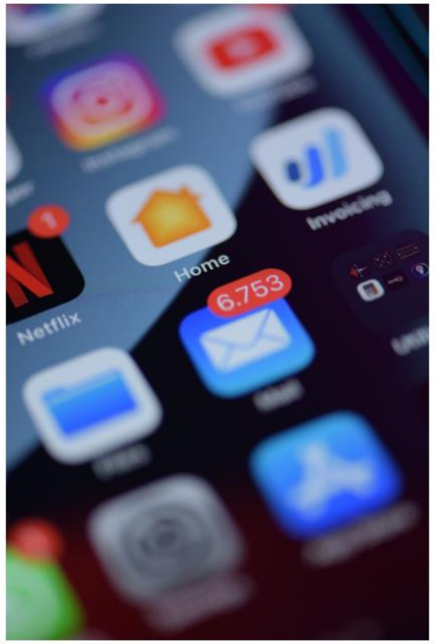
Machine Learning with Scikit-learn

Supervised Machine Learning

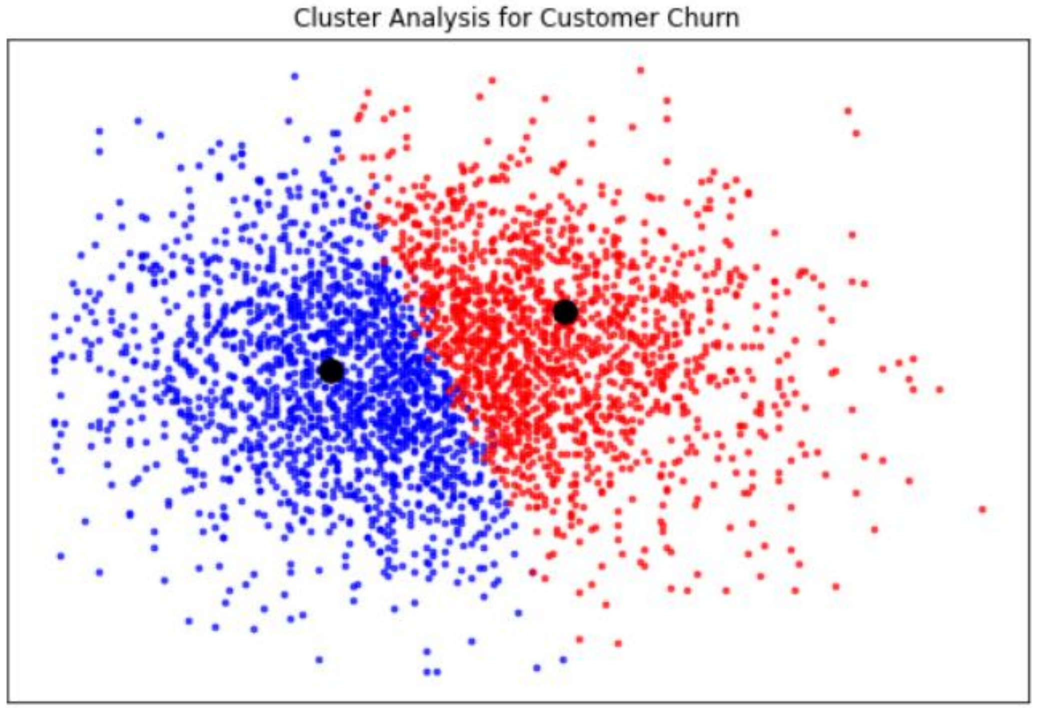
What is machine learning?

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



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

```python
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])
```

Classifying labels of unseen data

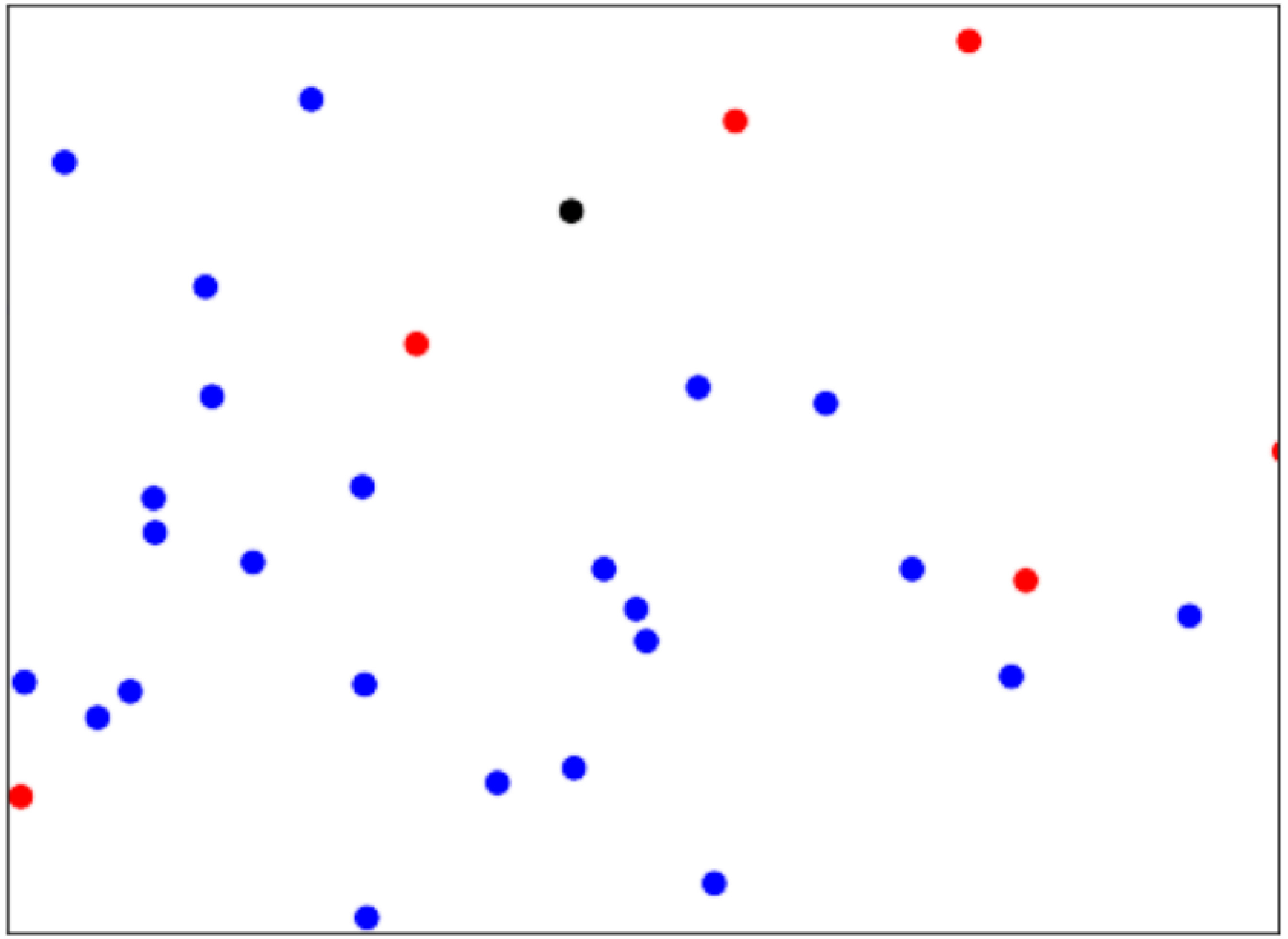
1. Build a model
2. Model learns from the labeled data we pass to it
3. Pass unlabeled data to the model as input
4. Model predicts the labels of the unseen data

- Labeled data = training data

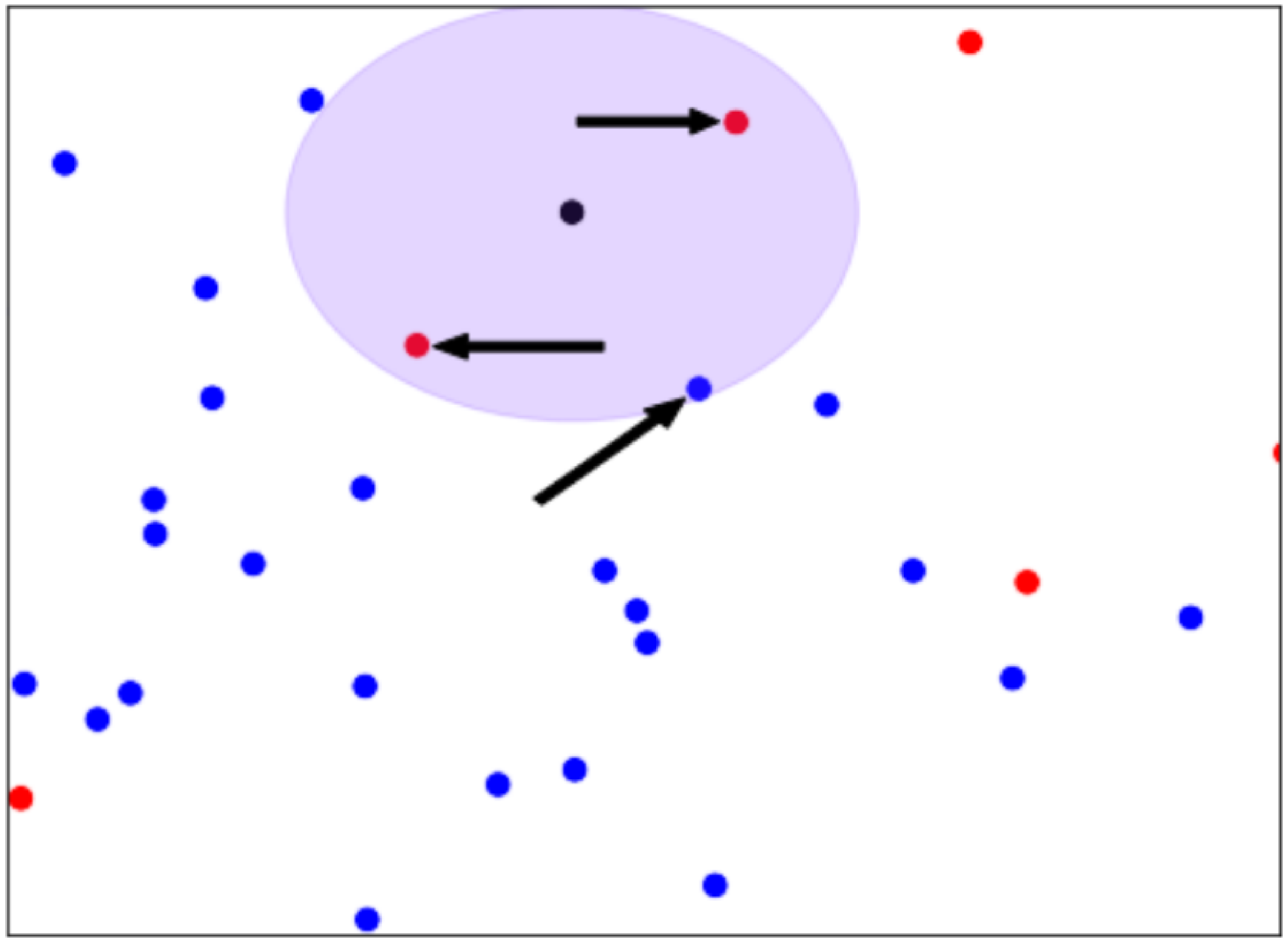
k-Nearest Neighbors

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

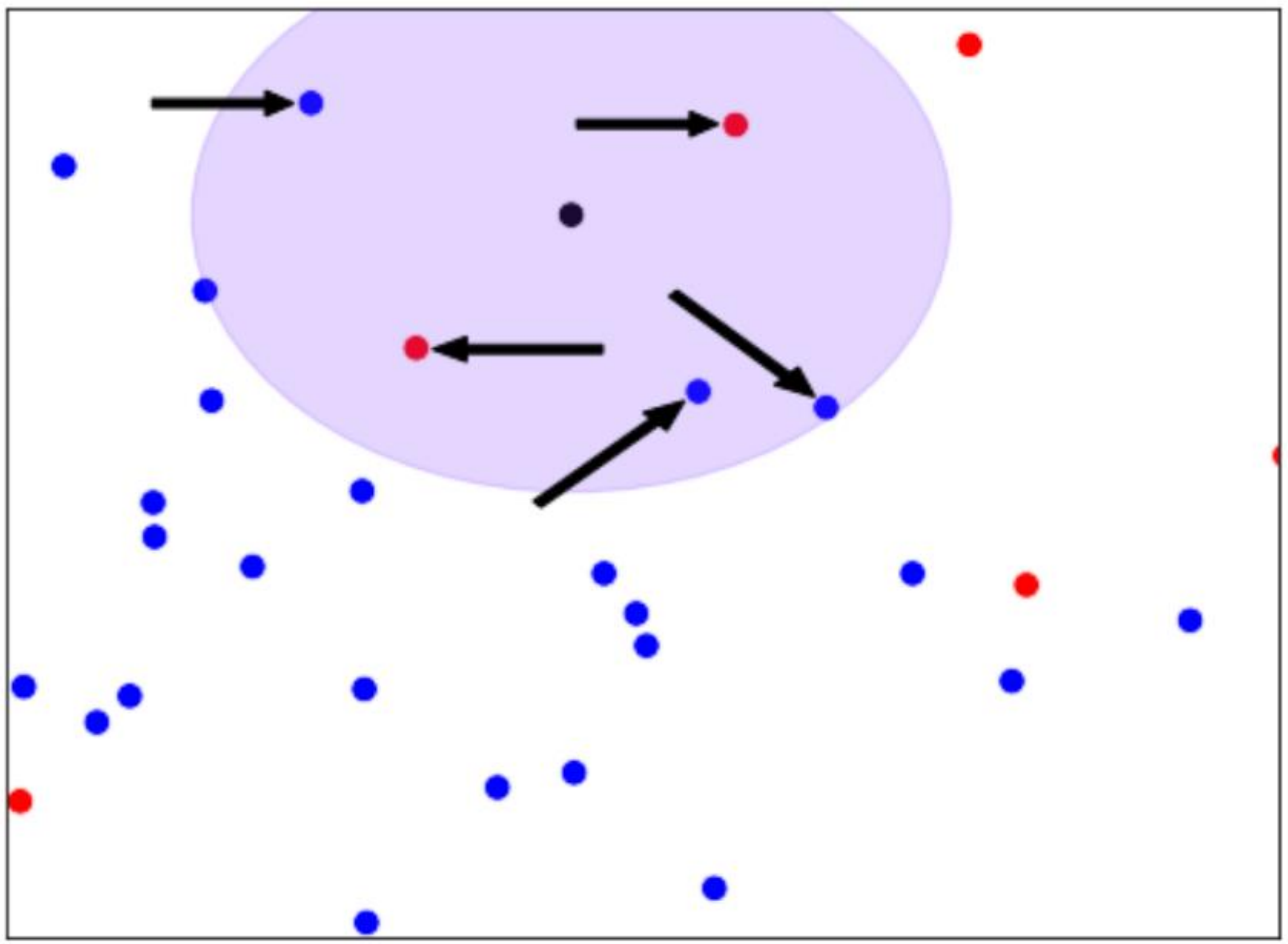
k-Nearest Neighbors



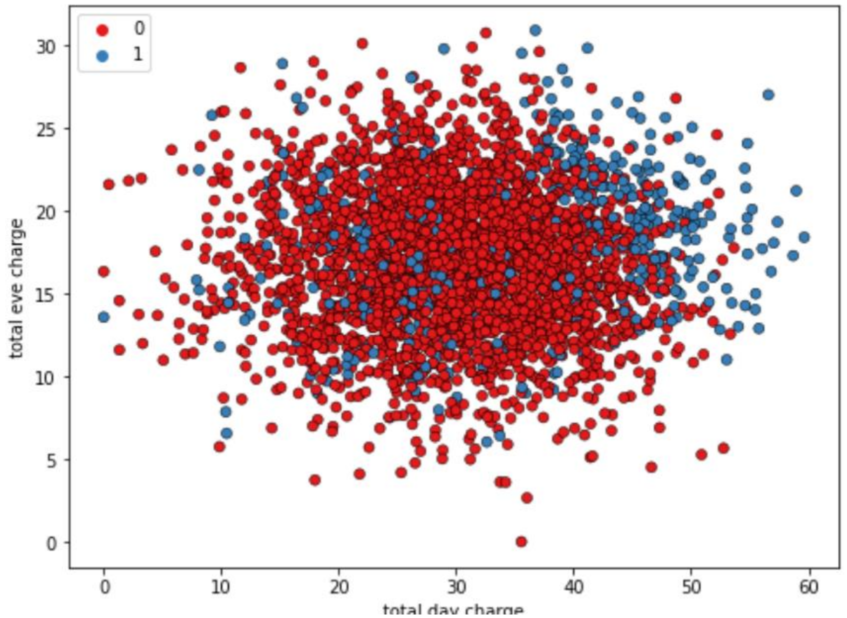
k-Nearest Neighbors



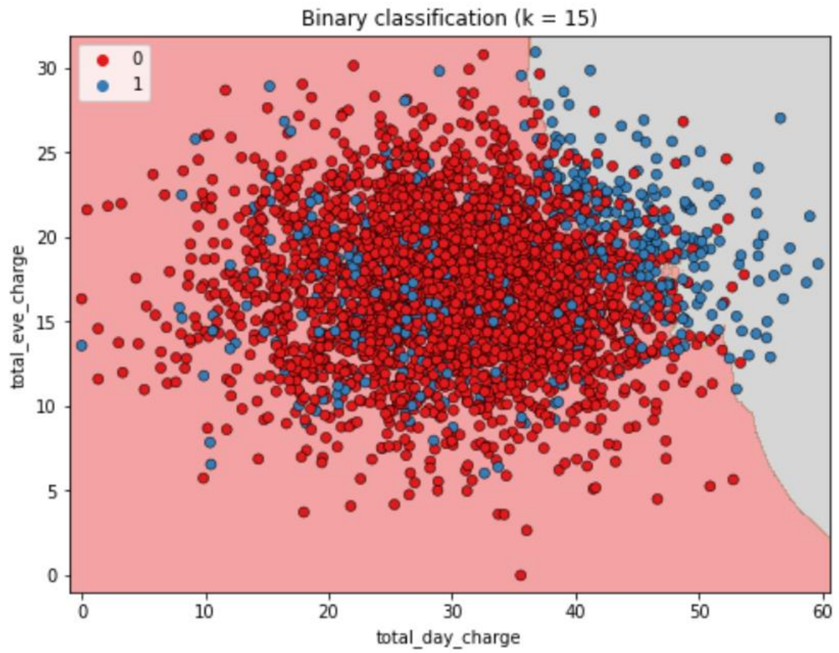
k-Nearest Neighbors



KNN Intuition



To build intuition for KNN, let's look at this scatter plot displaying total evening charge against total day charge for customers of a telecom company. The observations are colored in blue for customers who have churned, and red for those who have not churned.



Here we have visualized the results of a KNN algorithm where the number of neighbors is set to 15. KNN creates a decision boundary to predict if customers will churn. Any customers in the area with a gray background are predicted to churn, and those in the area with a red background are predicted to not churn. This boundary would be used to make predictions on unseen data.

Using scikit-learn to fit a classifier

```python
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,)
```

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

Predicting on unlabeled data

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

```
(3, 2)
```

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

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

1 corresponds to churn and 0 corresponds to no churn

k-Nearest Neighbors: Fit

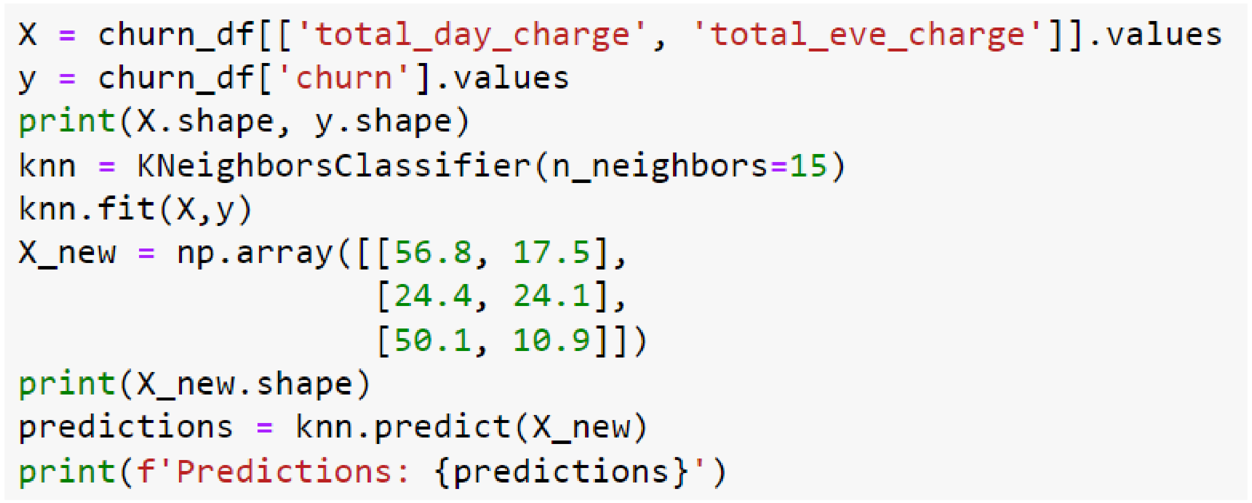
In this exercise, you will build your first classification model using the churn_df dataset, can be loaded from churn_df.csv.

The features to use will be "account_length" and "customer_service_calls". The target, "churn", needs to be a single column with the same number of observations as the feature data.

You will convert the features and the target variable into NumPy arrays, create an instance of a KNN classifier, and then fit it to the data.

Instructions

- Import KNeighborsClassifier from sklearn.neighbors.
- Create an array called X containing values from the "account_length" and "customer_service_calls" columns, and an array called y for the values of the "churn" column.
- Instantiate a KNeighborsClassifier called knn with 6 neighbors.
- Fit the classifier to the data using the .fit() method.

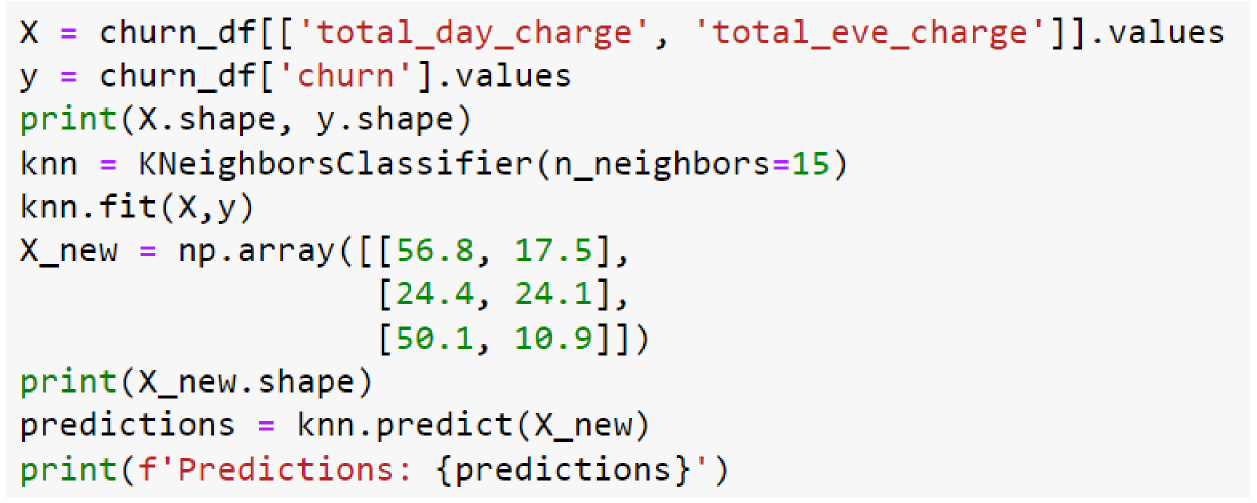


```
X = churn_df[['total_day_charge', 'total_eve_charge']].values
y = churn_df['churn'].values
print(X.shape, y.shape)
knn = KNeighborsClassifier(n_neighbors=15)
knn.fit(X,y)
X_new = np.array([[56.8, 17.5],
                  [24.4, 24.1],
                  [50.1, 10.9]])
print(X_new.shape)
predictions = knn.predict(X_new)
print(f'Predictions: {predictions}')
```

k-Nearest Neighbors: Predict

Now you have fit a KNN classifier, you can use it to predict the label of new data points. All available data was used for training, however, fortunately, there are new observations available, X_new.

The model knn, which you created and fit the data in the last exercise, will be used. You will use your classifier to predict the labels of a set of new data points:

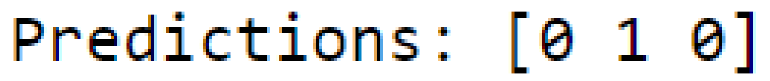


```python
X = churn_df[['total_day_charge', 'total_eve_charge']].values
y = churn_df['churn'].values
print(X.shape, y.shape)
knn = KNeighborsClassifier(n_neighbors=15)
knn.fit(X,y)
X_new = np.array([[56.8, 17.5],
                  [24.4, 24.1],
                  [50.1, 10.9]])
print(X_new.shape)
predictions = knn.predict(X_new)
print(f'Predictions: {predictions}')
```

```
X_new = np.array([[30.0, 17.5],
          [107.0, 24.1],
          [213.0, 10.9]])
```

Instructions

- Create y_pred by predicting the target values of the unseen features X_new.
- Print the predicted labels for the set of predictions.



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

Measuring model performance

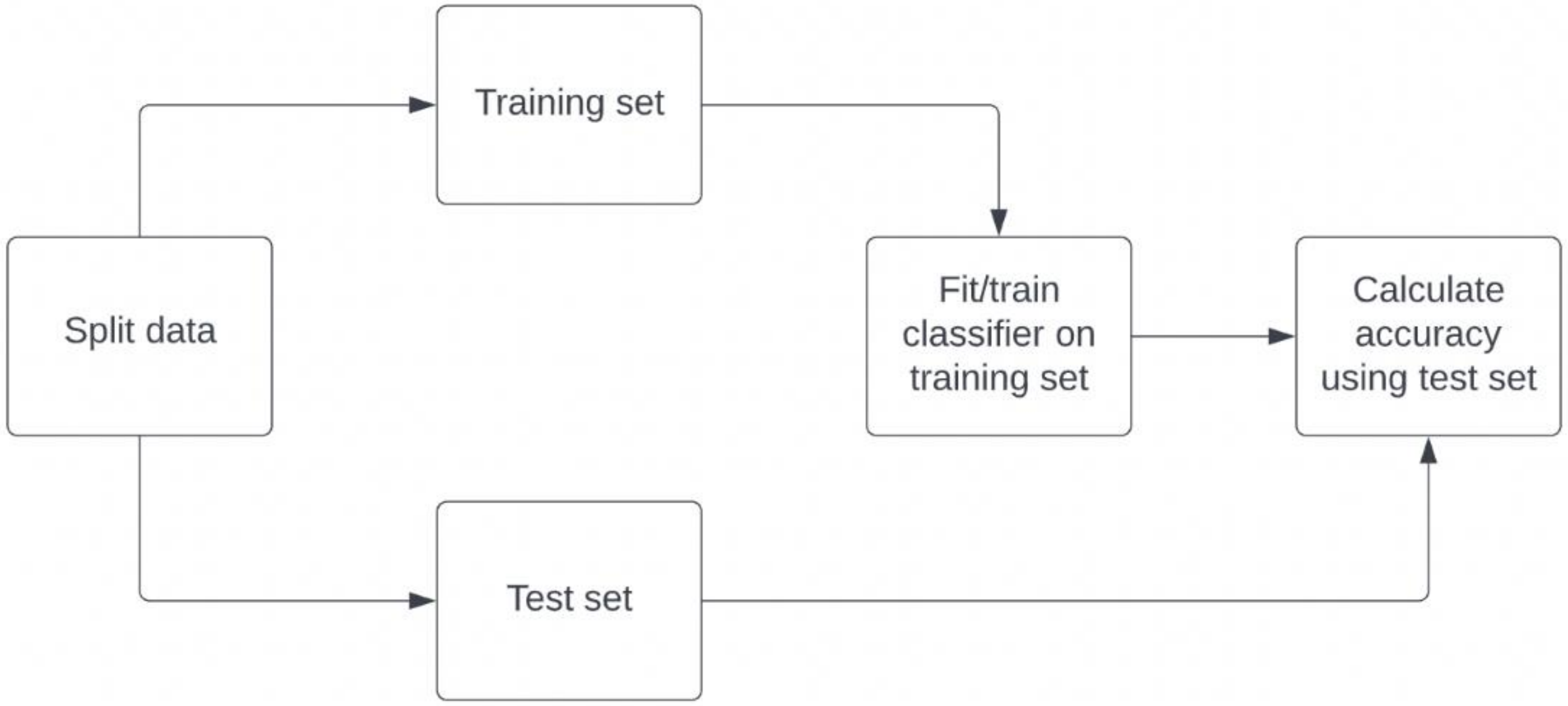
Measuring model performance

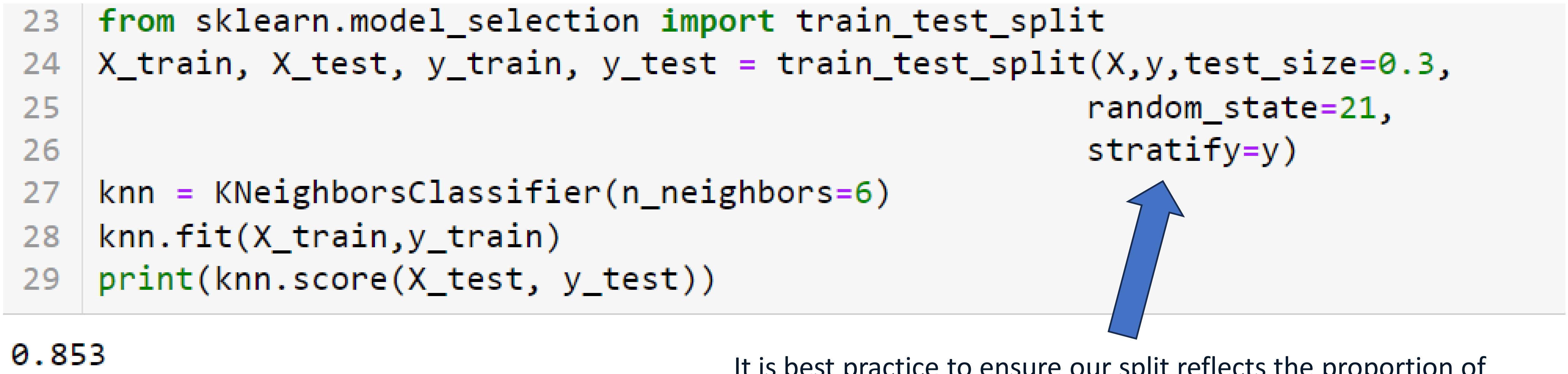
- In classification, accuracy is a commonly used metric
- Accuracy:

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

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

Computing accuracy


Training set
Split data
Fit/train classifier on training set
Calculate accuracy using test set
Test set



```python
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))
```

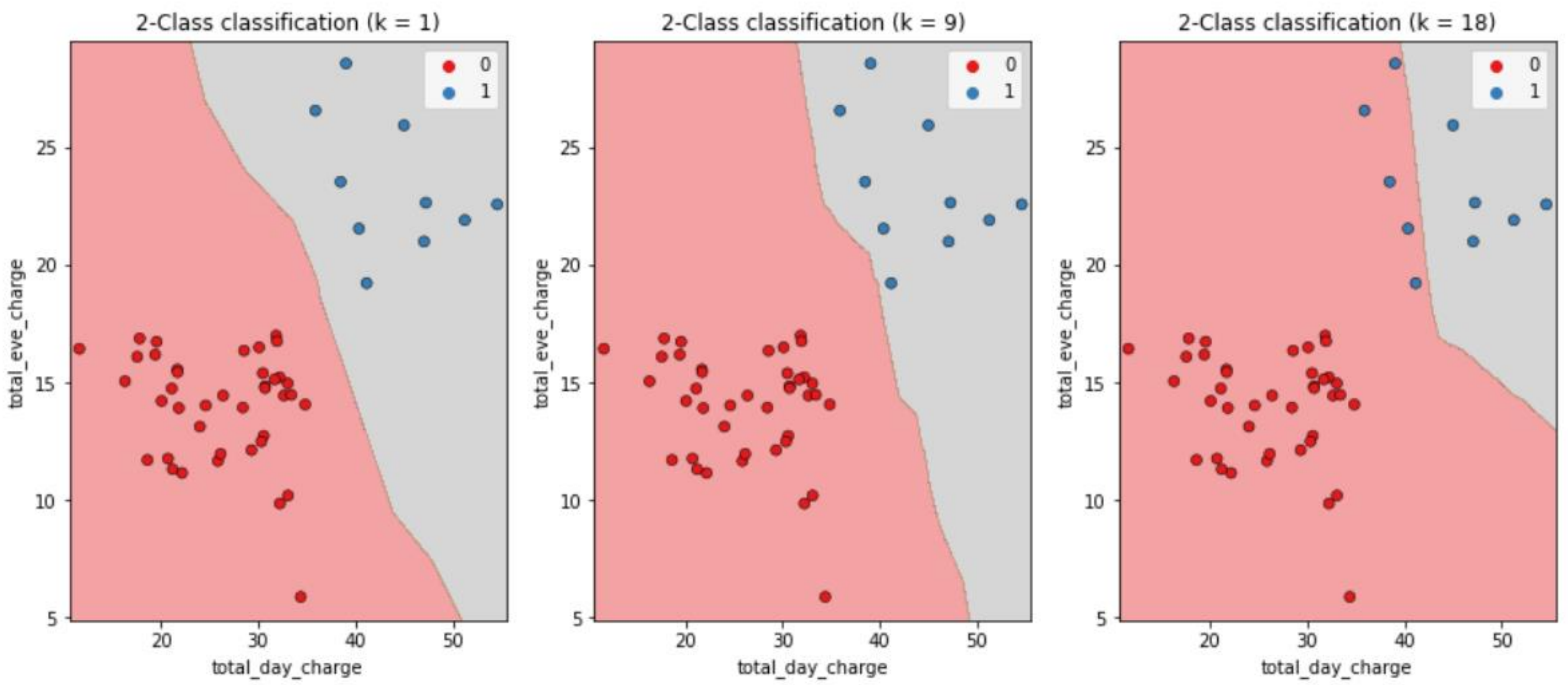
```
0.853
```

It is best practice to ensure our split reflects the proportion of labels in our data. So if churn occurs in 10% of observations, we want 10% of labels in our training and test sets to represent churn. We achieve this by setting stratify equal to y.

```python
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=2)
knn.fit(X_train,y_train)
print(knn.score(X_test, y_test))
```

Model complexity

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

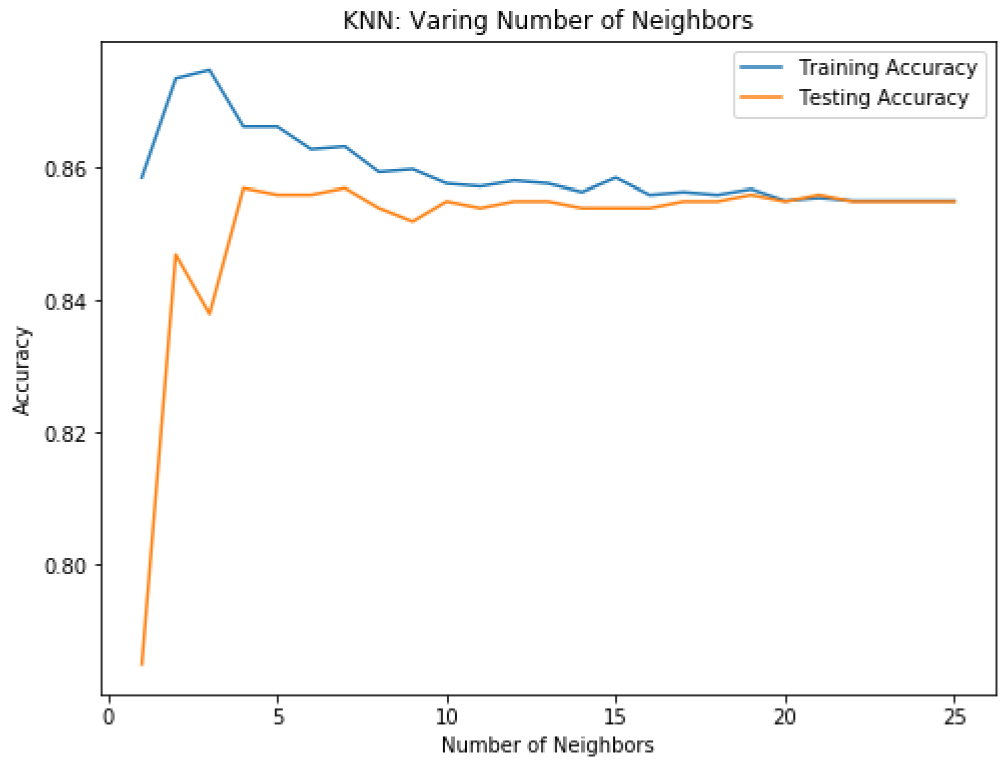


Recall that we discussed decision boundaries, which are thresholds for determining what label a model assigns to an observation. In the image shown, as k increases, the decision boundary is less affected by individual observations, reflecting a simpler model. Simpler models are less able to detect relationships in the dataset, which is known as underfitting. In contrast, complex models can be sensitive to noise in the training data, rather than reflecting general trends. This is known as overfitting.

```
train_accuracies = {}
test_accuracies = {}
neighbors = np.arange(1,26)
print(neighbors)
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)

#print(train_accuracies.values())
print(test_accuracies.values())
my_train = list(train_accuracies.values())
my_test = list(test_accuracies.values())

plt.figure(figsize=(8,6))
plt.title('KNN: Varing Number of Neighbors')
plt.plot(neighbors, my_train, label='Training Accuracy')
plt.plot(neighbors, my_test, label='Testing Accuracy')
plt.legend()
plt.xlabel('Number of Neighbors')
plt.ylabel('Accuracy')
plt.show()
```



Here's the result! As k increases beyond 15 we see overfitting where performance plateaus on both test and training sets, as indicated in this plot. Peak occurs at around 16.

Exercise

NumPy arrays have been created for you containing the features as X and the target variable as y. You will split them into training and test sets, fit a KNN classifier to the training data, and then compute its accuracy on the test data using the .score() method.

Instructions:

- Import train_test_split from sklearn.model_selection.
- Create Numpy arrays having all columns, except churn, as features X, and target variable, churn, as target variable y
- Split X and y into training and test sets, setting test_size equal to 20%, random_state to 42, and ensuring the target label proportions reflect that of the original dataset.
- Fit the knn model to the training data.
- Compute and print the model's accuracy for the test data.

```
# Import the module
from sklearn.model_selection import train_test_split

X = churn_df.drop("churn", axis=1).values
y = churn_df["churn"].values
```

Overfitting and underfitting

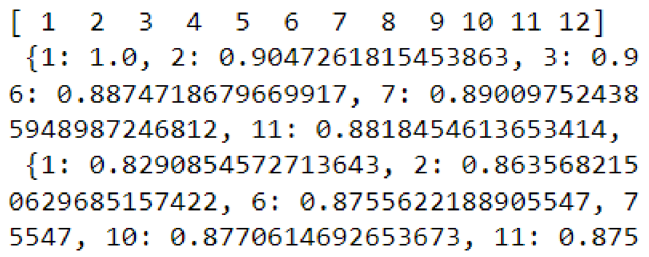
Interpreting model complexity is a great way to evaluate performance when utilizing supervised learning. Your aim is to produce a model that can interpret the relationship between features and the target variable, as well as generalize well when exposed to new observations.

You will generate accuracy scores for the training and test sets using a KNN classifier with different n_neighbor values, which you will plot in the next exercise.

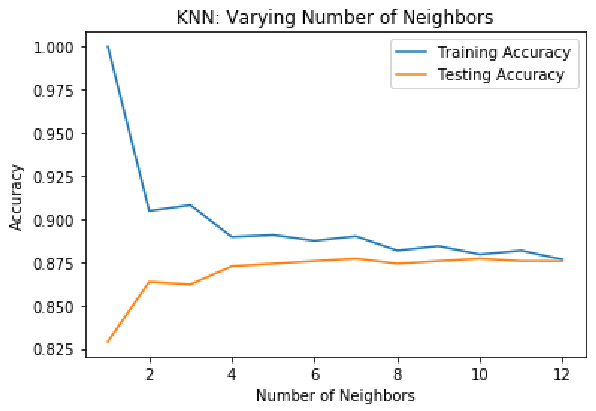
The training and test sets have been created from the churn_df dataset and preloaded as X_train, X_test, y_train, and y_test.

Instructions

- Create neighbors as a numpy array of values from 1 up to and including 12.
- Instantiate a KNN classifier, with the number of neighbors equal to the neighbor iterator.
- Fit the model to the training data.
- Calculate accuracy scores for the training set and test set separately using the .score() method, and assign the results to the index of the train_accuracies and test_accuracies dictionaries, respectively.
- Visual the accuracy



```
[ 1  2  3  4  5  6  7  8  9 10 11 12]
 {1: 1.0, 2: 0.9047261815453863, 3: 0.9
6: 0.8874718679669917, 7: 0.89009752438
5948987246812, 11: 0.8818454613653414,
 {1: 0.8290854572713643, 2: 0.863568215
0629685157422, 6: 0.8755622188905547, 7
5547, 10: 0.8770614692653673, 11: 0.875
```



Regression with Scikit-Learn

Predicting blood glucose levels

```python
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'>
```

```
diabetes_df = pd.read_csv('diabetes.csv', index_col = 0)
print(diabetes_df.shape)

#subset row for bmi not equal to 0
diabetes_df =
print(diabetes_df.shape)

#subset for glucose not equal to 0
diabetes_df =
print(diabetes_df.shape)
diabetes.head()
```

Making predictions from a single feature

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

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

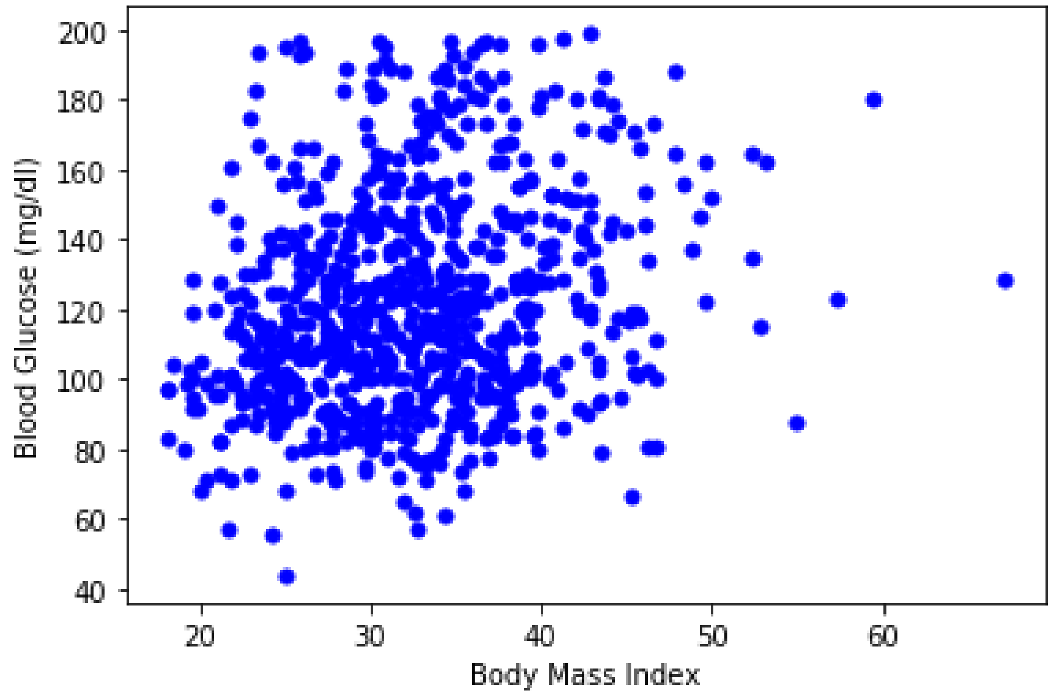
```
X_bmi = X_bmi.reshape(-1, 1)
print(X_bmi.shape)
```

```
(752, 1)
```

To start, let's try to predict blood glucose levels from a single feature: body mass index. To do this, we slice out the BMI column of X, which is the fourth column, storing as the variable X_bmi. Checking the shape of y and X_bmi, we see that they are both one-dimensional arrays. This is fine for y, but our features must be formatted as a two-dimensional array to be accepted by scikit-learn. To convert the shape of X_bmi we apply NumPy’s .reshape method, passing minus one followed by one. Printing the shape again shows X_bmi is now the correct shape for our model.

Plotting glucose vs. body mass index

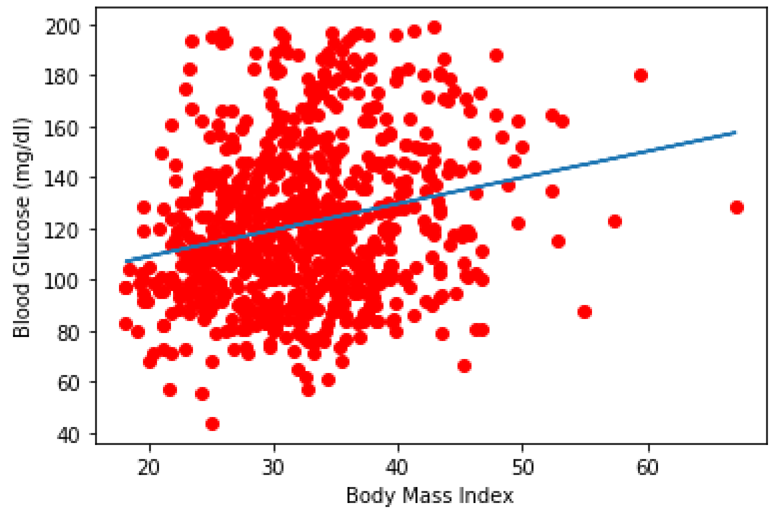
```python
import matplotlib.pyplot as plt
plt.scatter(X_bmi, y)
plt.ylabel("Blood Glucose (mg/dl)")
plt.xlabel("Body Mass Index")
plt.show()
```



You might get different color. But you can change the color by having color = ‘blue’ in scatter()

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()
```



Let’s practice

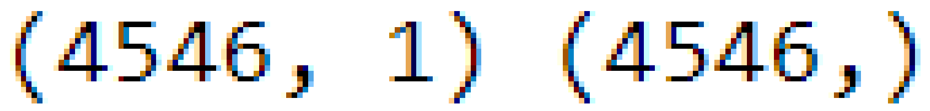
Creating features

In this exercise, you will work with a dataset called sales_df (loaded from sales_df.csv), which contains information on advertising campaign expenditure across different media types, and the number of dollars generated in sales for the respective campaign. The dataset has been preloaded for you. Here are the first two rows:

	tv	radio	social_media	sales
1	13000.0	9237.76	2409.57	46677.90
2	41000.0	15886.45	2913.41	150177.83

You will use the advertising expenditure as features to predict sales values, initially working with the "radio" column. However, before you make any predictions you will need to create the feature and target arrays, reshaping them to the correct format for scikit-learn.

- Create X, an array of the values from the sales_df DataFrame's "radio" column.
- Create y, an array of the values from the sales_df DataFrame's "sales" column.
- Reshape X into a two-dimensional NumPy array.
- Print the shape of X and y.



```
(4546, 1) (4546,)
```

Building a linear regression model
Now you have created your feature and target arrays, you will train a linear regression model on all feature and target values.

As the goal is to assess the relationship between the feature and target values there is no need to split the data into training and test sets.

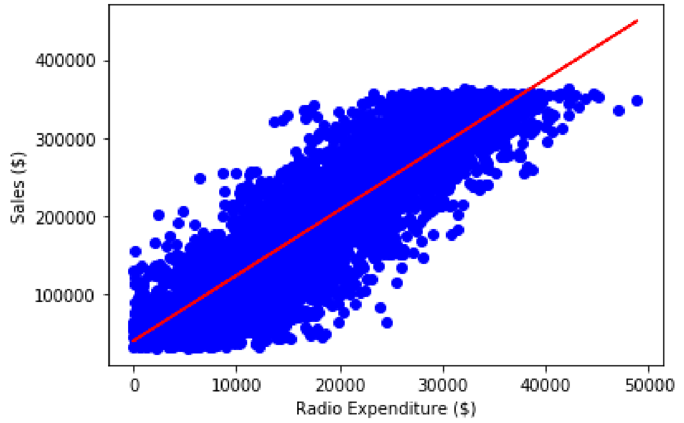
```
[ 95491.17119147 117829.51038393 173423.38071499 291603.11444202
 111137.28167129]
```

- Instantiate a linear regression model.
- Predict sales values using X, storing as predictions.
- Print five prediction values

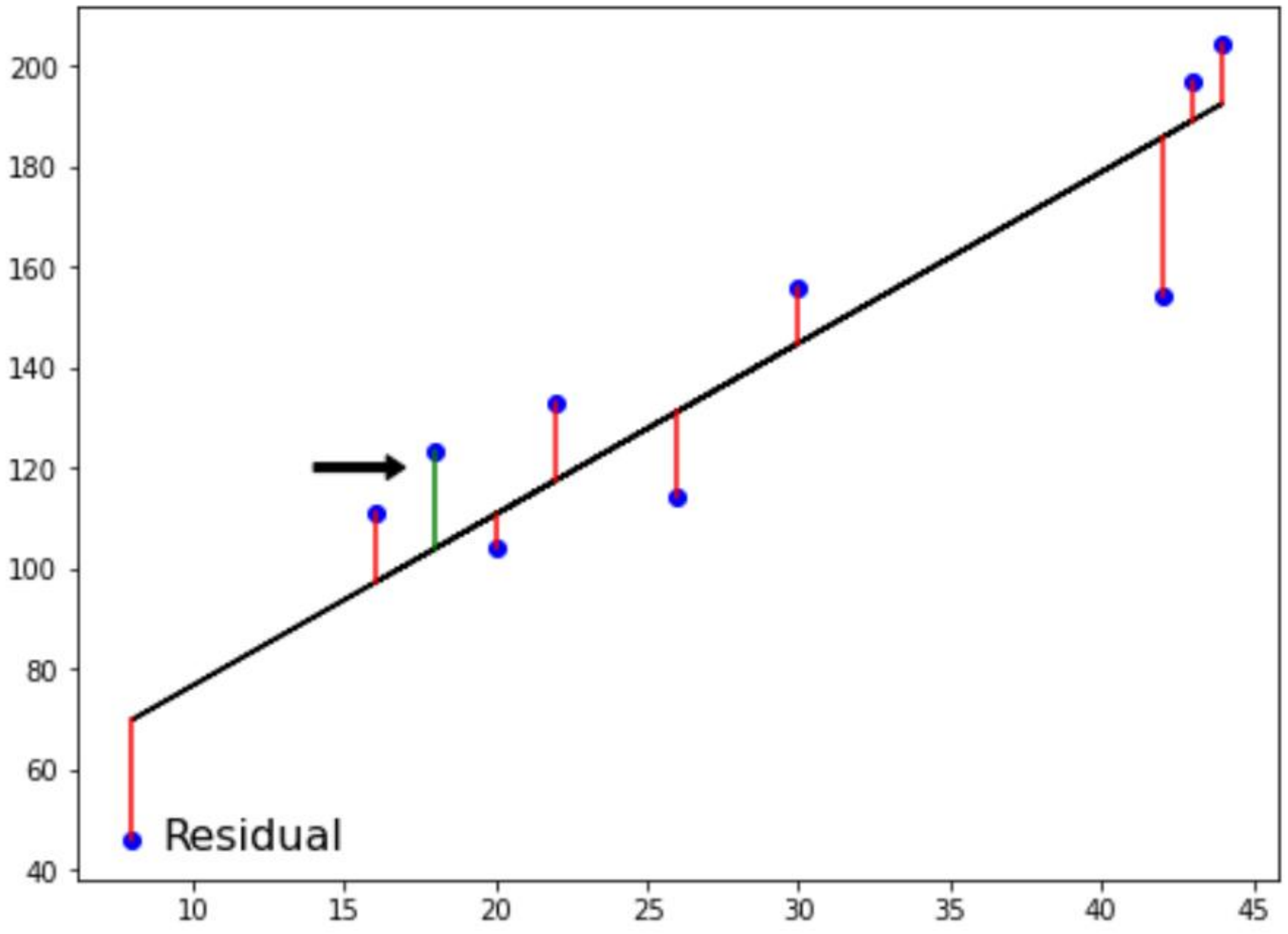
Visualizing a linear regression model
Now you have built your linear regression model and trained it using all available observations, you can visualize how well the model fits the data. This allows you to interpret the relationship between radio advertising expenditure and sales values.

The variables X, an array of radio values, y, an array of sales values, and predictions, an array of the model's predicted values for y given X, have all been preloaded for you from the previous exercise.

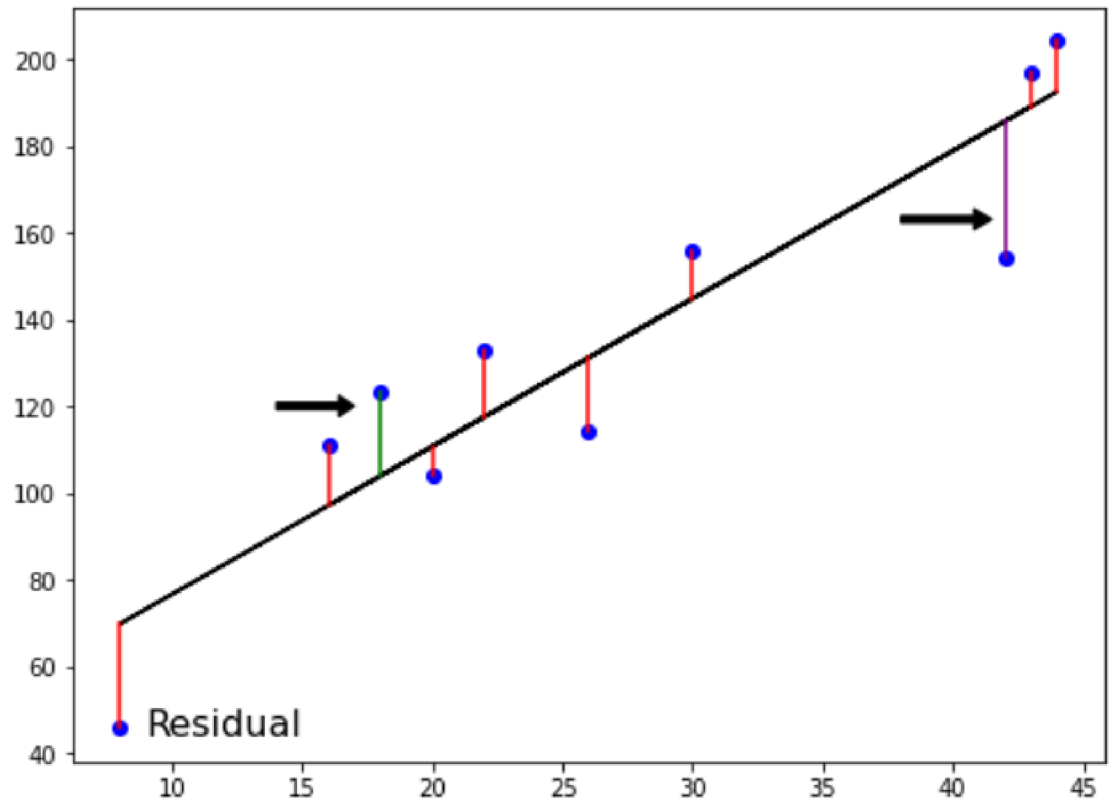
- Create a scatter plot visualizing y against X, with observations in blue.
- Draw a red line plot displaying the predictions against X.
- Display the plot.



The loss function



Ordinary Least Squares



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

Ordinary Least Squares (OLS): minimize RSS

The distance is called a residual. We could try to minimize the sum of the residuals, each negative residual. To avoid this, we square the residuals. By adding all the squared residuals, we calculate the residual sum of squares, or RSS. This type of linear regression is called Ordinary Least Squares, or OLS, where we aim to minimize the RSS.

Linear regression in higher dimensions

$$y = a_1x_1 + a_2x_2 + b$$

- 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

$$y = a_1x_1 + a_2x_2 + a_3x_3 + ... + a_nx_n + b$$

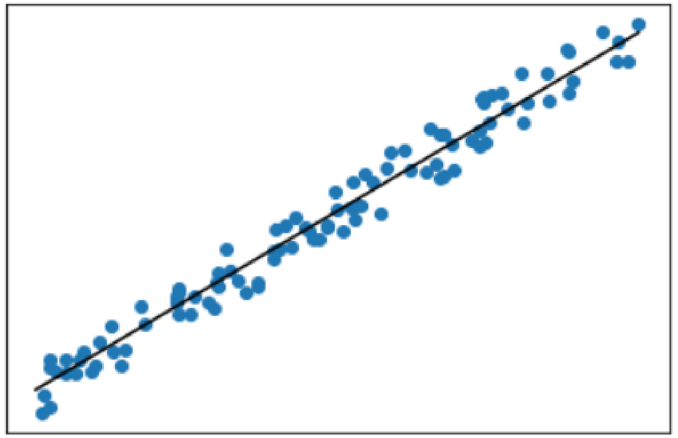
- scikit-learn works exactly the same way:
  - Pass two arrays: features and target

Linear regression using all features

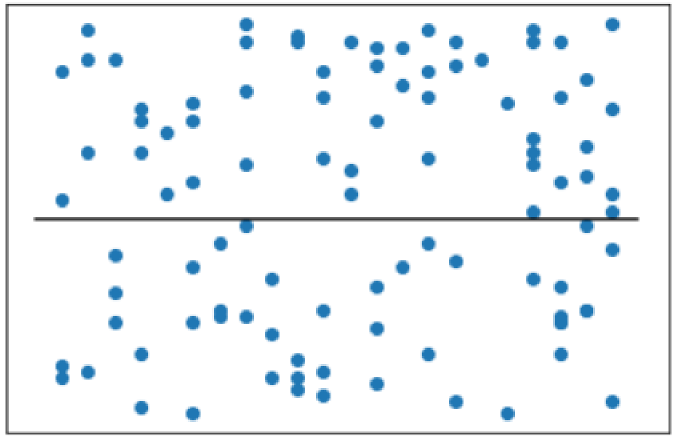
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
                                                    random_state=42)
reg_all = LinearRegression()
reg_all.fit(X_train, y_train)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

To compute R-squared, we call the model’s .score() method, passing the test features and targets. Here the features only explain about 35 percent of blood glucose level variance.

RMSE in scikit-learn

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

```
24.028109426907236
```

The model has an average error for blood glucose levels of around 24 milligrams per deciliter.

Fit and predict for regression

Now you have seen how linear regression works, your task is to create a multiple linear regression model using all of the features in the sales_df dataset. As a reminder, here are the first two rows:

```
        tv     radio  social_media      sales
0  16000.0   6566.23       2907.98   54732.76
1  13000.0   9237.76       2409.57   46677.90
2  41000.0  15886.45       2913.41  150177.83
3  83000.0  30020.03       6922.30  298246.34
```

You will then use this model to predict sales based on the values of the test features.

- Create X, an array containing values of all features in sales_df, and y, containing all values from the "sales" column.
- Instantiate a linear regression model.
- Fit the model to the training data.
- Create y_pred, making predictions for sales using the test features.
- Print the first two values of y_pred and y_test

```
Predictions: [53176.66154234 70996.19873235], Actual Values: [55261.28 67574.9 ]
```

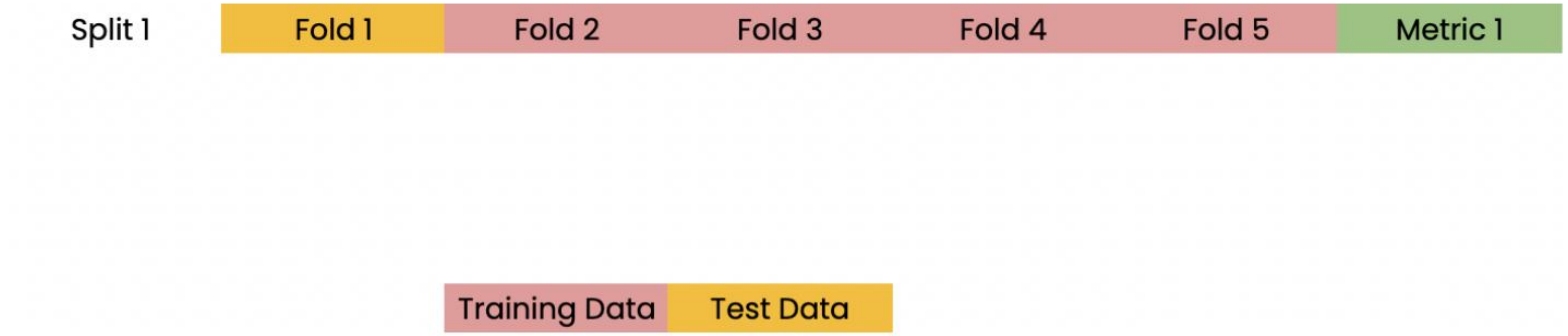
- Calculate the model's R-squared score by passing the test feature values and the test target values to an appropriate method.
- Calculate the model's root mean squared error using y_test and y_pred.
- Print r_squared and rmse.

```
R^2: 0.9990152104759367
RMSE: 2944.4331996000956
```

Cross-validation motivation

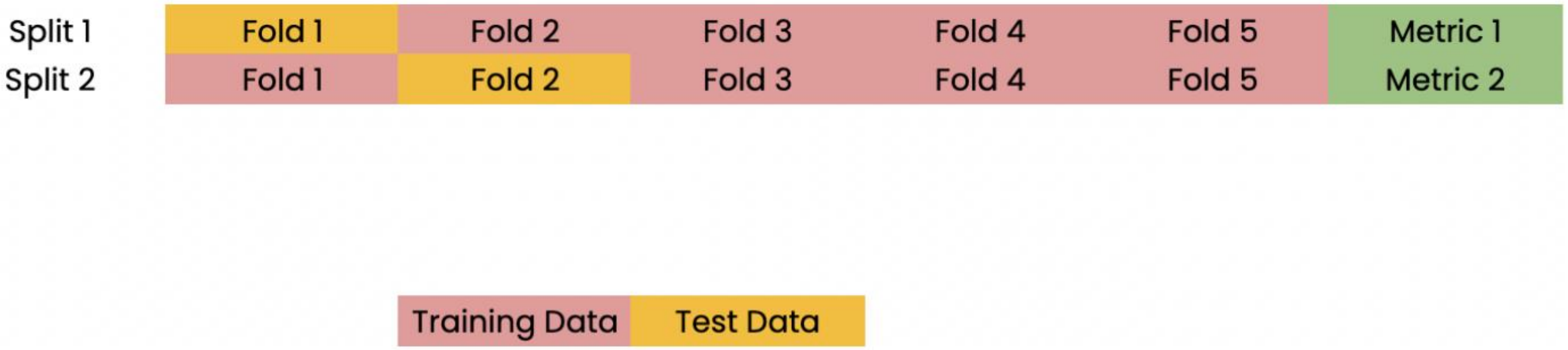
- Model performance is dependent on the way we split up the data
- Not representative of the model's ability to generalize to unseen data
- Solution: Cross-validation!

Cross-validation basics



We begin by splitting the dataset into five groups or folds. Then we set aside the first fold as a test set, fit our model on the remaining four folds, predict on our test set, and compute the metric of interest, such as R-squared.

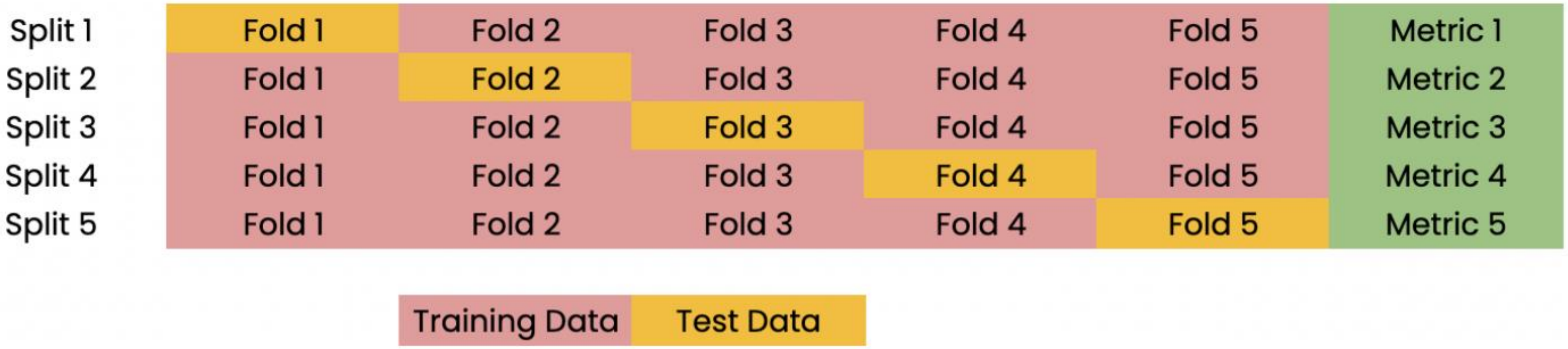
Cross-validation basics



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

🟥 Training Data 🟨 Test Data

Cross-validation basics



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

🟥 Training Data 🟨 Test Data

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

```python
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
```

Cross-validation for R-squared

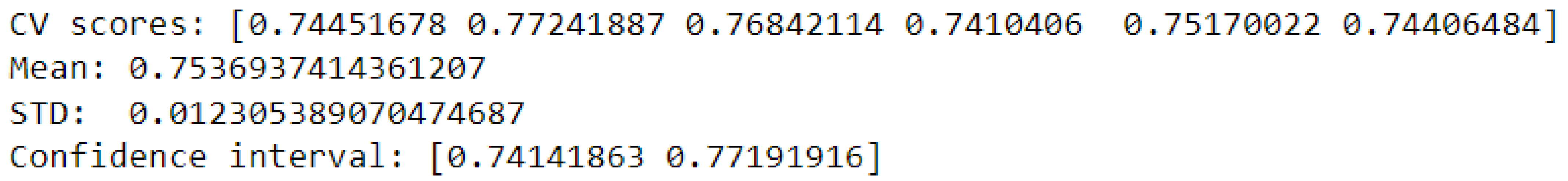
Cross-validation is a vital approach to evaluating a model. It maximizes the amount of data that is available to the model, as the model is not only trained but also tested on all of the available data.

In this exercise, you will build a linear regression model, then use 6-fold cross-validation to assess its accuracy for predicting sales using social media advertising expenditure. You will display the individual score for each of the six-folds.

The sales_df dataset shall be split into y for the target variable (sales), and X for the features (radio and social media).

Instructions

- Import KFold and cross_val_score.
- Create X and y according to the requirements above.
- Create kf by calling KFold(), setting the number of splits to six, shuffle to True, and setting a seed of 5.
- Perform cross-validation using reg on X and y, passing kf to cv.
- Print the cv_scores.
- Calculate and print the mean of the cv_scores results.
- Calculate and print the standard deviation of cv_scores.
- Display the 95% confidence interval for your results using np.quantile().



```
CV scores: [0.74451678 0.77241887 0.76842114 0.7410406  0.75170022 0.74406484]
Mean: 0.7536937414361207
STD:  0.012305389070474687
Confidence interval: [0.74141863 0.77191916]
```

Regularized regression

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

Lasso regression

- Loss function = OLS loss function +

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

- 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

Lasso regression in scikit-learn

```
from sklearn.linear_model import Lasso

diabetes_df = pd.read_csv('diabetes.csv', index_col = 0)
diabetes_df = diabetes_df[diabetes_df['bmi'] != 0]
diabetes_df = diabetes_df[diabetes_df['glucose'] != 0]
X = diabetes_df.drop('glucose', axis=1).values
y = diabetes_df['glucose'].values
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
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)
```

```
[0.35622500675820773, 0.3461828537090022, 0.201448239274153, 0.18595115472492285, 0.14542319216659483]
```

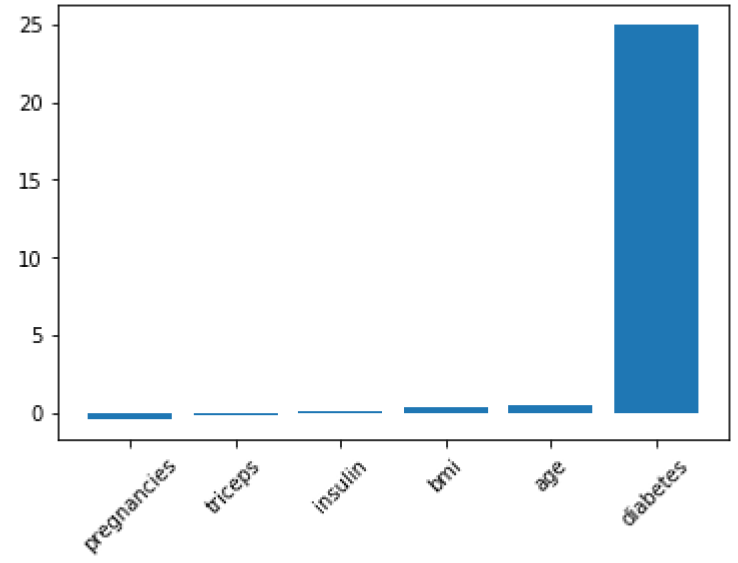
Lasso regression for feature selection

- 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

```
diabetes_df = pd.read_csv('diabetes.csv', index_col = 0)
diabetes_df = diabetes_df[diabetes_df['bmi'] != 0]
diabetes_df = diabetes_df[diabetes_df['glucose'] != 0]
X = diabetes_df.drop('glucose', axis=1).values
y = diabetes_df['glucose'].values


X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

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()
```

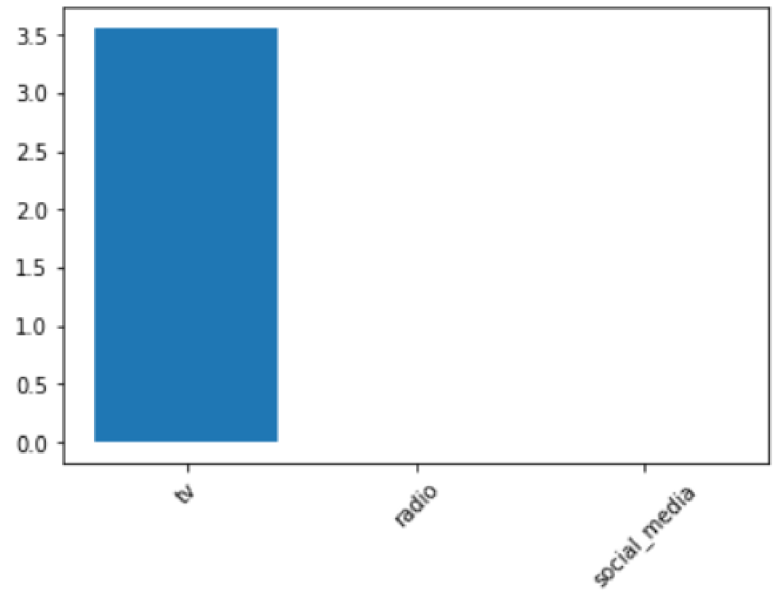


Lasso regression for feature importance
Earlier, you saw how lasso regression can be used to identify important features in a dataset.

In this exercise, you will fit a lasso regression model to the sales_df data and plot the model's coefficients.

- The feature variables (all columns except sales) and target variable (sales) arrays have to be created as X and y, along with sales_columns, which contains the dataset's feature names.
- Instantiate a Lasso regressor with an alpha of 0.3.
- Fit the model to the data.
- Compute the model's coefficients, storing as lasso_coef.

```
[ 3.56256962 -0.00397035  0.00496385]
```


tv
radio
social_media