Machine Learning with Scikit-learn part 2

How good is your model?

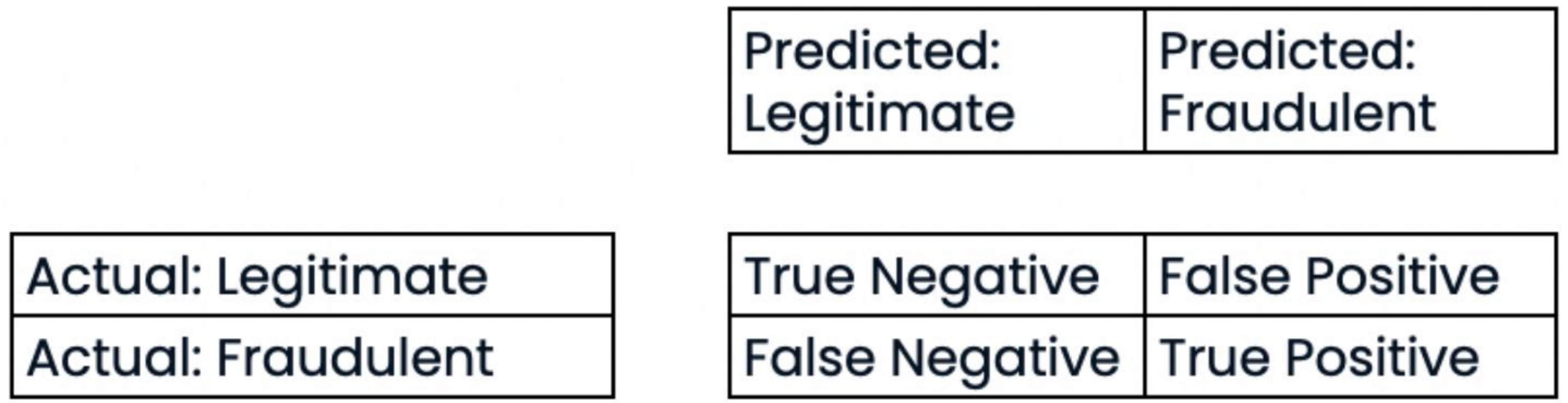
- Classification metrics
  - Measuring model performance with accuracy:
    - Fraction of correctly classified samples
    - Not always a useful metric

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

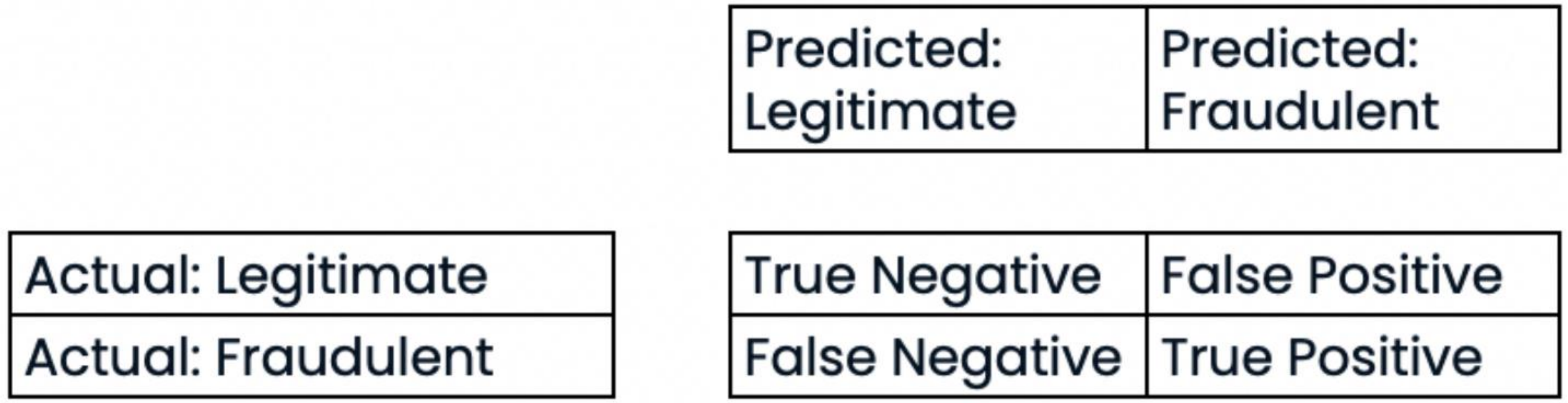
The false negatives are the number of legitimate transactions incorrectly labeled

The false positives are the number of transactions incorrectly labeled as fraudulent.

The true negatives are the number of legitimate transactions correctly labeled

The true positives are the number of fraudulent transactions correctly labeled

Assessing classification performance



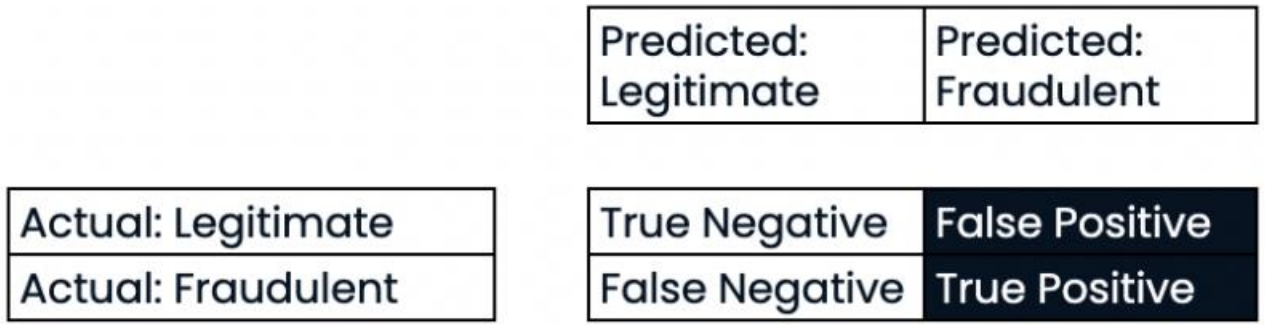
	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

- Accuracy:

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

Firstly, we can retrieve accuracy: it's the sum of true predictions divided by the total sum of the matrix.

Precision



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

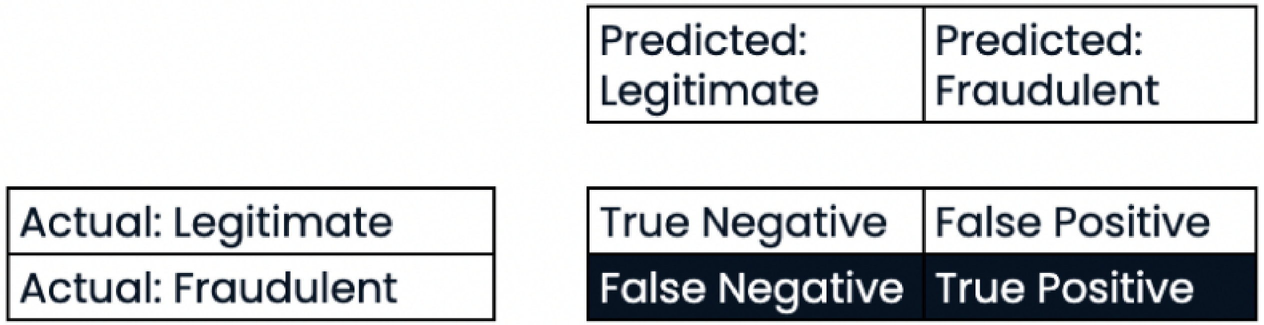
- Precision

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

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

Usually, the class of interest is called the positive class. As we aim to detect fraud, the positive class is an illegitimate transaction. So why is the confusion matrix important? There are other important metrics we can calculate from the confusion matrix. Precision is the number of true positives divided by the sum of all positive predictions. It is also called the positive predictive value. In our case, this is the number of correctly labeled fraudulent transactions divided by the total number of transactions classified as fraudulent. High precision means having a lower false positive rate.

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

Recall is the number of true positives divided by the sum of true positives and false negatives. This is also called sensitivity. High recall reflects a lower false negative rate. For our classifier, it means predicting most fraudulent transactions correctly.

F1 score

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

The F1-score is the harmonic mean of precision and recall. This metric gives equal weight to precision and recall, therefore it factors in both the number of errors made by the model and the type of errors. The F1 score favors models with similar precision and recall, and is a useful metric if we are seeking a model which performs reasonably well across both metrics.

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)
```

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

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

True Negative	False Positive
False Negative	True Positive

Classification report in scikit-learn

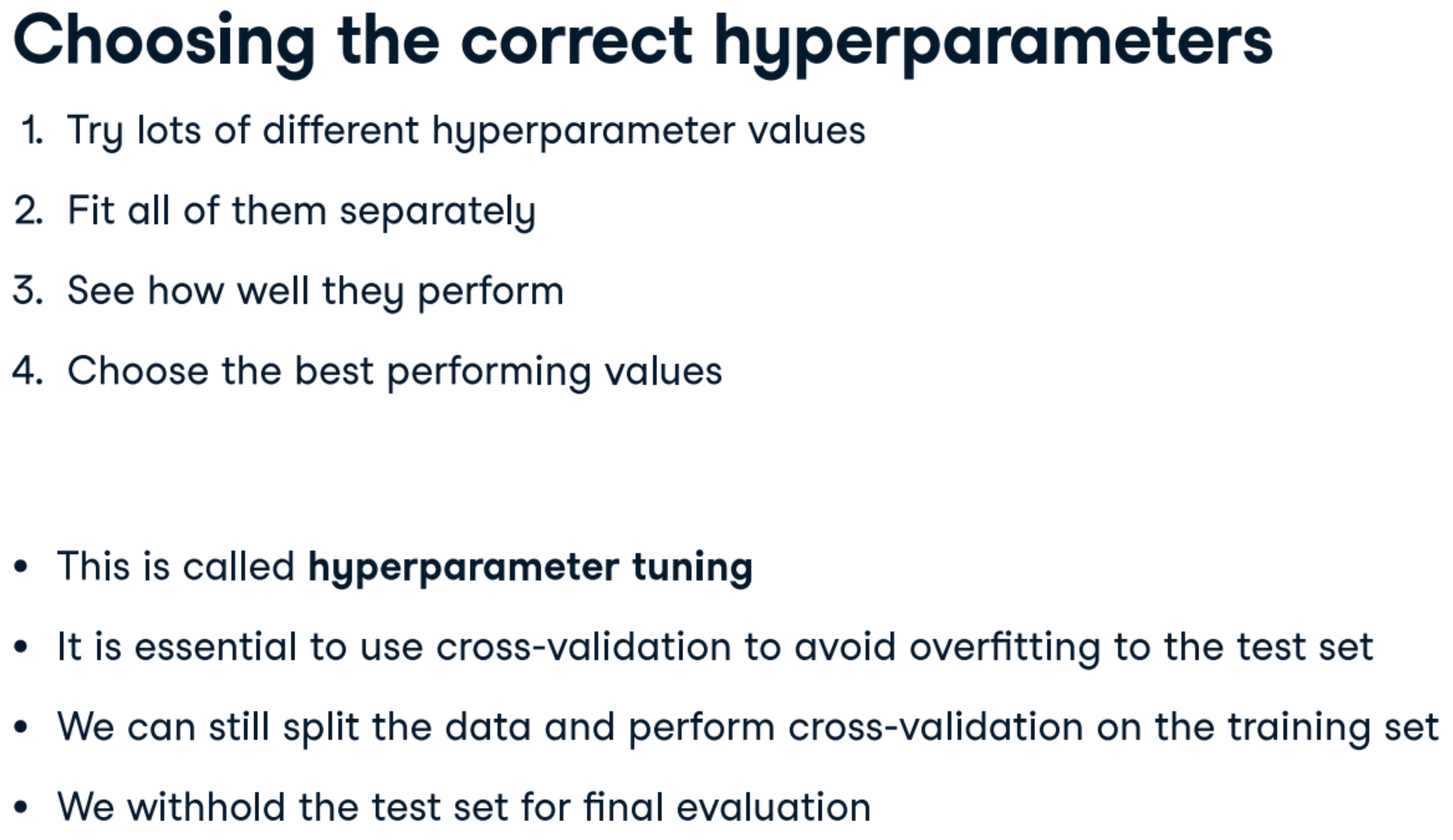
```python
print(classification_report(y_test, y_pred))
```

```
              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
```

This report includes precision and recall by class (churn: 1 or no churn: 0), point-seven-six and point-one-six for the churn class respectively, which highlights how poorly the model's recall is on the churn class. Support represents the number of instances for each class within the true labels. (Class imbalance)

Hyperparameter tuning

- Lasso regression: choosing alpha
- KNN: choosing n_neighbors
- Hyperparameters: parameters we specify before fitting the model

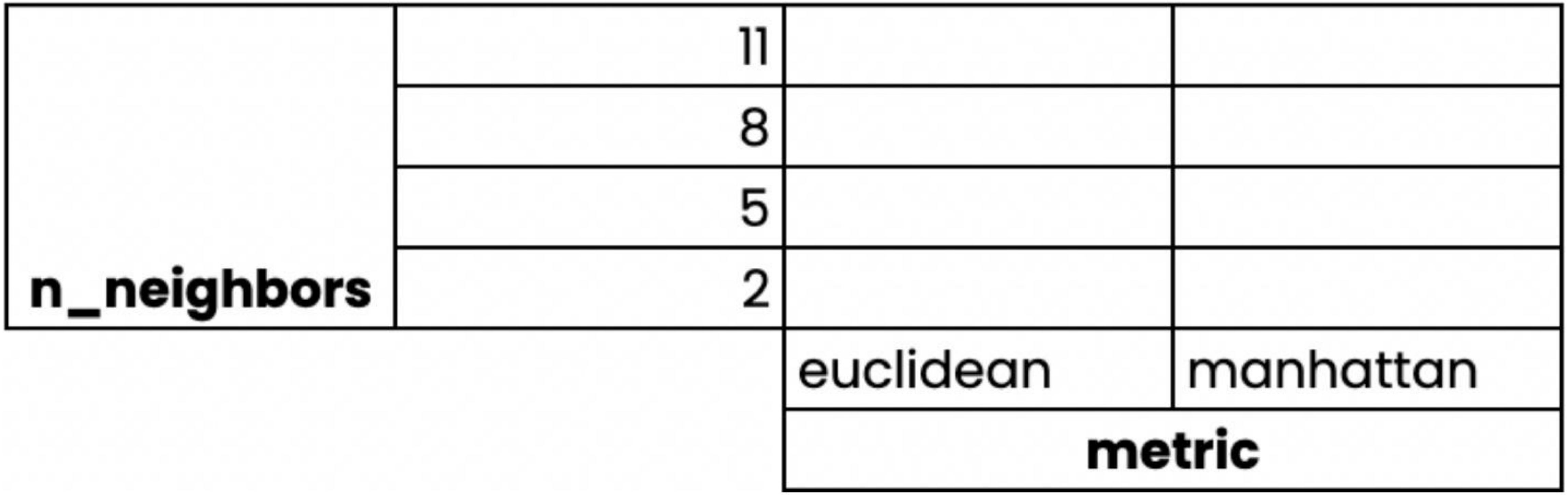


Choosing the correct hyperparameters

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



	11		
	8		
	5		
n_neighbors	2		
		euclidean	manhattan
		metric	

We perform k-fold cross-validation for each combination of hyperparameters. The mean scores for each combination are shown here. By default, KNN uses Euclidean distance but Manhattan distance can be selected by setting p = 1.

For example, **`.KNeighborsClassifier`**(*n_neighbors=5*, *p=1*)

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	

Preprocessing Data

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

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

- pandas: `get_dummies()`

Music dataset

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

```python
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
```

We will be working with a music dataset in this chapter, for both classification and regression problems. Initially, we will build a regression model using all features in the dataset to predict song popularity. There is one categorical feature, genre, with ten possible values.

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)
```

Linear regression with dummy variables

```python
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]
```

```
music_df = pd.read_csv('music.csv', index_col = 0)
music_dummies = pd.get_dummies(music_df['genre'], drop_first=True)

#music_dummies.head()
music_dummies = pd.concat([music_df, music_dummies], axis = 1)
music_dummies = music_dummies.drop('genre', axis=1)
#music_dummies.head()
print(music_dummies.columns)

#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')

linreg_cv2 = cross_val_score(linreg, X_train, y_train, cv=kf)
print(np.sqrt(-linreg_cv))
print(linreg_cv2)
```

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_df = pd.read_csv('music_unclean.csv', index_col = 0)

```
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

```python
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
```

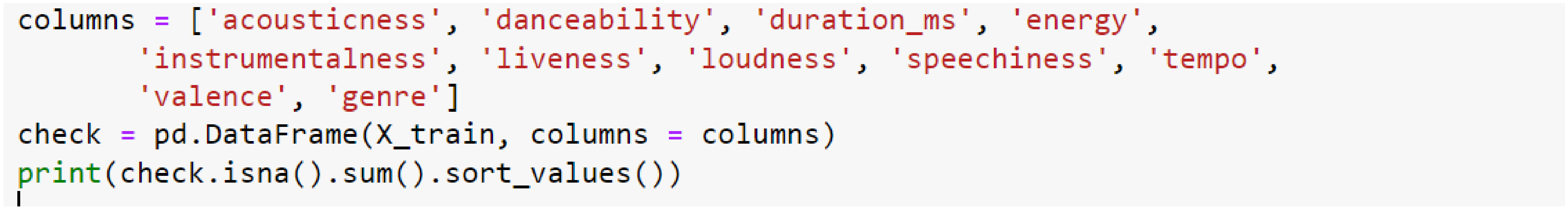
Imputing values

- 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

```python
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)
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)
```

```python
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred = knn.predict(X_test)
print(knn.score(X_test, y_test))
```



```python
columns = ['acousticness', 'danceability', 'duration_ms', 'energy',
       'instrumentalness', 'liveness', 'loudness', 'speechiness', 'tempo',
       'valence', 'genre']
check = pd.DataFrame(X_train, columns = columns)
print(check.isna().sum().sort_values())
```

Imputing within a pipeline

```python
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
```

```python
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)
```

```
from sklearn.pipeline import Pipeline
from sklearn.linear_model import LogisticRegression
music_df = pd.read_csv('music_unclean.csv', index_col =
0)
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
X_train, X_test, y_train, y_test =
train_test_split(X,y,test_size=0.3,random_state=42)

steps = [('imputation', SimpleImputer()),
        ('Log_reg', LogisticRegression())]
pipeline = Pipeline(steps)
pipeline.fit(X_train, y_train)
y_pred = pipeline.predict(X_test)
print(confusion_matrix(y_test, y_pred))
print(pipeline.score(X_test,y_test))
```

Handling Missing Data, Imputing Data and Pipelining

You are going to tidy the music_df dataset. You will create a pipeline to impute missing values and build a KNN classifier model, then use it to predict whether a song is of the "Rock" genre.

In this exercise specifically, you will drop missing values accounting for less than 5% of the dataset, and convert the "genre" column into a binary feature.

Instructions:

- Load the music_df from music_unclean.csv and print the number of missing values for each column in the music_df dataset, sorted in ascending order.
- Remove values for all columns with 50 or fewer missing values.
- Convert music_df["genre"] to values of 1 if the row contains "Rock", otherwise change the value to 0.
- X = all columns except genre
- y = genre
- Split data with test_size = 30 and random state set to 42
- Instantiate an imputer.
- Instantiate a KNN classifier with three neighbors.
- Create steps, a list of tuples containing the imputer variable you created, called "imputer", followed by the knn model you created, called "knn".
- Create a pipeline using the steps you previously defined.
- Fit the pipeline to the training data.
- Make predictions on the test set.
- Calculate and print the confusion matrix.

Repeat the previous task again without using pipeline.