

# Preprocessing data

SUPERVISED LEARNING WITH SCIKIT-LEARN



**George Boorman**

Core Curriculum Manager, 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	1
Anime	0
Blues	0
Classical	0
Country	0
Electronic	0
Hip-Hop	0
Jazz	0
Rap	0
Rock	0



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

We create binary features for each genre. As each song has one genre each row will have a 1 in one of the columns and 0's in the rest. If a song is not in the first 9 genres then implicitly it is a rock song. That means we implicitly need 9 features. So we can delete the rock column.

# 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

If we don't do this we are duplicating information which might be an issue for some models.

# Dealing with categorical features in Python

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

To create dummy variables the above 2 methods can be used.

# Music dataset

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

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

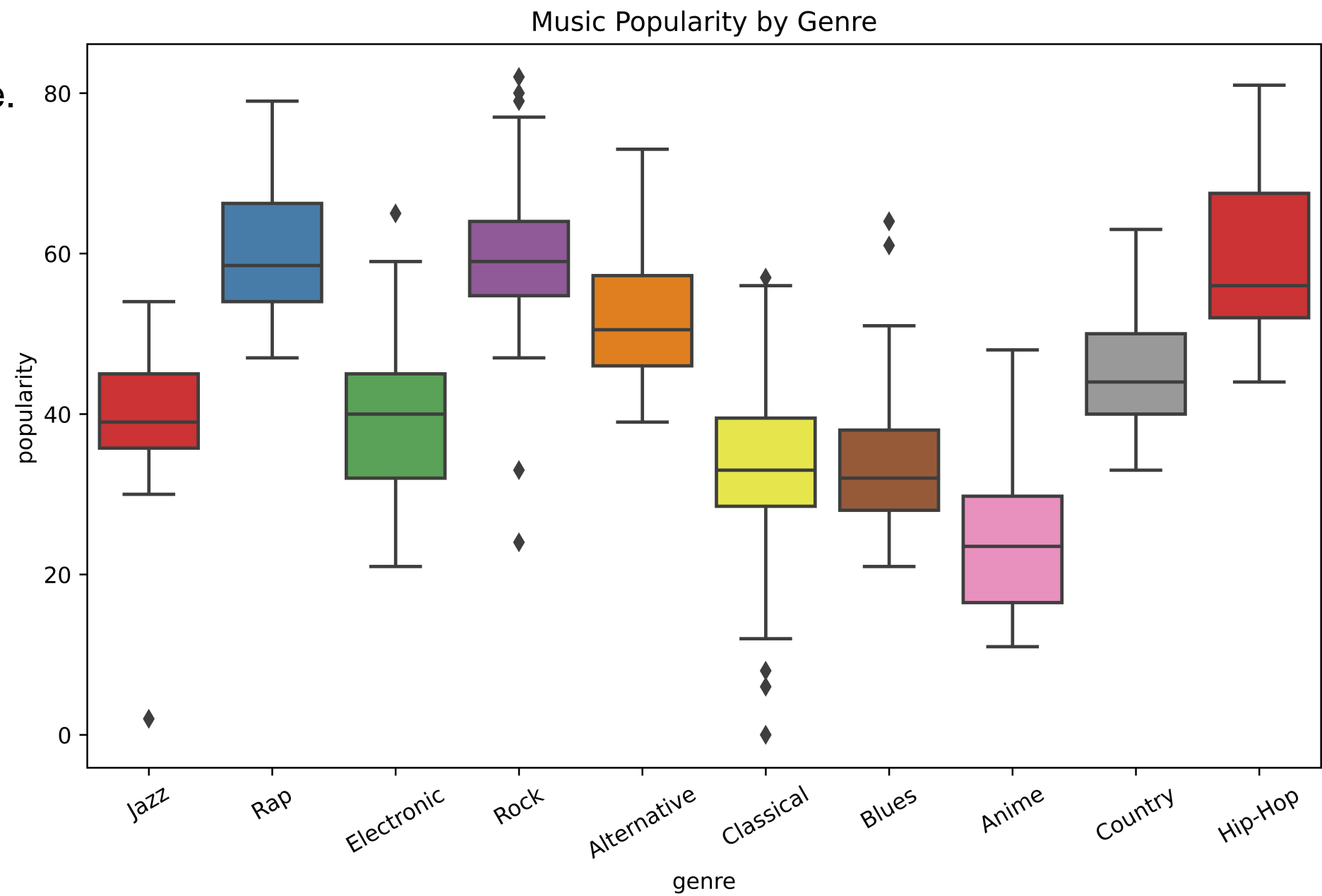
	popularity	acousticness	danceability	...	tempo	valence	genre
0	41.0	0.6440	0.823	...	102.619000	0.649	Jazz
1	62.0	0.0855	0.686	...	173.915000	0.636	Rap
2	42.0	0.2390	0.669	...	145.061000	0.494	Electronic
3	64.0	0.0125	0.522	...	120.406497	0.595	Rock
4	60.0	0.1210	0.780	...	96.056000	0.312	Rap

There is one categorical feature genre with 10 possible values.



# EDA w/ categorical feature

This box plot shows how popularity varies with genre. Lets encode this feature by using dummy variables.



# 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())
```

As we need only to keep 9 out of 10 binary features we can keep the `drop_first` argument to `true`.

	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)
```

We can see that pandas create 9 new binary features. To bring these binary features back into our original dataframe we can use `pd.concat` passing a list containing the `music_df` and our dummies dataframe and setting `axis=1`.

Lastly we can remove the original genre column using `dataframe.drop` passing the column and `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')
```

If the dataframe has only one categorical feature we can pass the entire dataframe thus skipping the step of combining variables. If we don't specify a column the new dataframe's binary columns will have the original feature names prefixed so they will start as `genre_` as shown above. The original genre column is automatically dropped. Once we have dummy variables we can fit models as before.

# 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))
```

We here call `cross_val_score` and we set `scoring = "neg_mean_squared_error"` which returns the negative MSE. This is because scikit-learn's cross validation metrics presume a higher score is better.

So MSE is changed to negative to counteract this.

We can get the training RMSE by taking the square root and converting into positive achieved by calling `numpy.sqrt` and passing our scores with a (-) sign in front.

```
[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

# 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

Here each feature is missing between 8 and 200 values.

# 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

A common approach is to remove missing observations accounting for 5% of all data. For this we use `.dropna` method, passing a list of columns with less than 5% missing values to the `subset` argument.

```
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
```

If there are missing values in our subset column the entire row is removed.

# 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*

Imputing missing data means making an educated guess as to what the missing values could be. We can impute the mean of all non-missing entries for a given feature. We can also use other values like the median.

We must split our data before imputing to avoid data leakage.

# Imputation with scikit-learn

Here is a workflow for imputation to predict song popularity.

As we use different imputation methods for categorical and numeric features we first split them storing as X\_cat and X\_num respectively along with our target array as y.

```
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)
```

We call `.fit_transform` to impute the training categorical features missing values. For the test categorical features we call `.transform`.

By using the same value for the `random_state` argument in splitting the target array's values remain unchanged. To impute missing categorical values we instantiate a simple imputer setting strategy as "most frequent". By default simple imputer expects `numpy.nan` to represent missing values.

# Imputation with scikit-learn

For our numerical data we import another imputer. By default it fills the values with the mean.

```
imp_num = SimpleImputer()  
X_train_num = imp_num.fit_transform(X_train_num) We fit and transform the training features and transform  
X_test_num = imp_num.transform(X_test_num) the test features.  
X_train = np.append(X_train_num, X_train_cat, axis=1) We then combine our training data using  
X_test = np.append(X_test_num, X_test_cat, axis=1) np.append passing our two arrays and setting  
axis = 1. We repeat this to our test data as well.
```

- Imputers are known as transformers  
Because of their ability to transform the data.

# Imputing within a pipeline

We can also impute using a pipeline which is an object used to run a series of transformations and build a model in a single workflow.

```
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
```

Here we perform binary classification to predict whether a song is rock or from another genre. We drop missing values accounting for less than 5% of our data. We convert the values in our genre column which will be our target to a 1 if rock, else 0, using `numpy.where`. We then create `X` and `y`.

# 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
```

To build a pipeline we construct a list of steps containing tuples where the step name is specified as strings and instantiate the transformer or model.

We pass this list when instantiating a pipeline. We then split our data and fit the pipeline to the training data, as with any other model. Finally we compute accuracy. Note that in a pipeline each step but the last must be a transformer.

# Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

# Centering and scaling

SUPERVISED LEARNING WITH SCIKIT-LEARN



**George Boorman**  
Core Curriculum Manager

Data imputation is one of the several important preprocessing steps for machine learning. Now lets cover another which is centering and scaling our data.



# 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

Here we use `df.describe` to checkout the ranges of some of our feature variables in our music dataset. We can see that the ranges vary widely. Duration ranges from 0 to 1.62mn. Speechiness contains only decimal places and loudness only have negative values.

# 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 To achieve this we can normalize or standardize our data often referred to as scaling or centering.
- Normalizing or standardizing (scaling and centering)

# How to scale our data

There are various ways 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    The normalized data have a minimum of 0 and maximum of 1.
- Can also *normalize* so the data ranges from -1 to +1    Here we center the data so that it ranges from -1 to +1
- See scikit-learn docs for further details

In this video we perform standardization but scikit learn has functions available for other types of scaling.

# Scaling in scikit-learn

To scale our features we import StandardScaler

After that we create our feature and target arrays.

```
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()
```

 Instantiation of a StandardScaler object.

```
X_train_scaled = scaler.fit_transform(X_train)
```

 Here we call it the fit\_transform method.

```
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))
```

Before scaling we split our data to avoid data leakage. We then instantiate a StandardScaler object and call it the fit\_transform method passing our training features. Next we use scaler.transform on the test features.

Looking at the mean and the SD of the columns of both the original and the scaled data verifies the changes taken place.

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

# Scaling in a pipeline

We can also put a scaler in a pipeline. Here we got a pipeline object to scale our data and we use a KNN model with 6 neighbors. We then split our data, fit the pipeline to our training set and predict on our test set.

```
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

You get an accuracy of 0.81. Lets compare this to using unscaled data.

# 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

Here we fit a KNN model to our unscaled training data and print the accuracy. It's only 0.53. So just by scaling the data we improved the accuracy by over 50%.

# CV and scaling in a pipeline

Lets also look at how we can use cross validation with a pipeline. We first, build our pipeline. We then specify our hyperparameter space by creating a dictionary.

```
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)
```

The key is the pipeline step name followed by a double underscore followed by the hyperparameter name. The corresponding value is a list or an array of the values to try for that particular hyperparameter. In this case we are tuning `n_neighbors` in the KNN model. Next we split our data into training and test sets.

We then perform a grid search over our parameters by instantiating the `GridSeachCV` object passing our pipeline and setting the `param_grid` argument equal to `parameters`. We then fit it to our training data. Lastly we make predictions using our test set.

# Checking model parameters

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

```
0.8199999999999999
```

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

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

Printing GridSearchCV's best score attribute we see that the score is slightly better than our previous model's performance. Printing the best parameters the optimal model has 12 neighbors.

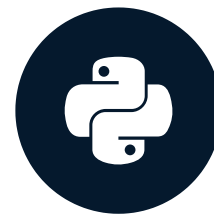


# Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

# Evaluating multiple models

SUPERVISED LEARNING WITH SCIKIT-LEARN



**George Boorman**  
Core Curriculum Manager, DataCamp

We've covered all parts of the supervised learning workflow. But how do we decide which model to use in the first place? This is a complex question and the answer depends on our situation.

# Different models for different problems

**Some guiding principles** These are the guiding principles in making this decision.

- **Size of the dataset** Models such as artificial neural networks require a lot of data to perform well.
  - Fewer features = simpler model, faster training time
  - Some models require large amounts of data to perform well
- **Interpretability** We may need an interpretable model so we can explain to stakeholders how predictions were made.
  - Some models are easier to explain, which can be important for stakeholders
  - Linear regression has high interpretability, as we can understand the coefficients
- **Flexibility** In linear regression we can calculate and interpret the model coefficients. Alternately flexibility might be important to get the most accurate predictions.
  - May improve accuracy, by making fewer assumptions about data
  - KNN is a more flexible model, doesn't assume any linear relationships

Generally flexible models make fewer assumptions about the data. For example a KNN model doesn't assume a linear relationship between the features and the target.

# It's all in the metrics

Scikit learn allows the same models to be used for most models. This makes it easy to compare them.

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

Therefore one approach is to select several models and a metric and then evaluate their performance without any form of hyperparameter tuning.

# A note on scaling

- Models affected by scaling: These models are affected by the scaling of our data.
  - KNN
  - Linear Regression (plus Ridge, Lasso)
  - Logistic Regression
  - Artificial Neural Network

Therefore it is generally best to scale our data before evaluating models out of the box.

- Best to scale our data before evaluating models

# Evaluating classification models

We will evaluate 3 models for binary classification of sound genre: KNN, logistic regression, and a new model called the Decision Tree classifier.

```
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)
```

As usual we create our feature and target arrays, and then split our data. We then scale our features using the `scaler.fit_transform` method on the training set.

And the `.transform` method on the test set.

# 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()
```

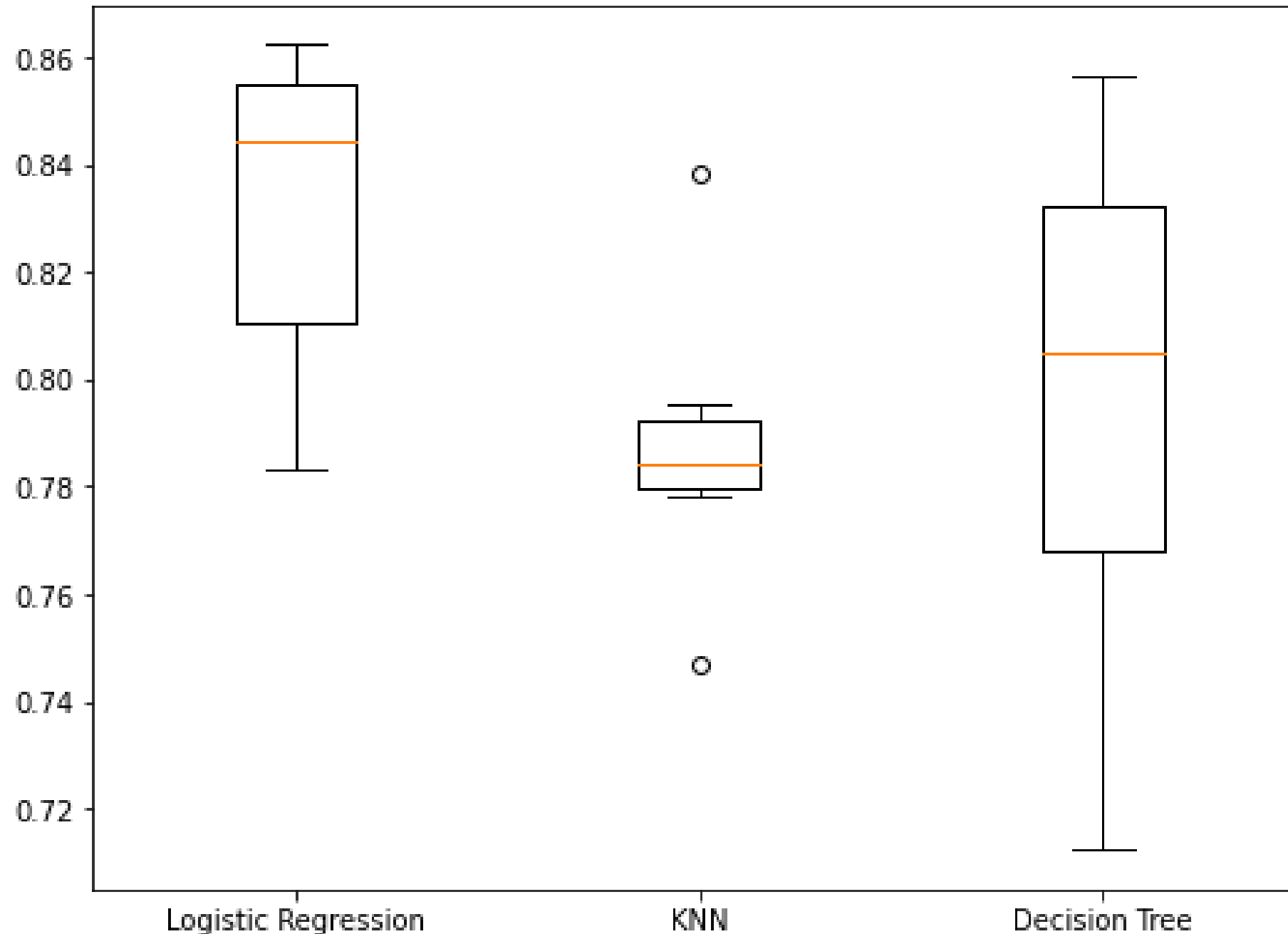
We create a dictionary with our model names as strings for the keys and instantiate models as the dictionary's values. We also create an empty list to store the results. Now we loop through the models in our models dictionary using `.values` methods. Inside the loop we instantiate a `KFold` object. Next we perform cross validation using the model being iterated along with our scaled training features and target training array. By default the scoring here is accuracy. We then append the cross validation results to our results list. Lastly outside the loop we create a boxplot of our results and set the labels argument equal to a call of `models.keys()` to retrieve each model's names.

# Visualizing results

The output shows us the range of cross validation accuracy scores.

We can also see each model's median cross validation score represented by the orange line in each box.

We can see logistic regression as the best median score.





# 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
```

To evaluate on the test set we loop through the names and values of the dictionary using the `.items` method. Inside the loop we fit the model, calculate accuracy, and print it. Logistic regression performs best for this problem if we are using accuracy as the metric.

# Let's practice!

SUPERVISED LEARNING WITH SCIKIT-LEARN

# Congratulations

SUPERVISED LEARNING WITH SCIKIT-LEARN



**George Boorman**

Core Curriculum Manager, 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