Supervised learning with scikit-learn
by datacamp

Unsupervised learning
-uncovering hidden patterns from unlabeled data
-clustering is one branch of unsupervised learning

Supervised learning
-values to be predicted are already known
-model is built with aim of accurately predicting values of previously unseen data
-learns from a labeled dataset where the input data is accompanied by corresponding outputs (ie target labels)
-the model attempts to learn a mapping between the input features and target labels, so later it can make predictions on unseen data

Types of supervised learning
-classification > target variable consists of categories
-example a binary classification (fraudulent or non-fraudulent
-regression > target variable is continuous
-used to predict continuous values
-example prediciting price of property using features like size and/or number of bedrooms

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 DF or NumPy array
*this all requires exploratory data analysis (EDA) first

scikit-learn syntax
#import ‘Model’ which is a type of algorithm for our supervised learning problem
#example model, k-Nearest Neighbors model uses distance between obsevations to predict labels or values
from sklearn.module import Model
#create a variable, here we will name it ‘model’
#instantiate the Model

#instantiate means creating a model from a predefined blueprint or template
#also referred to as creating an object or an instance
#in this case we are creating a specific machine learning model class
model = Model()
#fit the model
model.fit(X, y)
#X is an array of our features
#y is an arrary of our target labels
#here we will use ‘X_new’ an example array of six elements
predictions = model.predict(X_new)
print(predictions)
output > array([0, 0, 0, 0, 1, 0]) #example could be an email spam classifier > this would be saying element 5 (index 4) is spam

How to build a classification model (also called a classifier) to predict the labels of unseen data
Four steps:

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

*as the classifier leanrs from the labeled data, we call this the training data
labeled data = training data

k-Nearest Neighbors (KNN)
“lazy learner” algorithm
popular for classification problems
predicts label of a data point by majority voting
makes prediction based on what label the majority of nearest neighbors have
k is a pre-determined value
too small can lead to noisy predictions
too large and can smooth out decision boundaries and result in biased predictions
k represents the amount of voting neighbors when making a prediction on an unseen new data point
example if set k=3
then a new data point is classified by the Euclidean distance of its three nearest neighbors
new point takes on the label of the majority of nearest neighbors have
mostly used for classification but can be used for regression if algorithm takes the weighted average of the target values

KNN example - customer churn eval between day charge vs night charge customers

```
from sklearn.neighbors import KNeighborsClassifier
#split data into X, 2D array of our features and y, 1D array of our target values
X = churn_df[['total_day_charge', 'total_eve_charge']].values
y = churn_df['churn'].values #churn Series perfect for binary classification, values already of 0 and 1
*scikit-learn requires that the features are in an array where each column is a feature and each row is an observation
*similarly the target needs to be a single column with the same number of observations as the feature data
*using the .values attribute converts both X and y into NumPy arrays
good practice to print the shapes of X and y to ensure same size
print(X.shape, y.shape)
#instantiate our KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=15)
#fit the model
#'fitting' is training the model of the labeled data
knn.fit(X,y)
predctions = knn.predict(X_new)
print('Predictions: {}'.format(predictions))
```

Measuring model performance
*in classification, accuracy is a commonly used metric
accuracy = correct predictions / total observations
measuring accuracy on the labeled data will not be indicative of the models skill on unseen data
best practice > split data into a training and test set
fit classifier on training set
then calculate accuracy using test set

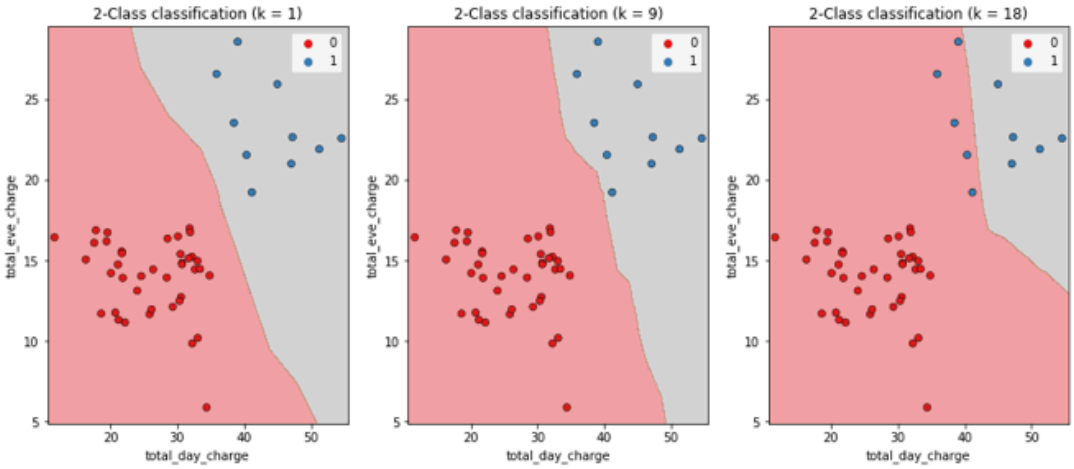
example

```
from sklearn.model_selection import train_test_split
#common to use 20-30% of data as test set
#0.3 is 30% for our example
**best practice to ensure our split reflects the proportion of labels in our data
**what this mean > if churn occurs in 10% of observations, then we want 10% of labels in our training and test sets to churn
#achieve this by using the 'stratify' argument and setting it equal to y
#test_train_split returns four arrays > training data, test data, training labels, and test labels > set as X_train, X_test, y_train, y_test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=21, stratify=y)
#instantiate a KNN model
knn = KNeighborsClassifier(n_neighbors=6)
```

#train model
knn.fit(X_train, y_train)
#check accuracy > do this with the .score() method passing the X_test and y_test arguments
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



Can also interpret k using a model complexity curve
with a KNN model, we can calculate accuracy on the training and test sets using incremental k values
create empty dictionaries to store our train and test accuracies
train_accuracies = {}
test_accuracies = {}
create an array containing the range of k values
neighbors = np.arange(1, 26)
use a for loop to repeat our previous workflow
building several models using a different number of neighbors
for neighbor in neighbors:
loop through our neighbors array
inside the loop, instantiate a KNN model with n_neighbors equal to the neighbor iterator
    knn = KNeighborsClassifier(n_neighbors=neighbor)
fit to the training data
    knn.fit(X_train, y_train)
calculate training and test set accuracy
then store in their respective dictionaries

```
        train_accuracies[neighbor] = knn.score(X_train, y_train)
        test_accuracies[neighbor] = knn.score(X_test, y_test)
then plot 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()
```

Introduction to regression
target variable has continuous values

Example - women's health data to predict blood glucose levels
*dataset > Series diabetes status [0 for no, 1 for yes]
create feature and target arrays
**use all the features in the dataset except our target ('blood glucose levels')
X = diabetes_df.drop('glucose', axis=1).values
for y take the target column's values attribute
y = diabetes_df['glucose'].values
confirm X and y are NumPy arrays
print(type(X), type(y))

Same example - Attempt to make prediction from one feature (BMI)
slice out BMI column
X_bmi == X[:, 3]
check shapes of y and X_bmi
print(y.shape, X_bmi.shape)
output > shows that both shapes are 1D arrays
**this is ok for y, but X must be a 2D array
change X to a 2D array
***sidebar - Why does X need to be a 2D array?
In supervised learning models, the input features (often denoted as X) are typically represented as a 2D array (also known as a matrix) because it allows for the efficient handling of multiple data points and their corresponding feature values. There are several reasons why X is represented as a 2D array:

1. **Multiple Data Points:**
   In supervised learning, you typically have multiple data points (samples) in your dataset. Each data point consists of a set of features (attributes) that represent the input to the model. A 2D array allows you to organize and represent these

multiple data points efficiently.

2. **Consistent Data Structure:**
   Representing X as a 2D array ensures a consistent data structure, where each row corresponds to a single data point, and each column represents a specific feature. This makes it easier to perform operations and computations on the entire dataset.

3. **Vectorized Operations:**
   Many machine learning algorithms are optimized to perform vectorized operations, which can be efficiently executed on 2D arrays. Vectorized operations allow the model to process multiple data points simultaneously, leading to faster computations and better performance.

4. **Integration with Libraries:**
   Popular machine learning libraries, such as NumPy, scikit-learn, and TensorFlow, are designed to work with 2D arrays. These libraries provide numerous built-in functions and methods that operate on 2D arrays, making it easier to implement and train supervised learning models.

5. **Input Requirements for Models:**
   Many supervised learning models, such as linear regression, decision trees, and neural networks, expect the input features to be represented as a 2D array. This format allows the models to access and process individual features for each data point during training and prediction.

6. **Multivariate Features:**
   Often, the input features (X) in supervised learning are multivariate, meaning each data point consists of multiple feature values. Representing X as a 2D array allows you to handle these multivariate features in a structured manner.

While representing X as a 2D array is common in many supervised learning scenarios, it is worth noting that some algorithms or models may have specific requirements regarding the input data shape. For example, time series data may require a different structure (e.g., 3D array) to capture the temporal dependencies.

In summary, representing X as a 2D array in supervised learning provides an efficient and consistent way to organize and process the input features for multiple data points, making it easier to implement and train various machine learning models.

**Return to example

changing X_bmi from 1D array to 2D array
X_bmi = X_bmi.reshape(-1, 1)
now plot
plt.scatter(X_bmi, y)
plt.ylabel('Blood Glucose (mg/dl)')
plt.xlabel('Body Mass Index')
plt.show()

generally shows us that as bmi increases blood glucose levels also tend to increase
now fit a regression model to our data
we'll use linear regression which fits a straight line to our data
from sklearn.linear_model import LinearRegression
#instantiate our regression model
reg = LinearRegression()
**since we are modeling the relationship between bmi and glucose rather than predicting target values for new observations,
we fit the model to all of our feature observations
*just as we did for our classification problem
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()

How does linear regression work?
Regression mechanics
the concept is to fit a line to the data
in 2 dimensions this takes the form of y = ax + b
in simple linear regression (ie using only one feature)
y = target
x = single feature
a, b = parameters/coefficients of the model
a and b are the model parameters that we want to learn
a and b are also called the model coefficients
a is also called the slope
y is also called the intercept

How do we choose a and b?
We can define an error function for any given line and then choose the line that minimizes this function
Error function has many names
error function = loss function = cost function

Loss function
goal is to have line as close to the observations as possible
ie minimize the vertical distance between the fit and the data
to do this we calculate the vertical distance between each observation and the line
this distance is called the residual
goal is to minimize the sum of the residuals
on initial glance this is not possible because the positive and negative residuals would cancel each other out
we get around this by squaring all residuals and in turn eliminating all negatives
adding all the squared residuals, we calculate the residual sum of squares (RSS)
this type of linear regression is called ordinary least squares (OLS)
aim is to minimize the RSS

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

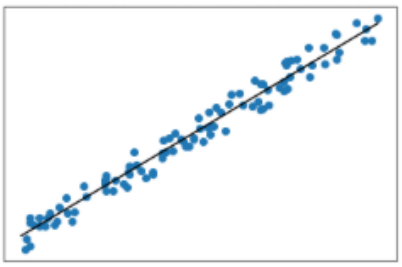
Linear regression in higher dimensions
two features and one target takes this form
y = a1x1 + a2x2 + b
to fit a linear regression model we specify three variables a1, a2, and b (the intercept)
adding more features is known as multiple linear regressions
fittin a multiple linear regression model means specifying a coefficient (a) for each of the features
for multiple linear regression, scikit expects one variable for each feature and one for target values (need to pass two arrays)
y = a1x1 + a2x2 + a3x3 + ... + anxn + b

Example - predict blood glucose levels using all of the features from the dataset
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)
**note the scikit linear regression model performs OLS under the hood

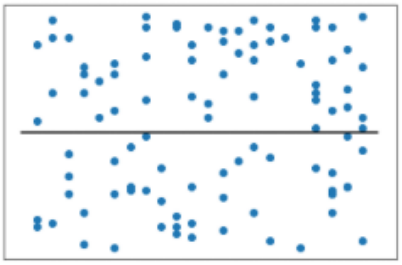
R-squared
default metric of linear regression
quantifies the amount of variance in the target variable that is explained by the

features
values range from 0 to 1
1 meaning the features completely explain the target's variance

High R^2:



- Low R^2:



to compute R-squared
reg_all.score(X_test, y_test)

Mean squared error and root mean squared error
another way to assess a regression model's performance is to take the mean of the residual sum of squares
measured in target value units squared

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

we use RMSE to measure in the same units as the target value

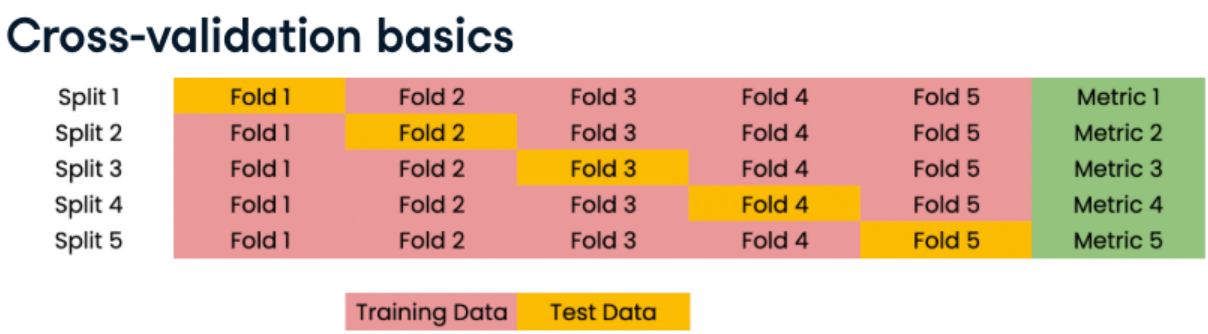
$$RMSE = \sqrt{MSE}$$

from sklearn.metrics import mean_squared_error
mean_squared_error(y_test, y_pred, squared=False)
'squared' argument to False, returns the square root of the MSE

Cross-validation motivation
R-squared returned is dependent on the way that we split up the data
test set since random may not be representative of the model's ability to generalize to unseen data
to combat this dependence on what is essentially a random split, we use a technique called cross-validation

Cross-validation basics
split data into 5 groups called 'folds' and label them fold 1, fold 2, ...

set fold 1 as test set and fit our model on the remaining four folds, predict on our test set
then compute metric of interest (like R-squared)
**repeat process on each fold group
at the end we'll have 5 values of interest (ie 5 R-squareds)



**can use any desired amount of folds
5 folds = 5-fold CV
10 folds = 10-fold CV
k folds = k-fold CV
*however always a trade-off > more folds = higher computational expense (because we are fitting/predicting multiple times)

from sklearn.model_selection import cross_val_score, KFold
KFold allows us to set a seed and shuffle our data, making our results repeatable downstream
kf = KFold(n_splits=6, shuffle=True, random_state=42)
'n_splits' argument default is 5, for our example we've chosen 6
will give us six folds
'shuffle' argument set to True will shuffle our dataset before splitting into folds
instantiate the model
reg = LinearRegression()
cv_results = cross_val_score(reg, X, y, cv=kf)
*the first three positional arguments (the model, the feature data, the target data) need to specify folds here as well
we did this in kf, so here we set cv=kf
**length of the array is the number of folds utilized
default score for linear regression is R-squared
print(cv_results)
output > 6 r-squares
can then run summary statistics on this array
print(np.mean(cv_results), np.std(cv_results))
calculate CI interval as well
print(np.quantile(cv_results, [0.025, 0.975]))

Regularized regession
regularization is a technique used to avoid overfitting
fitting a linear regression model minimizes a loss function to chooses a coefficient (a) for each feature and the intercept (b)
large coefficients can lead to overfitting
common practice to alther the loss function so that it penalizes large coefficients
this is what regularization does

Ridge regression is a type of regularized regression
loss function = OLS loss function + alpha * sum of the squared value of each coefficient
*when minimizing the loss funciton, models are penalized for coefficients with large positive or negative values
*need to choose the alpha in order to fit and predict
essentially we can select the alpha for which our model perfomrs best
similar to picking k in KNN
alpha is known as a hyperparameter, ie a variable used for selecting a model's parameters
alpha controls model complexity
**when alpha is 0, we are performing OLS, where large coefficients are not penalized and overfitting may occur
a high alpha means the large coefficients are significantly penalized; this can lead to underfitting

from sklearn.linear_model import Ridge
to highlight the impact of different alpha values, we create an empty list for our scores, then loop through a list of different alpha values
scores = [ ]
for alpha in [0.1, 1.0, 10.0, 100.0, 1000.0]:
#intstantiate Ridge, setting the 'alpha' argument to the iterator (also called alpha in this example)
    ridge = Ridge(alpha=alpha)
#train the data
    ridge.fit(X_train, y_train)
#predict
    y_pred = ridge.predict(X_test)
#save the model's R-squared value to the scores lis
    scores.append(ridge.score(X_test, y_test))
print(scores)
output > in our example we see .28>>>.19, performance gets worse as alpha increases

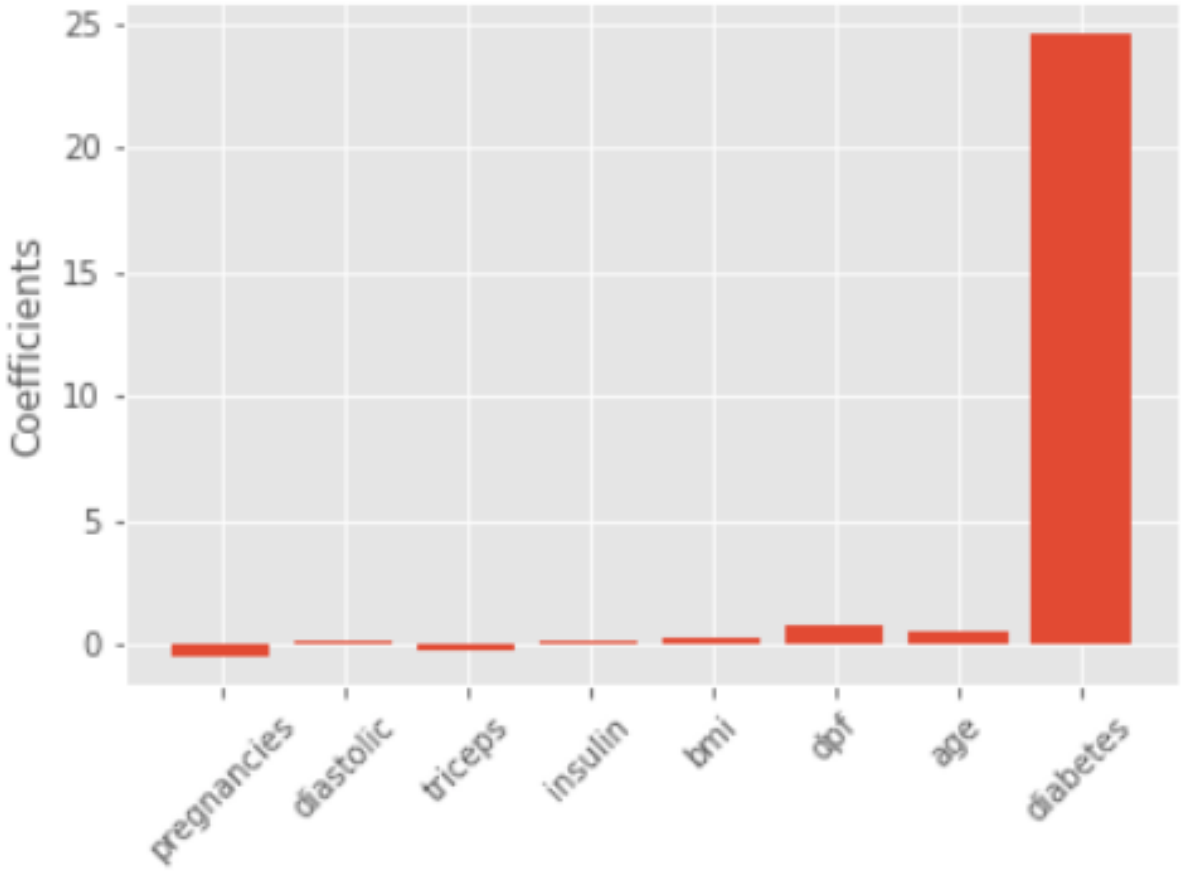
Lasso regression

another type of regularized regression
loss function = OLS loss function + alpha * sum of absolute value of each coefficient

example - similar to Ridge workflow
```
from sklearn.linear_model import Lasso
scores = [ ]
for alpha in {0.01, 1.0, 10.0, 20.0, 50.0]:
#instantiate Lasso, and setting alpha argument to iterator (in this case also called 'alpha')
        lasso = Lasso(alpha=alpha)
        lasso.fit(X_train, y_train)
        lasso_pred = lasso.predict(X_test)
#save the model's values and append them to the 'scores' list
        scores.append(lasso.score(X_test, y_test))
print(scores)
```

Lasso can be used to assess feature importance
**this is because it tends to shrink teh coefficients of less important features to zero
the features whose coefficients are not shrunk to zero are selected by the lasso algorithm

Example - Lasso for feature selection in scikit
```
from sklearn.linear_model import Lasso
X = diabetes_df.drop('glucose', axis=1).values
#reminder 'axis' argument states if we are dropping rows (axis=0) or columns (axis=1)
y = diabetes_df['glucose'].values
names = diabetes_df.drop('glucose', axis=1).columns
#used .columns attribute to access the feature names and store them as variable names
lasso = Lasso(alpha=0.1)
#alpha argument within the instantitation
lasso_coef = lasso.fit(X, y).coef__
#extract the coefficients using the .coef_ attribute
plt.bar(names, lasso_coef)
plt.xticks(rotation=45)
plt.show()
```



not surprising but shows us that a diabetes diagnosis is the most important predictor of our target variable
graph works as a great sanity check

Another example

```
# Import Lasso
from sklearn.linear_model import Lasso

# Instantiate a lasso regression model
lasso = Lasso(alpha=0.3)

# Fit the model to the data
lasso.fit(X, y)

# Compute and print the coefficients
lasso_coef = lasso.fit(X, y).coef_
print(lasso_coef)
plt.bar(sales_columns, lasso_coef)
plt.xticks(rotation=45)
```

plt.show()

How good is your model?
Classification metrics
-measuring model performance with accuracy
accuracy metric has its shortcomings
example - fraudulent bank transactions where presumed 99% are legitmate and 1% are fraudulent
we could build a classifier that predicts all transactions are legitmate
this model's accuracy would be 99%!
but it would be useless in determining fraudulent transactions
**this is called class imbalance (ie uneven frequency of classes)

Confusion matrix for assessing a binary classification performance
this is a 2x2 matrix

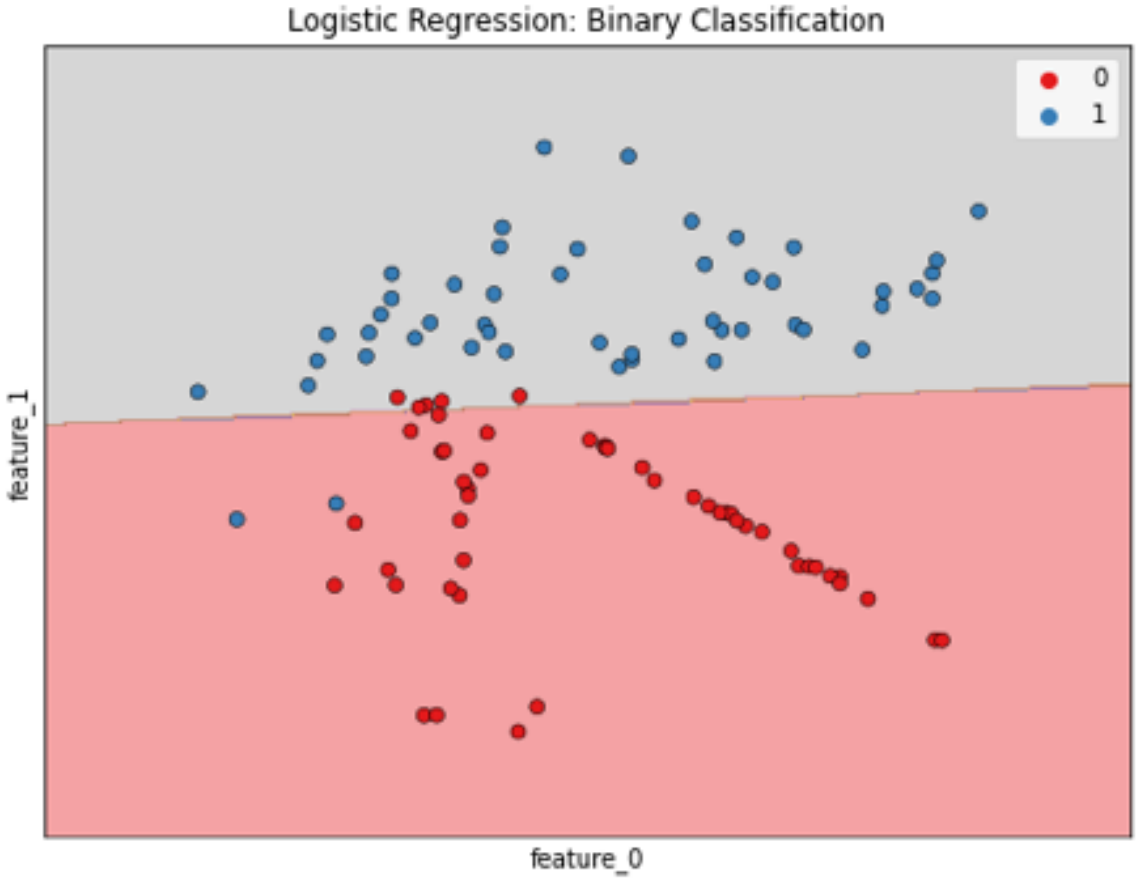
	Predicted: Legitimate	Predicted: Fraudulent
Actual: Legitimate	True Negative	False Positive
Actual: Fraudulent	False Negative	True Positive

The class of interest is called the positive class
here we aim to detect fraud > the positive class is an illegitamate transaction
confusion matrix still gives us the accuracy matrix > (tp + tn) / (tp + tn + fp + fn)
also get precision > tp / tp +fp
precision is also called the positive predictive value
high precision = lower false positive rate
our example high precision translates to fewer legitimate transactions being classified as fraudulent
also get recall > tp / tp + fn
also called sensitivity
high recall = lower false negative rate
our example high recall means predicting most fraudulent transactions correctly
also get F1 score
F1 Score = 2 * ((precision * recall) / (precision + recall))
F1 Score is the harmonic mean of precision and recall
gives equal weight to precision and recall, which means it factors in both the number of errors made by the model and the type of errors
*F1 score favors models with similar precision and recall
is a useful metric if we are seeking a model which performs reasonably well across

both metrics

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

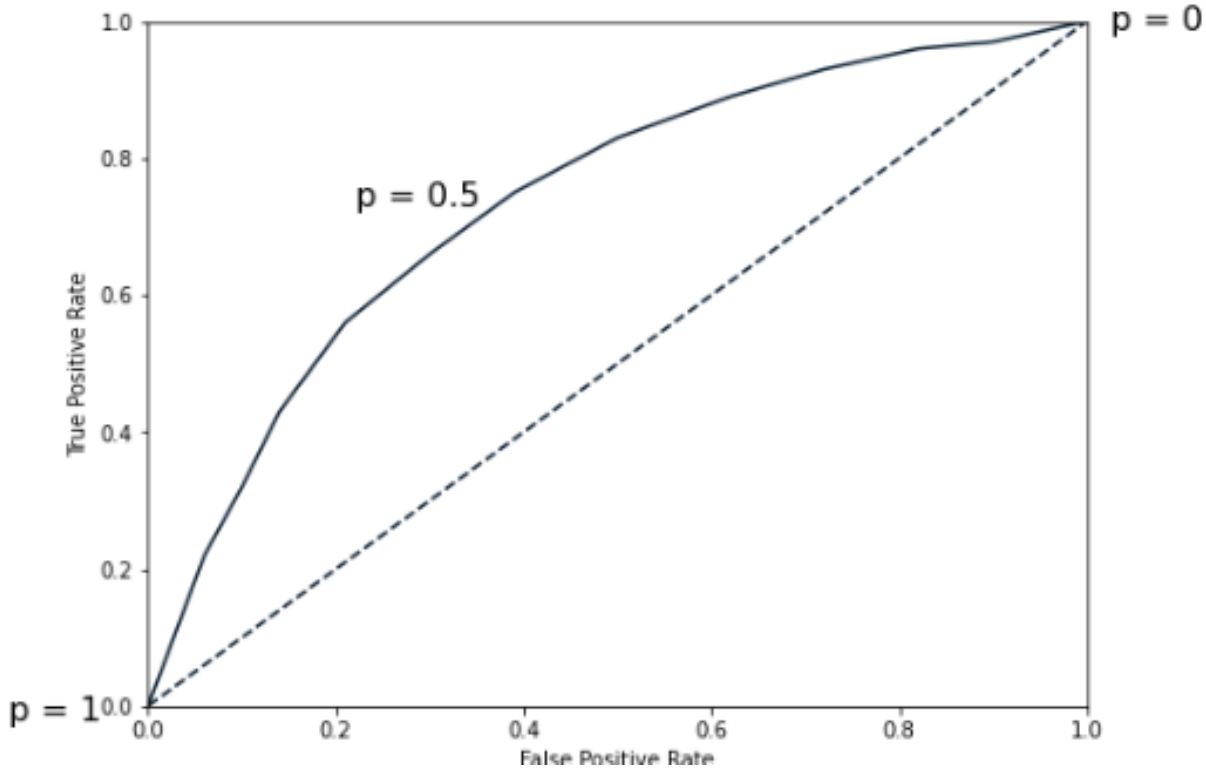
Logisitic regression is used for classification
logistic regression ouputs probabilities
model calculates the probability (p) that an observation belongs to a binary class
if p > 0.5, data is labeled 1
if p < 0.5, data is labeled 0
example with diabetes dataset
p > 0.5, labeled with diabetes
p < 0.5, labeled without diabetes
creates a linear decision boundary



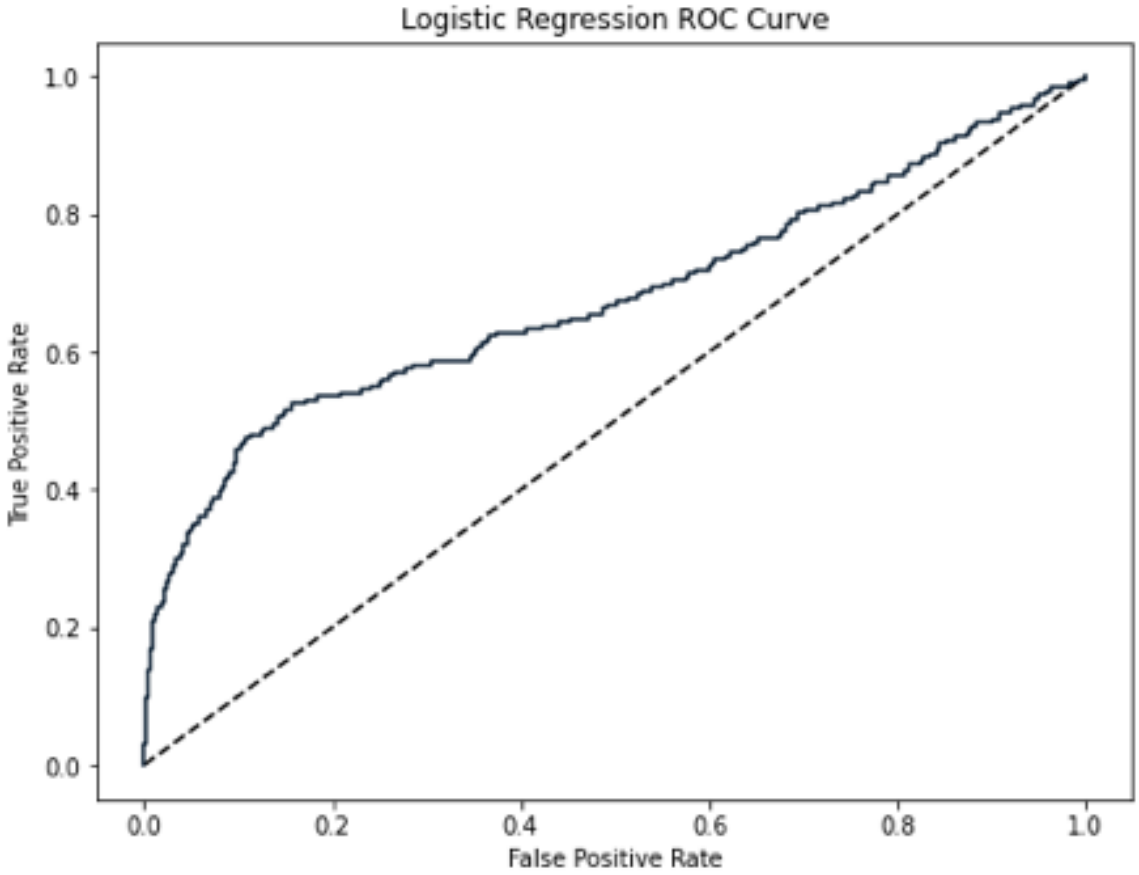
from sklearn.linear_model import LogisticRegression

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)
logreg.predict(X_test)
We can predict probabilities of each instance belonging to a class by calling logistic regression's 'predict_proba' method
returns a 2D array with probabilities for both classes
in our example its the probability of churn or not churn
#slice the second column, representing the positive class probabilities
y_pred_probs = logreg.predict_proba(X_test)[:, 1]
print(y_pred_probs[0])
output > 0.089 probability that the first observation has churned

Default probability threshold is 0.5
what happens if we change this?
The ROC curve
receiver operating characteristic
this allows us to visualize how different thresholds affect the true positive and false positive rates
when threshold equals zero > model predicts one for all observations, meaning it will correctly predict all positive values and incorrectly predict all negative values
when threshold equals one > model predicts zero for all observations, which means that both true and false positive rates are zero



```
from sklearn.metrics import roc_curve
#pass test labels as first argument, predicted probabilities as second argument
#unpack the results into three variables: false positive rate, true positive rate, and thresholds
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('Logisitic Regression ROC Curve')
plt.show()
```



So how do we quantify the model's performance based on this plot?
if we have a model with 1 for tpr and 0 for fpr, then we have the perfect model
to calculate in between > we measure the area under the curve (AUC)
ouput referred to as scores
range is from zero to one, with one being ideal
above gives us a score of p=0.67
this is only 34% better than a model making random guesses

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

calculated by using the model's predict_proba method on X_test

How to optimize your model?
hyperparameter tuning
examples are alpha and n_neighbors
how to choose 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

**when fitting different hyperparameter values, we use cross-validation to avoid overfitting the hyperparameters to the test set
still split the data, but perform cross-validation on the training set
we withhold the test set and use it for evaluating the tuned model

One tuning method - Grid search cross-validation
choose a grid of possible hyperparameter values to try
example - KNN model, search across different n_neighbors and a type of metric
perform k-fold CV for each combination

	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	

```
from sklearn.model_selection import GridSearchCV
#instantiate KFold
kf = KFold(n_splits=5, shuffle=True, random_state=42)
#create a dictionary specifying the names and values of the hyperparameters as keys and values
param_grid = {'alpha': np.arange(0.0001, 1, 10), 'solver': ['sag', 'lsqr']}
#'solver' hyperparameter refers to the optimization algorithm used to find the optimal weights or coefficients for the model during the training process
#there are many options for solver and vary depending on the model
#instantiate Ridge
ridge = Ridge()
#call GridSearchCV and pass it our model, the grid we wish to tune over, and set our cross-validation
ridge_cv = GridSearchCV(ridge, parm_grid, cv=kf)
```

#this returns a GridSearch object that we can then fit to the training data
#this fit performs the actual cross-validated grid search
ridge_cv.fit(X_train, y_train)
#.best_params_ and .best_score_ attributes will retrieve the hyperparameters that perform best along with that specific mean cross-validation score
print(ridge_cv.best_params_, ridge_cv.best_score_)

GridSearch has limitations
it doesn't scale well
10 fold CV, 3 hyperparameters, 30 total values = 900 fits

Another way - RandomizedSearchCV
picks random hyperparameter values rather than exhaustively searching through all options
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()
#'n_iter' argument determines the number of hyperparameter values tested
#example five-fold CV with n_iter set to 2 will perform 10 fits
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_)

Evaluate on the test set
by passing it to a call of the random search object's .score method
test_score = ridge_cv.score(X_test, y_test)
print(test_score)
**in this example it actually performs slightly better than the best score in our grid search

Preprocessing data
remember that scikit requires numeric data and no missing values
in real-world data this is rarely the case
this is where preprocessing comes in

example - dealing with categorical features
convert categorical features into numeric values
how does this work?
we use what is called 'dummy' variables
converts cat feature into a binary feature
where 0 = observation was not that cat
and 1 = observation was that cat

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

each row is one song > each song has one genre
this means we get one 1 on each row and nine 0’s
**be careful of duplicating information
***Not fully grasping this concept on why and when a duplicate column would be created
More info:
Duplicates can form with dummy variables when encoding categorical variables using one-hot encoding or dummy encoding. One-hot encoding is a common technique used to convert categorical variables into numerical format, representing each category as a binary column (0 or 1).

Here's how duplicates can form with dummy variables:

1. **Categorical Variables:**
   Let's say you have a categorical variable called "Color" with possible categories: "Red," "Blue," and "Green."

2. **Dummy Encoding:**
   Dummy encoding creates binary columns for each category in the original categorical variable. For example, using one-hot encoding, you'll create three binary columns: "Red," "Blue," and "Green." Each row in the dataset will have a 1 in the corresponding binary column for the color it belongs to and 0 in the other two binary columns.

   ```
   Original Data:
   Color
   Red
   Blue
   Green
   Red
   Blue
```

```
Dummy Encoding:
Red   Blue   Green
1     0      0
0     1      0
0     0      1
1     0      0
0     1      0
```

3. **Duplicate Categories:**
   Now, imagine you have a dataset where two rows have the same categorical value, say "Red." When dummy encoding is performed, two binary columns will have 1 for the "Red" category, resulting in duplicate columns.

```
Original Data:
Color
Red
Blue
Green
Red
Blue
Red

Dummy Encoding:
Red   Blue   Green
1     0      0
0     1      0
0     0      1
1     0      0
0     1      0
1     0      0
```

4. **Duplicate Columns:**
   In the dummy encoded result, you can see that the first and the last rows both have 1 in the "Red" column, indicating the same category. This leads to duplicate columns in the dataset, which can cause issues during model training and affect the model's performance.

To avoid duplicates when using dummy variables, you should consider one of the following approaches:

- Drop one of the dummy columns for each categorical variable to avoid multicollinearity. This is known as "dummy variable trap."
- Use "drop_first=True" argument while performing one-hot encoding in libraries like pandas or scikit-learn. This automatically drops the first dummy column to avoid the trap.

By properly handling duplicate columns, you can ensure that the dummy encoding represents the categorical variable accurately and doesn't introduce redundancies in the data.

Can create dummy variables with scikit or pandas
scikit > OneHotEncoder()
pandas > get_dummies()

example - music dataset (‘popularity’: target variable and ‘genre’: categorical feature)
import pandas as pd
music_df = pd.read_csv(‘music.csv’)
#pass the categorical column
#to avoid duplicates and to increase computing efficiency we only need nine out of ten binary features
#argument ‘drop_first’ set to True takes care of this
music_dummies = pd.get_dummies(music_df[‘genre’], drop_first=True)
#bring these binary features back into our original DF with pd.concat
#bring in as columns so use ‘axis’ argument set to 1
music_dummies = pd.concat([music_df, music_dummies]), axis=1)
#remove original categorical ‘genre’ column
music_dummies = music_dummies.drop(‘genre’, axis=1)

pandas and Python are smart
if only one categorical feature, we can pass the entire DF
pandas will prefix (in our example - genre_Rap and so on) the new binary columns and get drop the original categorical column

once our dummy variables are set, we can fit models as before
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 = (X, y, test_size=0.2, random_state=42)
#create a kf object
kf = KFold(n_splits=5, shuffle=True, random_state=42)
#instantiate a linear regression model

linreg = LinearRegression()
#call cross_val_score
linreg_cv = cross_val_score(linreg, X_train, y_train, cv=kf,
scoring='neg_mean_squared_error')
#we set scoring equal to neg_mean_squared_error > this returns negative MSE
#why? - scikit CV metrics presume a higher score is better, so MSE is changed to negative to counteract this
#calculate the training RMSE and convert to positive
print(np.sqrt(-linreg_cv))

Handling missing data
inspect your dataset
print(music_df.isna(),sum().sort_values())

Dropping missing data
common approach is to remove missing observations accounting for less than 5% of all data
example
#use the .dropna() method and pass the columns with less than 5% missing data to the subset argument
music_df = music_df.dropna(subset['cols_with_<5%', ...])
*if there are missing values in our subset column, the entire row is removed

Another option - imputing values
imputation > use subject-matter expertise to replace missing data with educated guesses
common to use the 'mean'
can also use th median
for categorical values, we typically use the most frequent value (the mode)
**must split our data first, to avoid data leakage
must split our data before imputing to avoid leaking test set information to our model

Sidebar - **More on data leakage
Data leakage, also known as data snooping or data peeking, occurs when information from outside the training dataset is used inappropriately to create a machine learning model or make predictions, leading to overly optimistic or biased results. It is a critical issue in machine learning as it can severely impact the model's performance and generalization to new, unseen data.

Data leakage can take different forms, but the common theme is that information that should not be available at the time of prediction is inadvertently included in the training or evaluation process. This extra information can cause the model to

learn patterns or relationships that do not exist in the real world, leading to inaccurate or unrealistic predictions.

Some common examples of data leakage include:

1. **Train-Test Split Leakage:**
   Splitting the data into training and testing sets should be done before any data preprocessing or feature engineering. If feature scaling, imputation, or other data transformations are applied before splitting, information from the testing set might have leaked into the training set, leading to overfitting and optimistic performance estimates.

2. **Target Leakage:**
   Target leakage occurs when the target variable (the variable to be predicted) is inadvertently included as a feature during model training. For example, predicting credit card defaults based on past due payments would introduce target leakage, as past due payments are caused by credit card defaults.

3. **Temporal Leakage:**
   In time-series data, temporal leakage occurs when information from the future is used to predict the past. For instance, using future stock prices to predict past stock prices would introduce temporal leakage.

4. **Information Leakage:**
   Including variables in the model that would not be available at the time of prediction can lead to information leakage. For instance, using future customer behavior data to predict current customer churn.

To prevent data leakage, it is essential to maintain a strict separation between training and testing data, avoid using information that would not be available at the time of prediction, and be mindful of any data transformations or feature engineering steps that could inadvertently include future or target-related information.

Data leakage can lead to models that perform well on the training data but fail to generalize to new data, making them unreliable for real-world applications. Ensuring data integrity and preventing leakage is critical for building robust and trustworthy machine learning models.

example - imputation workflow
#different imputation methods for numeric and categorical features
#split and store features via categorical or numeric (here we use X_cat or X_num)
from sklearn.impute import SimplImputer

```
X_cat = music_df['genre'].values.reshape(-1, 1)
X_num = music_df.drop(['genre', 'popularity'], axis=1).values
y = music_df['popularity'].values
#make separate training and test sets for the categorical features and the numeric features
#key to use the same random seed for both sets, this ensures target (y) array's values remain unchanged
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)
#instantiate a SimpleImputer()
#use 'strategy' argument set to 'most_frequent' to use mode to impute missing categorical values
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)
```

Why we use fit_transform
After imputing the missing values, the next step is to transform the dataset to make it suitable for training a machine learning model. This is where the `fit_transform` method comes into play.

The reason we use `fit_transform` instead of just `fit` after imputing is that `fit_transform` combines two steps in one:

1. **Fit:**
   During the "fit" step, the imputer estimates the parameters required for imputing the missing values. For example, in the case of mean imputation, the imputer calculates the mean of each feature based on the available data and stores it as an internal attribute.

2. **Transform:**
   The "transform" step uses the estimated parameters from the "fit" step to fill in the missing values in the dataset with the appropriate imputed values.

By using `fit_transform`, we avoid having to perform these two steps separately. It simplifies the process and ensures that the same estimated parameters obtained during the "fit" step are used consistently when transforming the data.

Why we also need to transform the test set
After imputing missing values in the training set, it is essential to also transform the test set using the same imputer to ensure consistency and prevent data

leakage. The reasons for transforming the test set after imputation are as follows:

1. **Consistency with Training Set:**
   During the training phase, the imputer estimated parameters (e.g., mean, median, or most frequent value) from the available data in the training set. These estimated parameters were used to fill in the missing values. By transforming the test set with the same imputer, you ensure that the imputed values in the test set are consistent with the training set. This consistency is crucial because the model has been trained on the training set using the imputed values, and it expects the same data format during prediction.

2. **Preventing Data Leakage:**
   If you were to use a separate imputer for the test set, there is a risk of data leakage. Data leakage occurs when information from the test set, or future data, inadvertently leaks into the training phase. It can lead to over-optimistic model performance and unrealistic predictions. By using the same imputer for both the training and test sets, you avoid this potential issue.

3. **Maintaining Data Integrity:**
   Transforming the test set ensures that the entire dataset (both training and test sets) is consistent and properly preprocessed. This consistency helps maintain the integrity of the data and ensures that the test set is in the same format as the training set when the model was trained.

This ensures that both sets have consistent imputed values before training and testing the machine learning model.

Example cont'd from above:
now impute the numeric data
#instantiate another imputer
imp_num = SimpleImputer()
#fit and transform the training features
#transform the test features
X_train_num = imp_num.fit_transform(X_train_num)
X_test_num = imp_num.transform(X_test_num)
#now combine our training data using numpy append() method
#pass the two arrays and set as columns
#repeat for the test data
X_train = np.append(X_train_num, X_train_cat, axis=1)
X_test = np.append(X_test_num, X_test_cat, axis=1)
**due to their ability to transform our data, imputers are known as transformers

Imputing with a pipeline

a pipeline is an object used to run a series of transformations and build a model in a single workflow
example
from sklearn.pipeline import Pipeline
#dropn missing values accounting for less that 5% of our data
music_df = music_df.dropna(subset=[‘genre’, ‘popularity’, ‘loudness’, ‘liveness’, ‘tempo’])
#convert values in ‘genre’, our target, to a 1 if Rock, else 0 with the np.where() method
music_df[‘genre’] = np.where(music_df[‘genre’] == ‘Rock’, 1, 0)
#create X and y
X = music_df.drop(‘genre’, axis=1).values
y = music_df[‘genre’].values
#next to build a pipeline, construct a list of steps containing tuples with the step names specified as strings
steps = [(‘imputation’, SimpleImputer()), (‘logistic_regression’, LogisticRegression())]
#instantiate a Pipeline
pipeline = Pipeline(steps)
#split and fit our data
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)
#compute accuracy
pipeline.score(X_test, y_test)

**in a pipeline, each step but the last must be a transformer
what does this mean?
In the context of building a pipeline in machine learning, the statement "each step but the last must be a transformer" means that all the steps in the pipeline, except the final step, should be instances of transformer objects. A transformer is an object that implements the `fit` and `transform` methods, which allows it to preprocess the data and perform feature engineering.

The pipeline is a convenient way to chain multiple data preprocessing steps and machine learning models together. It ensures that the data flows seamlessly through each step, allowing you to combine data preprocessing and model training into a single workflow.

Here's an example of a pipeline with two steps, where the first step is a transformer, and the last step is a machine learning model:

```python
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression

# Step 1: Define the pipeline with a list of tuples containing step names and
transformers
steps = [
    ('scaling', StandardScaler()),    # Transformer for data scaling
    ('model', LogisticRegression())   # Machine learning model
]

# Step 2: Create the pipeline by passing the steps list to Pipeline
pipeline = Pipeline(steps)

# Step 3: Split the data into training and test sets (not shown here)

# Step 4: Fit the pipeline to the training data
pipeline.fit(X_train, y_train)

# Step 5: Compute accuracy on the test data using the pipeline
accuracy = pipeline.score(X_test, y_test)
print("Accuracy:", accuracy)
```

In this example, the pipeline consists of two steps: data scaling and logistic regression. The `StandardScaler` is a transformer that performs data scaling, and `LogisticRegression` is a machine learning model. The pipeline ensures that the data is first scaled using the `StandardScaler` before being passed to the `LogisticRegression` model for training.

The last step in the pipeline, in this case, is the machine learning model (`LogisticRegression`). Since it is not a transformer but a model, it is exempt from the requirement of being a transformer.

By enforcing that all steps but the last should be transformers, pipelines ensure that the data preprocessing and feature engineering are applied consistently during both training and prediction, helping to avoid data leakage and potential issues with inconsistent preprocessing.

Centering and scaling
also referred to as normalizing or standardizing our data
many ML models us some form of distance to inform them
**if we have features on far larger scales, they can disproportionately influence

our model
example is KNN, which explicitly uses distance when making predictons

***Several ways to scale our data
Standardization > given any column, we can subtract the mean and divide by the variance so that all features are centered around 0 and have a variance of 1
Normalization > subtract the minimum and divide by the range of the data so the normalized dataset has minimum 0 and maximum 1
Or we can also 'normalize' by centering our data so that the range is -1 to 1
**scikit has multiple functions availabe for other types of scaling

example - standardization
```
from sklearn.preprocessing import StandardScaler
#create feature and target arrays
X = music_df.drop('genre', axis=1).values
y = music_df['genre'].values
#before scaling, split our data to avoid data leakage
X_train, _test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
#instantiate StandardScaler object
scaler = StandardScaler()
#fit and transform
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))
```
*looking at the mean and std of the columns verifies the change has taken place

example - 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))
```
output > 0.81
compared to unscaled 0.53
**nearly 30% increase in our models accuracy just by scaling

example - CV and scaling in a pipeline
```
from sklearn.model_selection import GridSearchCV
steps = [('scaler', StandardScaler()), ('knn', KNeighborsClassifier())]
```

```
pipeline = Pipleline (steps)
#**create key (pipeline step name followed by a double underscore, followed by the desired hyperparameter)
#value to the key is a list or an array of the values to try for that particular hyperparameter
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)
print(cv.best_score_) > ouput 82%
print(cv.best_params_) ouput knn 12
```

Another example - calculate R-squared off pipeline

```
# Import StandardScaler
from sklearn.preprocessing import StandardScaler

# Create pipeline steps
steps = [("scaler", StandardScaler()),
        ("lasso", Lasso(alpha=0.5))]

# Instantiate the pipeline
pipeline = Pipeline(steps)
pipeline.fit(X_train, y_train)

# Calculate and print R-squared
print(pipeline.score(X_test, y_test))
```

How do we decide which model to use in the first place?
complex question that depends on the situation
but there are some guiding principles:

1. size of the dataset
    1. fewer features = simpler model and faster training time
    2. some models (like Artificial Neural Networks) require large amounts of data to perform well
2. interpretability
    1. who are the stakeholders?
    2. do we need to explain granularly how the model got the predictions?
    3. linear regression has high interpretability because we can understand the coefficients
3. flexibility
    1. may improve accuracy, by making fewer assumptions about the data
    2. KNN is a more flexible model because it does not assume a linear

relationship between the features and the target

It's all in the metrics
scikit conveniently allows the same methods to be used on most models
this makes for easy comparing
Regression model performance:
- RMSE
- R-squared

Classification model performance:
- accuracy
- confusin matrix
- precision, recall, F1-score
- ROC AUC

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

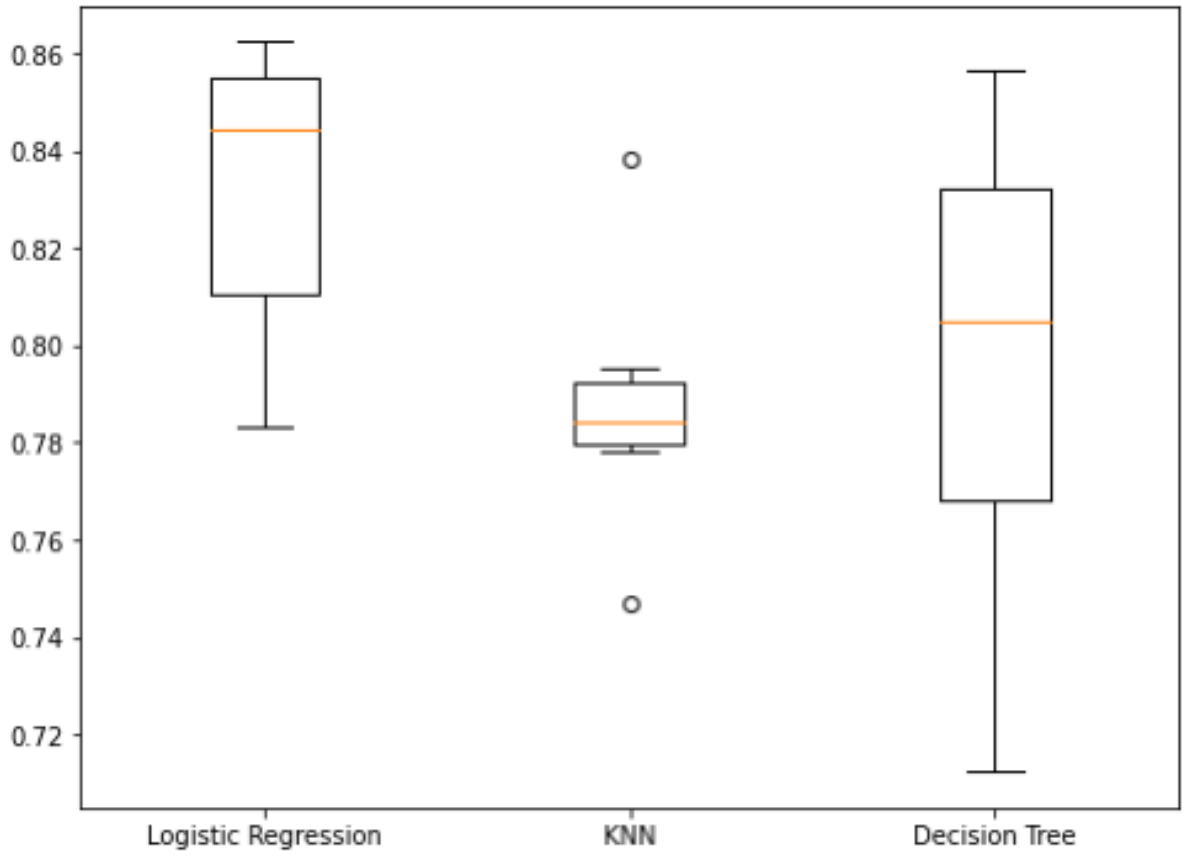
Remember - best to scale our data before evaluating models
Models affected by scaling:
- KNN
- Linear Regression including Ridge and Lasso
- Logistic Regression
- Artificial Neural Network

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)
```

```
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()
```

here we are creating a dictionary with our model names as strings for the keys, an instantiate models as the dictionary's values
create an empty list to store the results
loop through the models in our models dictionary using the .values() method
inside the loop we instantiate a KFold object
perform cv, using the mode being iterated, along with our scaled training features and target training array
set cv argument to our KFold variable 'kf'
default cross_val_score is accuracy
append results to our results list
create a boxplot
set the labels argumen equal to a call of models.keys() to retrieve each model's name



orange line in each box represents each model's median cross-validation score

Test set performance

```
#loop through the names and values of the dictionary using the .items() method
for name, model in models.items():
#fit the model
        model.fit(X_train_scaled, y_train)
#calculate metric accuracy
        test_score = model.score(X_test_scaled, y_test)
        print('{} Test Set Accuracy: {}'.format(name, test_score))
```

Another example

```
# Create models dictionary
models = {"Logistic Regression": LogisticRegression(), "KNN":
KNeighborsClassifier(), "Decision Tree Classifier": DecisionTreeClassifier()}
results = []

# Loop through the models' values
for model in models.values():

  # Instantiate a KFold object
```

```
  kf = KFold(n_splits=6, random_state=12, shuffle=True)

  # Perform cross-validation
  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()
```