# Model construction

Yacine Debbabi

December 6, 2020

## Contents



## 1 Load the data

Immediately split out a test set from the data (e.g. 20%) first, and look the sort of data present at hand.

```
df = pd.read_csv("housing.csv")

# Split test data first
us = uniform(size=len(df))
df_test = df[us>0.8]
df = df[us<=.8]

# Explore data
df.head(5) # quick peek
df.info() # shows the dataset size, the types (for potential downcasting) and the amount
    of corrupted entries
df.describe() # shows orders of magnitude
df.hist(bins=100, figsize=(15,10)); plt.show() # shows feature distributions
```

## 2 Build a data processor

Rather than using SK-Learn pipelines, build a simple class to transform dataframes (i.e., clean/filter data and compute additional features). Store fitted components as class members for later re-use.

```python
class DataManager:

    def __init__(self):
        self.scaler = None

    def transform(self, df):

        # Clean data
        pre_cleaning_len = len(df)
        df = df.query("housing_median_age<52 and median_house_value < 500001").dropna()
        print("Dataframe cleaning: len(df):", pre_cleaning_len, "->", len(df))

        # Compute features

        df["rooms_per_house"] = df.total_rooms/df.households
        df["bedrooms_per_house"] = df.total_bedrooms/df.households
        df["bedrooms_per_room"] = df.total_bedrooms/df.total_rooms

        # Scale and eliminate outliers
        num_cols = [c for c in df.columns if is_numeric_dtype(df[c])]

        if self.scaler is None:
            self.scaler = RobustScaler().fit(X=df[num_cols])

        df[[c+"_S" for c in num_cols]] = self.scaler.transform(df[num_cols])

        print("Dataframe transformed")

        return df

dm = DataManager()
df = dm.transform(df)
df_test = dm.transform(df_test)
```

# 3 Implement the data processing logic

Clean the data set.

- Remove incomplete entries or fill missing values (e.g., by default values or population median).
- Eliminate/clip corrupted entries (e.g. unrealistic values and obvious measurement errors).

Randomly sample a smaller dataset from the training data to accelerate data exploration.

# 4 Engineer predictive features

## 4.1 Classification

- Plot the (continuous) feature distribution conditional on the class outcome (and possibility on an additional categorical feature). A predictive feature would show different distribution supports for the different classes.
- Compute class outcome frequencies conditional on (discrete) feature values. A predictive feature would yield significant differences in class distributions.

### 4.2 Regression

- Plot the features against the response using a Gaussian smoother.
- Transform features to achieve a linear dependency of the response on the feature.

### 4.3 For all problems

- Transform features into "normally" distributed features (e.g., handle tail-distributions by applying a log or Box-Cox transformation). HOW TO CALIBRATE THE INDEX?
- Scale the features (using a RobustScaler) so they are mostly distributed between -1 and 1 to facilitate learning tasks.
- Encode unordered categorical data using SKLearn's OneHotEncoder. This effectively adjust the intercept of linear model; but does not change the relation of the response with the other features. Note one should only encode $C-1$ values if there are $C$ possible categories to avoid multicollinearity.
- Encode ordered categorical data into a normalized score showing a monotonic relation with the response or the class probability.

## 5 Fit and diagnostic a linear model

Fit a linear model and test its out-of-sample performance.

- Plot the correlation matrix of the features and the response.
- Identify a small set of features which correlate well with the response but not too much among each other.
- Evaluate OOS performance (e.g. $R^2$/RMSE for linear regressions, and ROC AUC score for classifications).
- Verify features are significant via p-values.
- Verify feature correlation does not introduce coefficient instability via the matrix conditioning number.
- Verify the model assumptions.
- Calibrate regularization parameters.

```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

features = [
    "bedrooms_per_room_bc_S",
    "median_income_bc_S"
]
target = "median_house_value_bc"

X, y = df[features].values, df[target].values

reg = LinearRegression().fit(X, y)
cv_scores = cross_val_score(reg, X, y, cv=5)
print("cv_score:", "%.2f"%(100*np.mean(cv_scores)), "+-", "%.2f"%(100*np.std(cv_scores)))
```

# 6 Stack model predictions

Combine distinct model ouputs into a final model using StackingRegressor.

```
from sklearn.ensemble import StackingRegressor

sr = StackingRegressor(
   estimators=[("1", reg), ("2", gs.best_estimator_)],
   cv=5,
).fit(X, y)

sr.score(X, y)
```