End-to-End Maching Learning Project: Predicting House Prices in California

BY HAMZA JAKOUK

Tabel Of Contents

INTRODUCTION

In this project, we are going to use the 1990 California Census dataset to study and try to understand how the different attributes can make the house prices get higher or lower. How does the location impact? How about the size of the house? The age?

This dataset has a lot of information that can help us. The main goal is to build a Machine Learning Model in python that can learn from this data and make predictions of the housing price in any district, given all the other metrics provided in the dataset.

The project will be divided in 2 main parts: First, we'll take a deep dive in the data, clean it up, and make a big EDA to gather insights and create hypotheses that might be helpful to the model.

After that, we'll jump through hoops to create a ML model capable of making the best possible prediction on the house prices

frame the problem

1. IT IS A SUPERVISED LEARNING TASK. BECAUSE GIVEN LABELED TRAINING EXAMPLES (EACH INSTANCE COMES WITH THE EXPECTED OUTPUT, I.E., THE DISTRICT'S MEDIAN HOUSING PRICE).

2. IT IS A REGRESSION TASK. BECAUSE ASKED TO PREDICT A VALUE

3. THERE IS NO CONTINUOUS FLOW OF DATA COMING IN THE SYSTEM, THERE IS NO PARTICULAR NEED TO ADJUST TO CHANGING DATA RAPIDLY, AND THE DATA IS SMALL ENOUGH TO FIT IN MEMORY, SO PLAIN BATCH LEARNING SHOULD DO JUST FINE

get the data

Now let’s load the data using pandas. we write a small function to load the data :

```
>> import pandas as pd

 def load_housing_data(housing_path):
 csv_path = os.path.join(housing_path,
"housing.csv")
 return pd.read_csv(csv_path)
```

LET’S TAKE A LOOK AT THE TOP FIVE ROWS USING THE DATAFRAME’S HEAD() METHOD :

```
>> housing = load_housing_data(os.path.join("datasets", "housing"))
 housing.head()
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value	ocean_proximity
0	-122.23	37.88	41.0	880.0	129.0	322.0	126.0	8.3252	452600.0	NEAR BAY
1	-122.22	37.86	21.0	7099.0	1106.0	2401.0	1138.0	8.3014	358500.0	NEAR BAY
2	-122.24	37.85	52.0	1467.0	190.0	496.0	177.0	7.2574	352100.0	NEAR BAY
3	-122.25	37.85	52.0	1274.0	235.0	558.0	219.0	5.6431	341300.0	NEAR BAY
4	-122.25	37.85	52.0	1627.0	280.0	565.0	259.0	3.8462	342200.0	NEAR BAY

Each row represents one district. There are 10 attributes :longitude, latitude, housing_median_age, total_rooms, total_bed rooms, population, households, median_income, median_house_value, and ocean_proximity

BEFORE MAKING THE CHANGES IDENTIFIED ABOVE, LET'S MAKE THE GOOD-OLD CHECK FOR NULLS :

```
>> housing.isnull ().sum()
```

```
longitude             0
latitude              0
housing_median_age    0
total_rooms           0
total_bedrooms      207
population            0
households            0
median_income         0
median_house_value    0
ocean_proximity       0
dtype: int64
```

Variables with null values

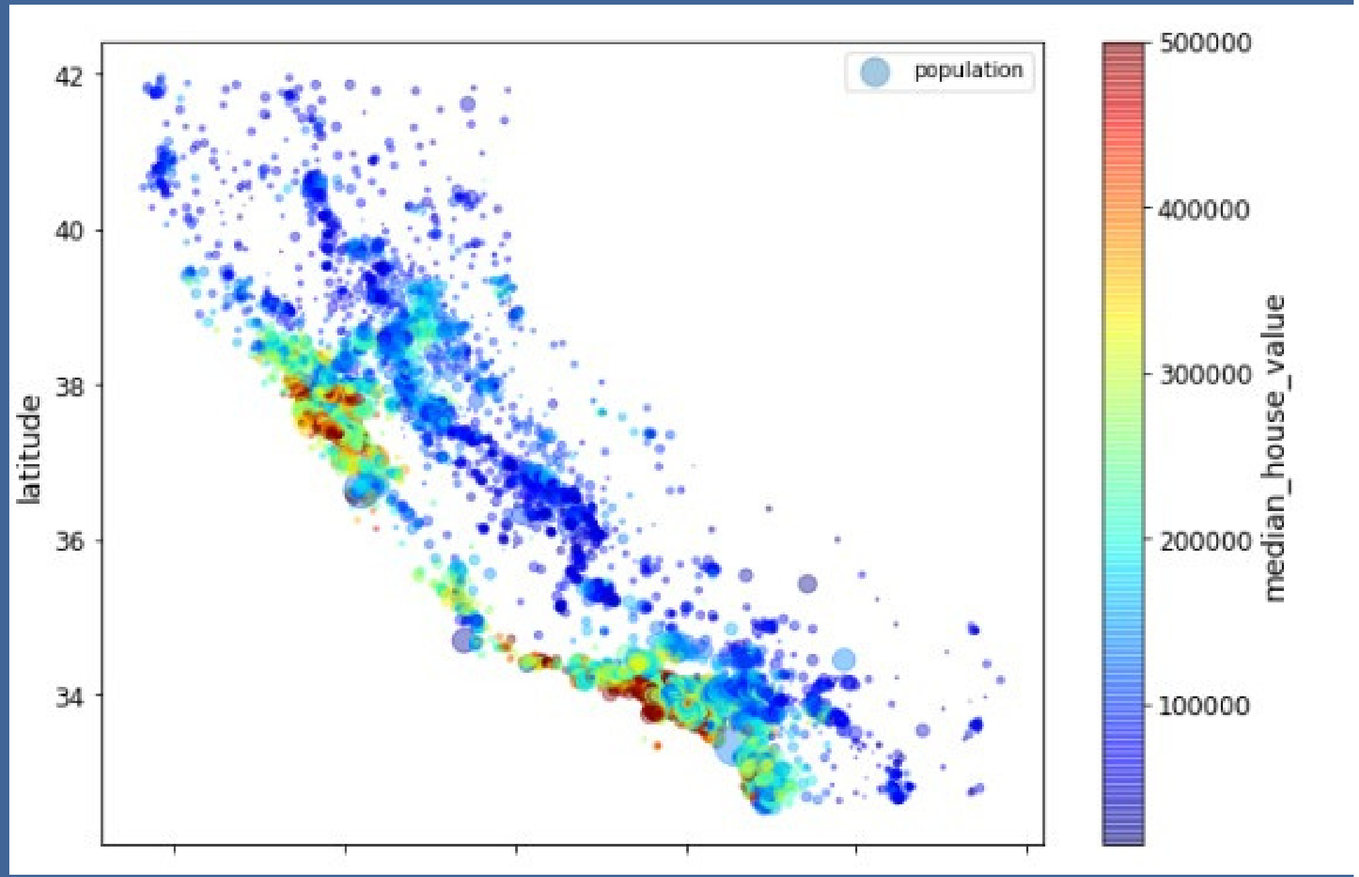
We have 207 records of missing values of "total bedrooms". This number is very small, and the missing values won't jeopardize the model: pretty much every "classic" solution, such as dropping the rows, filling with the mean/median, etc. will give a similar result in the final model.

create a test set

```
>>housing["income_cat"] = pd.cut(housing["median_income"],
                                bins=[0., 1.5, 3.0, 4.5, 6., np.inf],
                                labels=[1, 2, 3, 4, 5])
>> from sklearn.model_selection import StratifiedShuffleSplit,
train_test_split
split = StratifiedShuffleSplit(n_splits=1, test_size=.2,
random_state=42)
    for train_index, test_index in split.split(housing,
housing["income_cat"]):
       strat_train_set = housing.loc[train_index]
       strat_test_set = housing.loc[test_index]
   housing.drop("income_cat", axis = 1, inplace= True)
```

DISCOVER AND VISUALIZ DATA TO GAIN INSIGHT

```
>> housing.plot(kind="scatter", x =
"longitude", y= 'latitude', alpha= .4,
            s=housing["population"]/100,
label="population", figsize=(10,7),
            c="median_house_value",
cmap=plt.get_cmap("jet"), colorbar=True,
            )
    plt.legend()
```



This image tells us that the housing prices are very much related to the location (e.g., close to the ocean) and to the population density

Looking for Correlations

Since the dataset is not too large, we can easily compute the standard correlation coefficient (also called Pearson's r) between every pair of attributes using the corr() method:

```
>> corr_matrix = housing.corr()

 corr_matrix["median_house_val
ue"].sort_values(ascending=
False)
```

```
median_house_value     1.000000
median_income          0.688075
total_rooms            0.134153
housing_median_age     0.105623
households             0.065843
total_bedrooms         0.049686
population            -0.024650
longitude             -0.045967
latitude              -0.144160
Name: median_house_value, dtype: float64
```

The most promising attribute to predict the median house value is the median income,

pipeline transforming

As you can see, there are many data transformation steps that need to be executed in the right order. Fortunately, Scikit-Learn provides the Pipeline class to help with such sequences of transformations. Here is a small pipeline for the numerical attributes:

```
>> from sklearn.pipeline import Pipeline
    from sklearn.preprocessing import StandardScaler
    from sklearn.impute import SimpleImputer

    housing_num = housing.drop("ocean_proximity", axis
= 1)
    num_pipeline = Pipeline([
    ("imputer", SimpleImputer(strategy= "median")),
    ("atr_adder", CombinedAttributesAdder()),
   ("std_transform", StandardScaler())
    ])

    housing_num_tr =
num_pipeline.fit_transform(housing_num)

>> # one hot encoding
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.compose import ColumnTransformer
    num_columns = list(housing_num)
    cat_attribs = ["ocean_proximity"]
    fully_pipline = ColumnTransformer([("num_pipe",
 num_pipeline, num_columns),
                                ("cat", OneHotEncoder(), cat_attribs)])
    housing_prepared =
fully_pipline.fit_transform(housing)
```

SELECT AND TRAIN A MODEL :

Training and Evaluating on the Training Set The good news is that thanks to all these previous steps, things are now going to be much simpler than we might think.

Let’s train a Random Forest model

```
>> # RandomForestRegressor => best model before regularization
from sklearn.ensemble import RandomForestRegressor
forest_reg = RandomForestRegressor(n_estimators=100, random_state=42)
forest_reg.fit(housing_prepared, housing_labels)
>>> display_scores(forest_rmse_scores)
```

```
Scores: [49519.80364233 47461.9115823  50029.02762854 52325.28068953
 49308.39426421 53446.37892622 48634.8036574  47585.73832311
 53490.10699751 50021.5852922 ]
Mean: 50182.303100336096
Standard deviation: 2097.0810550985693
```

this is much better: Random Forests look very promising. However, note that the score on the training set is still much lower than on the validation sets, meaning that the model is still overfitting the training set. Possible solutions for overfitting are to simplify the model, constrain it (i.e., regularize it), or get a lot more training data. Before you dive much deeper into Random Forests

FINE-TUNE YOUR MODEL :

```
>> from sklearn.model_selection import GridSearchCV
param_grid = [
    {'n_estimators': [3, 10, 30], 'max_features': [2, 4, 6, 8]},
 {'bootstrap': [False], 'n_estimators': [3, 10], 'max_features': [2, 3, 4]}
   ]
forest_reg = RandomForestRegressor()
grid_search = GridSearchCV(forest_reg, param_grid, cv=5,
scoring="neg_mean_squared_error",
                           return_train_score = True)
grid_search.fit(housing_prepared, housing_labels)
>> grid_search.best_params_
```

{'MAX_FEATURES': 8, 'N_ESTIMATORS': 30}

, we obtain the best solution by setting the max_features hyperpara- meter to 8 and the n_estimators hyperparameter to 30. The RMSE score for this combination is 49,682, which is slightly better than the score we got earlier using the default hyperparameter values (which was 50,182). we have successfully fine-tuned our best model

feature importances

we will often gain good insights on the problem by inspecting the best models. For example, the RandomForestRegressor can indicate the relative importance of each attribute for making accurate predictions:

```
>> feature_importances = random_grid.best_estimator_.feature_importances_

extra_attribs = ["rooms_per_hhold", "pop_per_hhold", "bedrooms_per_room"]
cat_encoder = fully_pipline.named_transformers_["cat"]
cat_one_hot_attribs = list(cat_encoder.categories_[0])
attributes = list(housing_num) + extra_attribs + cat_one_hot_attribs
sorted(zip(feature_importances, attributes), reverse=True)
```

```
[(0.35818534365808613, 'median_income'),
 (0.1526770881864163, 'INLAND'),
 (0.10957380961488956, 'pop_per_hhold'),
 (0.0724624299084352, 'longitude'),
 (0.06593470548509851, 'bedrooms_per_room'),
 (0.06418082482074416, 'latitude'),
 (0.05183467600359977, 'rooms_per_hhold'),
 (0.04371420973501114, 'housing_median_age'),
 (0.01653900682873527, 'total_rooms'),
 (0.01634664834561291, 'population'),
 (0.015544703892966315, 'total_bedrooms'),
 (0.015073882209962081, 'households'),
 (0.01076608161694577, '<1H OCEAN'),
 (0.004307721214967066, 'NEAR OCEAN'),
 (0.0027772238190798084, 'NEAR BAY'),
 (8.16446594500139e-05, 'ISLAND')]
```

With this information, we may want to try dropping some of the less useful features (e.g., apparently only one ocean_proximity category is really useful, so we could try dropping the others). You should also look at the specific errors that your system makes, then try to under- stand why it makes them and what could fix the problem (adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.).

EVALUATE ON TEST DATA

Now is the time to evaluate the final model on the test set. There is nothing special about this process; just get the predictors and the labels from our test set, run the full_pipeline to transform the data (call transform()) and evaluate the final model on the test set:

```
>> X_test = strat_test_set.drop(["income_cat", "median_house_value"], axis=1)
    y_test = strat_test_set["median_house_value"].copy()
    x_test_pr = fully_pipline.transform(X_test)

>> test_pred = random_grid.predict(x_test_pr)
    rmse = mean_squared_error(y_test, test_pred, squared= False)
    rmse
```

46974.16288833782

In some cases, such a point estimate of the generalization error will not be quite enough to convince you to launch: what if it is just 0.1% better than the model currently in production? You might want to have an idea of how precise this estimate is. For this, you can compute a 95% confidence interval for the generalization error using scipy.stats.t.interval():

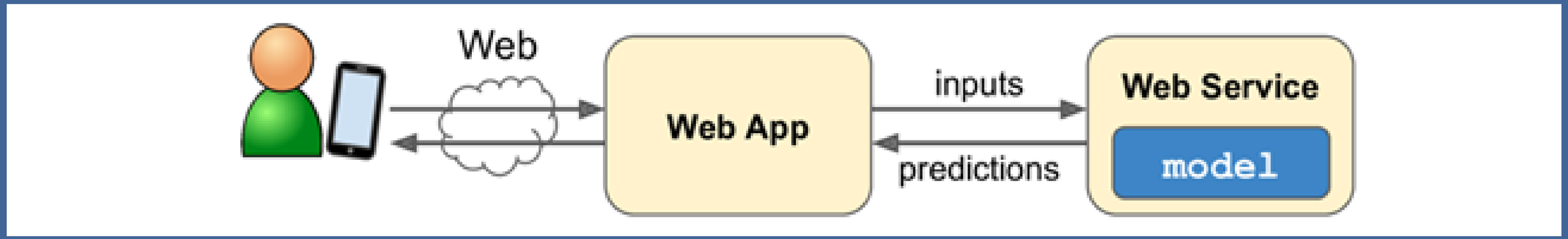
```
>> from scipy import stats

confidence = 0.95
squared_errors = (test_pred - y_test) ** 2
np.sqrt(stats.t.interval(confidence, len(squared_errors) - 1,
                         loc=squared_errors.mean(),
                         scale=stats.sem(squared_errors)))
```

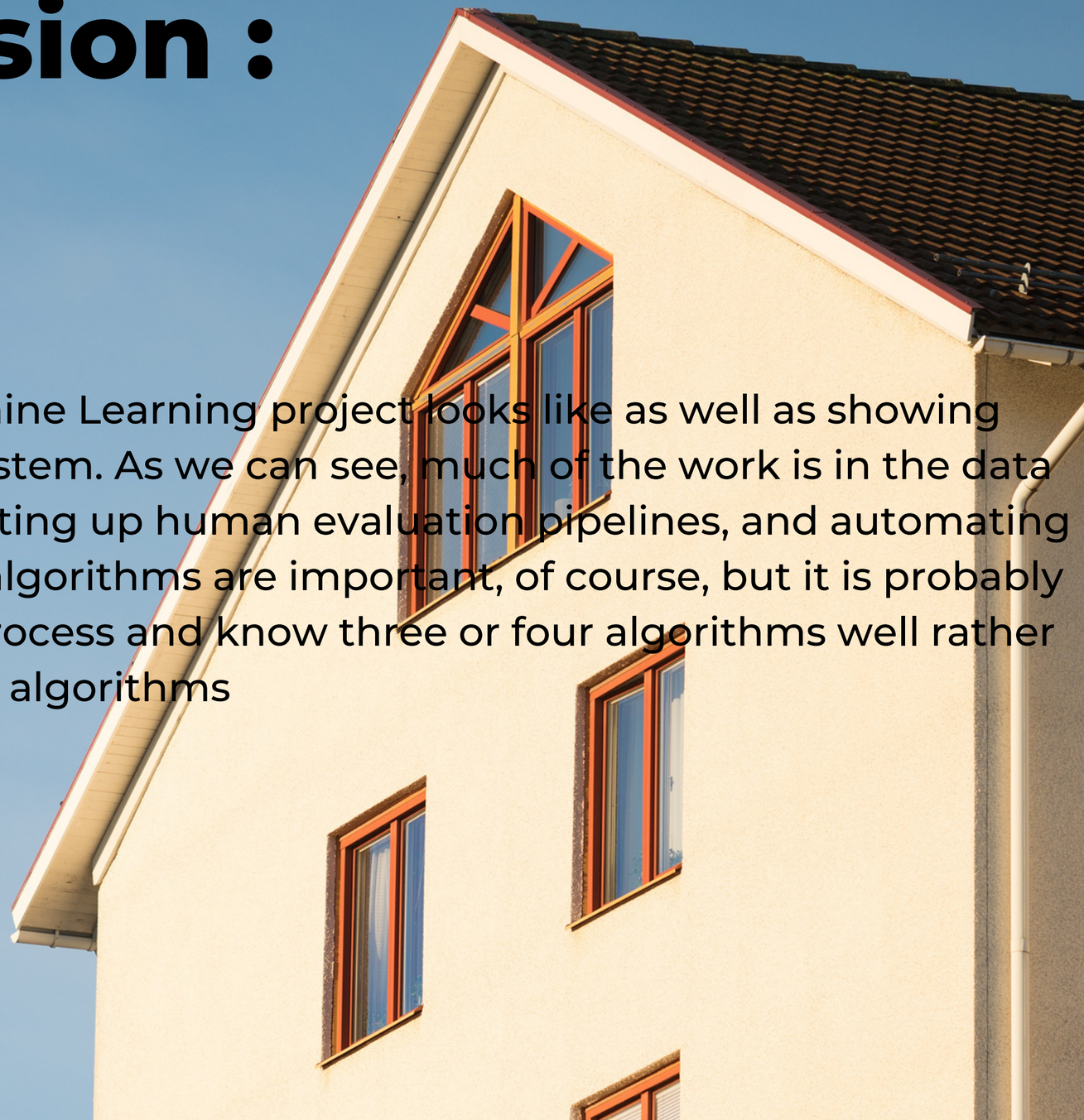
array([44990.28259489, 48877.58617354])

Launch, Monitor, and Maintain Your System :

Now we can deploy our model to the production environment. One way to do this is to save the trained Scikit-Learn model (e.g., using joblib), including the full preprocessing and prediction pipeline, then load this trained model within our production environment and use it to make predictions by calling its predict() method. perhaps the model will be used within a website



Conclusion :



this project gave us a good idea of what a Machine Learning project looks like as well as showing some of the tools we can use to train a great system. As we can see, much of the work is in the data preparation step: building monitoring tools, setting up human evaluation pipelines, and automating regular model training. The Machine Learning algorithms are important, of course, but it is probably preferable to be comfortable with the overall process and know three or four algorithms well rather than to spend all your time exploring advanced algorithms