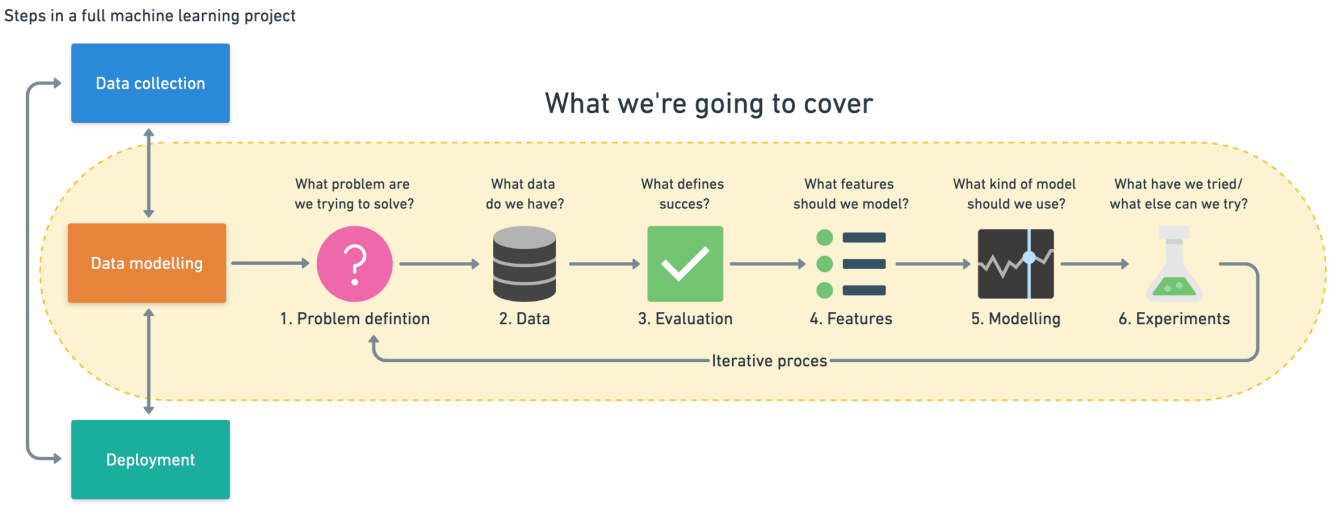
Scikit-Learn

Table of contents

Scikit Learn Introduction

- Scikit Learn (SkLearn): Python Machine Learning Library, built on Numpy & Matplotlib
- Machine Learning = Computer is writting it own function (or ML Models/Algorithms) based on I/P & O/P data.



Readings

- A 6 Step Field Guide for Building Machine Learning Projects

(Back to top)

Scikit Learn Workflow

Get data ready

4 main things we have to do:

- 1.1. Split the data into features and labels (Usually `X` and `y`)
- 1.2. Imputing: Filling or disregarding missing values
- 1.3. Feature Encoding: Converting non-numerical values to numerical values
- 1.4. Feature Scaling: making sure all of your numerical data is on the same scale

1.1. Split Data into X and y

- Before split, Drop all rows with Missing Values in y.

```
# Drop the rows with missing in the "Price" column
car_sales_missing.dropna(subset=["Price"], inplace=True)
```

- Split Data into X and y

```
# Create X (features matrix)
X = car_sales.drop("Price", axis = 1) # Remove 'Price' column (y)

# Create y (lables)
y = car_sales["Price"]
```

- Split X, y into Training & Test Sets

```
np.random.seed(42)

# Split the data into training and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y, test_size=0.2)
```

1.2. Imputing

- Fill missing values with Scikit-Learn `SimpleImputer()` transforms data by filling missing values with a given strategy

```
from sklearn.impute import SimpleImputer #Help fill the missing values
from sklearn.compose import ColumnTransformer

# Fill Categorical values with 'missing' & numerical values with mean
cat_imputer = SimpleImputer(strategy="constant", fill_value="missing")
num_imputer = SimpleImputer(strategy="mean")

# Define different column features
categorical_features = ["Make", "Colour"]
```

```
numerical_feature = ["Odometer (KM)"]

imputer = ColumnTransformer([
    ("cat_imputer", cat_imputer, categorical_features),
    ("num_imputer", num_imputer, numerical_feature)])
```

Note: We use fit_transform() on the training data and transform() on the testing data.

- In essence, we learn the patterns in the training set and transform it via imputation (fit, then transform).
- Then we take those same patterns and fill the test set (transform only).

```
# learn the patterns in the training set and transform it via imputation
(fit, then transform)
filled_X_train = imputer.fit_transform(X_train)
# take those same patterns and fill the test set (transform only)
filled_X_test = imputer.transform(X_test)
```

- Convert back the filled columns back to Data Frame

```
# Get our transformed data array's back into DataFrame's
car_sales_filled_train = pd.DataFrame(filled_X_train,
                                      columns=["Make", "Colour", "Odometer
(KM)"])

car_sales_filled_test = pd.DataFrame(filled_X_test,
                                      columns=["Make", "Colour", "Odometer
(KM)"])
```

1.3. Feature Encoding: Converting categorical features into numerical values

- Note: **Needs to inspect numerical features to check their data are categorical or not** → need to convert into categorical also.
- For example: "Door" feature, although, is numerical in type, but actually categorical feature since only 3 options: (4,5,3)

```
# Inspect whether "Door" is categorical feature or not
# Although "Door" contains numerical values
car_sales["Doors"].value_counts()

# Conclusion: "Door" is categorical feature since it has only 3 options:
(4,5,3)
4    856
5     79
3     65
Name: Doors, dtype: int64
```

```
# Turn the categories into numbers
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer

categorical_features = ["Make", "Colour", "Doors"]

one_hot = OneHotEncoder()
transformer = ColumnTransformer([("one_hot",
                                  one_hot,
                                  categorical_features)],
remainder="passthrough")

# Fill train and test values separately
transformed_X_train = transformer.fit_transform(car_sales_filled_train)
transformed_X_test = transformer.transform(car_sales_filled_test)

transformed_X_train.toarray()
```

1.4. Feature Scaling

- For example: predict the sale price of cars
  - The number of kilometres on their odometers varies from 6,000 to 345,000
  - The median previous repair cost varies from 100 to 1,700.
  - A machine learning algorithm may have trouble finding patterns in these wide-ranging variables
- To fix this, there are two main types of feature scaling:
  - **Normalization** (also called `min-max scaling`): This rescales all the numerical values to between 0 and 1 → `MinMaxScaler` from Scikit-Learn.
  - **Standardization**: This subtracts the mean value from all of the features (so the resulting features have 0 mean). It then scales the features to unit variance (by dividing the feature by the standard deviation). → `StandardScalar` class from Scikit-Learn.
- Note:
  - Feature scaling usually isn't required for your target variable + encoded feature variables
  - Feature scaling is usually not required with tree-based models (e.g. Random Forest) since they can handle varying features

Readings

- Feature Scaling- Why it is required?
- Feature Scaling with scikit-learn
- Feature Scaling for Machine Learning: Understanding the Difference Between Normalization vs. Standardization

(Back to top)

Choose the right estimator

- Scikit-learn uses **estimator** as another term for machine learning model or algorithm
- Based on the .score() + ML Map to choose right estimator
- Map: https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

1. **Structured data (tables)** → ensemble methods (combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator)
2. **Unstructured data (image, audio, text, video)** → deep learning or transfer learning

2.1 Choose the right estimator for Regression Problem:

```
# Let's try the Ridge Regression Model
from sklearn.linear_model import Ridge

#Setup random seed
np.random.seed(42) #to make sure result is reproducible

#instantiate Ridge Model
model = Ridge()
model.fit(X_train, y_train)

# Check the score of the Ridge model on test data
model.score(X_test, y_test) #Return R^2 of the regression
```

2.2 Choose the right estimator for Classification Problem:

```
# Import the LinearSVC estimator class
from sklearn.ensemble import RandomForestClassifier

# Setup random seed
np.random.seed(42)

# Instantiate Random Forest Classifier
clf = RandomForestClassifier(n_estimators=100)

#Fit the model to the data (training the machine learning model)
clf.fit(X_train, y_train)

# Evaluate Random Forest Classifier (use the parterns the model has learnt)
clf.score(X_test, y_test) #Return the mean accuracy on the given test data
and labels.
```

(Back to top)

Make predictions using ML model

3.1 Predict for Classification Models

1. Using `predict()`

```
# Use a trained model to make predictions
y_preds = clf.predict(X_test)
```

Predict a single value: "predict" method always expects a 2D array as the format of its inputs. And putting 12 into a double pair of square brackets makes the input exactly a 2D array:

```
clf.predict([[12]])
```

2. Using `predict_proba()`

- `predict_proba()` returns the probabilities of a classification label.

```
clf.predict_proba(X_test) #[x% prob class = 0, y% prob class = 1]

array([[0.89, 0.11],
       [0.49, 0.51],
       [0.43, 0.57],
       [0.84, 0.16],
       [0.18, 0.82]])
```

- This output `[0.89, 0.11]` means the model is predicting label 0 (index 0) with a probability score of 0.89.
- Because the score is over 0.5, when using predict(), a label of 0 is assigned.

3.2 Predict for Regression Models

- `predict()` can also be used for regression models

(Back to top)

Evaluate a Machine Learning Model

- Tips: Google 'scikit learn evaluate a model'
- 3 ways to evaluate Scikit Learn Models

1. Estimator `score` method
2. The `scoring` parameter
3. Problem-specific metric function
    - Classification Model Evaluation Metrics
    - Regression Model Evaluation Metrics

4.1 Evaluate a model with `Score` Method

- Note: Calling `score()` on a model instance will return a metric assosciated with the type of model you're using. The metric depends on which model you're using.

- Regression Model: `model.score(X_test, y_test) #score() = Return R^2 of the regression`
- Classifier Model: `clf.score(X_test, y_test) #score() = Return the mean accuracy on the given test data and labels.`

4.2 Evaluating a model using the `scoring` parameter

- This parameter can be passed to methods such as `cross_val_score()` or `GridSearchCV()` to tell Scikit-Learn to use a specific type of scoring metric.
- `cross_val_score()` vs `score()`

```
from sklearn.model_selection import cross_val_score
cross_val_score(clf, X, y, cv = 10) #by default = 10-fold => split X,y into
10 different dataset and train 5 different models

array([0.90322581, 0.80645161, 0.87096774, 0.9       , 0.86666667,
       0.76666667, 0.7       , 0.83333333, 0.73333333, 0.8       ])
```

- `cross_val_score()` returns an array where as `score()` only returns a single number

```
# Using score()
clf.score(X_test, y_test)
```

- Figure 1.0: using score(X_test, y_test), a model is trained using the training data or 80% of samples, this means 20% of samples aren't used for the model to learn anything
- Figure 2.0: using 5-fold cross-validation, instead of training only on 1 training split and evaluating on 1 testing split, 5-fold cross-validation does it 5 times. On a different split each time, returning a score for each

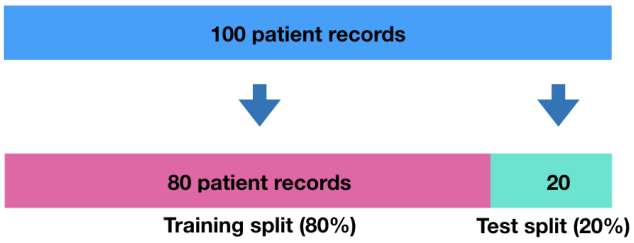


Figure 1.0: Model is trained on training data, and evaluated on the test data.

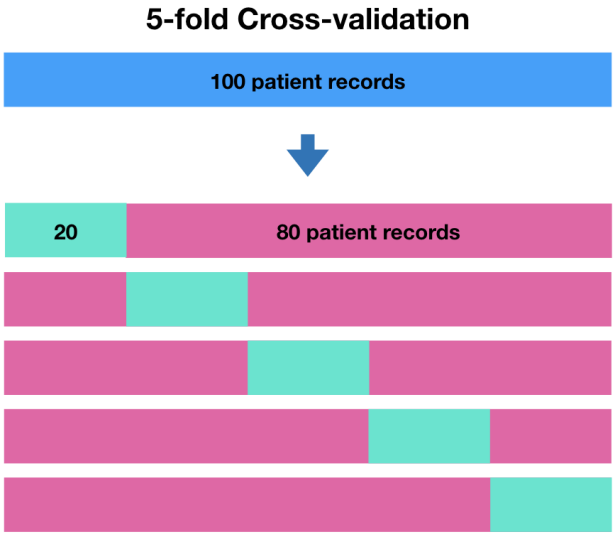


Figure 2.0: Model is trained on training data, and evaluated on the test data.

- **Note#1**: if you were asked to report the accuracy of your model, even though it's lower, you'd prefer the cross-validated metric over the non-cross-validated metric.
- **Note#2**:`cross_val_score(clf, X, y, cv=5, scoring=None) # default scoring`: by default, scoring set to `None`, i.e: `cross_val_score()` will use the same metric as score()
  - For Ex: clf which is an instance of RandomForestClassifier uses mean accuracy as the default score() metric, so `cross_val_score()` will use mean accuracy also
  - You can change the **evaluation score** of `cross_val_score()` uses by changing the `scoring` parameter.

- Cross-Val score for **Classification Model**

```
cv_acc = cross_val_score(clf, X,y, scoring="accuracy")
print(f'The cross-validated accuracy is: {np.mean(cv_acc)*100:.2f}%')
#The cross-validated accuracy is: 82.48%

cv_precision = cross_val_score(clf, X,y, scoring="precision")
print(f'The cross-validated precision is:
{np.mean(cv_precision)*100:.2f}%')
#The cross-validated precision is: 80.86%

cv_recall = cross_val_score(clf, X,y, scoring="recall")
print(f'The cross-validated recall is: {np.mean(cv_recall)*100:.2f}%')
#The cross-validated recall is: 84.24%

cv_f1 = cross_val_score(clf, X,y, scoring="f1")
print(f'The cross-validated f1 is: {np.mean(cv_f1)*100:.2f}%')
#The cross-validated f1 is: 84.15%
```

- Cross-Val score for **Regression Model**

```
cv_r2 = cross_val_score(model, X, y, cv=5, scoring=None) #default score
function= R^2
np.mean(cv_r2) #0.6243870737930857

# Mean Absolute Error
cv_mae = cross_val_score(model, X,y, cv=5,
scoring="neg_mean_absolute_error")
np.mean(cv_mae) #-3.003222869345758

# Mean Squared Error
cv_mse = cross_val_score(model, X,y, cv=5,
scoring="neg_mean_squared_error")
np.mean(cv_mse) #-21.12863512415064
```

4.3 Evaluating with Problem-Specific Metric Function

Classification Model Evaluation Metrics

Four of the main evaluation metrics/methods you'll come across for classification models are:

1. Accuracy: default metric for the score() function within each of Scikit-Learn's classifier models
2. Area under ROC curve
3. Confusion matrix
4. Classification report

4.3.1. Accuracy

```
print(f"Heart Disease Classifier Cross-Validated Accuracy:
{np.mean(cross_val_score)*100:.2f}%")

Heart Disease Classifier Cross-Validated Accuracy: 82.48%
```

4.3.2. Area under the receiver operating characteristic curve (AUC/ROC)

- Area Under Curve (AUC)
- Receiver Operating Characteristic (ROC) Curve

ROC curves are a comparison of a model's true positive rate (TPR) vs a model's false positive (FPR).

- True Positive = Model predicts 1 when truth is 1
- False Positive = Model predicts 1 when truth is 0
- True Negative = Model predicts 0 when truth is 0
- False Negative = Model predicts 0 when truth is 1

True Positive Rate (**TPR**) is a synonym for recall and is therefore defined as follows:

$$TPR = \frac{TP}{TP + FN}$$

False Positive Rate (**FPR**) is defined as follows:

$$FPR = \frac{FP}{FP + TN}$$

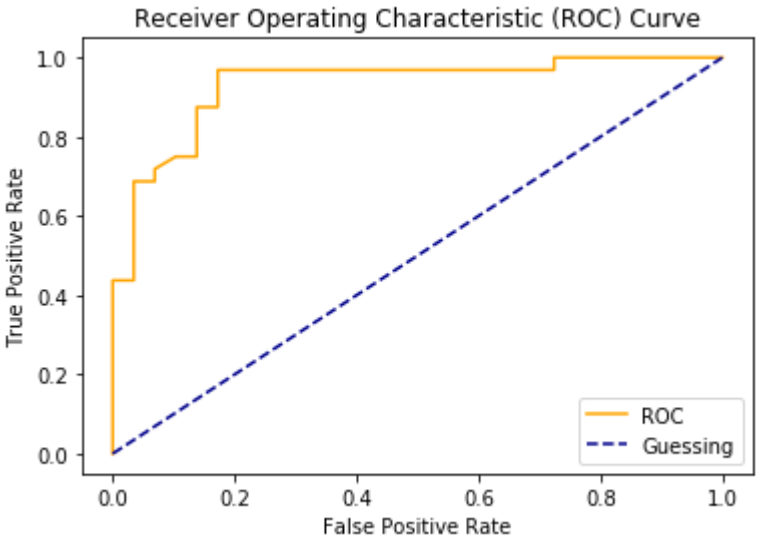
Scikit-Learn lets you calculate the information required for a ROC curve using the `roc_curve` function

```
from sklearn.metrics import roc_curve

# Make predictions with probabilities
y_probs = clf.predict_proba(X_test)

# Keep the probabilites of the positive class only
y_probs = y_probs[:, 1]

# Calculate fpr, tpr and thresholds using roc_curve from Scikit-learn
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
```

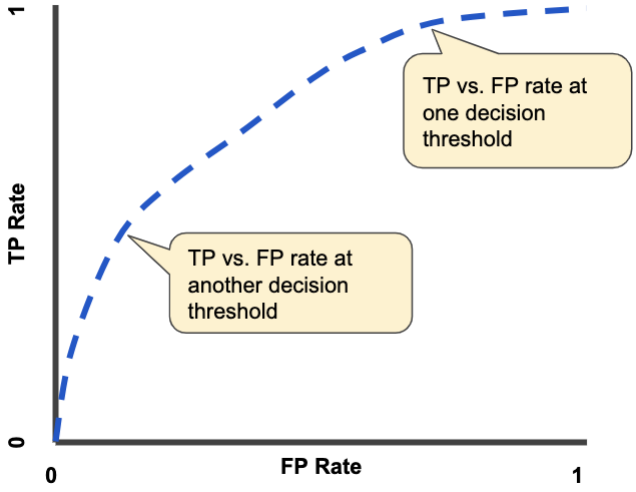
Since Scikit-Learn doesn't have a built-in function to plot a ROC curve, quite often, you'll find a function (or write your own) like the one below

```
# Create a function for plotting ROC curves
import matplotlib.pyplot as plt

def plot_roc_curve(fpr, tpr):
    """
    Plots a ROC curve given the false positive rate (fpr)
    and true positive rate (tpr) of a model.
    """
    #Plot roc curve
    plt.plot(fpr, tpr, color="orange", label="ROC") # x = fpr, y = tpr
    #Plot line with no predictive power (baseline)

    #This line means that prob of classified correctly the positives = prob
of classified NOT correctly as positives
    plt.plot([0,1], [0,1], color="darkblue", linestyle="--",
label="Guessing") # x = [0,1], y=[0,1]

    #Customize the plot
    plt.xlabel("False positive rate (fpr)")
    plt.ylabel("True positive rate (tpr)")
    plt.title("Receiver Operating Characteristics (ROC) Curve")
    plt.legend()
    plt.show()
```

```
plot_roc_curve(fpr, tpr)
```



- Key take-away: our model is doing far better than guessing.
- Curve plots TPR vs. FPR at different classification thresholds. Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives.

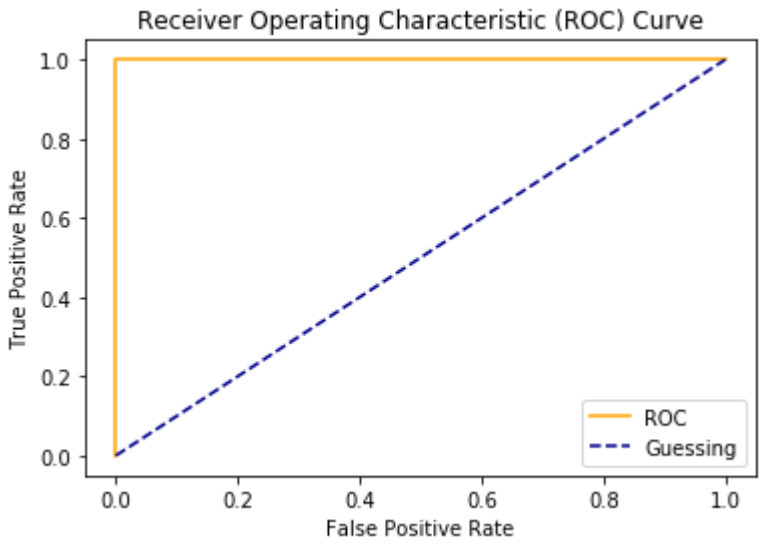


- The maximum ROC AUC score you can achieve is 1.0 and generally, the closer to 1.0, the better the model.
- `AUC (Area Under Curve)` = A metric you can use to quantify the ROC curve in a single number. Scikit-Learn implements a function to caculate this called roc_auc_score().

```
from sklearn.metrics import roc_auc_score

roc_auc_score(y_test, y_probs)

0.93049
```

- The most ideal position for a ROC curve to run along the top left corner of the plot.
- This would mean the model predicts only true positives and no false positives. And would result in a ROC AUC score of 1.0.
- You can see this by creating a ROC curve using only the y_test labels.

```
# Plot perfect ROC curve
fpr, tpr, thresholds = roc_curve(y_test, y_test)
plot_roc_curve(fpr, tpr)
```



This means that the top left corner of the plot is the "ideal" point - a false positive rate of zero, and a true positive rate of one.

Readings

- ROC and AUC, Clearly Explained!
- Classification: ROC Curve and AUC

4.3.3. Confusion Matrix

- A confusion matrix is a quick way to compare the labels a model predicts and the actual labels it was supposed to predict.

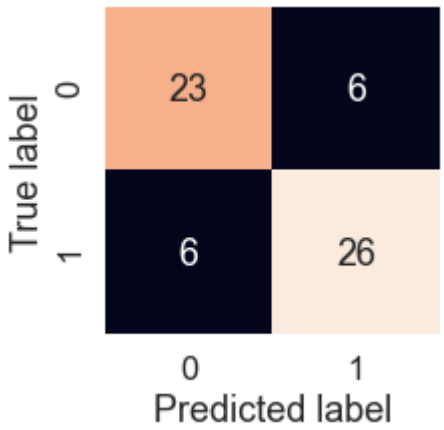
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test, y_preds)
```

- Another way is to use with `pd.crosstab()`.

```
pd.crosstab(y_test,
            y_preds,
            rownames=["Actual Label"],
            colnames=["Predicted Label"])
```

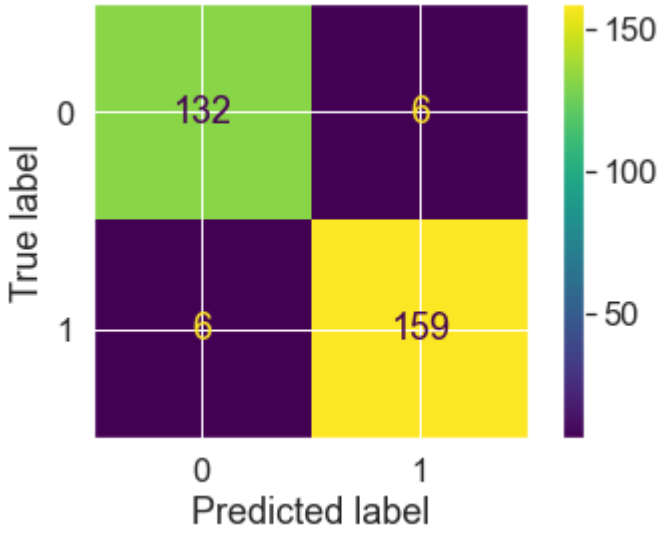
- An even more visual way is with Seaborn's `heatmap()` plot.

```
# Plot a confusion matrix with Seaborn
import seaborn as sns

# Set the font scale
sns.set(font_scale=1.5)

# Create a confusion matrix
conf_mat = confusion_matrix(y_test, y_preds)

# Create a function to plot confusion matrix
def plot_conf_mat(conf_mat):
    """
    Plots a confusion matrix using Seaborn's heatmap().
    """
    fig, ax = plt.subplots(figsize=(3, 3))
    ax = sns.heatmap(conf_mat,
                     annot=True, # Annotate the boxes
                     cbar=False)
    plt.xlabel('Predicted label')
    plt.ylabel('True label');

plot_conf_mat(conf_mat)
```



- Scikit-Learn has an implementation of plotting a confusion matrix in plot_confusion_matrix()

```
from sklearn.metrics import plot_confusion_matrix

plot_confusion_matrix(clf, X, y)
```



4.3.4. Classification Report

- **Precision**: proportion of positive identifications (model predicted class 1) are actually correct → No false postives, Precision = 1.0
- **Recall**: proportion of actual positives are correctly classified → No false negatives, Recall = 1.0
- **F1 Score**: a combination of precision and recall → Perfect model F1 score = 1.0
- **Support**: the number of samples each metric was calculated on. (for Ex below: class 0 has 29 samples, class 1 has 32 samples)
- **Accuracy**: The accuracy of the model in decimal form. Perfect accuracy = 1

- **Marco Avg**: the average precision, recall and F1 score of each class (0 & 1) => Drawback: does not reflect class imbalance (i.e: maybe 0 samples maybe more outweight 1 samples)
- **Weighted Avg**: same as Marco Avg, except: each metric is calculated w.r.t how many samples there are in each class. This metric will favour majority class (i.e: the class which has more samples)

```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_preds))
```

```
              precision    recall  f1-score   support

           0       0.79      0.79      0.79        29
           1       0.81      0.81      0.81        32

    accuracy                           0.80        61
   macro avg       0.80      0.80      0.80        61
weighted avg       0.80      0.80      0.80        61
```

- Alternately, you can use `sklearn.metrics`

```
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score

# Evaluate the classifier
print("Classifier metrics on the test set")
print(f"Accuracy: {accuracy_score(y_test, y_preds)*100:.2f}%")
print(f"Precision: {precision_score(y_test, y_preds)}")
print(f"Recall: {recall_score(y_test, y_preds)}")
print(f"F1: {f1_score(y_test, y_preds)}")

#Classifier metrics on the test set
#Accuracy: 85.25%
#Precision: 0.8484848484848485
#Recall: 0.875
#F1: 0.8615384615384615
```

Example of Imbalanced Classes

For example, let's say there were 10,000 people. And 1 of them had a disease. You're asked to build a model to predict who has it.

You build the model and find your model to be 99.99% accurate. Which sounds great! ...until you realise, all its doing is predicting no one has the disease, in other words all 10,000 predictions are false.

In this case, you'd want to turn to metrics such as precision, recall and F1 score.

```
#Where precision and recall become valuable

disease_true = np.zeros(10000)
disease_true[0] =1 #Only 1 positive case

disease_preds = np.zeros(10000)#Model predicts every case as 0

pd.DataFrame(classification_report(disease_true, disease_preds,
output_dict=True))
```

	0.0	1.0	accuracy	macro avg	weighted avg
precision	0.99990	0.0	0.9999	0.499950	0.99980
recall	1.00000	0.0	0.9999	0.500000	0.99990
f1-score	0.99995	0.0	0.9999	0.499975	0.99985
support	9999.00000	1.0	0.9999	10000.000000	10000.00000

- Precision: 99% for class 0, but 0% for class 1

Ask yourself, although the model achieves 99.99% accuracy, is it useful?

To summarize:

- **Accuracy** is a good measure to start with if all classes are balanced (e.g. same amount of samples which are labelled with 0 or 1)
- **Precision and recall** become more important when classes are imbalanced.
- If false positive predictions are worse than false negatives, aim for higher precision.
- If false negative predictions are worse than false positives, aim for higher recall.

Regression Model Evaluation Metrics

Regression Model evaluation metrics: https://scikit-learn.org/stable/modules/model_evaluation.html#regression-metrics

1. R^2 (r-squared) or coefficient of determination. → Maximize
2. Mean Absolute Error (MAE) → Minimize
3. Mean Squared Error (MSE) → Minimize

4.3.1. R^2 (r-squared) or coefficient of determination

- What R-squared does: Compares your model predictions to the mean of the targets.
- Values can range from negative infinity (a very poor model) to 1.
- For example, if all your model does is predicting the mean of the targets, it's R^2 value would be 0. And if your model perfectly predicts a range of numbers it 's R^2 value would be 1.

```
from sklearn.metrics import r2_score
y_preds = model.predict(X_test)
r2_score(y_test, y_preds) #Can indicate how well the model is predicting,
but can't give how far the prediction is => MAE

0.8654448653350507
```

4.3.2. Mean Absolute Error (MAE)

- MAE is the average of the absolute diff btw predictions and actual values.
- MAE gives a better indication of how far off each of your model's predictions are on average.

```
from sklearn.metrics import mean_absolute_error

mae = mean_absolute_error(y_test, y_preds)
2.1226372549019623
```

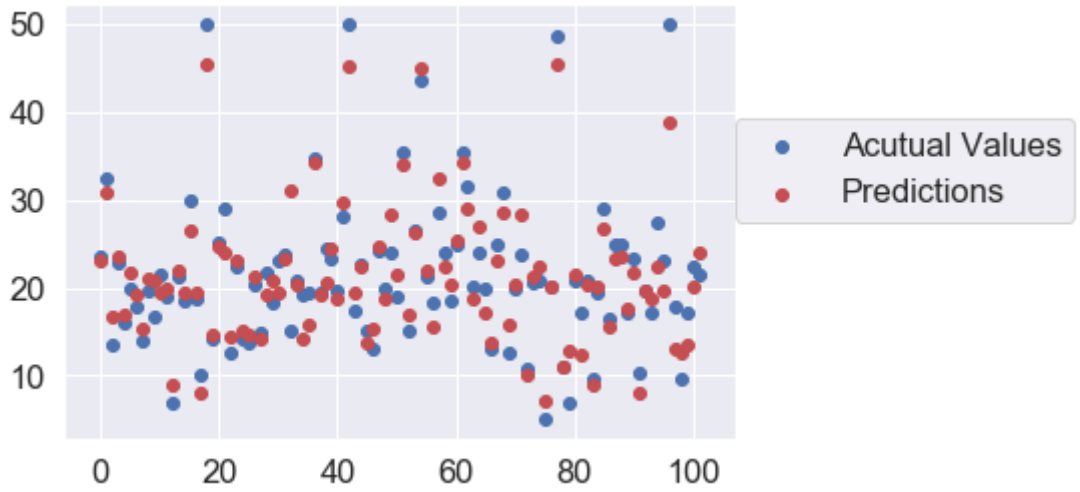
- Our model achieves an MAE of 2.123. This means, on average our models predictions are 2.123 units away from the actual value.

```
df = pd.DataFrame(data={"actual values": y_test,
                        "predicted values" : y_preds})
df["differences"] = df["predicted values"] - df["actual values"]
df.head(5)
```

	actual values	predicted values	differences
173	23.6	23.081	-0.519
274	32.4	30.574	-1.826
491	13.6	16.759	3.159
72	22.8	23.460	0.660
452	16.1	16.893	0.793

- Visualize the results

```
fig, ax = plt.subplots()
x = np.arange(0, len(df), 1)
ax.scatter(x, df["actual values"], c='b', label="Acutual Values")
ax.scatter(x, df["predictions"], c='r', label="Predictions")
ax.legend(loc=(1, 0.5));
```



4.3.3. Mean Squared Error (MAE)

- MSE will always be higher than MAE because is squares the errors rather than only taking the absolute difference into account.

```
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_preds) #9.867437068627442

#Calculate MSE by hand
squared = np.square(df["differences"])
squared.mean() #9.867437068627439
```

(Back to top)

Enhance Model

5.1 Terms and Methods

- First predictions = **baseline predictions**
- First model = **baseline model**

From a **data** perspective:

- Could we collect more data ? (Generally, the more data, the better)
- Could we improve our data ?

From a **model** perspective:

- Is there a better model we could use ?
- Could we improve the current model ?

5.1.1 Hyperparameter vs Parameters

- **Parameters** = model find these patterns in data
- **Hyperparameters** = settings on a model you can adjust to (potentially) improve its ability to find the patterns

5.1.2. Three Methods to adjust Hyperparameters:

1. By Hand
2. Randomly with RandomSearchCV
3. Exhaustively with GridSearchCV

- Take `clf = RandomForestClassifier()` as an example. We're going to try and adjust below Hyperparameters of the classifer:
  - `max_depth`: (the maximum depth of the tree)
  - `max_features`: (the number of features to consider when looking for the best split)
  - `min_samples_leaf`: (the minimum number of samples required to be at a leaf node)
  - `min_samples_split`
  - `n_estimators`

- Create a function to evaluate `y_true` vs `y_preds`:

```
def evaluate_preds(y_true, y_preds):
    """
    Performs evaluation comparison on y_true labels vs y_preds labels
    on a classification model
    """
    accuracy = accuracy_score(y_true, y_preds)
    precision = precision_score(y_true, y_preds)
    recall = recall_score(y_true, y_preds)
    f1 = f1_score(y_true, y_preds)
    metric_dict = {"accuracy" : round(accuracy,2),
                   "precision": round(precision,2),
                   "recall": round(recall,2),
                   "f1": round(f1,2)}
```

```
    print(f"Acc: {accuracy * 100:.2f}%")
    print(f"Precision: {precision:.2f}")
    print(f"Recall: {recall:.2f}")
    print(f"F1 score: {f1:.2f}")

    return metric_dict
```

- Baseline model:

```
clf = RandomForestClassifier()
clf.fit(X_train, y_train)

# Make Baseline Predictions
y_preds = clf.predict(X_test)

# Evaludate the classifier on the validation set

baseline_metrics = evaluate_preds(y_test, y_preds)
Acc: 83.61%
Precision: 0.84
Recall: 0.84
F1 score: 0.84
```

5.2 Hyperparameter Tuning by hand

- Fine-tune the model with `n_estimators=100, max_depth=10`

```
clf = RandomForestClassifier(n_estimators=100, max_depth=10)#More work
taken if adjust by hand like this)
clf.fit(X_train, y_train)

# Make Baseline Predictions
y_preds = clf.predict(X_test)

# Evaludate the classifier on the validation set

baseline_metrics = evaluate_preds(y_test, y_preds)
Acc: 85.25%
Precision: 0.85
Recall: 0.88
F1 score: 0.86
```

5.3 Hyperparameter Tuning with RandomizedSearchCV

- Scikit-Learn's `RandomizedSearchCV` allows us to randomly search across different hyperparameters to see which work best.
- It also stores details about the ones which work best!

- First, we create a grid (dictionary) of hyperparameters we'd like to search over.

```
# Hyperparameter grid RandomizedSearchCV will search over
grid = {"n_estimators": [10,100, 200, 500, 1000, 1200],
        "max_depth": [None, 5,10,20,30],
        "max_features": ["auto", "sqrt"],
        "min_samples_split": [2,4,6],
        "min_samples_leaf": [1,2,4]}
```

- Since we're going over so many different models, we'll set `n_jobs` to `-1` of `RandomForestClassifier(n_jobs=-1)` so Scikit-Learn takes advantage of all the cores (processors) on our computers.

```
from sklearn.model_selection import RandomizedSearchCV

#Instantiate RandomForestClassifier
clf = RandomForestClassifier(n_jobs=-1) #The number of jobs to run in
parallel
#since we're going over so many different models, we'll set n_jobs to -1 of
RandomForestClassifier so Scikit-Learn takes advantage of all the cores
(processors) on our computers.

# Setup RandomizedSearchCV
rs_clf = RandomizedSearchCV(estimator=clf,
                            param_distributions=grid,
                            n_iter=10, #number of models to try
                            cv=5,
                            verbose=2,
                            random_state=42, # set random_state to 42 for
reproducibility
                            refit=True) # set refit=True (default) to refit
the best model on the full dataset )

# Fit the RandomizedSearchCV version of clf
rs_clf.fit(X_train, y_train);

#Fitting 5 folds for each of 10 candidates, totalling 50 fits
```

- `n_iter = 10` in `RandomizedSearchCV` => randomly select 10 combo of Hyperparameter to create 10 models based on the selected hyperparameter
- `cv = 5` => for each combo of Hyperparameters, the data will be splitted 5 times with cv
- Total = 50 models

```
rs_clf.best_params_

{'n_estimators': 100,
 'min_samples_split': 2,
```

```
 'min_samples_leaf': 4,
 'max_features': 'auto',
 'max_depth': 30}
```

- `.best_params_` to return the best hyperparameter combo
- we can make prediction with the best hyperparameter combo

```
rs_y_preds = rs_clf.predict(X_test) #predict() in this case will use the
best_params_
rs_metrics = evaluate_preds(y_test, rs_y_preds)
Acc: 85.25%
Precision: 0.85
Recall: 0.88
F1 score: 0.86
```

5.4 Hyper-parametter Tuning with GridSearchCV

- The main difference between `RandomizedSearchCV` and `GridSearchCV` is
  - `RandomizedSearchCV` searches across a grid of hyperparameters randomly (stopping after n_iter combinations).
  - `GridSearchCV` searches across a grid of hyperparamters exhaustively
- Based on `.best_params_` from `RandomizedSearchCV`, we will reduce the search space from`grid` of hyper-parameters for `GridSearchCV`.

```
# Another hyperparameter grid similar to rs_clf.best_params_
grid_2 = {'n_estimators': [1200, 1500, 2000],
          'max_depth': [None, 5, 10],
          'max_features': ['auto', 'sqrt'],
          'min_samples_split': [4, 6],
          'min_samples_leaf': [1, 2]}
```

- `n_estimators` has 3, `max_depth` has 3, `max_features` has 2,`min_samples_leaf` has 2, `min_samples_split` has 2.
- That's 3x3x2x2x2 = 72 models in total.

```
from sklearn.model_selection import GridSearchCV

# Setup GridSearchCV
gs_clf = GridSearchCV(estimator=clf,
                      param_grid=grid_2,
                      cv=5,
                      verbose=2)

# Fit the RandomizedSearchCV version of clf
gs_clf.fit(X_train, y_train);
```

- Fitting 5 folds (`cv=5`) for each of 72 candidates, totalling 360 models.

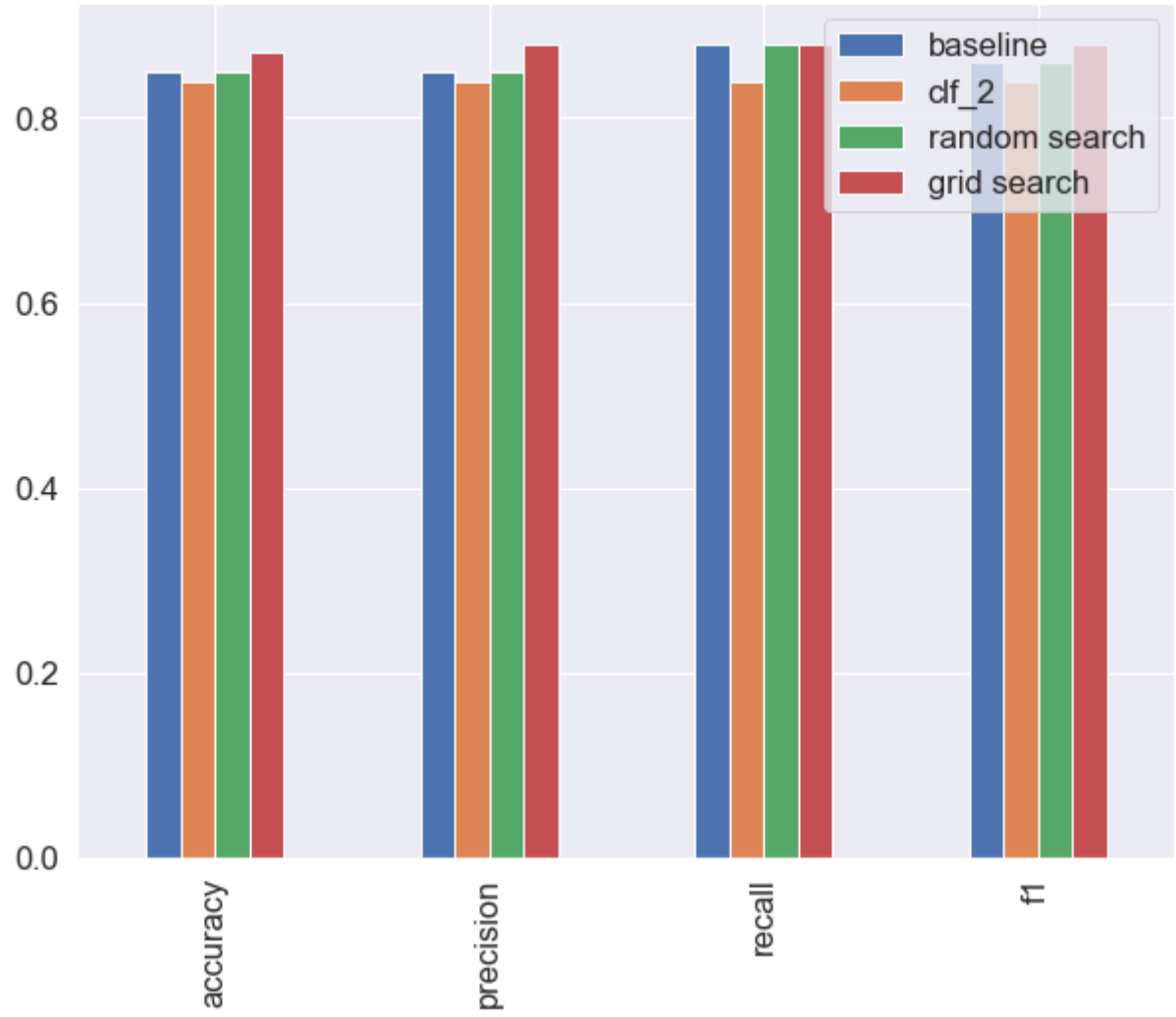
```
rs_clf.best_params_

{'max_depth': 30,
 'max_features': 'sqrt',
 'min_samples_leaf': 4,
 'min_samples_split': 2,
 'n_estimators': 50}
```

- we can make prediction with the best hyperparameter combo

```
rs_y_preds = rs_clf.predict(X_test) #predict() in this case will use the
best_params_
rs_metrics = evaluate_preds(y_test, rs_y_preds)
Acc: 86.89%
Precision: 0.88
Recall: 0.88
F1 score: 0.88
```

- Let's compare our different model metrics

```
compare_metrics = pd.DataFrame({"baseline": baseline_metrics,
                                "clf_2": clf_2_metrics,
                                "random search": rs_metrics,
                                "grid search": gs_metrics
                                })
compare_metrics.plot.bar(figsize=(10,8));
```



(Back to top)

Sklearn Pipeline

- Now we've dropped the rows with no labels and split our data into `X` and `y`, let's create a `Pipeline()` (or a few of them) to fill the rest of the missing values, encode them if necessary (turn them into numbers) and fit a model to them.
- A `Pipeline()` in Scikit-Learn is a class which allows us to put multiple steps, such as filling data and then modelling it, together sequentially.
- More specifically, we'll go through the following steps:
    - Step 1: Define categorical, door and numeric features.
    - Step 2: Build transformer `Pipeline()`'s for imputing missing data and encoding data.
    - Step 3: Combine our transformer `Pipeline()`'s with `ColumnTransformer()`.
    - Step 4: Build a `Pipeline()` to preprocess and model our data with the `ColumnTransformer()` and `RandomForestRegressor()`.
    - Step 5: Split the data into train and test using `train_test_split()`.
    - Step 6: Fit the preprocessing and modelling `Pipeline()` on the training data.
    - Step 7: Score the preprocessing and modelling `Pipeline()` on the test data.
- Let's start with steps 1. and 2.
- We'll build the following:
    - A categorical transformer to fill our categorical values with the value `'missing'` and then one encode them.

    - A door transformer to fill the door column missing values with the value `4`.
    - A numeric transformer to fill the numeric column missing values with the mean of the rest of the column.
- All of these will be done with the `Pipeline()` class.

```
# Pipeline module import
from sklearn.pipeline import Pipeline

# Getting data ready
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.compose import ColumnTransformer


# Define different features and transformer Pipeline
categorical_features = ["Make", "Colour"]
categorical_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="constant", fill_value="missing")),
    ("onehot", OneHotEncoder(handle_unknown="ignore")) #transform
])

door_feature = ["Doors"]
door_transformer = Pipeline(steps=[
    # Create door transformer (fills all door missing values with 4)
    ("imputer", SimpleImputer(strategy="constant", fill_value=4))
])

numeric_feature = ["Odometer (KM)"]
numeric_transformer = Pipeline(steps=[
    ("imputer", SimpleImputer(strategy="mean"))
])

# Setup Preprocessing steps (fill missing values, then convert to numbers)
preprocessor = ColumnTransformer(
    transformers=[
        ("categorical", categorical_transformer, categorical_features),
        ("door", door_transformer, door_feature),
        ("numeric", numeric_transformer, numeric_feature)
    ]
)

# Creating a preprocessing and modelling Pipeline
model = Pipeline(steps=[
    ("preprocessor", preprocessor),# this will fill our missing data and
make sure it's all numbers
    ("model", RandomForestClassifier()) # this will model our data

])
```

- `Pipeline()`'s main input is steps which is a list (`[(step_name, action_to_take)]`) of the step name, plus the action you'd like it to perform.

```
model.fit(X_train, y_train)
```

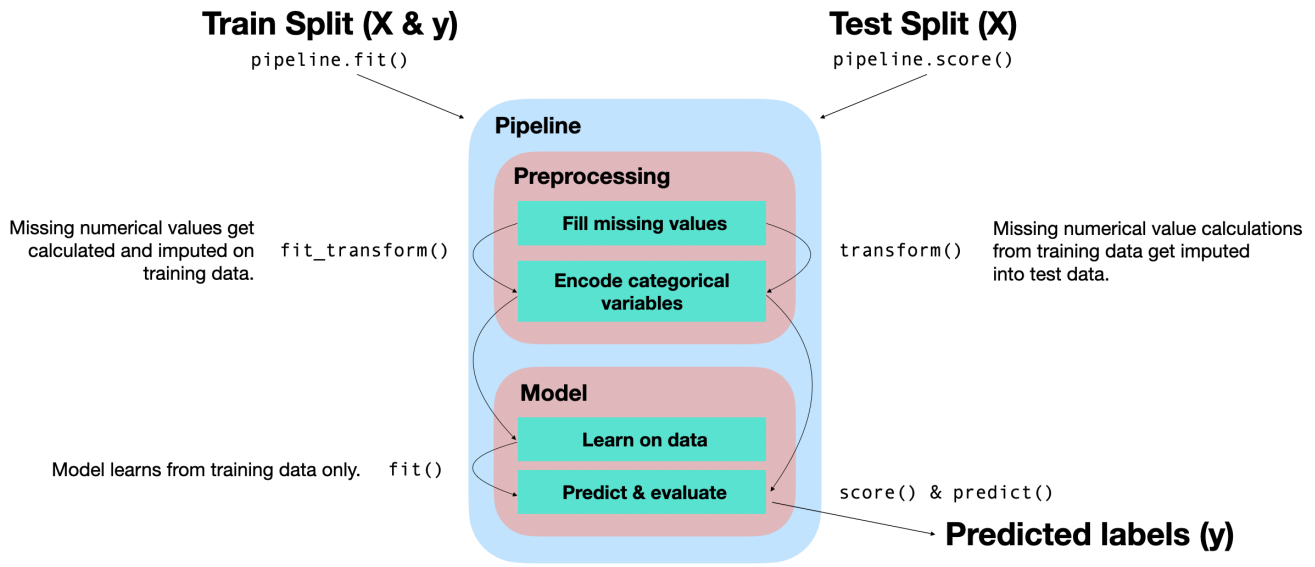
Pipeline behind the scenes

- When filling **numerical data**, it's important **not** to use values from the test set to fill values in the training set. Since we're trying to predict on the test set, this would be like taking information from the future to fill in the past.
- Let's have an example.
- In our case, the `Odometer (KM)` column is missing values. We could fill every value in the column (before splitting it into train and test) with the `mean()`. But this would result in using information from the test set to fill the training set (because we fill the whole column before the split).
- Instead, we split the data into train and test sets first (still with missing values). Then calculate the `mean()` of the `Odometer (KM)` column on the training set and use it to fill the **training set** missing values *as well as* the **test set** missing values.
- Now you might be asking, how does this happen?
- Well, behind the scenes, `Pipeline()` calls a couple of methods:

1. `fit_transform()` - in our case, this computes the `mean()` of the `Odometer (KM)` column and then transforms the rest of the column on the **training data**. It also stores the `mean()` in memory.
2. `transform()` - uses the saved `mean()` of the `Odometer (KM)` column and transforms the **test** values.

The magic trick is:

- `fit_transform()` is only ever used when calling `fit()` on your `Pipeline()` (in our case, when we used `model.fit(X_train, y_train)`.
- `transform()` is only ever used when calling `score()` or `predict()` on your `Pipeline()` (in our case, `model.score(X_test, y_test)`.



- This means, when our missing **numerical values** get calculated and filled (using `fit_transform()`), they only happen on the training data (as long as you only pass `X_train` and `y_train` to

`model.fit()`).
- And since they get saved in memory, when we call `model.score(X_test, y_test)` and subsequently `transform()`, the test data gets preprocessed with information from the training set (using the past to try and predict the future, not the other way round).

What about categorical values?

- Since they usually don't depend on each other, categorical values are okay to be filled across sets and examples.
- Okay, knowing all this, let's cross-validate our model pipeline using cross_val_score().
- Since our `model` is an instance of `Pipeline()`, the same steps as we discussed above happen here with the `cross_val_score()`.

`Pipeline()` for `GridSearchCV` or `RandomizedSearchCV`

- It's also possible to use `GridSearchCV` or `RandomizedSearchCV` with our `Pipeline`.
- The main difference is when creating a hyperparameter grid, you **have to add a prefix to each hyperparameter**.
- The prefix is the name of the Pipeline step you'd like to alter, followed by two `underscores`.
    - For example, to adjust `n_estimators` of "model" in the Pipeline, you'd use: `"model__n_estimators"`.
- `--`: means up to one level from preprocessor -> numeric_transformer > imputer: adjust strategy

```
# Use GridSearchCV with our regression Pipeline
from sklearn.model_selection import GridSearchCV
#Grid of Hyper-parameters will be use in GridSearchCV
pipe_grid = {
    # -- : means up to one level
    #from preprocessor -> numeric_transformer > imputer: adjust strategy
    "preprocessor__numeric__imputer__strategy": ["mean", "median"],
    "model__n_estimators": [100,1000],
    "model__max_depth": [None, 5],
    "model__max_features": ["auto"],
    "model__min_samples_split": [2,4]
}

gs_model = GridSearchCV(model, pipe_grid, cv=5, verbose=2)
gs_model.fit(X_train, y_train)
```

Readings:

- **Reading:** Scikit-Learn Pipeline() documentation.
- **Reading:** Imputing missing values before building an estimator (compares different methods of imputing values).
- **Practice:** Try tuning model hyperparameters with a `Pipeline()` and `GridSearchCV()`.

(Back to top)

Save and Load Model

Two ways to save and load machine learning models:

1. With Python's `pickle` module
2. With the `joblib` module

Pickle

```
import pickle

#Save an existing model to file
pickle.dump(gs_clf, open('gs_random_forest_model_1.pkl', "wb"))

# Load a saved model
loaded_pickle_model = pickle.load(open("gs_random_forest_model_1.pkl",
"rb"))
```

Joblib

```
from joblib import dump, load

# Save model to file
dump(gs_clf, filename="gs_random_forest_model_1.joblib")

# Import a save joblib model
loaded_job_model = load(filename="gs_random_forest_model_1.joblib")
```

(Back to top)