In [1]:
```
%%html

<h2>Scikit-learn steps</h2>
```

Scikit-learn steps

In [1]:
```
%%html

<table>
<tr>
<th>No.</th>
<th>Steps</th>
<th>Description</th>
<th>function</th>
</tr>

<tr>
<td>
 1.
</td>
 <td>
 Get the data & visualise the basic statistics
</td>
 <td>
 Mainly used to understand the data
</td>
 <td>
 Pandas has built in visulaization. Also, Matplotlib is used.
</td>
</tr>

<tr>
<td>
 2.
</td>
 <td>
 Create a train-test dataset & label
```

```
</td>
 <td>
Required for supervised & semi-supervised problem
</td>
 <td>
sklearn.<b>model_selection</b> import <b>train_test_split</b>
<br/>
train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
<hr/>
strat_train_set, strat_test_set = train_test_split(housing, test_size=0.2, random_state=42, <b>stratify=housing['in
</td>
</tr>

<tr>
<td>
 3.
</td>
 <td>
 Explore & visualize to gain insight from the data
</td>
 <td>
 Look for correlation by visualizing it
</td>
 <td>
 pandas has built in visulaization.
</td>
</tr>

<tr>
<td>
 4.
</td>
 <td>
 Clean the Data
</td>
 <td>
How to work with missing features. For example, total_bedrooms feature
<br/>
1. Get rid of the entire row which has missing value in the total_bedrooms column
<br/>
2. Get rid of the whole feature (column) -> drop total_bedrooms
```

```
<br/>
3. Set the the missing value (Zero, the mean, the median, etc.). This is called imputation.
</td>
 <td>
Option 1 & 2 are mainly done using panda NaN functions. E.g. df.dropna(), df.drop, df.fillna()
<hr/>
Option 3 can be done with pandas or imputer function from the sklearn.
from sklearn.<b>impute</b> import <b>SimpleImputer</b>
<br/>
imputer = SimpleImputer(strategy="median")
<br/>
housing_num = <b>housing.select_dtypes(include=[np.number])</b>
<br/>
X = imputer.<b>fit_transform</b>(housing_num)
</td>
</tr>

<tr>
<td>
 5.
</td>
 <td>
Handling Text and Categorical Attributes/Features
</td>
 <td>
OrdinalEncoder and OneHotEncoder from sklearn
</td>
 <td>
housing_cat = housing[['ocean_proximity']]
<br/>
from sklearn.<b>preprocessing</b> import <b>OneHotEncoder</b>
<br/>
onehot_encoder = OneHotEncoder()
<br/>
housing_cat_encoded = onehot_encoder.fit_transform(housing_cat)
</td>
</tr>

<tr>
<td>
 6
```

```
</td>
 <td>
Feature Scaling & Transformation
</td>
 <td>
    <br/>
There are two approaches to feature scaling:<b> min-max scaling and standardization</b>
<br/>
<b>min-max</b> scales the data ranging between minimum to maximum we define
<hr/>
<b>Standardization</b> is done by subtracting the value from the mean and dividing it by SD.
It\'s less affected by outliers, however, it won't restrict the values between certain ranges.
<hr/>
Both approaches will not work for feature distribution that has heavy tail. The solution would be replacing the fea
bucketizing the feature, etc.
<hr/>
As we are transforming the data,  we need to reverse the transformation to get the actual values
</td>
 <td>
 from sklearn.preprocessing import MinMaxScaler
<br/>
min_max_scaler = <b>MinMaxScaler(feature_range=(-1,1))</b>
<br/>
housing_num_min_max_scaled = min_max_scaler.<b>fit_transform(housing_num)</b>
<hr/>
from sklearn.preprocessing import StandardScaler
<br/>
std_scaler = StandardScaler()
<br/>
housing_num_std_scaled =  std_scaler.fit_transform(housing_num)
<hr/>
target_scaler = StandardScaler()
<br/>
...
<br/>
...
<br/>
scaled_predictions = model.predict(some_new_data)
<br/>
 predictions = traget_scaler.<b>inverse_transform</b>(scaled_predictions)
</td>
```

```
</tr>

<tr>
<td>
 7
</td>
 <td>
 Custom transformer (optional)
</td>
 <td>
 This is an optional step where feature scaling with standard transformers is not sufficient.
<hr>
 We can create a basic custom transformer using the function transformer.
 For replacing heavy-tailed distribution with its logarithm, we can use the function transformer.
<hr>
If we would like to tranformer to be trainable: learning some parameter in the fit() and using them later in the tr
We need to write a custom class
TransformerMixin parent class brings the fir_transform method so we do not need to specifically write here

</td>
 <td>
from sklearn.preprocessing import FunctionTransformer
<br/>
log_transformer = <b>FunctionTransformer(np.log, inverse_func=np.exp)</b>
<br/>
log_transformer.transform(housing['population'])
<hr>

from sklearn.base import BaseEstimator, TransformerMixin
<br/>
from sklearn.utils.validation import check_array, check_is_fitted
<br/>
<br/>
<b>class custom_class_name(BaseEstimator, TransformerMixin):</b>
<br/>
<br/>
    def __init__(self, with_mean=True):
        <br/>
        self.with_mean=True
<br/>
<br/>
```

```
    <b>def fit(self, X, y=None):</b>
        <br/>
        return self
<br/>
<br/>
    <b>def transform(self, X):</b>
        <br/>
        return transformed_data
<br/>
<br/>
custom_class_name_clone = custom_class_name()
<br/>
<br/>
scaled = custom_class_name_clone.fit_transform(df)
</td>
</tr>

<tr>
<td>
 8
</td>
 <td>
Transformation pipelines
</td>
 <td>
 Scikit-learn provides the pipeline class to help with sequence of transformations, mainly for numerical data.
<hr>
So far we've dealt with numerical & categorical features separately. We can combine them using the ColumnTransforme
</td>
 <td>
from sklearn.pipeline import Pipeline
<br/>
<b>num_pipeline = Pipeline([</b>
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler()),
])
<br/>
housing_num_prepared = num_pipeline.<b>fit_transform(housing_num)</b>
<br/>
df_test = pd.DataFrame(housing_num_prepared, columns=num_pipeline.get_feature_names_out(), index = housing_num.inde
<hr>
```

```
from sklearn.pipeline import Pipeline
<br/>
from sklearn.preprocessing import OneHotEncoder
<br/>
from sklearn.compose import ColumnTransformer
<br/>
<br/>
num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms",
               "total_bedrooms", "population", "households", "median_income"]
<br/>
<br/>
cat_attribs = ["ocean_proximity"]
<br/>
<br/>
<b>num_pipeline = Pipeline</b>([
    ("impute", SimpleImputer(strategy="median")),
    ("standardize", StandardScaler())
])
<br/>
<br/>
<b>cat_pipeline = Pipeline</b>([
    ("impute", SimpleImputer(strategy="most_frequent")),
    ("one_hot_encoding", OneHotEncoder(handle_unknown="ignore"))
])
<br/>
<br/>
<b>preprocessing = ColumnTransformer</b>([
    ("num", num_pipeline, num_attribs),
    ("cat", cat_pipeline, cat_attribs)
])
<br/>
<br/>
housing_prepared = <b>preprocessing.fit_transform(housing)</b>

</td>
</tr>

<tr>
<td>
 9
```

```
</td>
 <td>
Train a model by adding to the training pipeline & calling the fit method
</td>
 <td>
We already have the trnasformation pipeline and we can add model training to the pipeline. Then fir on the data
</td>
 <td>

<b>tree_reg = Pipeline([</b>
    ('preprocessing', preprocessing),
    ('decision_tree_regressor',DecisionTreeRegressor(random_state=42))
])
<b>tree_reg.fit</b>(housing, housing_labels)
</td>
</tr>

<tr>
<td>
 10
</td>
 <td>
Efficient evaluation using cross validation (Optinal)
</td>
 <td>
We can better evaluate the model using k fold cross validation. In this, training set is randomly splits into 10 no
Then it train & evalaute the decision tree model 10 times, picking a different fold for evaluation every time and u
The result is an array containing the 10 evaluation scores.
<img src='images/cross_validation.png' width=250/>

</td>
 <td>
from sklearn.model_selection import cross_val_score
<br/>
tree_rmse = <b>-cross_val_score</b>(tree_reg, housing, housing_labels, scoring="neg_root_mean_squared_error", <b>cv
<br/>
</td>
</tr>

<tr>
<td>
```

```
 11
</td>
 <td>
 Finetune the model (Optional)
</td>
 <td>
 Normally we need to manually tune the hyperparameters to get the best fit model. Scikit-learn provides tuning meth
GridsearchCV is used when the search space is realtively small and randomizedSearchCV is preferred when the search

    </td>
 <td>
from sklearn.model_selection import GridSearchCV
<br/>
from sklearn.ensemble import RandomForestRegressor
<br/>
<br/>
<b>full_pipeline = Pipeline([
    ('preprocessing', preprocessing),
    ('random_forest', RandomForestRegressor(random_state=42))
])</b>
<br/>
<br/>
# Hyperparameters of random_forest: search space has only two items
<br/>
<b>param_grid = [</b>
    {'random_forest__max_features': [4, 6, 8]}
]
<br/>
<br/>
grid_search = <b>GridSearchCV</b>(full_pipeline, param_grid, cv=3, scoring='neg_root_mean_squared_error')
<br/>
<br/>
grid_search.<b>fit</b>(housing, housing_labels)

<hr>

from sklearn.model_selection import RandomizedSearchCV
<br/>
from scipy.stats import randint
<br/>
<br/>
```

full_pipeline = Pipeline([
    ('preprocessing', preprocessing),
    ('random_forest', RandomForestRegressor(random_state=42))
])

param_distribs = {'random_forest__max_features': randint(low=2, high=4)}

rnd_search = RandomizedSearchCV(
    full_pipeline, param_distributions=param_distribs, n_iter=1, cv=3,
    scoring='neg_root_mean_squared_error', random_state=42)

rnd_search.fit(housing, housing_labels)
</td>
</tr>

<tr>
<td>
 12
</td>
 <td>
 Analysing the best models and their errors (Optional)
</td>
 <td>
We can analyse the feature's importance so that we can drop and keep the features which are less or more important.
</td>
 <td>
final_model = rnd_search.best_estimator_

feature_importances = final_model["random_forest"].feature_importances_

sorted(zip(feature_importances, final_model['preprocessing'].get_feature_names_out()), reverse=True)
</td>
</tr>

<tr>

```
<td>
 13
</td>
 <td>
 Evaluate your system on the test set
</td>
 <td>
It helps to identify how the system will perform on unseen data.
</td>
 <td>
X_test = strat_test_set.drop("median_house_value", axis=1)
<br/>
<br/>
y_test = strat_test_set["median_house_value"].copy()
<br/>
<br/>
final_predictions = final_model.predict(X_test)
<br/>
<br/>
final_rmse = mean_squared_error(y_test, final_predictions, squared=False)
print(final_rmse)
</td>
</tr>

<tr>
<td>
 14
</td>
 <td>
 Saving and loading the model
</td>
 <td>

</td>
 <td>
import joblib
<br/>
    <br/>
# Save the final model
    <br/>
joblib.dump(final_model, "my_california_housing_model.pkl")
```

```
    <br/>
    <br/>

# Load the saved model
    <br/>
final_model_reloaded = joblib.load("my_california_housing_model.pkl")
</td>
</tr>

</table>
```

No.	Steps	Description	function
1.	Get the data & visualise the basic statistics	Mainly used to understand the data	Pandas has built in visulaization. Also, Matplotlib is used.
2.	Create a train-test dataset & label	Required for supervised & semi-supervised problem	sklearn.**model_selection** import **train_test_split** train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42) --- strat_train_set, strat_test_set = train_test_split(housing, test_size=0.2, random_state=42, **stratify=housing['income_cat']**)
3.	Explore & visualize to gain insight from the data	Look for correlation by visualizing it	pandas has built in visulaization.
4.	Clean the Data	How to work with missing features. For example, total_bedrooms feature 1. Get rid of the entire row which has missing value in the total_bedrooms column 2. Get rid of the whole feature (column) -> drop total_bedrooms 3. Set the the missing value (Zero, the mean, the median, etc.). This is called imputation.	Option 1 & 2 are mainly done using panda NaN functions. E.g. df.dropna(), df.drop, df.fillna() --- Option 3 can be done with pandas or imputer function from the sklearn. from sklearn.**impute** import **SimpleImputer** imputer = SimpleImputer(strategy="median") housing_num = **housing.select_dtypes(include=[np.number])** X = imputer.**fit_transform**(housing_num)
5.	Handling Text and Categorical Attributes/Features	OrdinalEncoder and OneHotEncoder from sklearn	housing_cat = housing[['ocean_proximity']] from sklearn.**preprocessing** import **OneHotEncoder** onehot_encoder = OneHotEncoder() housing_cat_encoded = onehot_encoder.fit_transform(housing_cat)
6	Feature Scaling & Transformation	There are two approaches to feature scaling: **min-max scaling and standardization** **min-max** scales the data ranging between minimum to maximum we define --- **Standardization** is done by subtracting the value from the mean and dividing it by SD. It\'s less affected by outliers, however, it won't restrict the values between certain ranges. ---	from sklearn.preprocessing import MinMaxScaler min_max_scaler = **MinMaxScaler(feature_range=(-1,1))** housing_num_min_max_scaled = min_max_scaler.**fit_transform(housing_num)** --- from sklearn.preprocessing import StandardScaler std_scaler = StandardScaler() housing_num_std_scaled = std_scaler.fit_transform(housing_num) --- target_scaler = StandardScaler() ... ...

		Both approaches will not work for feature distribution that has heavy tail. The solution would be replacing the feature with it\'s logarithm, bucketizing the feature, etc. As we are transforming the data, we need to reverse the transformation to get the actual values	scaled_predictions = model.predict(some_new_data) predictions = traget_scaler.**inverse_transform**(scaled_predictions)
7	Custom transformer (optional)	This is an optional step where feature scaling with standard transformers is not sufficient. We can create a basic custom transformer using the function transformer. For replacing heavy-tailed distribution with its logarithm, we can use the function transformer. If we would like to tranformer to be trainable: learning some parameter in the fit() and using them later in the transform() just like other standard tranformers, We need to write a custom class TransformerMixin parent class brings the fir_transform method so we do not need to specifically write here	from sklearn.preprocessing import FunctionTransformer log_transformer = **FunctionTransformer(np.log, inverse_func=np.exp)** log_transformer.transform(housing['population']) from sklearn.base import BaseEstimator, TransformerMixin from sklearn.utils.validation import check_array, check_is_fitted **class custom_class_name(BaseEstimator, TransformerMixin):** def __init__(self, with_mean=True): self.with_mean=True **def fit(self, X, y=None):** return self **def transform(self, X):** return transformed_data custom_class_name_clone = custom_class_name() scaled = custom_class_name_clone.fit_transform(df)
8	Transformation pipelines	Scikit-learn provides the pipeline class to help with sequence of transformations, mainly for numerical data. So far we've dealt with numerical & categorical features separately. We can combine them using the ColumnTransformer class	from sklearn.pipeline import Pipeline **num_pipeline = Pipeline([** ("impute", SimpleImputer(strategy="median")), ("standardize", StandardScaler()),]) housing_num_prepared = num_pipeline.**fit_transform(housing_num)** df_test = pd.DataFrame(housing_num_prepared, columns=num_pipeline.get_feature_names_out(), index = housing_num.index) from sklearn.pipeline import Pipeline from sklearn.preprocessing import OneHotEncoder from sklearn.compose import ColumnTransformer num_attribs = ["longitude", "latitude", "housing_median_age", "total_rooms", "total_bedrooms", "population", "households", "median_income"]

			cat_attribs = ["ocean_proximity"] **num_pipeline = Pipeline**([("impute", SimpleImputer(strategy="median")), ("standardize", StandardScaler())]) **cat_pipeline = Pipeline**([("impute", SimpleImputer(strategy="most_frequent")), ("one_hot_encoding", OneHotEncoder(handle_unknown="ignore"))]) **preprocessing = ColumnTransformer**([("num", num_pipeline, num_attribs), ("cat", cat_pipeline, cat_attribs)]) housing_prepared = **preprocessing.fit_transform(housing)**
9	Train a model by adding to the training pipeline & calling the fit method	We already have the trnasformation pipeline and we can add model training to the pipeline. Then fir on the data	**tree_reg = Pipeline([** ('preprocessing', preprocessing), ('decision_tree_regressor',DecisionTreeRegressor(random_state=42))]) **tree_reg.fit**(housing, housing_labels)
10	Efficient evaluation using cross validation (Optinal)	We can better evaluate the model using k fold cross validation. In this, training set is randomly splits into 10 nooverlpaping folds is k=10. Then it train & evalaute the decision tree model 10 times, picking a different fold for evaluation every time and using the other 9 folds for training. The result is an array containing the 10 evaluation scores.  No description has been provided for this image	from sklearn.model_selection import cross_val_score tree_rmse = **-cross_val_score**(tree_reg, housing, housing_labels, scoring="neg_root_mean_squared_error", **cv=10**)
11	Finetune the model (Optional)	Normally we need to manually tune the hyperparameters to get the best fit model. Scikit-learn provides tuning methods along with cross-validation: Grid search cv and randomized search cv GridsearchCV is used when the search space is realtively small and randomizedSearchCV is preferred when the search space is large or continuous.	from sklearn.model_selection import GridSearchCV from sklearn.ensemble import RandomForestRegressor **full_pipeline = Pipeline([('preprocessing', preprocessing), ('random_forest', RandomForestRegressor(random_state=42))])** # Hyperparameters of random_forest: search space has only two items **param_grid = [** {'random_forest__max_features': [4, 6, 8]}] grid_search = **GridSearchCV**(full_pipeline, param_grid, cv=3, scoring='neg_root_mean_squared_error') grid_search.**fit**(housing, housing_labels) --- from sklearn.model_selection import RandomizedSearchCV from scipy.stats import randint

			full_pipeline = Pipeline([('preprocessing', preprocessing), ('random_forest', RandomForestRegressor(random_state=42))]) **param_distribs = {'random_forest__max_features': randint(low=2, high=4)}** rnd_search = **RandomizedSearchCV(** full_pipeline, **param_distributions=param_distribs, n_iter=1,** cv=3, scoring='neg_root_mean_squared_error', random_state=42) rnd_search.fit(housing, housing_labels)
12	Analysing the best models and their errors (Optional)	We can analyse the feature's importance so that we can drop and keep the features which are less or more important.	final_model = rnd_search.best_estimator_ feature_importances = final_model["random_forest"].feature_importances_ sorted(zip(feature_importances, final_model['preprocessing'].get_feature_names_out()), reverse=True)
13	Evaluate your system on the test set	It helps to identify how the system will perform on unseen data.	X_test = strat_test_set.drop("median_house_value", axis=1) y_test = strat_test_set["median_house_value"].copy() final_predictions = final_model.predict(X_test) final_rmse = mean_squared_error(y_test, final_predictions, squared=False) print(final_rmse)

14	Saving and loading the model		import joblib
			
			# Save the final model
			joblib.dump(final_model, "my_california_housing_model.pkl")
			
			# Load the saved model
			final_model_reloaded = joblib.load("my_california_housing_model.pkl")

In [ ]: