Lab 4 (Part II): Predicting median housing price of census blocks

Please open your notebook from the Part I of Lab 4 and continue working on it following the instructions below:

1. **Select and construct features.** Here, we are going to select and construct features for our machine learning model. Before we make any actions, we will get a data copy from our original training data:

```
train_set = strat_train_set.copy()
```

One way to select features is to focus on those that correlate with the output value. Let's code the following lines:

```
corr_matrix = train_set.corr()

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

The Pearson's correlation coefficient ranges from –1 to 1. When it is close to 1, it means that there is a strong positive correlation; for example, the median house value tends to go up when the median income goes up. When the coefficient is close to –1, it means that there is a strong negative correlation.

Another way to check for correlation between attributes is to use Pandas' *scatter_matrix* function, which plots every numerical attribute against every other numerical attribute. Since there are now 11 numerical attributes, you would get 121 plots, so let's just focus on a few promising attributes that seem most correlated with the median housing value.

```
from pandas.plotting import scatter_matrix


attributes = ["median_house_value", "median_income", "total_rooms",
              "housing_median_age"]
scatter_matrix(train_set[attributes],figsize=(12,8))
```

Based on this correlation analysis, it seems median income is an attribute that strongly correlates with the median house price, so let's zoom in a bit with the following code:

```
train_set.plot(kind="scatter", x= "median_income", y="median_house_value", alpha=0.1)
```

2. **Feature engineering**. We will try to create and extract some features for training our machine learning models. The housing.csv data already provides us with a number of attributes that can be directly used as features. However, we can create additional features based on the existing ones. For example, the attribute of total number of rooms in a census block may not be as useful as the number of rooms per household for predicting median house price. Similarly, the total number of bedrooms by itself is not very useful: you probably want to compare it to the number of rooms. And the population per household also seems like an interesting attribute combination to look at. Feature engineering is the step where a lot of your domain knowledge (e.g., geographic knowledge) applies. A better machine learning model can be developed if you add an important feature that captures an important aspect of a problem but was previously ignored by others.

```
train_set["rooms_per_household"] = train_set["total_rooms"] / train_set["households"]
train_set["bedrooms_per_room"] = train_set["total_bedrooms"] / train_set["total_rooms"]
train_set["population_per_household"] = train_set["population"] / train_set["households"]
```

We can perform the correlation analysis again:

```
corr_matrix = train_set.corr()
corr_matrix["median_house_value"].sort_values(ascending=False)
```

The result is not too bad. The new bedrooms_per_room attribute shows a fair negative correlation with the median house value. The number of rooms per household is also informative—generally, the larger the houses, the more expensive they are. The new population per household attribute does not show much correlation with the median house value.

3. **Data cleaning and imputation**: with the newly added features, our next step is to clean and impute the data. The data for this lab is already relatively clean. However, as we have noticed earlier, there are some missing values in our data. Let's use the following code to see this again:

```
train_set.info()
```

As you can see, both total_bedrooms and bedrooms_per_room have missing values. We can take a look at some of these records:

```
sample_incomplete_rows = train_set[train_set.isnull().any(axis=1)].head()
sample_incomplete_rows
```

We can remove these missing data records, or more extremely delete the whole attribute. Here, however, we will fill in these missing values with the median value of the data.

```
median_total_bedrooms = train_set["total_bedrooms"].median()

train_set["total_bedrooms"].fillna(median_total_bedrooms,inplace=True)

median_bedrooms_per_room = train_set["bedrooms_per_room"].median()

train_set["bedrooms_per_room"].fillna(median_bedrooms_per_room,inplace=True)
```

If you type *train_set.info()* again, you will see that all attributes have the same number of valid records.

4. **Handling texts and categories**: in machine learning and data science projects, you are likely to encounter **categorical data** or **textual data**. Categorical data are constructed based on a set of pre-defined categories usually represented in pre-defined textual strings. Textual data can be said to contain categorical data, but also include natural language texts in which texts can be constructed in any way. In this lab, we have one categorical attribute "ocean_proximity". Most numeric models cannot directly handle texts, so the typical approach is to convert texts to numbers. First, we extract this categorical attribute:

```
ocean_cate = train_set["ocean_proximity"]
```

To handle categorical data, we will convert them to some numbers, such as 1, 2, 3, 4, … However, many models will assume that categories with nearby numbers are more similar. To address this issue, we will use one-hot encoding. Scikit-Learn provides a OneHotEncoder encoder to convert integer categorical values into one-hot vectors. One hot vector is a vector where all other values are zero except one value is 1.

```
from sklearn.preprocessing import OneHotEncoder

cate_encoder = OneHotEncoder(sparse=False)

ocean_cate = ocean_cate.values.reshape(-1,1)   # We need 2D matrix here

ocean_cate_1hot = cate_encoder.fit_transform(ocean_cate)

ocean_cate_1hot
```

You can see the meaning of each value by:

```
cate_encoder.categories_
```

5. **Feature scaling.** Models often do not perform well when the input features have different scales. For example, in our lab, the median house age and population have very different scales in terms of their value ranges. There are two common ways to get all attributes to have the same scale: **min-max scaling** and **standardization**. Min-max scaling (also called normalization) is quite simple: values are shifted and rescaled so that they end up ranging from 0 to 1. We do this by subtracting the min value and dividing by the max minus the min. Standardization is quite different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms. However, standardization is much less affected by outliers. In this lab, we will use standardization.

   Before we do standardization, we will first separate labels from training data (since we don't want to standardize the labels, i.e., median house prices, that we want to predict).

   ```
   train_set_label = train_set["median_house_value"].copy()

   train_set = train_set.drop("median_house_value",axis=1)
   ```

   We will also drop the "*ocean_proximity*" attribute, and will add the one hot vector we created after the standardization process.

   ```
   train_set_num = train_set.drop("ocean_proximity", axis=1)
   ```

   Next, we will apply standardization to all the numeric features in our data:

   ```
   from sklearn.preprocessing import StandardScaler

   standard_scaler = StandardScaler()

   train_set_num_standarized = standard_scaler.fit_transform(train_set_num)
   ```

   Now, let's then combine the standardized features with the one-hot vectors of the categories:

   ```
   preparedData = np.c_[train_set_num_standarized, ocean_cate_1hot]
   ```

   You can take a look at the prepared data:

   ```
   preparedData
   ```

6. **Training and Evaluating on the Training Set**. With the training set prepared, we can now train a model and make predictions.

```
from sklearn.linear_model import LinearRegression
lin_reg = LinearRegression()
lin_reg.fit(preparedData, train_set_label)
```

Evaluate the performance of the model on the same training data.

```
from sklearn.metrics import mean_squared_error
train_predictions = lin_reg.predict(preparedData)
lin_mse = mean_squared_error(train_predictions, train_set_label)
lin_rmse = np.sqrt(lin_mse)
lin_rmse
```

You should see the root mean squared error of the model in the output. Note, strictly speaking, we are not supposed to train the model and evaluate the mode using the same dataset, as we did above. This is because the model already saw the data, so it is more likely to make correct predictions. However, we do it in this lab to make a point, which you will see in the following operations.

We can also see the performance of the trained model on some specific data records:

```
some_data = preparedData[:5]
some_data_label = train_set_label[:5]
print("Predictions:", list(lin_reg.predict(some_data)))
print("Labels:", list(some_data_label))
```

In the result, you should see some predictions are close but some are far away. Overall, its performance is OK. Now, let's train a DecisionTreeRegressor.

```
from sklearn.tree import DecisionTreeRegressor
tree_reg = DecisionTreeRegressor(random_state=42)
tree_reg.fit(preparedData, train_set_label)
```

Now, let’s evaluate its performance on the same training data:

```
train_predictions = tree_reg.predict(preparedData)

lin_mse = mean_squared_error(train_predictions, train_set_label)

lin_rmse = np.sqrt(lin_mse)

lin_rmse
```

The result suggests an RMSE of 0! This surprisingly good result, in fact, is due to overfitting. Remember that we are training and testing the model using the same data? Remember we didn’t do any regularization on the Decision Tree model? This setting means the decision tree can grow as complex as it can in order to perfectly fit the training data (which leads to overfitting).

To objectively evaluate our model, we should use validation data and we need to first divide the training data into training and validation. Scikit-learn already provides a function, = *cross_val_score*, for us to do so. In fact, this function combines the functions of dividing training dataset, fitting model, and evaluating model all together.

```
from sklearn.model_selection import cross_val_score

scores = cross_val_score(tree_reg, preparedData, train_set_label, scoring =
"neg_mean_squared_error", cv=10)

tree_rmse_scores = np.sqrt(-scores)
```

Note that the *cross_val_score* function takes a metric of *"neg_mean_squared_error"*. More information about the possible evaluation metrics can be found here: https://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter. “cv=10” means we will do 10-fold cross validation. Since we did 10-fold cross validation, the *tree_rmse_scores* object contains 10 values. We can define the following function to show the scores as well as their mean and standard deviation.

```
def display_scores(scores):

   print("Scores:", scores)

   print("Mean:",scores.mean())

   print("Standard deviation:", scores.std())


display_scores(tree_rmse_scores)
```

With this cross validation, we can see that the performance of the Decision Tree is just Okay.

Let’s do the same 10-fold cross-validation with the linear regression model:

```
scores = cross_val_score(lin_reg, preparedData, train_set_label, scoring =
"neg_mean_squared_error", cv=10)

lin_rmse_scores = np.sqrt(-scores)

display_scores(lin_rmse_scores)
```

Notice that cross-validation allows you to get not only an estimate of the performance of your model (RMSE here), but also a measure of how precise this estimate is (i.e., its standard deviation). Overall, the linear regression model performs better than the unregularized decision tree model.

Let’s try one more model, Random Forest. We haven’t learned it in our lectures yet, but let’s try it first in this lab. Random Forest is an ensemble model (a model consists of multiple models), which is basically a group of decision trees.

```
from sklearn.ensemble import RandomForestRegressor

forest_reg = RandomForestRegressor(n_estimators=10, random_state=42)

scores = cross_val_score(forest_reg, preparedData, train_set_label, scoring=
"neg_mean_squared_error",cv=10)

forest_rmse_scores = np.sqrt(-scores)

display_scores(forest_rmse_scores)
```

You should see the RMSE scores of the random forest model are smaller than the linear regression model. The random forest model contains a **hyperparameter** *n_estimators* which determines how many decision trees this forest should have (we have 10 decision trees in this forest). How can we choose a suitable value for this hyperparameter? One way is to try several different values manually, which is OK if you have only a small number of hyperparameters to try. A more convenient way is to use Scikit-Learn’s GridSearchCV to search for you. We will try it now.

```
from sklearn.model_selection import GridSearchCV

param_grid = [{'n_estimators': [3, 10, 30]}]

forest_reg = RandomForestRegressor(random_state=42)
```

```
grid_search = GridSearchCV(forest_reg, param_grid, cv=3,
scoring='neg_mean_squared_error', return_train_score=True)

grid_search.fit(preparedData, train_set_label)
```

Note, we used 3, instead of the original 10, for the "cv" parameter here, because doing a grid search with 10-fold cross validation will cost longer time. We can get the best parameter with the following command:

```
grid_search.best_params_
```

Let's also examine the performances of the model at different parameters:

```
cvres = grid_search.cv_results_

for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):

    print(np.sqrt(-mean_score), params)
```

Now, we have trained three different types of models, Linear Regression, Decision Tree, and Random Forest, and we have found that Random Forest with n_estimators = 30 performs the best after the experiments. We will train a final model using all of our training data (without splitting it into training and validation).

```
final_forest_model = RandomForestRegressor(n_estimators=30,random_state=42)

final_forest_model.fit(preparedData, train_set_label)
```

7. **Final testing (This step is optional for this lab)**. Finally, it is time to test the performance of our model on the testing data which the model has never seen before. To do that, we need to apply the same processing steps that we applied to our training data to the testing data. Here, we will create a function to process the data so that it would be easier if we want to re-apply the same data processing steps in the future.

```
test_data_label = strat_test_set["median_house_value"].copy()

test_data = strat_test_set.drop("median_house_value", axis=1)


def prepare_data(data_set):

    # add features

    data_set["rooms_per_household"] = data_set["total_rooms"]/data_set["households"]
    data_set["bedrooms_per_room"] = data_set["total_bedrooms"]/data_set["total_rooms"]
    data_set["population_per_household"] = data_set["population"]/data_set["households"]

    # fill missing values
    data_set["total_bedrooms"].fillna(median_total_bedrooms, inplace=True)
    data_set["bedrooms_per_room"].fillna(median_bedrooms_per_room, inplace=True)

    # one hot encoding
    this_cat_encoder = OneHotEncoder(categories=cate_encoder.categories_, sparse=False)
    ocean_cate = data_set["ocean_proximity"].values.reshape(-1,1)
    ocean_cate_1hot = this_cat_encoder.fit_transform(ocean_cate)

    # Standardization
    data_set_num = data_set.drop("ocean_proximity", axis = 1)
    standard_scaler = StandardScaler()
    data_set_num_standarized = standard_scaler.fit_transform(data_set_num)
    prepared_data_set = np.c_[data_set_num_standarized, ocean_cate_1hot]

    return prepared_data_set



prepared_test_data = prepare_data(test_data)
final_model_prediction = final_forest_model.predict(prepared_test_data)
final_mse = mean_squared_error(test_data_label, final_model_prediction)
print(np.sqrt(final_mse))
```

Please submit the following item:

- Lab4_ FirstName_LastName.ipynb