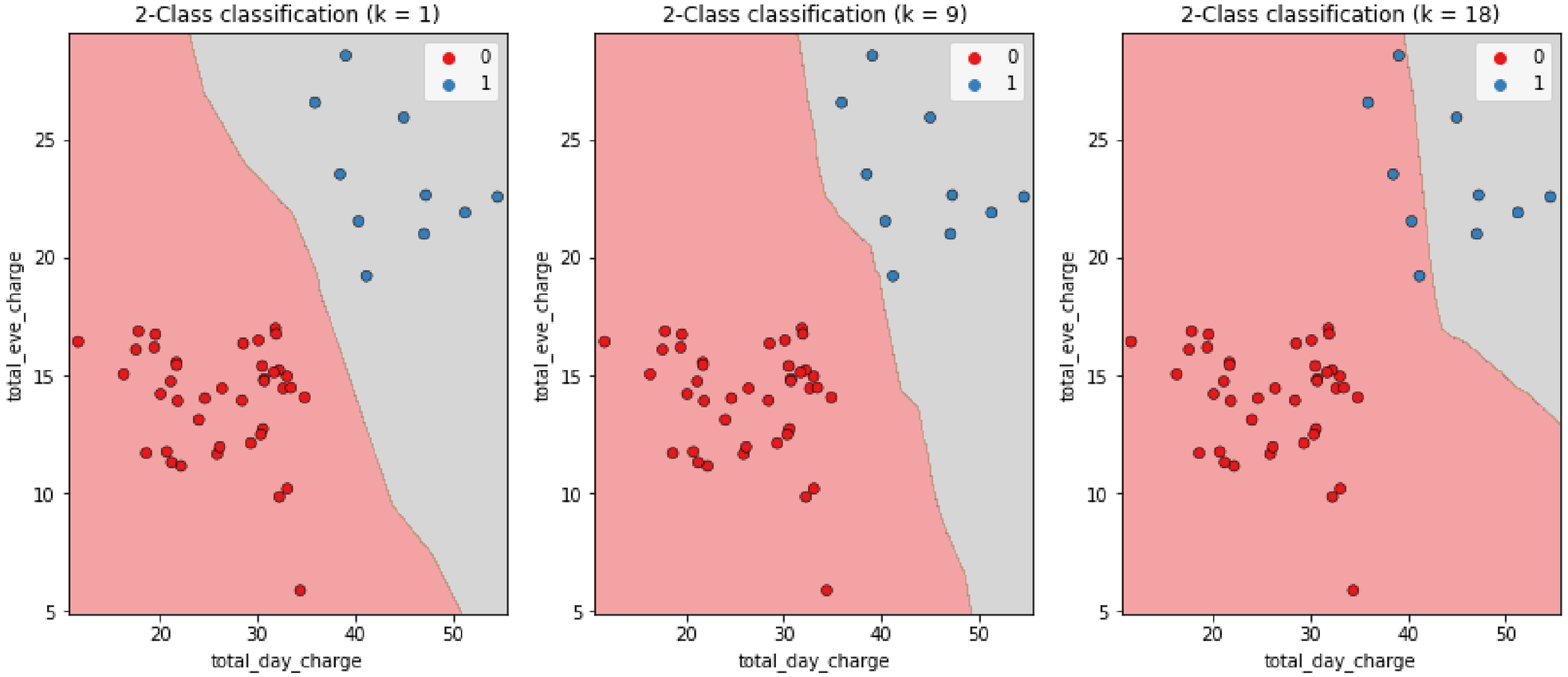
Train/test split

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=21, stratify=y)
knn = KNeighborsClassifier(n_neighbors=6)
knn.fit(X_train, y_train)
print(knn.score(X_test, y_test))
```

```
0.8800599700149925
```

Model complexity

- Larger k = less complex model = can cause underfitting
- Smaller k = more complex model = can lead to overfitting



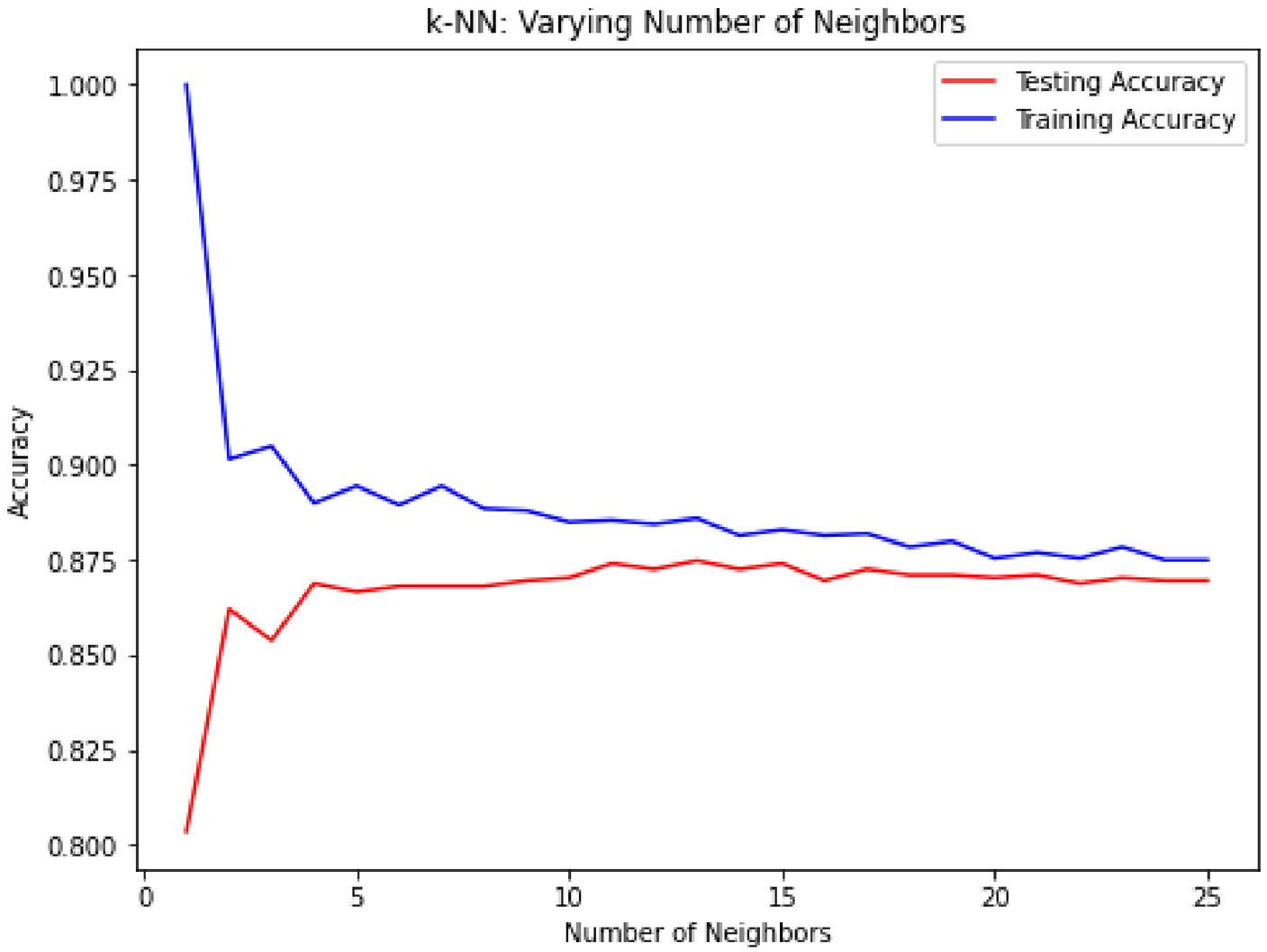
Model complexity and over/underfitting

```
train_accuracies = {}
test_accuracies = {}
neighbors = np.arange(1, 26)
for neighbor in neighbors:
    knn = KNeighborsClassifier(n_neighbors=neighbor)
    knn.fit(X_train, y_train)
    train_accuracies[neighbor] = knn.score(X_train, y_train)
    test_accuracies[neighbor] = knn.score(X_test, y_test)
```

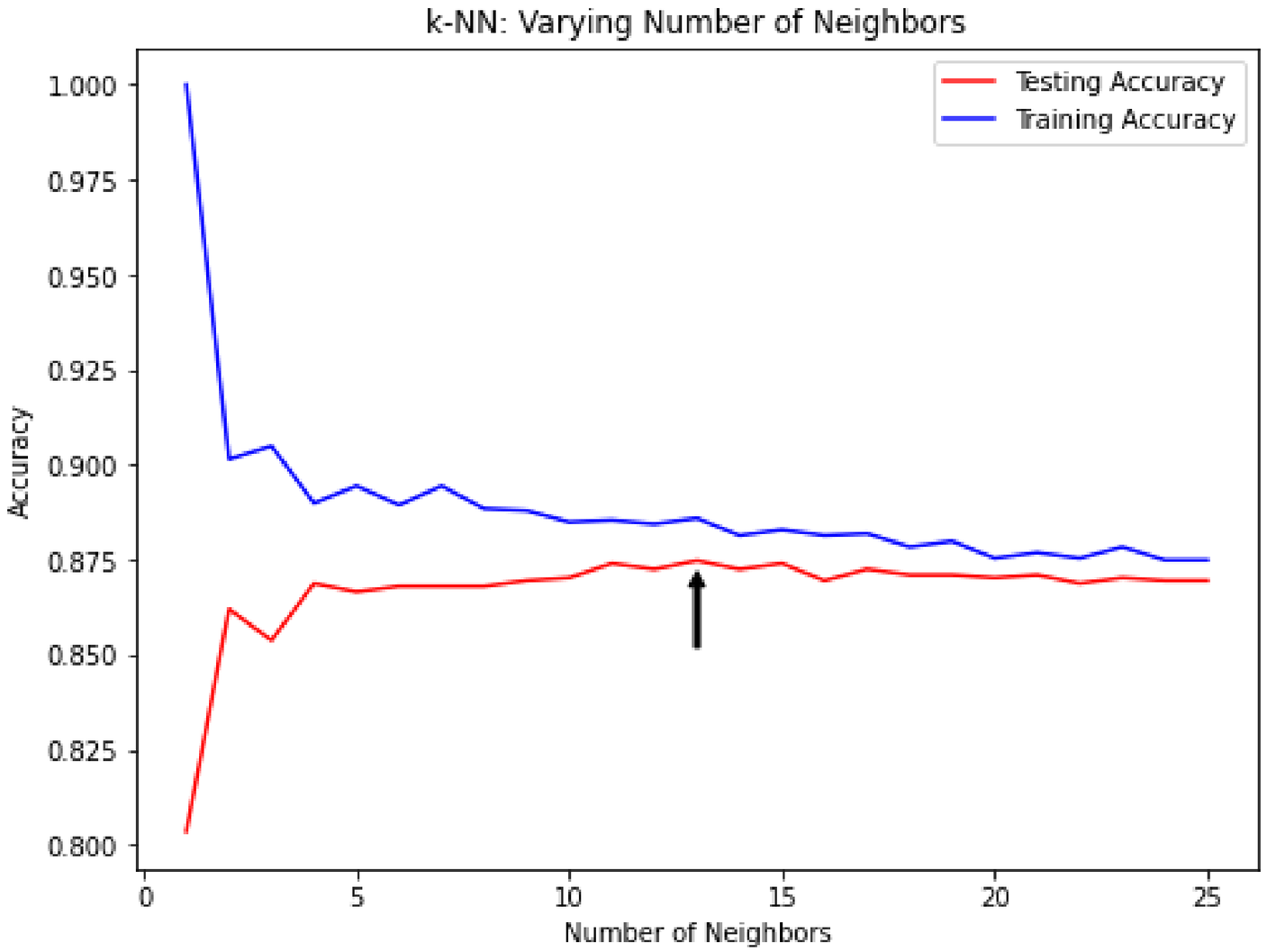
Plotting our results

```
plt.figure(figsize=(8, 6))
plt.title("KNN: Varying Number of Neighbors")
plt.plot(neighbors, train_accuracies.values(), label="Training Accuracy")
plt.plot(neighbors, test_accuracies.values(), label="Testing Accuracy")
plt.legend()
plt.xlabel("Number of Neighbors")
plt.ylabel("Accuracy")
plt.show()
```

Model complexity curve



Model complexity curve



Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Introduction to regression

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Predicting blood glucose levels

```
import pandas as pd
diabetes_df = pd.read_csv("diabetes.csv")
print(diabetes_df.head())
```

```
   pregnancies  glucose  triceps  insulin   bmi  age  diabetes
0            6      148       35        0  33.6   50         1
1            1       85       29        0  26.6   31         0
2            8      183        0        0  23.3   32         1
3            1       89       23       94  28.1   21         0
4            0      137       35      168  43.1   33         1
```

Creating feature and target arrays

```
X = diabetes_df.drop("glucose", axis=1).values
y = diabetes_df["glucose"].values
print(type(X), type(y))
```

```
<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

Making predictions from a single feature

```
X_bmi = X[:, 3]
print(y.shape, X_bmi.shape)
```

```
(752,) (752,)
```

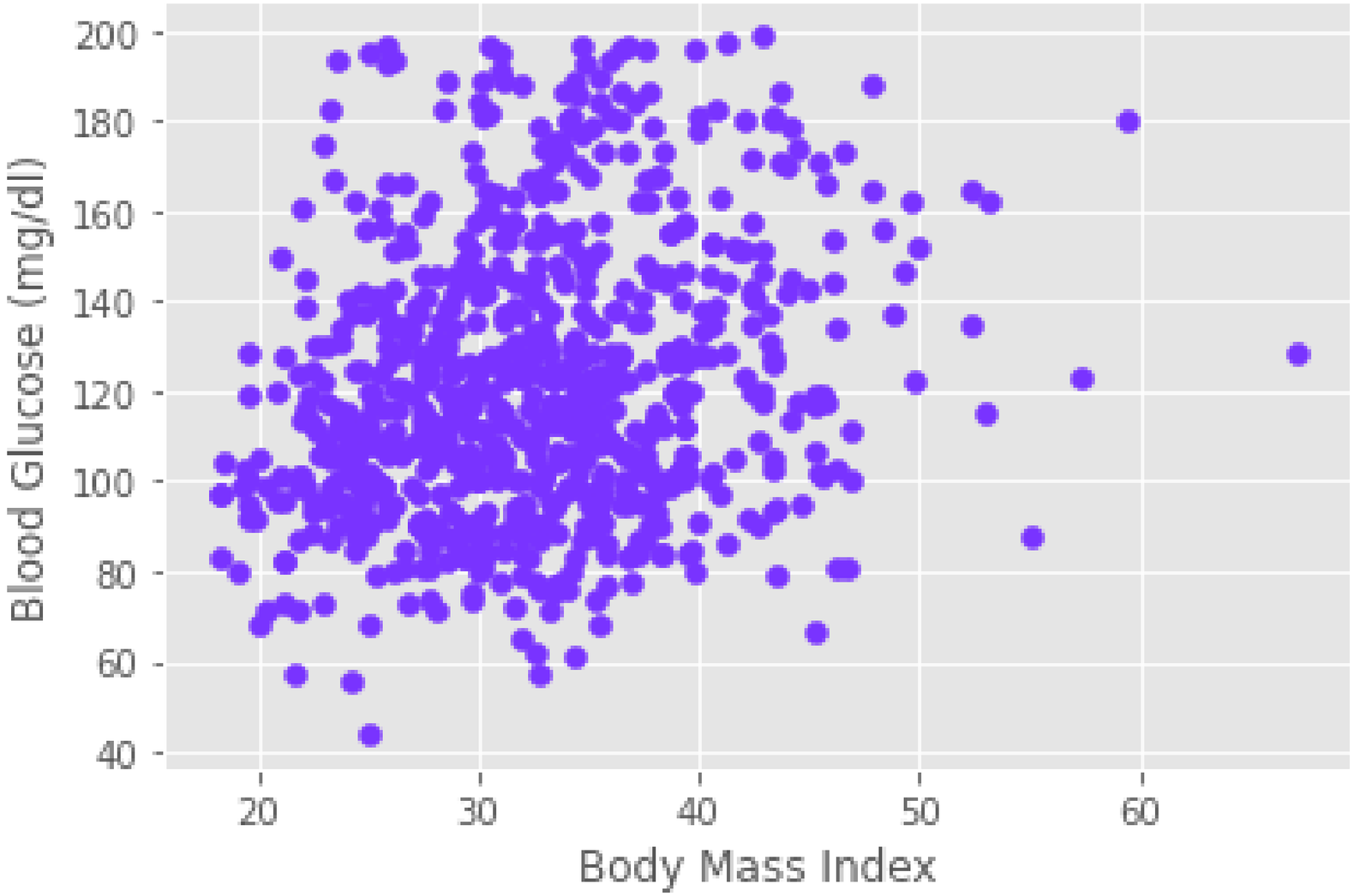
```
X_bmi = X_bmi.reshape(-1, 1)
print(X_bmi.shape)
```

```
(752, 1)
```

Plotting glucose vs. body mass index

```
import matplotlib.pyplot as plt
plt.scatter(X_bmi, y)
plt.ylabel("Blood Glucose (mg/dl)")
plt.xlabel("Body Mass Index")
plt.show()
```

Plotting glucose vs. body mass index



Fitting a regression model

```
from sklearn.linear_model import LinearRegression
reg = LinearRegression()
reg.fit(X_bmi, y)
predictions = reg.predict(X_bmi)
plt.scatter(X_bmi, y)
plt.plot(X_bmi, predictions)
plt.ylabel("Blood Glucose (mg/dl)")
plt.xlabel("Body Mass Index")
plt.show()
```

Fitting a regression model

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

The basics of linear regression

SUPERVISED LEARNING WITH SCIKIT-LEARN

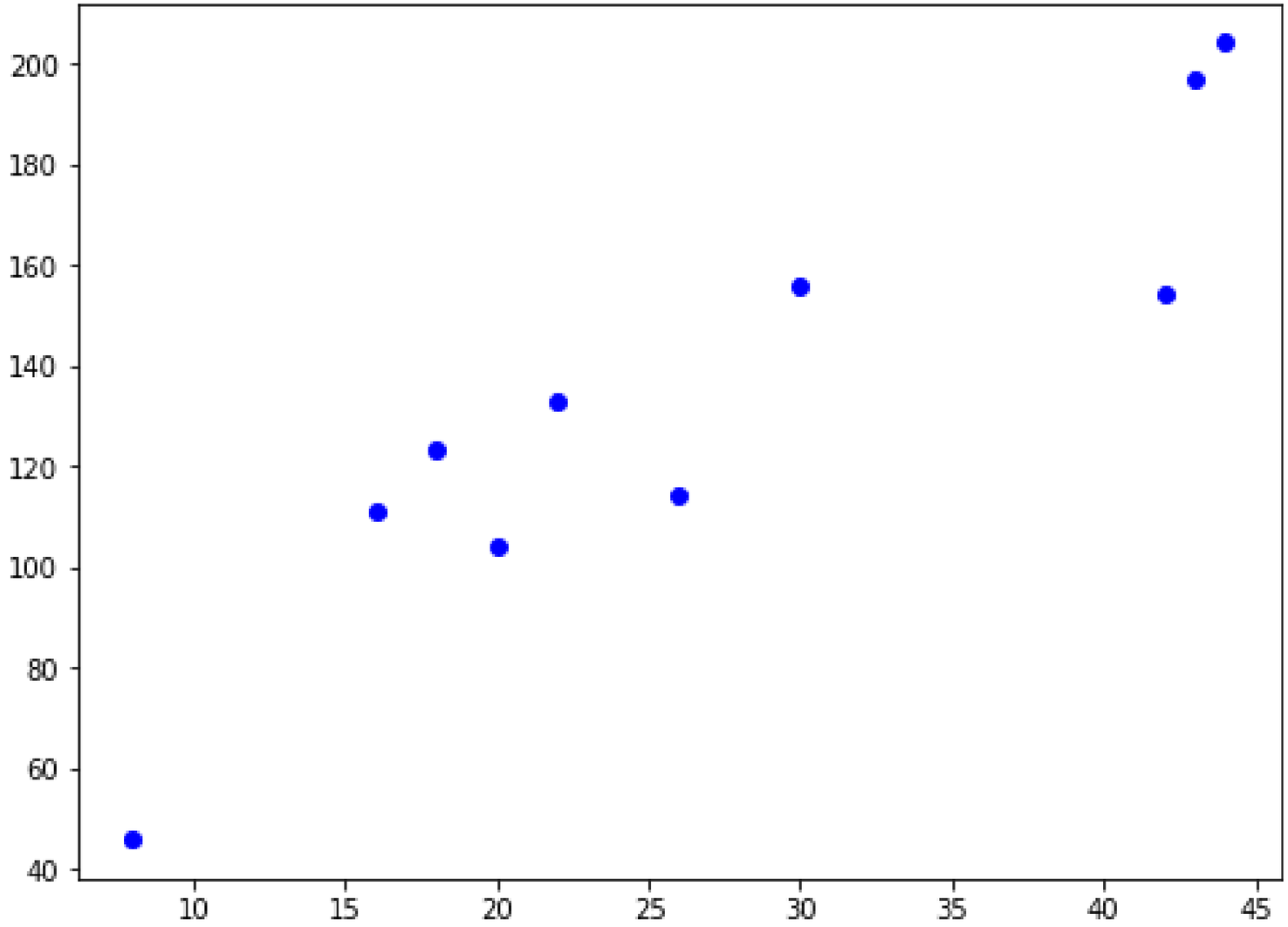
George Boorman
Core Curriculum Manager, DataCamp

datacamp

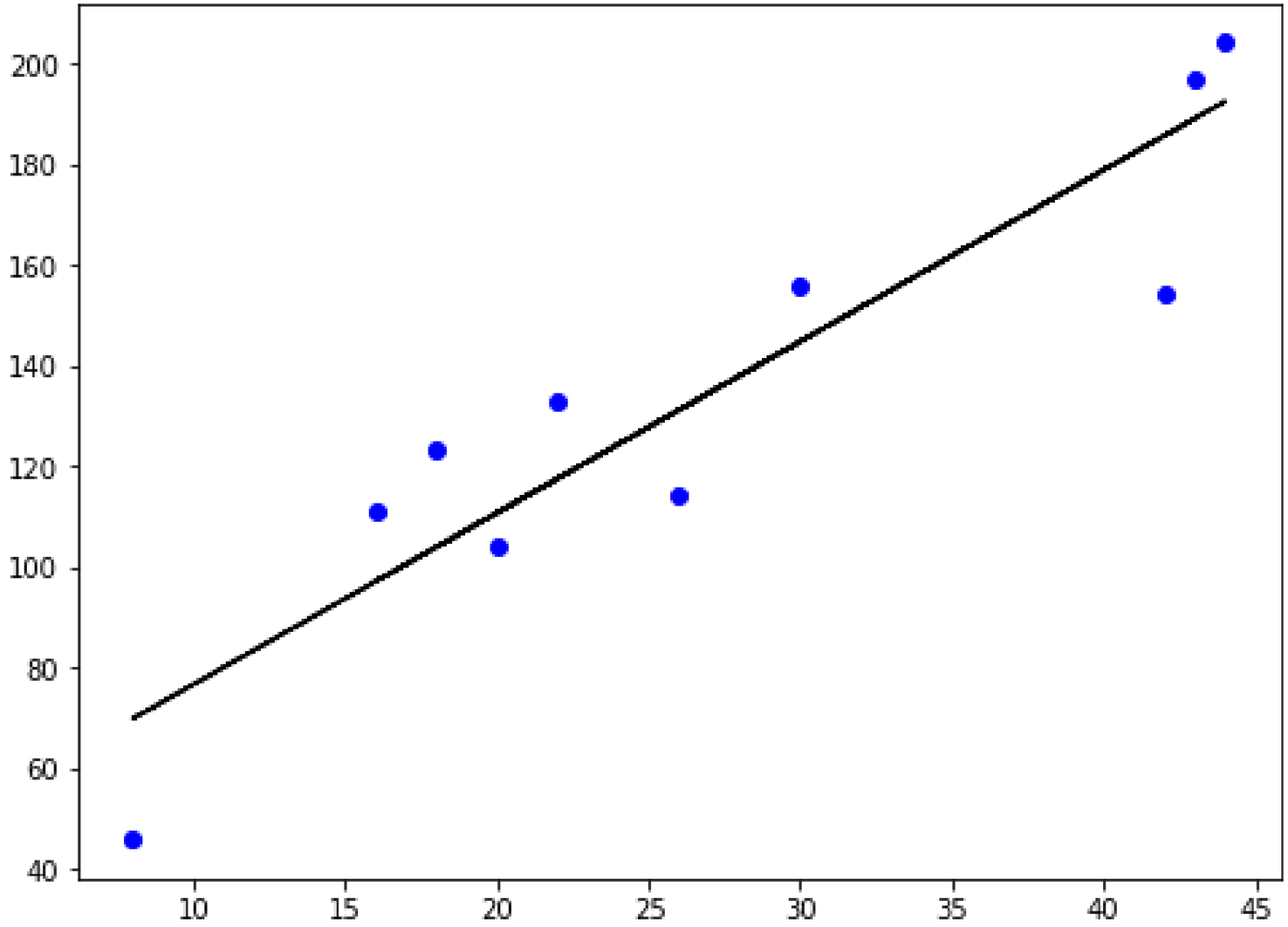
Regression mechanics

- $y = ax + b$
  - Simple linear regression uses one feature
    - y = target
    - x = single feature
    - a, b = parameters/coefficients of the model - slope, intercept
- How do we choose a and b?
  - Define an error function for any given line
  - Choose the line that minimizes the error function
- Error function = loss function = cost function

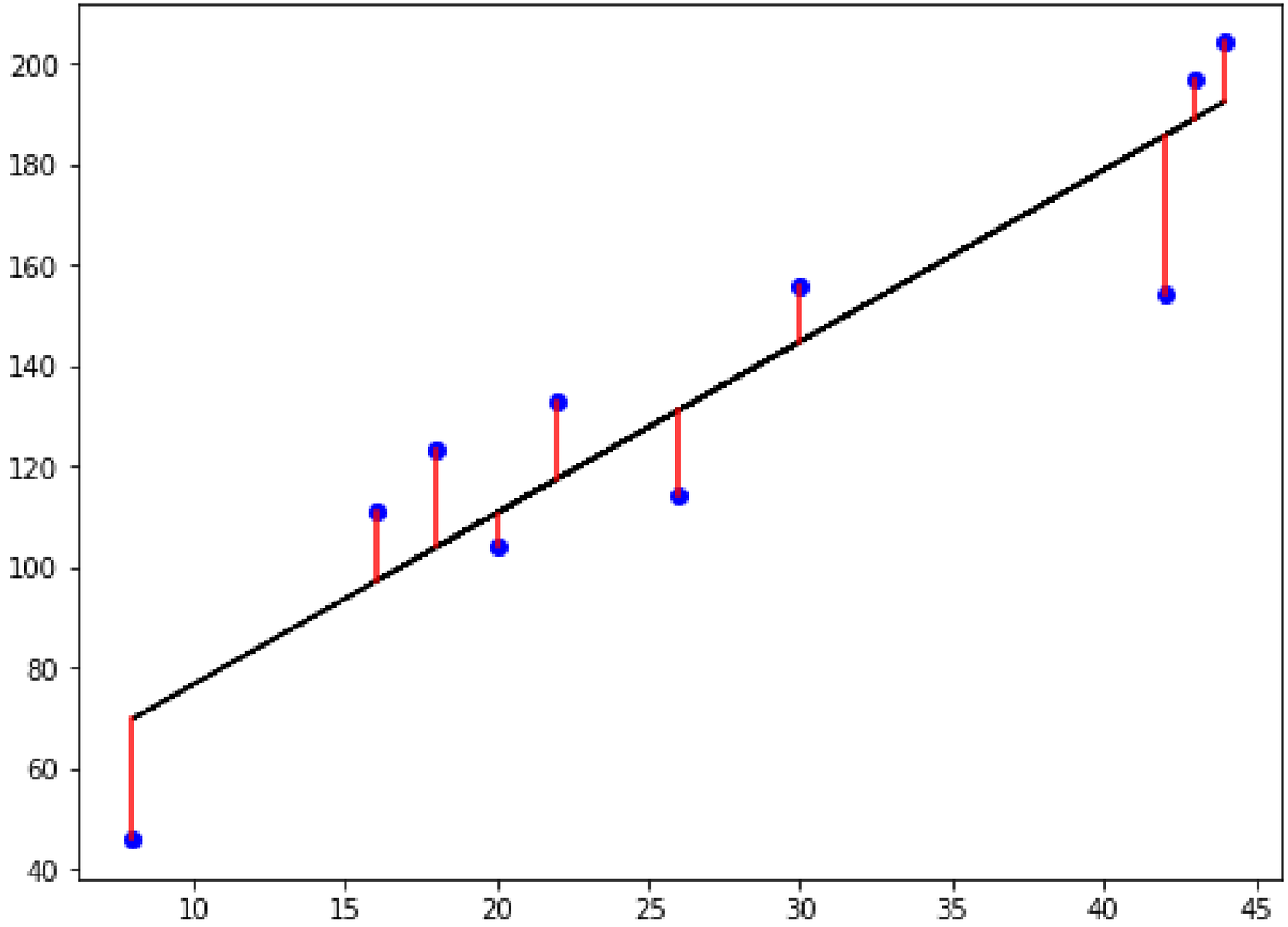
The loss function



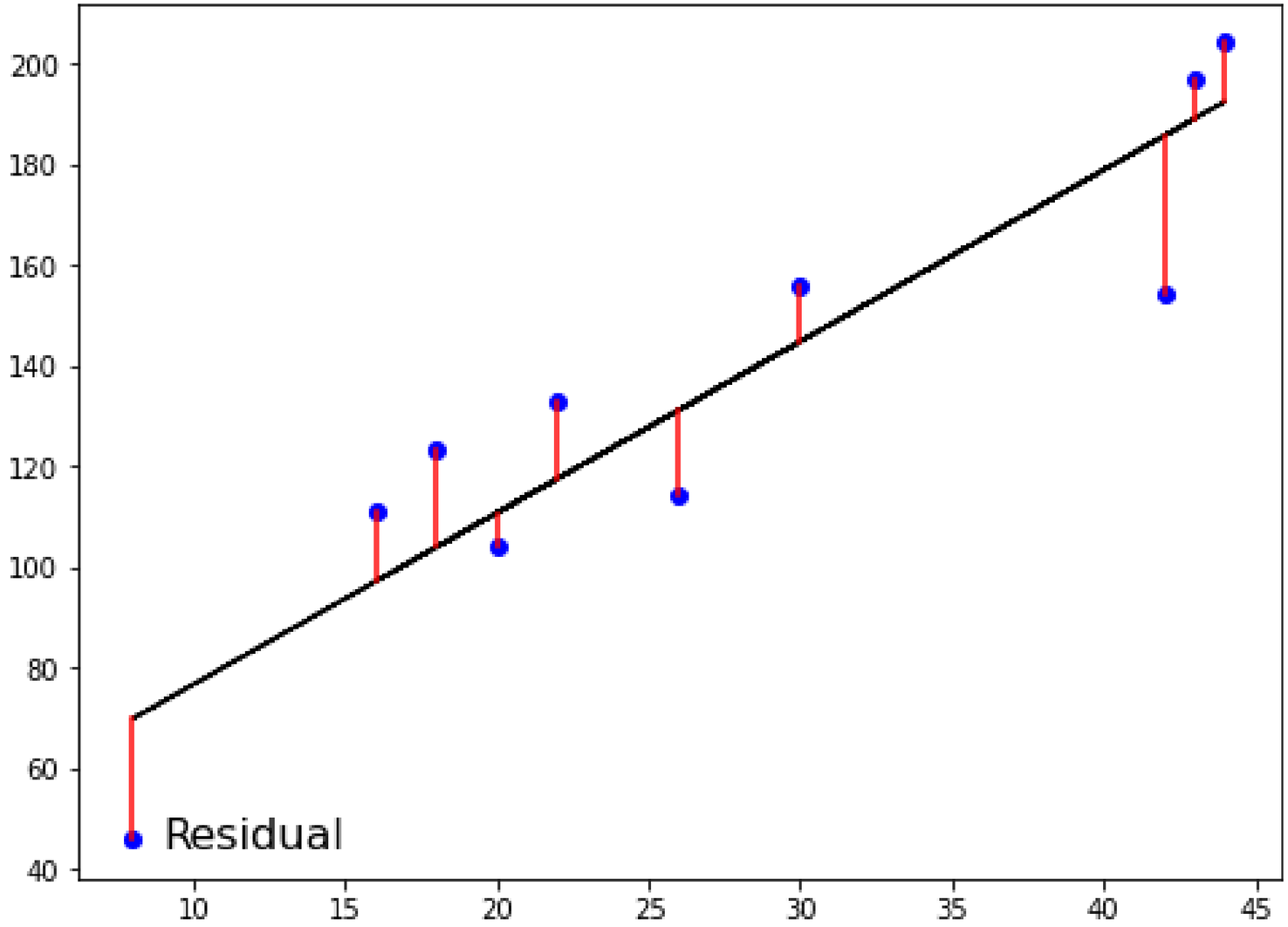
The loss function



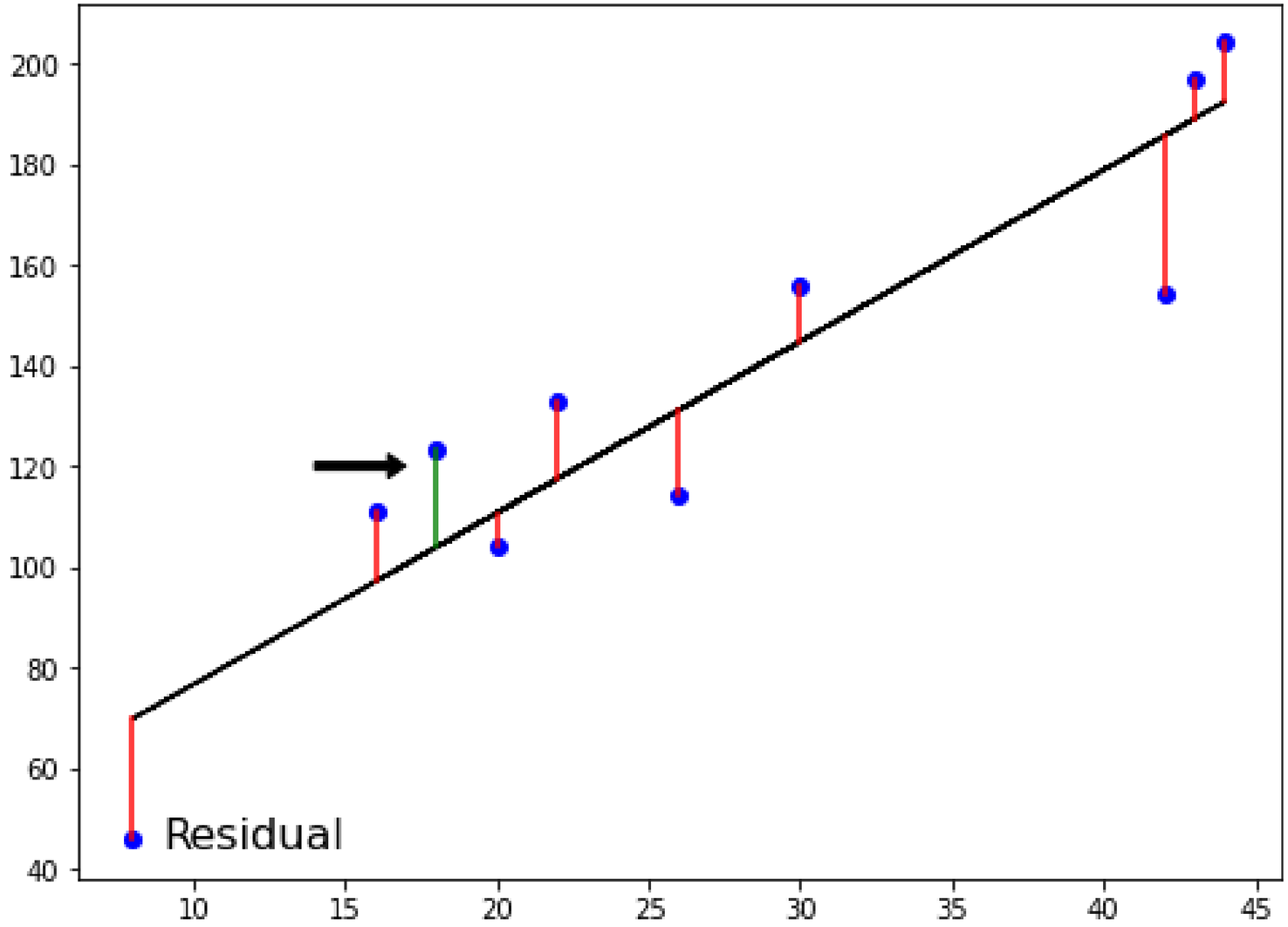
The loss function



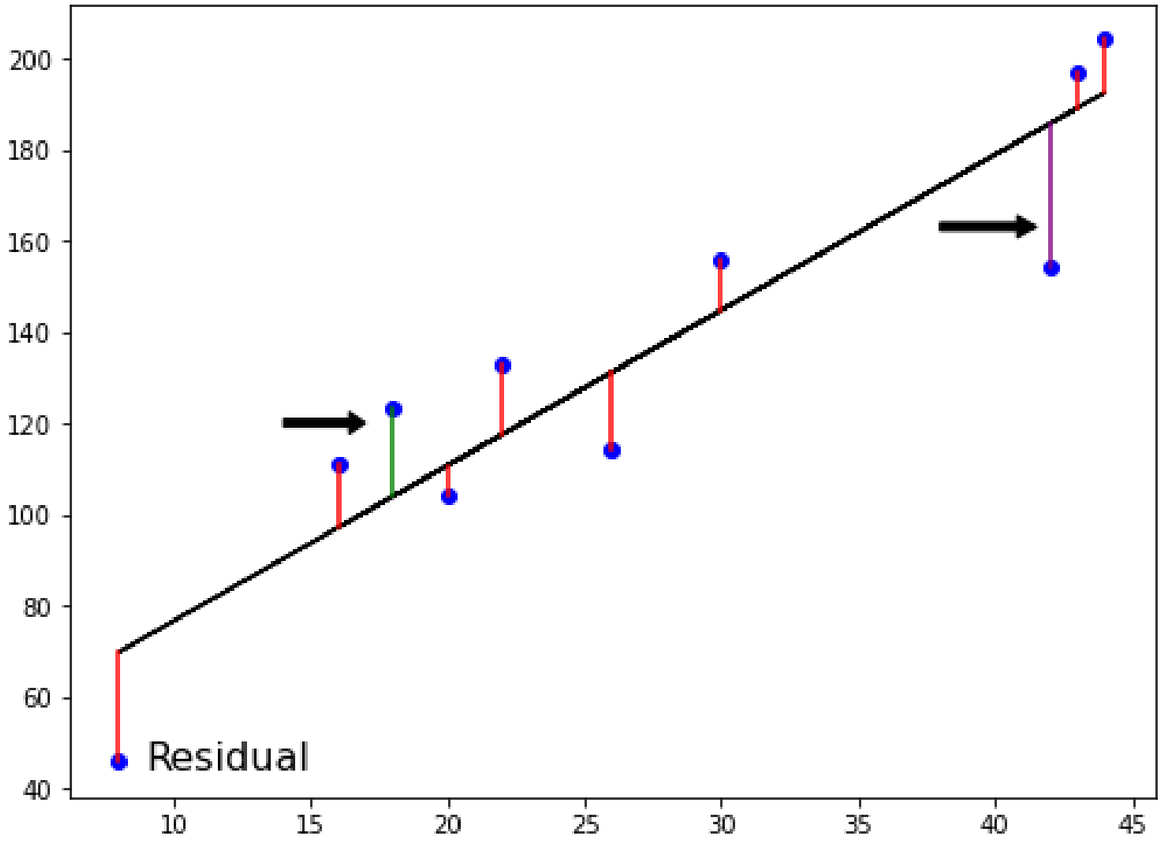
The loss function



The loss function



Ordinary Least Squares



$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Ordinary Least Squares (OLS): minimize RSS

Linear regression in higher dimensions

$$y = a_1 x_1 + a_2 x_2 + b$$

- To fit a linear regression model here:
  - Need to specify 3 variables: $a_1,\ a_2,\ b$
- In higher dimensions:
  - Known as multiple regression
  - Must specify coefficients for each feature and the variable b

$$y = a_1 x_1 + a_2 x_2 + a_3 x_3 + \ldots + a_n x_n + b$$

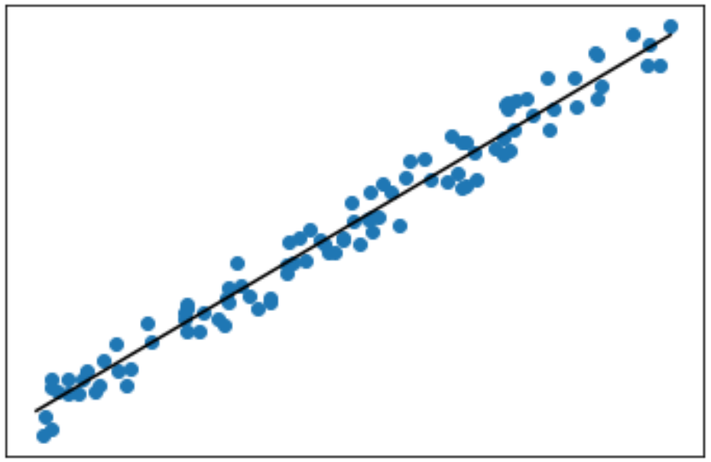
- scikit-learn works exactly the same way:
  - Pass two arrays: features and target

Linear regression using all features

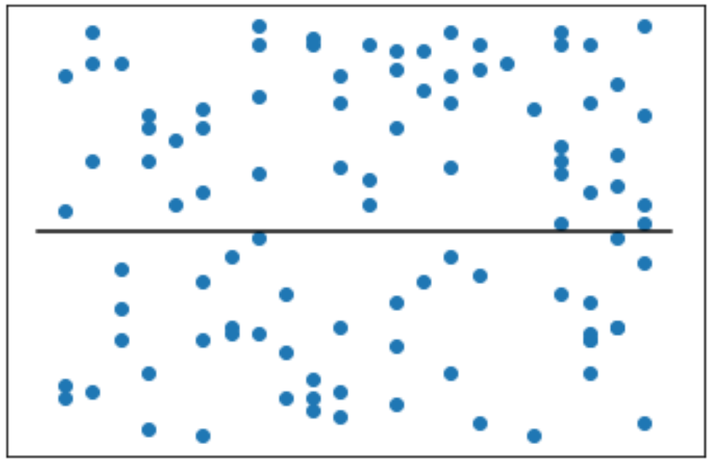
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)
reg_all = LinearRegression()
reg_all.fit(X_train, y_train)
y_pred = reg_all.predict(X_test)
```

R-squared

- R^2: quantifies the variance in target values explained by the features
  - Values range from 0 to 1
- High R^2:



- Low R^2:



R-squared in scikit-learn

```
reg_all.score(X_test, y_test)
```

```
0.356302876407827
```

Mean squared error and root mean squared error

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- MSE is measured in target units, squared

$$RMSE = \sqrt{MSE}$$

- Measure $RMSE$ in the same units at the target variable

RMSE in scikit-learn

```
from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred, squared=False)
```

```
24.028109426907236
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Cross-validation

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

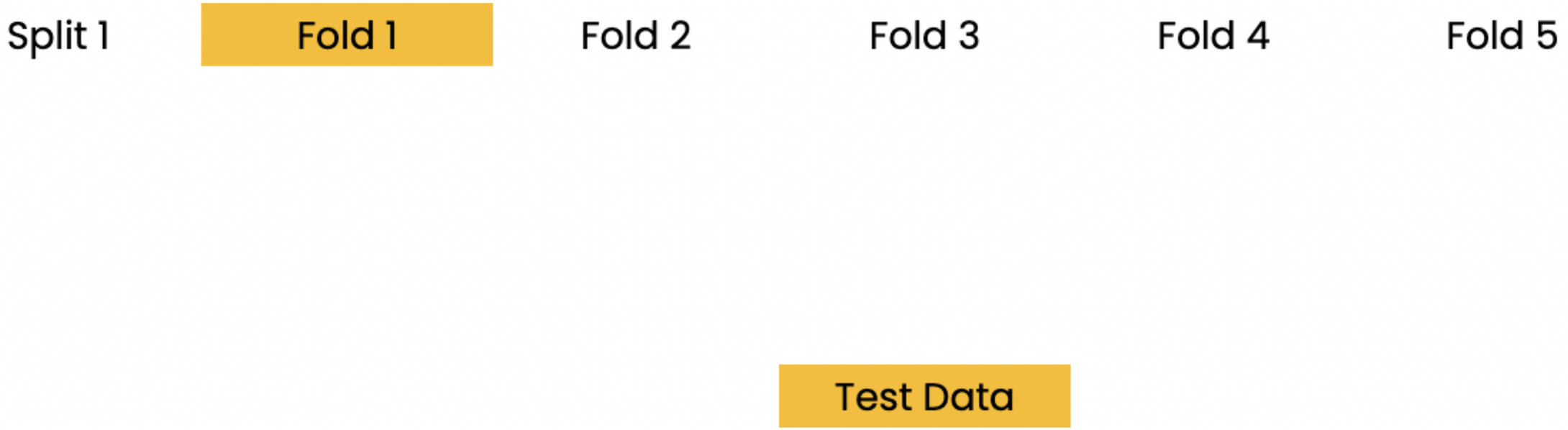
Cross-validation motivation

- Model performance is dependent on the way we split up the data
- Not representative of the model's ability to generalize to unseen data
- Solution: Cross-validation!

Cross-validation basics

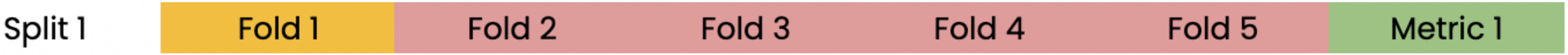
Split 1	Fold 1	Fold 2	Fold 3	Fold 4	Fold 5

Cross-validation basics

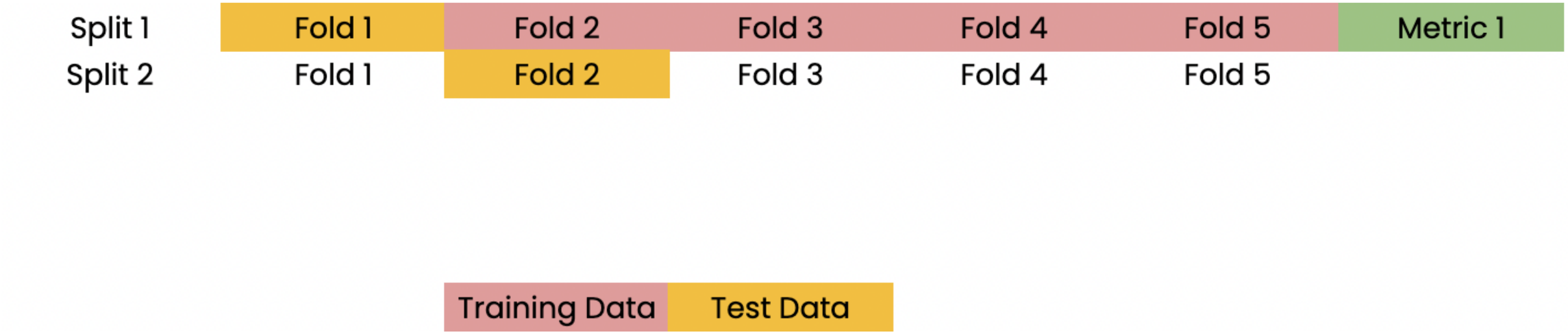


Cross-validation basics

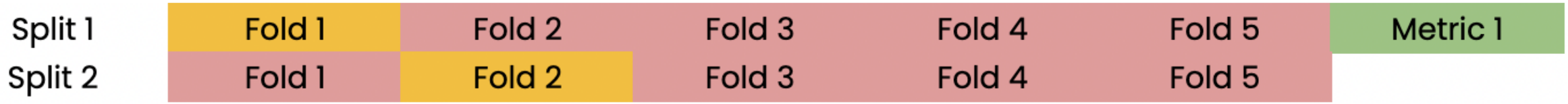
Cross-validation basics



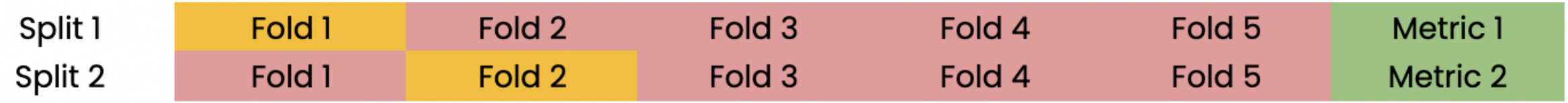
Cross-validation basics



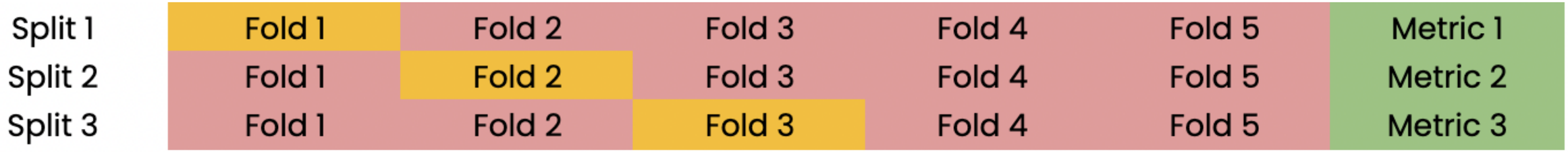
Cross-validation basics



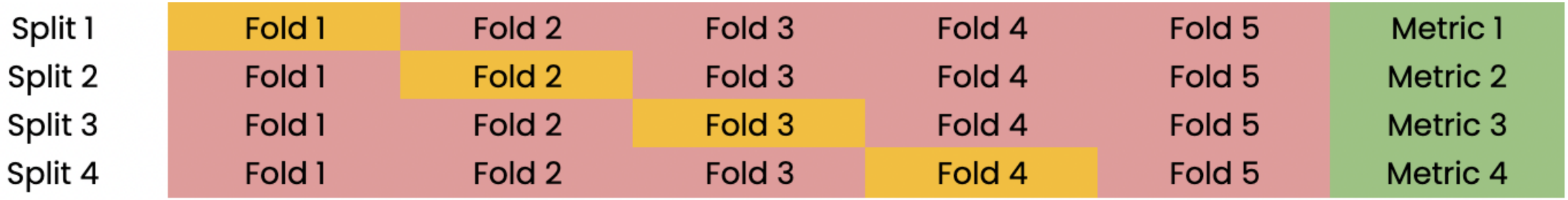
Cross-validation basics



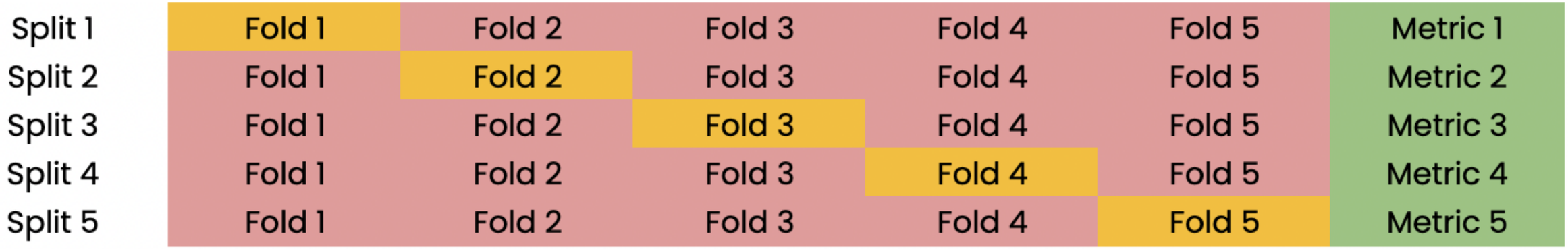
Cross-validation basics



Cross-validation basics



Cross-validation basics



Cross-validation and model performance

- 5 folds = 5-fold CV
- 10 folds = 10-fold CV
- k folds = k-fold CV
- More folds = More computationally expensive

Cross-validation in scikit-learn

```
from sklearn.model_selection import cross_val_score, KFold
kf = KFold(n_splits=6, shuffle=True, random_state=42)
reg = LinearRegression()
cv_results = cross_val_score(reg, X, y, cv=kf)
```

Evaluating cross-validation peformance

```
print(cv_results)
```

```
[0.70262578, 0.7659624, 0.75188205, 0.76914482, 0.72551151, 0.73608277]
```

```
print(np.mean(cv_results), np.std(cv_results))
```

```
0.7418682216666667 0.023330243960652888
```

```
print(np.quantile(cv_results, [0.025, 0.975]))
```

```
array([0.7054865, 0.76874702])
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Regularized regression

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Why regularize?

- Recall: Linear regression minimizes a loss function
- It chooses a coefficient, a, for each feature variable, plus b
- Large coefficients can lead to overfitting
- Regularization: Penalize large coefficients

Ridge regression

- Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} {a_i}^2$$

- Ridge penalizes large positive or negative coefficients
- α: parameter we need to choose
- Picking α is similar to picking `k` in KNN
- Hyperparameter: variable used to optimize model parameters
- α controls model complexity
  - α = 0 = OLS (Can lead to overfitting)
  - Very high α: Can lead to underfitting

Ridge regression in scikit-learn

```
from sklearn.linear_model import Ridge
scores = []
for alpha in [0.1, 1.0, 10.0, 100.0, 1000.0]:
    ridge = Ridge(alpha=alpha)
    ridge.fit(X_train, y_train)
    y_pred = ridge.predict(X_test)
    scores.append(ridge.score(X_test, y_test))
print(scores)
```

```
[0.2828466623222221, 0.2832063357480477, 0.2853000732200006,
 0.2642398481266813, 0.19292424694100963]
```

Lasso regression

- Loss function = OLS loss function +

$$\alpha * \sum_{i=1}^{n} |a_i|$$

Lasso regression in scikit-learn

```
from sklearn.linear_model import Lasso
scores = []
for alpha in [0.01, 1.0, 10.0, 20.0, 50.0]:
  lasso = Lasso(alpha=alpha)
  lasso.fit(X_train, y_train)
  lasso_pred = lasso.predict(X_test)
  scores.append(lasso.score(X_test, y_test))
print(scores)
```

```
[0.999916490711 23, 0.999617002842 23, 0.938822276710 69, 0.748553186762 32, -0.057410346400 16]
```

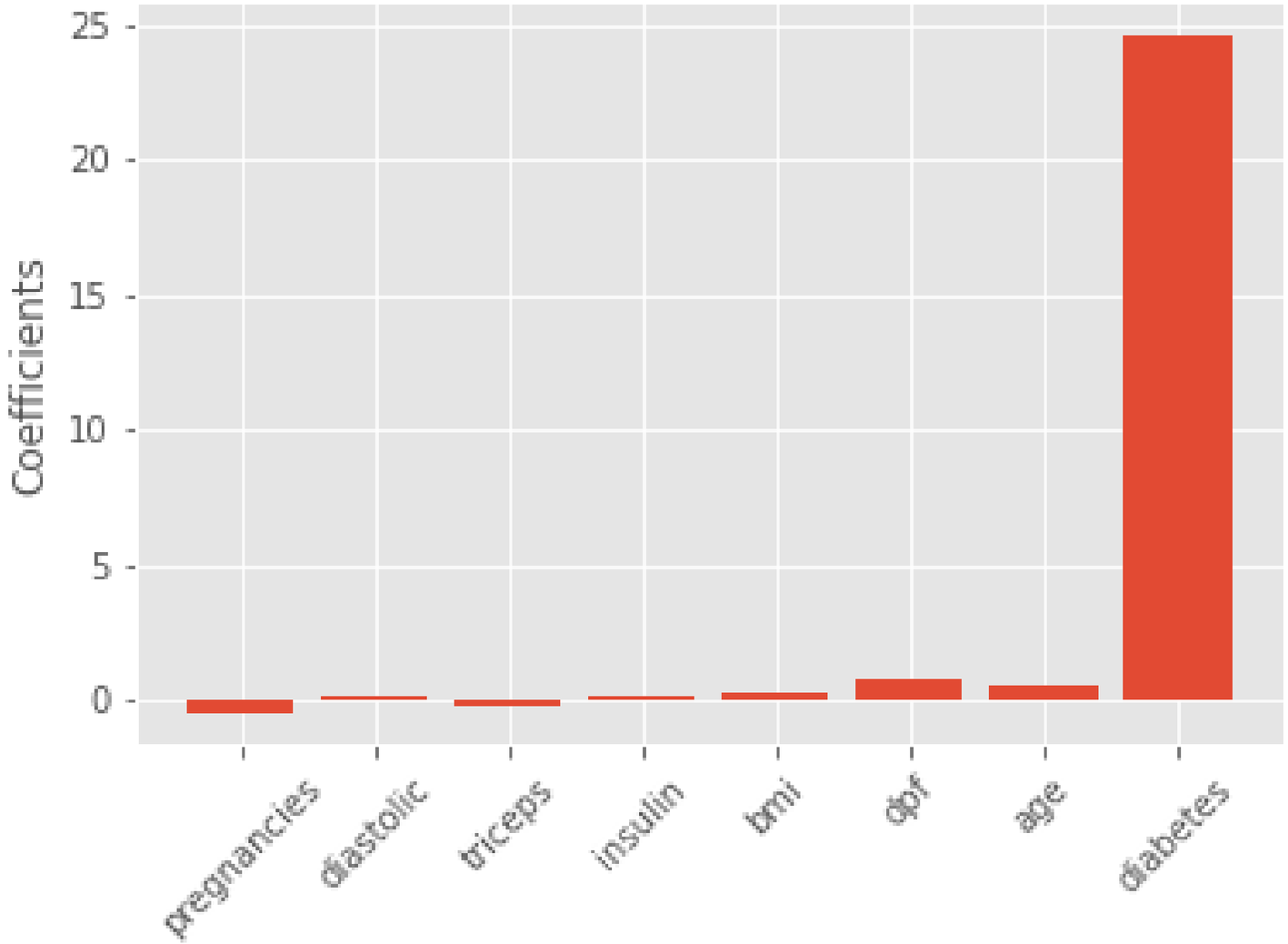
Lasso regression for feature selection

- Lasso can select important features of a dataset
- Shrinks the coefficients of less important features to zero
- Features not shrunk to zero are selected by lasso

Lasso for feature selection in scikit-learn

```
from sklearn.linear_model import Lasso
X = diabetes_df.drop("glucose", axis=1).values
y = diabetes_df["glucose"].values
names = diabetes_df.drop("glucose", axis=1).columns
lasso = Lasso(alpha=0.1)
lasso_coef = lasso.fit(X, y).coef_
plt.bar(names, lasso_coef)
plt.xticks(rotation=45)
plt.show()
```

Lasso for feature selection in scikit-learn



Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

How good is your model?

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Classification metrics

- Measuring model performance with accuracy:
  - Fraction of correctly classified samples
  - Not always a useful metric

Class imbalance

- Classification for predicting fraudulent bank transactions
  - 99% of transactions are legitimate; 1% are fraudulent
- Could build a classifier that predicts NONE of the transactions are fraudulent
  - 99% accurate!
  - But terrible at actually predicting fraudulent transactions
  - Fails at its original purpose
- Class imbalance: Uneven frequency of classes
- Need a different way to assess performance

Confusion matrix for assessing classification performance

- Confusion matrix

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

Assessing classification performance

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Accuracy:

$$\frac{tp + tn}{tp + tn + fp + fn}$$

Precision

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Precision

$$\frac{true\ positives}{true\ positives + false\ positives}$$

- High precision = lower false positive rate
- High precision: Not many legitimate transactions are predicted to be fraudulent

Recall

	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Recall

$$\frac{true\ positives}{true\ positives + false\ negatives}$$

- High recall = lower false negative rate
- High recall: Predicted most fraudulent transactions correctly

F1 score

- F1 Score: $2 * \frac{precision * recall}{precision + recall}$

Confusion matrix in scikit-learn

```
from sklearn.metrics import classification_report, confusion_matrix
knn = KNeighborsClassifier(n_neighbors=7)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
                                                    random_state=42)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
```

Confusion matrix in scikit-learn

```
print(confusion_matrix(y_test, y_pred))
```

```
[[1106   11]
 [ 183   34]]
```

Classification report in scikit-learn

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.86      0.99      0.92      1117
           1       0.76      0.16      0.26       217

    accuracy                           0.85      1334
   macro avg       0.81      0.57      0.59      1334
weighted avg       0.84      0.85      0.81      1334
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Logistic regression and the ROC curve

SUPERVISED LEARNING WITH SCIKIT-LEARN

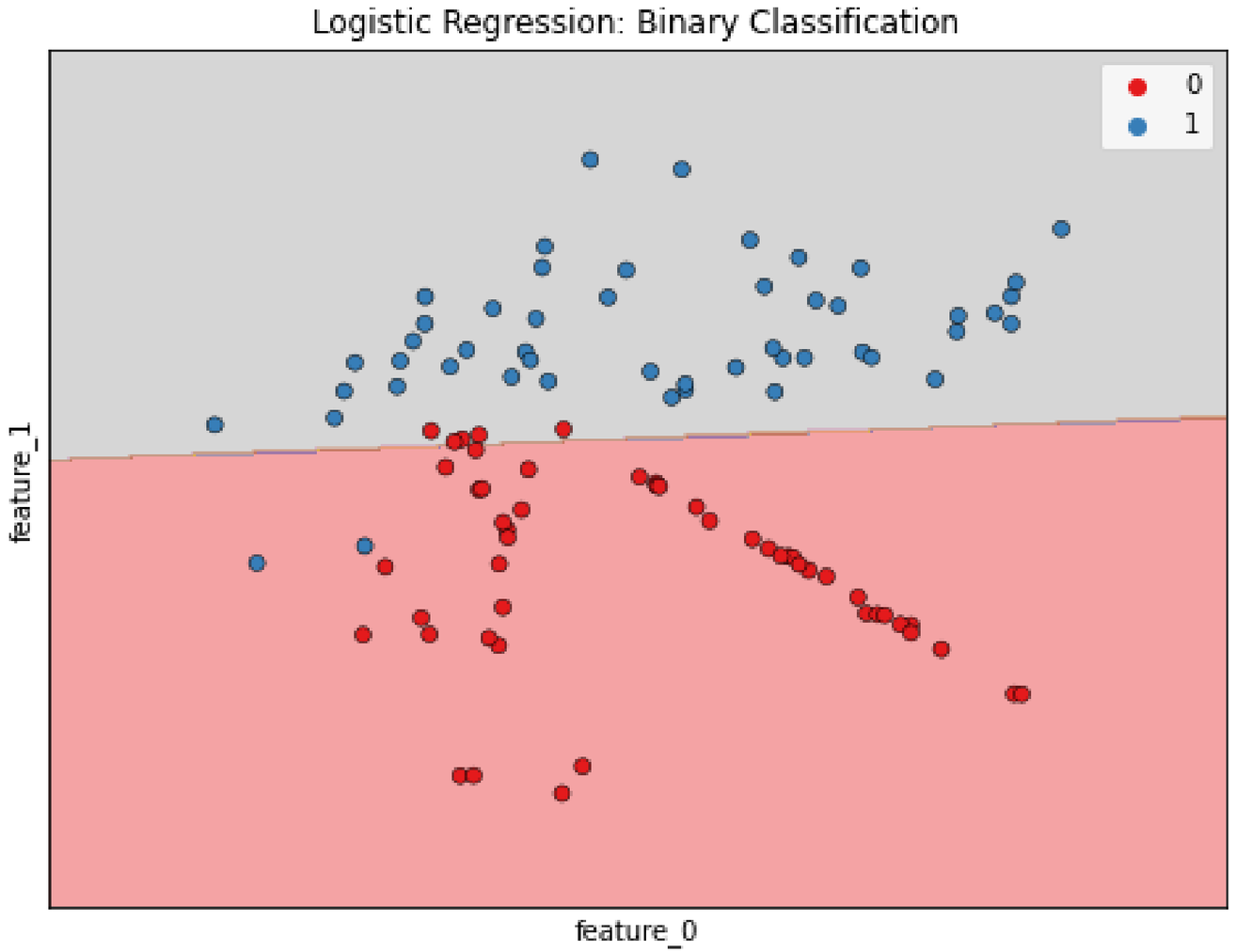
George Boorman
Core Curriculum Manager, DataCamp

datacamp

Logistic regression for binary classification

- Logistic regression is used for classification problems
- Logistic regression outputs probabilities
- If the probability, $p > 0.5$:
  - The data is labeled `1`
- If the probability, $p < 0.5$:
  - The data is labeled `0`

Linear decision boundary



Logistic regression in scikit-learn

```
from sklearn.linear_model import LogisticRegression
logreg = LogisticRegression()
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)
logreg.fit(X_train, y_train)
y_pred = logreg.predict(X_test)
```

Predicting probabilities

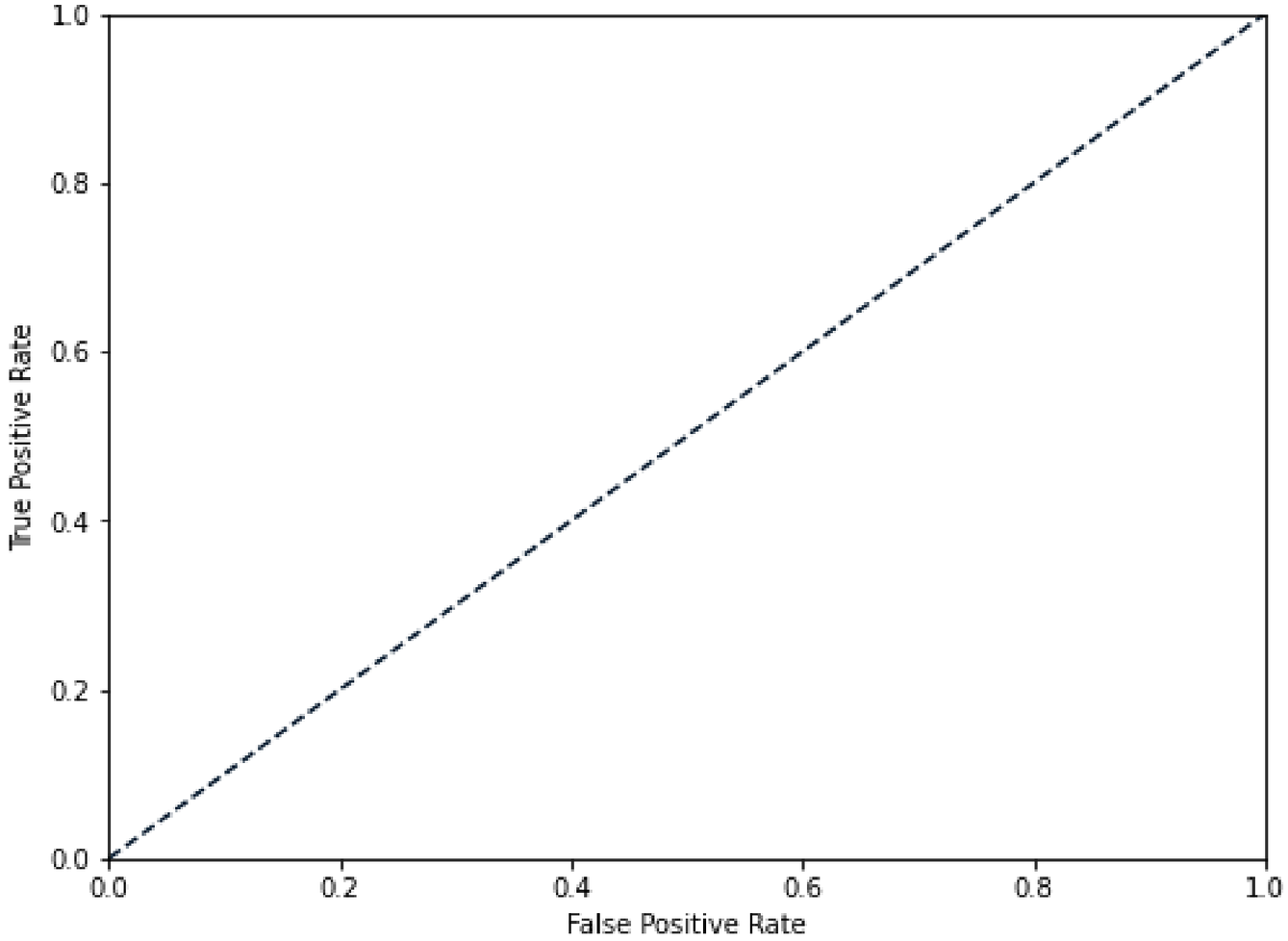
```
y_pred_probs = logreg.predict_proba(X_test)[:, 1]
print(y_pred_probs[0])
```

```
[0.08961376]
```

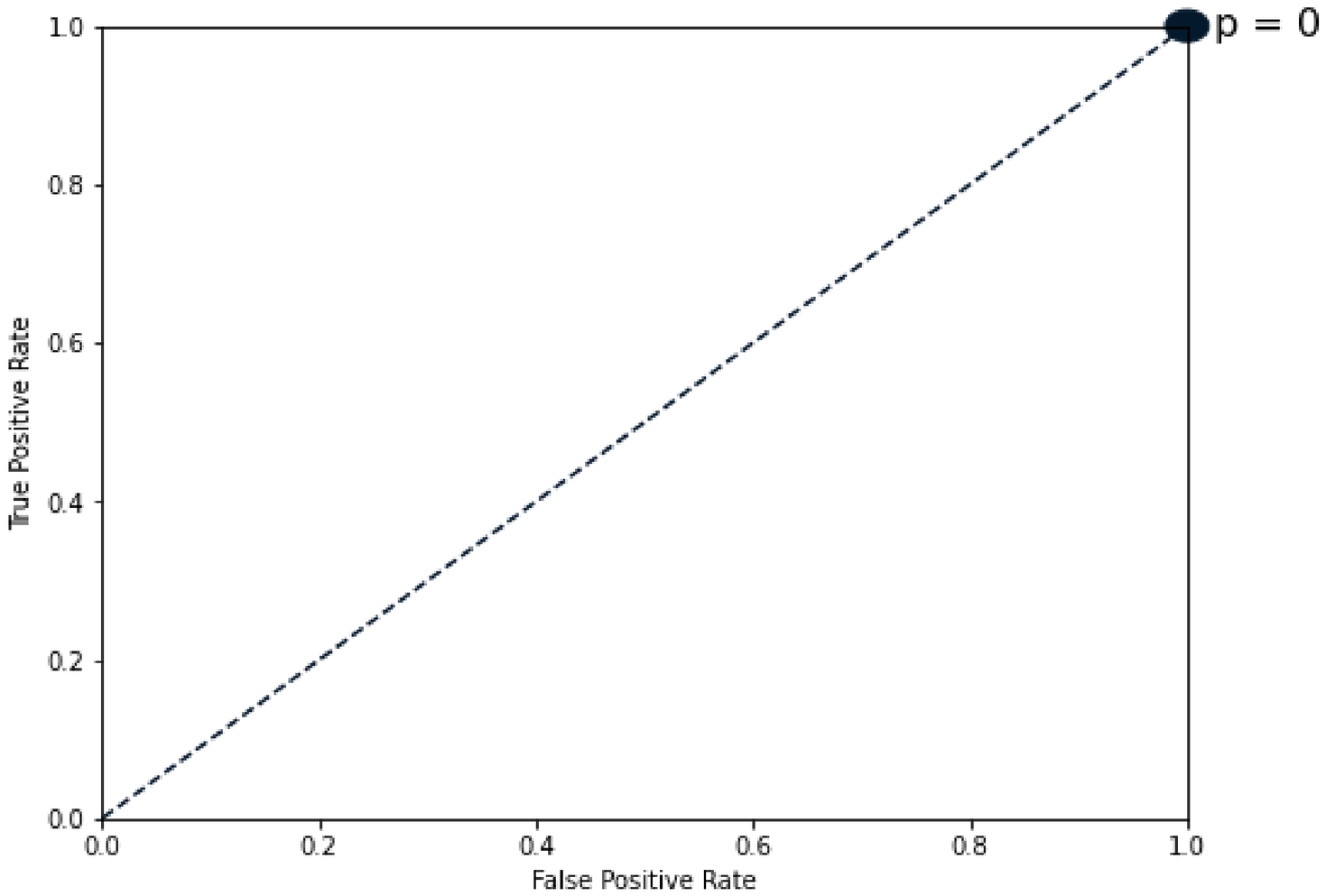
Probability thresholds

- By default, logistic regression threshold = 0.5
- Not specific to logistic regression
  - KNN classifiers also have thresholds
- What happens if we vary the threshold?

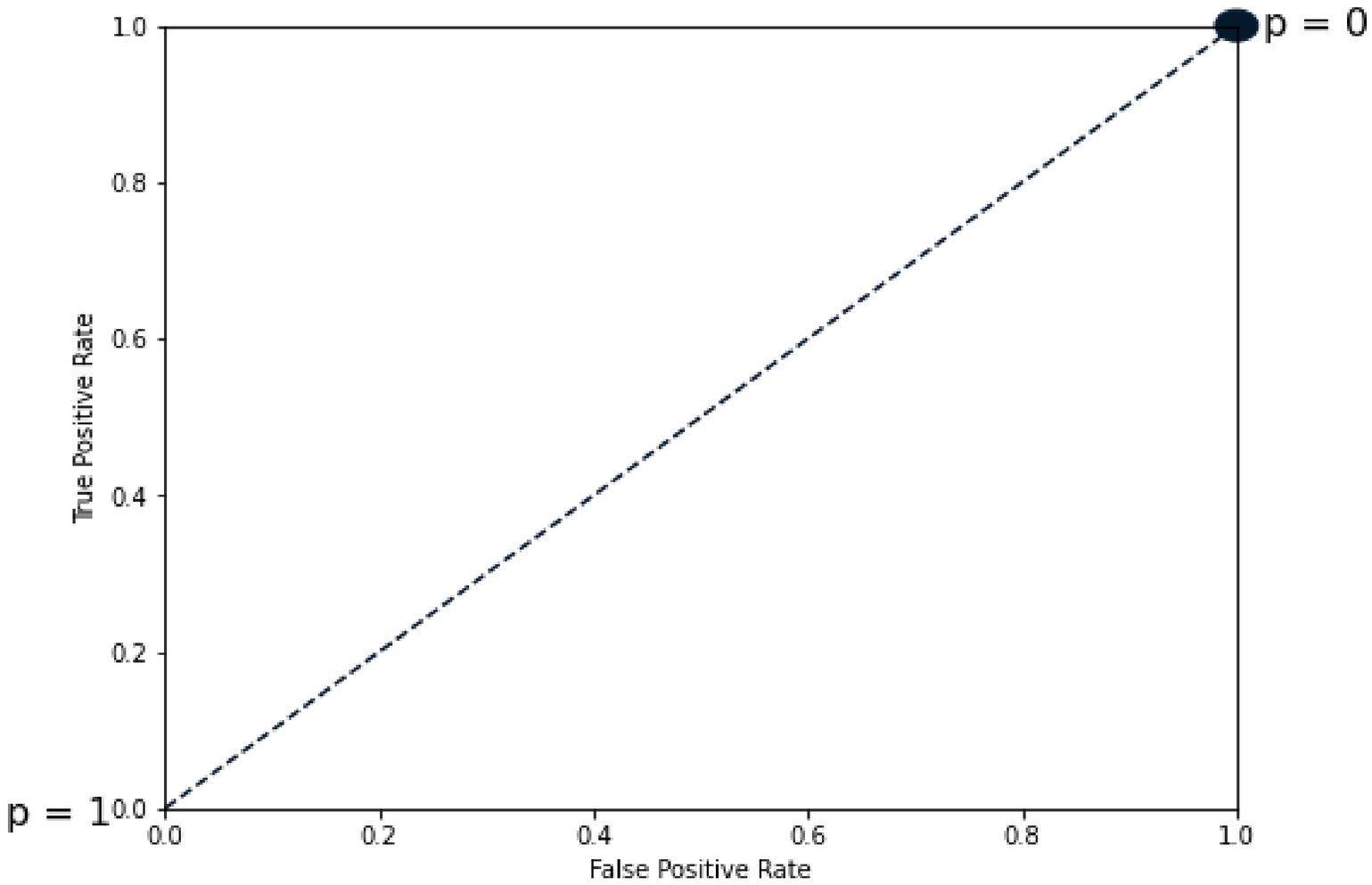
The ROC curve



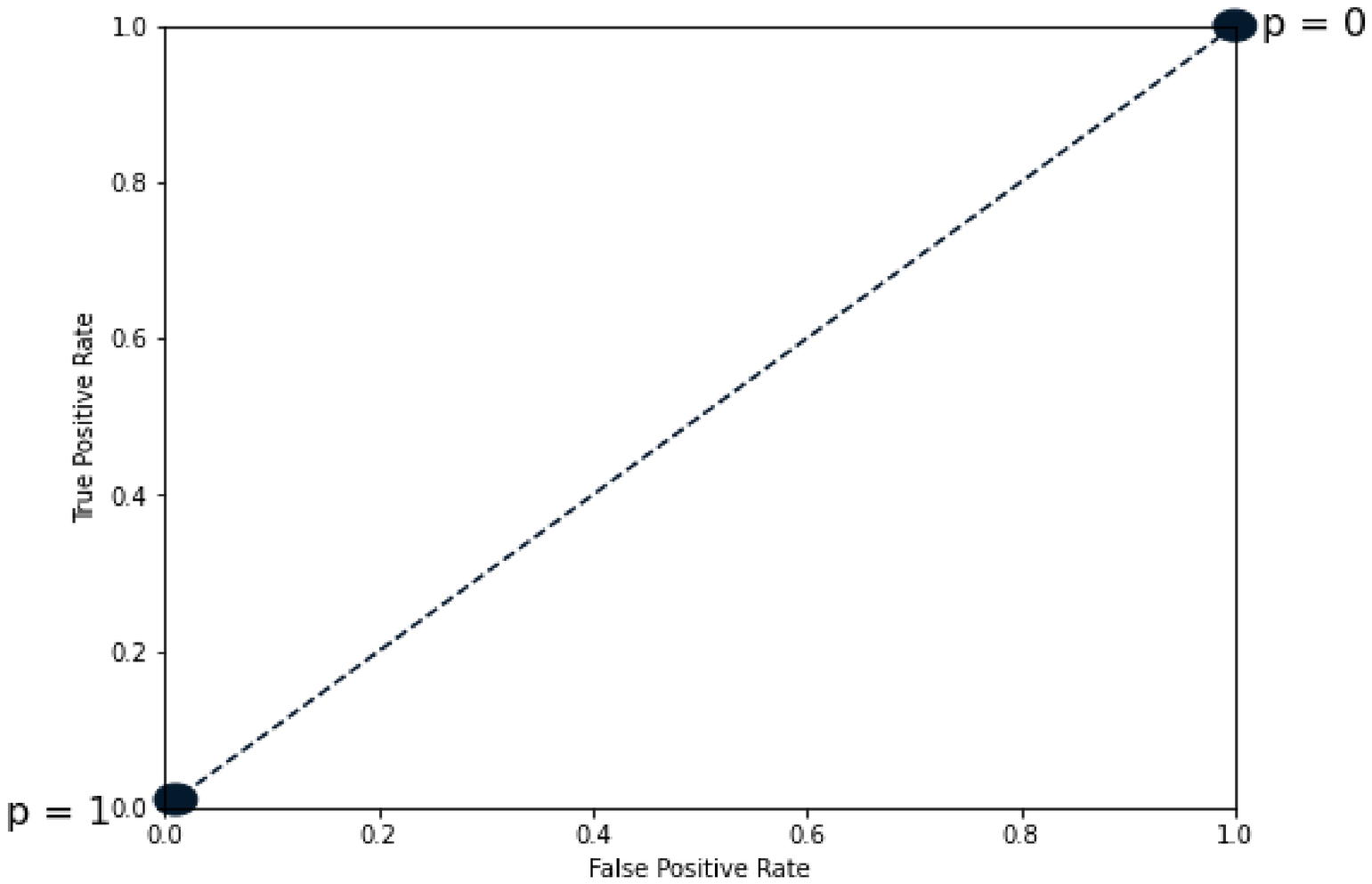
The ROC curve



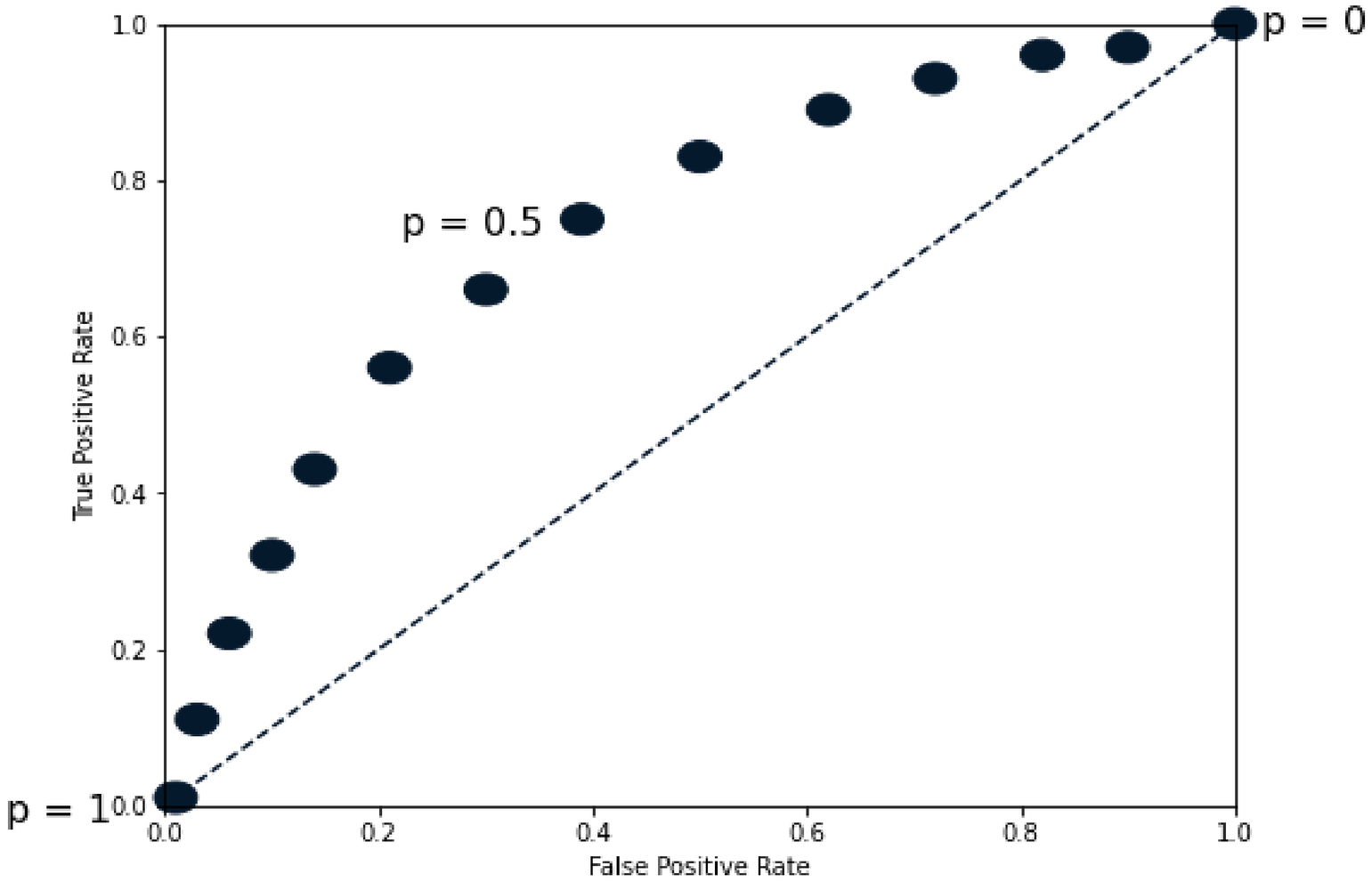
The ROC curve



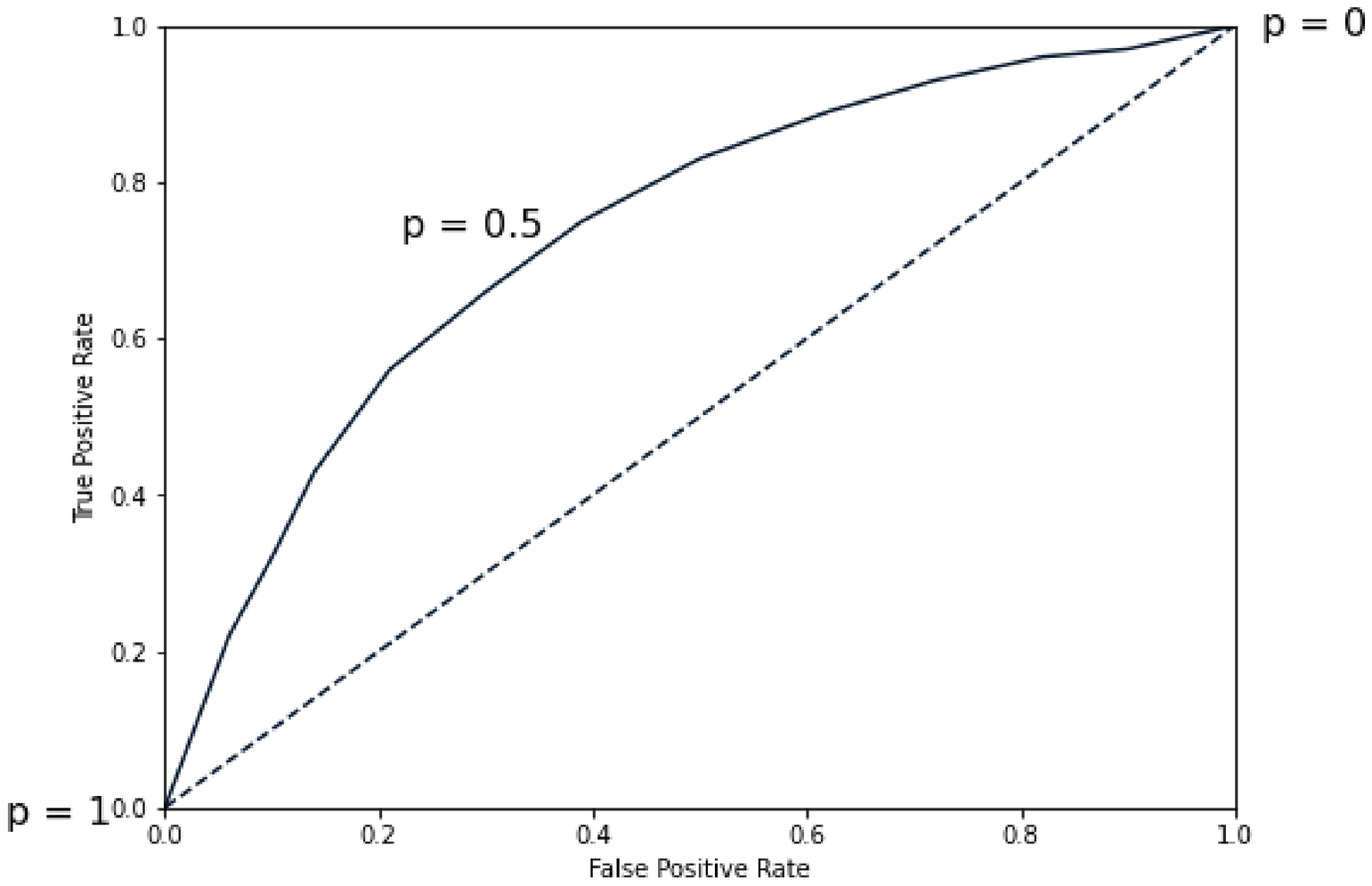
The ROC curve



The ROC curve



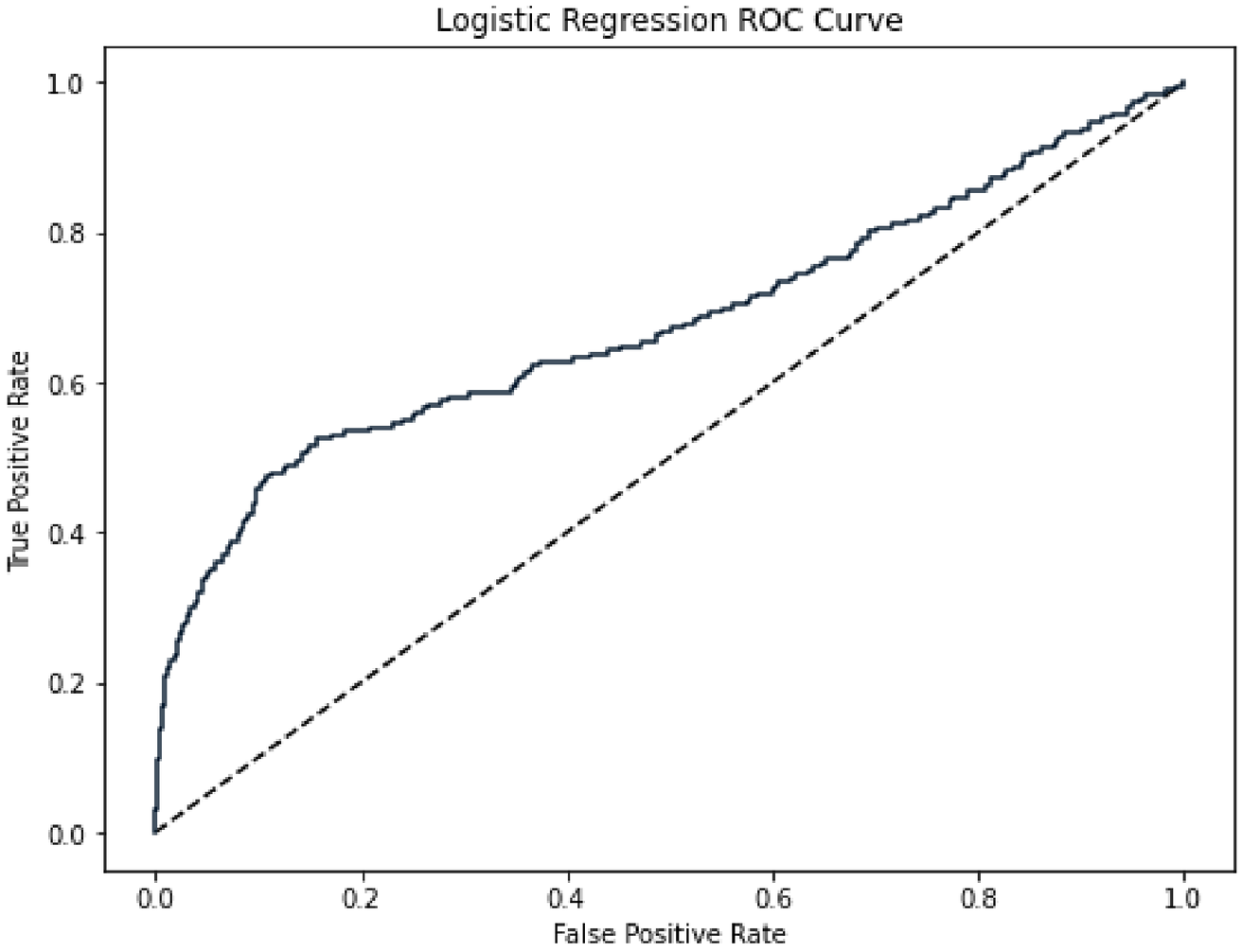
The ROC curve



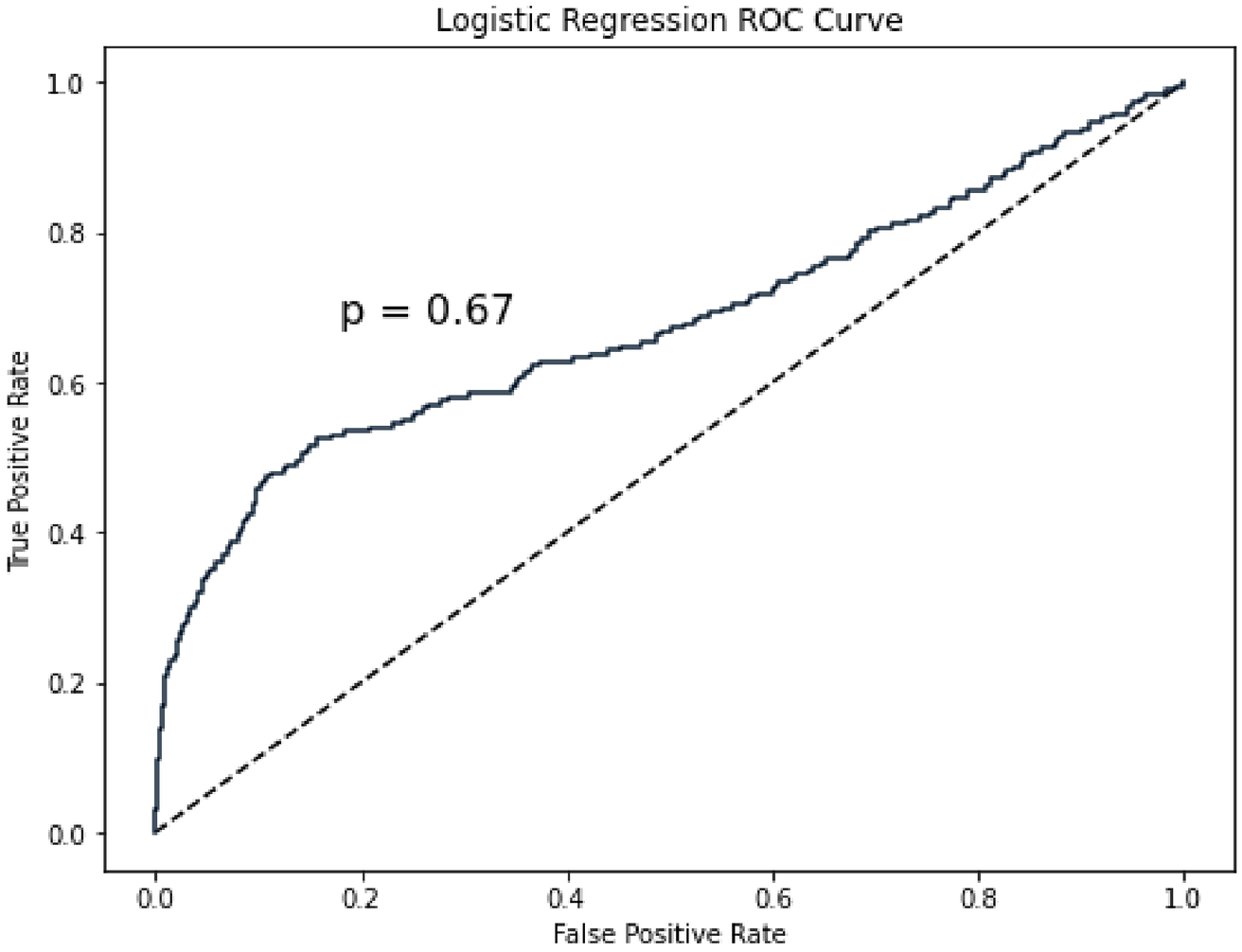
Plotting the ROC curve

```
from sklearn.metrics import roc_curve
fpr, tpr, thresholds = roc_curve(y_test, y_pred_probs)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr, tpr)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Logistic Regression ROC Curve')
plt.show()
```

Plotting the ROC curve



ROC AUC


Logistic Regression ROC Curve
p = 0.67
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0

ROC AUC in scikit-learn

```
from sklearn.metrics import roc_auc_score
print(roc_auc_score(y_test, y_pred_probs))
```

```
0.6700964152663693
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Hyperparameter tuning

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager

datacamp

Hyperparameter tuning

- Ridge/lasso regression: Choosing `alpha`
- KNN: Choosing `n_neighbors`
- Hyperparameters: Parameters we specify before fitting the model
  - Like `alpha` and `n_neighbors`

Choosing the correct hyperparameters

1. Try lots of different hyperparameter values
2. Fit all of them separately
3. See how well they perform
4. Choose the best performing values

- This is called **hyperparameter tuning**
- It is essential to use cross-validation to avoid overfitting to the test set
- We can still split the data and perform cross-validation on the training set
- We withhold the test set for final evaluation

Grid search cross-validation

n_neighbors	11		
	8		
	5		
	2		
		euclidean	manhattan
		metric	

Grid search cross-validation

n_neighbors	11	0.8716	0.8692
	8	0.8704	0.8688
	5	0.8748	0.8714
	2	0.8634	0.8646
		euclidean	manhattan
		metric	

Grid search cross-validation

	11	0.8716	0.8692
	8	0.8704	0.8688
	5	0.8748	0.8714
n_neighbors	2	0.8634	0.8646
		euclidean	manhattan
		metric	

GridSearchCV in scikit-learn

```
from sklearn.model_selection import GridSearchCV
kf = KFold(n_splits=5, shuffle=True, random_state=42)
param_grid = {"alpha": np.arange(0.0001, 1, 10),
              "solver": ["sag", "lsqr"]}
ridge = Ridge()
ridge_cv = GridSearchCV(ridge, param_grid, cv=kf)
ridge_cv.fit(X_train, y_train)
print(ridge_cv.best_params_, ridge_cv.best_score_)
```

```
{'alpha': 0.0001, 'solver': 'sag'}
0.7529912278705785
```

Limitations and an alternative approach

- 3-fold cross-validation, 1 hyperparameter, 10 total values = 30 fits
- 10 fold cross-validation, 3 hyperparameters, 30 total values = 900 fits

RandomizedSearchCV

```
from sklearn.model_selection import RandomizedSearchCV
kf = KFold(n_splits=5, shuffle=True, random_state=42)
param_grid = {'alpha': np.arange(0.0001, 1, 10),
              "solver": ['sag', 'lsqr']}
ridge = Ridge()
ridge_cv = RandomizedSearchCV(ridge, param_grid, cv=kf, n_iter=2)
ridge_cv.fit(X_train, y_train)
print(ridge_cv.best_params_, ridge_cv.best_score_)
```

```
{'solver': 'sag', 'alpha': 0.0001}
0.7529912278705785
```

Evaluating on the test set

```
test_score = ridge_cv.score(X_test, y_test)
print(test_score)
```

```
0.7564731534089224
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Preprocessing data

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

scikit-learn requirements

- Numeric data
- No missing values

- With real-world data:
  - This is rarely the case
  - We will often need to preprocess our data first

Dealing with categorical features

- scikit-learn will not accept categorical features by default
- Need to convert categorical features into numeric values
- Convert to binary features called dummy variables
  - 0: Observation was NOT that category
  - 1: Observation was that category

Dummy variables

genre
Alternative
Anime
Blues
Classical
Country
Electronic
Hip-Hop
Jazz
Rap
Rock

Dummy variables

genre
Alternative
Anime
Blues
Classical
Country
Electronic
Hip-Hop
Jazz
Rap
Rock

Alternative	Anime	Blues	Classical	Country	Electronic	Hip-Hop	Jazz	Rap	Rock
1	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	0	1

Dummy variables

genre
Alternative
Anime
Blues
Classical
Country
Electronic
Hip-Hop
Jazz
Rap
Rock

Alternative	Anime	Blues	Classical	Country	Electronic	Hip-Hop	Jazz	Rap
1	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0
0	0	0	0	1	0	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0

Dealing with categorical features in Python

- scikit-learn: `OneHotEncoder()`
- pandas: `get_dummies()`

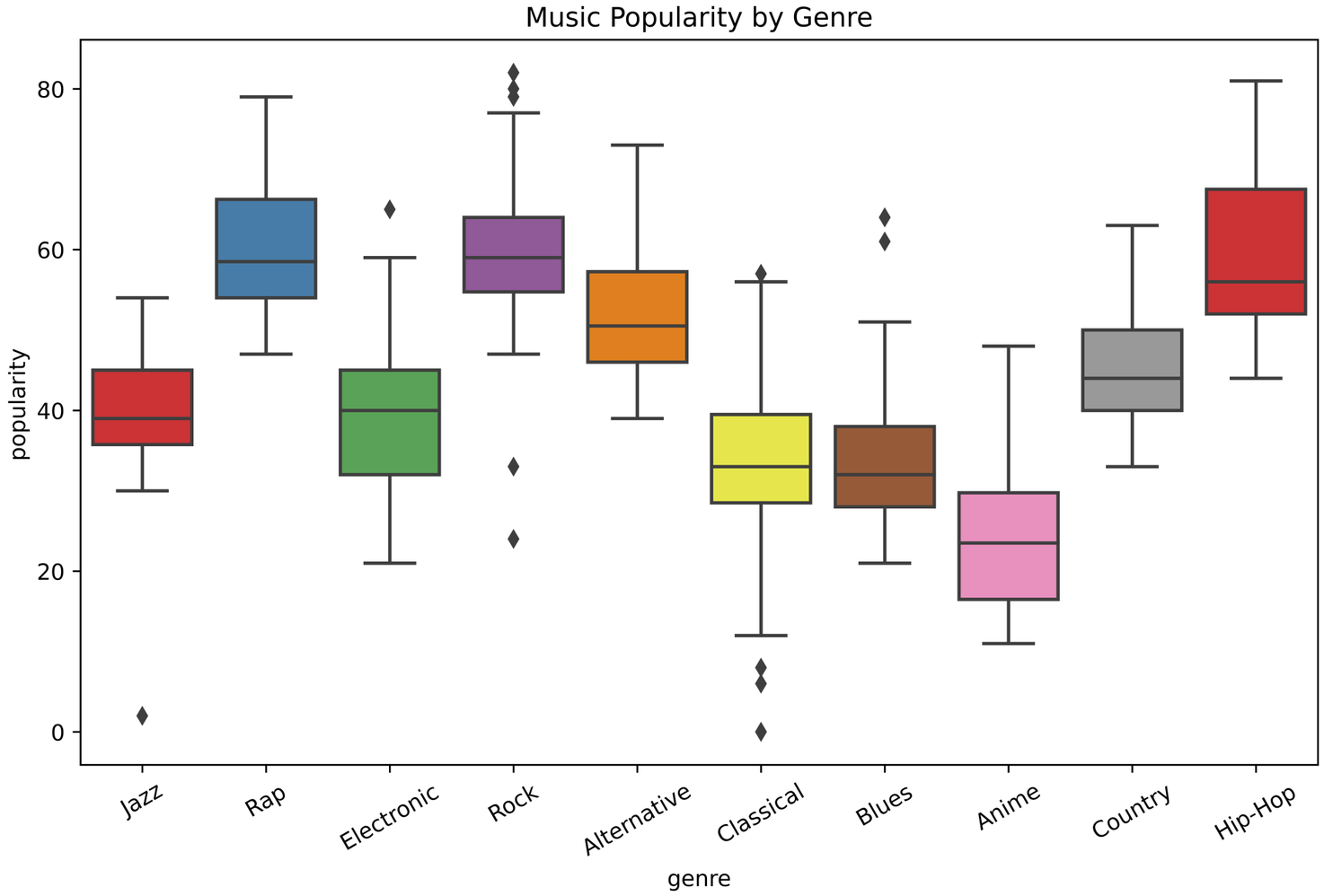
Music dataset

- `popularity` : Target variable
- `genre` : Categorical feature

```
print(music.info())
```

```
    popularity    acousticness    danceability    ...    tempo         valence    genre
0   41.0          0.6440          0.823           ...    102.619000    0.649      Jazz
1   62.0          0.0855          0.686           ...    173.915000    0.636      Rap
2   42.0          0.2390          0.669           ...    145.061000    0.494      Electronic
3   64.0          0.0125          0.522           ...    120.406497    0.595      Rock
4   60.0          0.1210          0.780           ...    96.056000     0.312      Rap
```

EDA w/ categorical feature



Encoding dummy variables

```
import pandas as pd
music_df = pd.read_csv('music.csv')
music_dummies = pd.get_dummies(music_df["genre"], drop_first=True)
print(music_dummies.head())
```

```
   Anime  Blues  Classical  Country  Electronic  Hip-Hop  Jazz  Rap  Rock
0      0      0          0        0           0        0     1    0     0
1      0      0          0        0           0        0     0    1     0
2      0      0          0        0           1        0     0    0     0
3      0      0          0        0           0        0     0    0     1
4      0      0          0        0           0        0     0    1     0
```

```
music_dummies = pd.concat([music_df, music_dummies], axis=1)
music_dummies = music_dummies.drop("genre", axis=1)
```

Encoding dummy variables

```
music_dummies = pd.get_dummies(music_df, drop_first=True)
print(music_dummies.columns)
```

```
Index(['popularity', 'acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
       'valence', 'genre_Anime', 'genre_Blues', 'genre_Classical',
       'genre_Country', 'genre_Electronic', 'genre_Hip-Hop', 'genre_Jazz',
       'genre_Rap', 'genre_Rock'],
      dtype='object')
```

Linear regression with dummy variables

```
from sklearn.model_selection import cross_val_score, KFold
from sklearn.linear_model import LinearRegression
X = music_dummies.drop("popularity", axis=1).values
y = music_dummies["popularity"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)
kf = KFold(n_splits=5, shuffle=True, random_state=42)
linreg = LinearRegression()
linreg_cv = cross_val_score(linreg, X_train, y_train, cv=kf,
                            scoring="neg_mean_squared_error")
print(np.sqrt(-linreg_cv))
```

```
[8.15792932, 8.63117538, 7.52275279, 8.6205778, 7.91329988]
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Handling missing data

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Missing data

- No value for a feature in a particular row
- This can occur because:
  - There may have been no observation
  - The data might be corrupt
- We need to deal with missing data

Music dataset

```
print(music_df.isna().sum().sort_values())
```

```
genre                8
popularity          31
loudness            44
liveness            46
tempo               46
speechiness         59
duration_ms         91
instrumentalness    91
danceability       143
valence            143
acousticness       200
energy             200
dtype: int64
```

Dropping missing data

```
music_df = music_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])
print(music_df.isna().sum().sort_values())
```

```
popularity           0
liveness             0
loudness             0
tempo                0
genre                0
duration_ms         29
instrumentalness    29
speechiness         53
danceability       127
valence            127
acousticness       178
energy             178
dtype: int64
```

Imputing values

- Imputation - use subject-matter expertise to replace missing data with educated guesses
- Common to use the mean
- Can also use the median, or another value
- For categorical values, we typically use the most frequent value - the mode
- Must split our data first, to avoid *data leakage*

Imputation with scikit-learn

```
from sklearn.impute import SimpleImputer
X_cat = music_df["genre"].values.reshape(-1, 1)
X_num = music_df.drop(["genre", "popularity"], axis=1).values
y = music_df["popularity"].values
X_train_cat, X_test_cat, y_train, y_test = train_test_split(X_cat, y, test_size=0.2,
                                                            random_state=12)
X_train_num, X_test_num, y_train, y_test = train_test_split(X_num, y, test_size=0.2,
                                                            random_state=12)
imp_cat = SimpleImputer(strategy="most_frequent")
X_train_cat = imp_cat.fit_transform(X_train_cat)
X_test_cat = imp_cat.transform(X_test_cat)
```

Imputation with scikit-learn

```
imp_num = SimpleImputer()
X_train_num = imp_num.fit_transform(X_train_num)
X_test_num = imp_num.transform(X_test_num)
X_train = np.append(X_train_num, X_train_cat, axis=1)
X_test = np.append(X_test_num, X_test_cat, axis=1)
```

- Imputers are known as transformers

Imputing within a pipeline

```
from sklearn.pipeline import Pipeline
music_df = music_df.dropna(subset=["genre", "popularity", "loudness", "liveness", "tempo"])
music_df["genre"] = np.where(music_df["genre"] == "Rock", 1, 0)
X = music_df.drop("genre", axis=1).values
y = music_df["genre"].values
```

Imputing within a pipeline

```
steps = [("imputation", SimpleImputer()),
         ("logistic_regression", LogisticRegression())]
pipeline = Pipeline(steps)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)
pipeline.fit(X_train, y_train)
pipeline.score(X_test, y_test)
```

```
0.7593582887700535
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Centering and scaling

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager

datacamp

Why scale our data?

```
print(music_df[["duration_ms", "loudness", "speechiness"]].describe())
```

```
       duration_ms     loudness      speechiness
count    1.000000e+03  1000.000000  1000.000000
mean     2.176493e+05    -8.284354     0.078642
std      1.137703e+05     5.065447     0.088291
min     -1.000000e+00   -38.718000     0.023400
25%      1.831070e+05    -9.658500     0.033700
50%      2.176493e+05    -7.033500     0.045000
75%      2.564468e+05    -5.034000     0.078642
max      1.617333e+06    -0.883000     0.710000
```

Why scale our data?

- Many models use some form of distance to inform them
- Features on larger scales can disproportionately influence the model
- Example: KNN uses distance explicitly when making predictions
- We want features to be on a similar scale
- Normalizing or standardizing (scaling and centering)

How to scale our data

- Subtract the mean and divide by variance
  - All features are centered around zero and have a variance of one
  - This is called **standardization**
- Can also subtract the minimum and divide by the range
  - Minimum zero and maximum one
- Can also *normalize* so the data ranges from -1 to +1
- See scikit-learn docs for further details

Scaling in scikit-learn

```
from sklearn.preprocessing import StandardScaler
X = music_df.drop("genre", axis=1).values
y = music_df["genre"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
print(np.mean(X), np.std(X))
print(np.mean(X_train_scaled), np.std(X_train_scaled))
```

```
19801.42536120538, 71343.52910125865
2.260817795600319e-17, 1.0
```

Scaling in a pipeline

```
steps = [('scaler', StandardScaler()),
         ('knn', KNeighborsClassifier(n_neighbors=6))]
pipeline = Pipeline(steps)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=21)
knn_scaled = pipeline.fit(X_train, y_train)
y_pred = knn_scaled.predict(X_test)
print(knn_scaled.score(X_test, y_test))
```

```
0.81
```

Comparing performance using unscaled data

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=21)
knn_unscaled = KNeighborsClassifier(n_neighbors=6).fit(X_train, y_train)
print(knn_unscaled.score(X_test, y_test))
```

```
0.53
```

CV and scaling in a pipeline

```
from sklearn.model_selection import GridSearchCV
steps = [('scaler', StandardScaler()),
         ('knn', KNeighborsClassifier())]
pipeline = Pipeline(steps)
parameters = {"knn__n_neighbors": np.arange(1, 50)}
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=21)
cv = GridSearchCV(pipeline, param_grid=parameters)
cv.fit(X_train, y_train)
y_pred = cv.predict(X_test)
```

Checking model parameters

```
print(cv.best_score_)
```

```
0.8199999999999999
```

```
print(cv.best_params_)
```

```
{'knn__n_neighbors': 12}
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Evaluating multiple models

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

Different models for different problems

Some guiding principles

- Size of the dataset
  - Fewer features = simpler model, faster training time
  - Some models require large amounts of data to perform well
- Interpretability
  - Some models are easier to explain, which can be important for stakeholders
  - Linear regression has high interpretability, as we can understand the coefficients
- Flexibility
  - May improve accuracy, by making fewer assumptions about data
  - KNN is a more flexible model, doesn't assume any linear relationships

It's all in the metrics

- Regression model performance:
  - RMSE
  - R-squared
- Classification model performance:
  - Accuracy
  - Confusion matrix
  - Precision, recall, F1-score
  - ROC AUC
- Train several models and evaluate performance out of the box

A note on scaling

- Models affected by scaling:
  - KNN
  - Linear Regression (plus Ridge, Lasso)
  - Logistic Regression
  - Artificial Neural Network

- Best to scale our data before evaluating models

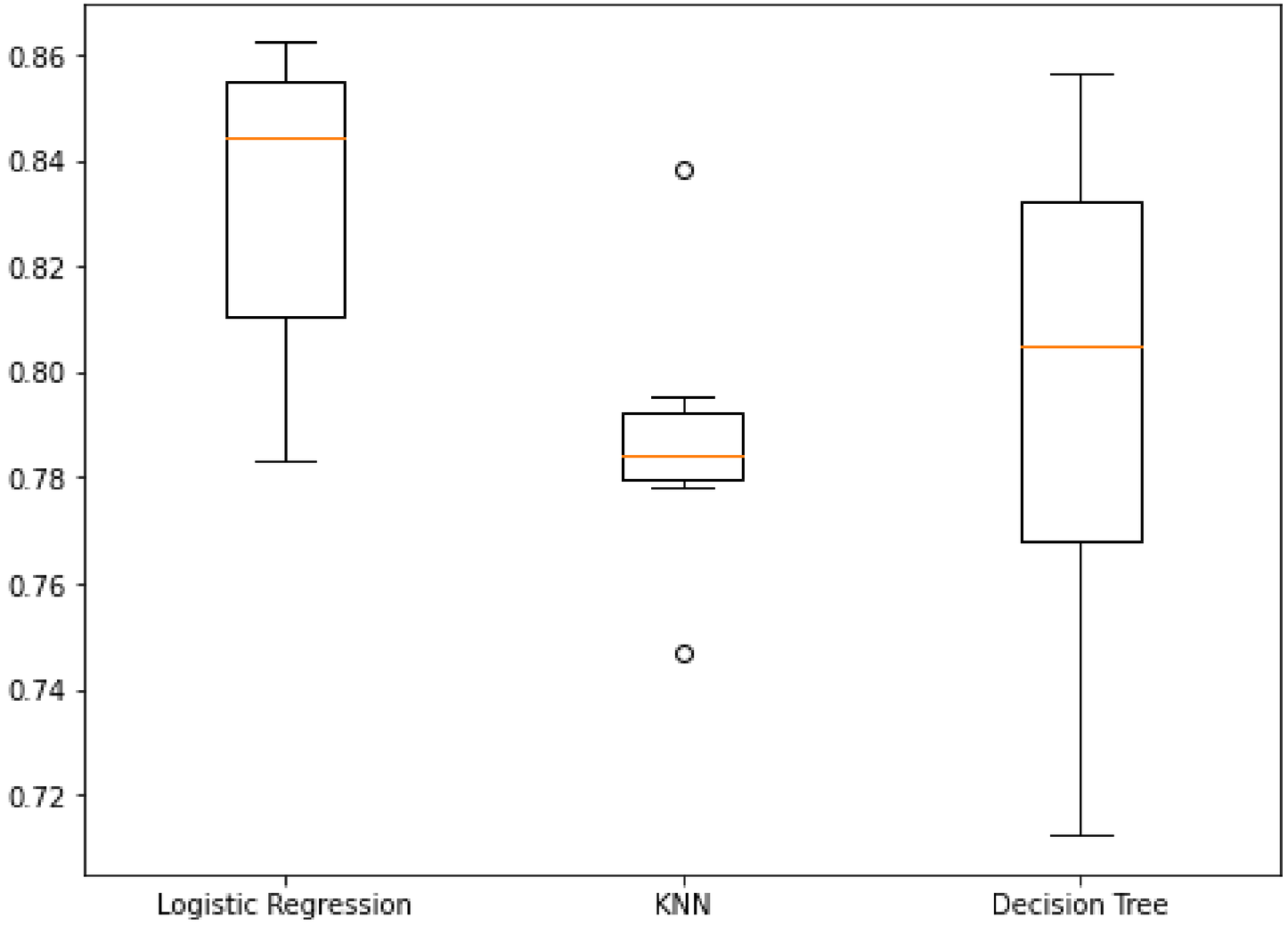
Evaluating classification models

```
import matplotlib.pyplot as plt
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import cross_val_score, KFold, train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
X = music.drop("genre", axis=1).values
y = music["genre"].values
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Evaluating classification models

```
models = {"Logistic Regression": LogisticRegression(), "KNN": KNeighborsClassifier(),
          "Decision Tree": DecisionTreeClassifier()}
results = []
for model in models.values():
     kf = KFold(n_splits=6, random_state=42, shuffle=True)
     cv_results = cross_val_score(model, X_train_scaled, y_train, cv=kf)
     results.append(cv_results)
plt.boxplot(results, labels=models.keys())
plt.show()
```

Visualizing results



Test set performance

```
for name, model in models.items():
  model.fit(X_train_scaled, y_train)
  test_score = model.score(X_test_scaled, y_test)
  print("{} Test Set Accuracy: {}".format(name, test_score))
```

```
Logistic Regression Test Set Accuracy: 0.844
KNN Test Set Accuracy: 0.82
Decision Tree Test Set Accuracy: 0.832
```

Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

Congratulations

SUPERVISED LEARNING WITH SCIKIT-LEARN

George Boorman
Core Curriculum Manager, DataCamp

datacamp

What you've covered

- Using supervised learning techniques to build predictive models
- For both regression and classification problems
- Underfitting and overfitting
- How to split data
- Cross-validation

What you've covered

- Data preprocessing techniques
- Model selection
- Hyperparameter tuning
- Model performance evaluation
- Using pipelines

Where to go from here?

- Machine Learning with Tree-Based Models in Python
- Preprocessing for Machine Learning in Python
- Model Validation in Python
- Feature Engineering for Machine Learning in Python
- Unsupervised Learning in Python
- Machine Learning Projects

Thank you!

SUPERVISED LEARNING WITH SCIKIT-LEARN