Chapter 2: End to End Machine Learning Project

K Abhiroop Tejomay

2019:09:22

The main steps involved in a machine learning project are as follows(think of it as a project checklist):

1. Look at the big picture.
2. Get the data.
3. Discover and visualize the data to gain insights.
4. Prepare the data for Machine Learning algorithms.
5. Select a model and train it.
6. Fine-tune your model.
7. Present your solution.
8. Launch, monitor, and maintain your system.

We will look at each step.

1. Look at the Big Picture

1.1 Frame the Problem

- The first question to ask is what exactly the business objective is.
- The next question is to ask what the current solution looks like.
- With all this information, frame the problem:
    - Is it supervised, unsupervised, semisupervised or reinforcement?
    - Is is a classification or a regression task?
    - Should you use batch or online learning techniques?

1.2 Select a Performance Measure

- A typical performance measure for regression tasks is Root Mean Square Error(RMSE)

$$\mathrm{RMSE}(\mathbf{X}, h) = \sqrt{\frac{1}{m}\sum_{i=1}^{m}\left(h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2} \tag{1}$$

- Common Machine Learning Notations:

  - m is the number of instances in the dataset
  - $x^{(i)}$ is a vector of all the feature values (excluding the label) of the ith instance in the dataset, and $y^{(i)}$ is its label (the desired output value for that instance).
  - X is a matrix containing all the feature values (excluding labels) of all instances in the dataset. There is one row per instance, and the ith row is equal to the transpose of $x^{(i)}$, noted $(x^{(i)})^T$
  - h is your system's prediction function, also called a hypothesis.
  - RMSE(X,h) is the cost function measured on the set of examples using your hypothesis h.
- Suppose that there are many outlier districts. In that case, you may consider using the mean absolute error (MAE, also called the average absolute deviation) instead of RMSE.

$$\mathrm{MAE}(\mathbf{X}, h) = \frac{1}{m}\sum_{i=1}^{m}\left|h\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right| \tag{2}$$

- Both the RMSE and the MAE are ways to measure the distance between two vectors: the vector of predictions and the vector of target values.
- RMSE uses the l_2 norm and MAE uses the l_1 norm. The higher the norm index, the more it focuses on large values and neglects small ones. This is why the RMSE is more sensitive to outliers than the MAE. But when outliers are exponentially rare (like in a bell-shaped curve), the RMSE performs very well and is generally preferred.
- In the case of classification, measures like accuracy, precision, recall, f1 score and roc curve are used.

1.3 Check the Assumptions

- Lastly, it is good practice to list and verify the assumptions that have been made so far (by you or others)this can help you catch serious issues early on.

2. Get the Data

2.1 Create the Workspace

- Check the book out for creating the workspace.

2.2 Take a quick look at the Data Structure

- The main methods to have a quick look at the Data Structure are:

```
DataFrame.head()
DataFrame.info()
DataFrame.describe()
DataFrame['categorical_attribute'].value_counts()
```

- Another quick way to get a feel of the type of data you are dealing with is to plot a histogram for each numerical attribute.

```
%matplotlib inline   # only in a Jupyter notebook
import matplotlib.pyplot as plt
housing.hist(bins=50, figsize=(20,15))
plt.show()
```

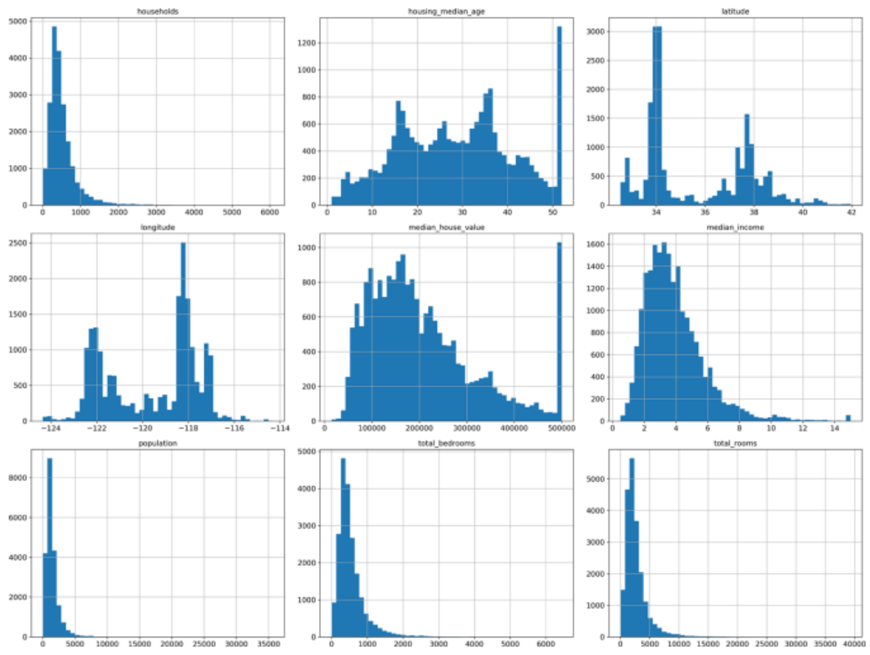

Figure 1: Histogram for Each Numerical Attribute

- Draw preliminary conclusions from the histogram plot.

2.3 Create a Test Set

- Before you look at the data any further, you need to create a test set, put it aside, and never look at it.
- This is because we need to avoid *data snooping bias*: if you look at the test set, you may stumble upon some seemingly interesting pattern in the test data that leads you to select a particular kind of Machine Learning model.
- Scikit-Learn provides `train_test_split` to split the dataset into train and test sets.

```python
from sklearn.model_selection import train_test_split

train_set, test_set = train_test_split(housing, test_size=0.2, random_state=42)
```

- You need to make sure that the test set is representative of the orginal dataset. So you could apply Stratified sampling instead based on a promising feature.

```python
from sklearn.model_selection import StratifiedShuffleSplit

split = StratifiedShuffleSplit(n_splits=1, test_size=0.2, 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]
```

3. Discover and Visualize the Data to Gain Insights

- Apply Univariate Analysis(PDFs and CDFs), Histograms and Pie Charts.
- Plot Box Plots.
- Plot Pair Plots (Multivariate Analysis).
- Look for Correlations.

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

3.1 Experiment with Attribute Combinations

- One last thing you may want to do before preparing the data for Machine Learning algorithms is to try out various attribute combinations(Go to *4.3 Custom Transformers* section of the Notes to see implementation).
- Check the correlation matrix again with the new features created.

4. Prepare the Data for Machine Learning Algorithms

4.1 Data Cleaning

- Most Machine Learning algorithms cannot work with missing features. Things you could do are:
  1. Get rid of the corresponding districts.
  2. Get rid of the whole attribute.
  3. Set the values to some value (zero, the mean, the median, etc.).

You can accomplish these easily using DataFrame's `dropna()`, `drop()`, and `fillna()` methods:

```python
housing.dropna(subset=["total_bedrooms"])     # option 1
housing.drop("total_bedrooms", axis=1)        # option 2
median = housing["total_bedrooms"].median()  # option 3
```

- Don't forget to save the median value that you have computed. You will need it later to replace missing values in the test set when you want to evaluate your system, and also once the system goes live to replace missing values in new data.
- Scikit-Learn provides a handy class to take care of missing values: `SimpleImputer`.

```python
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy="median")
```

- Median can only be computed on numerical attributes so pass the DataFrame with only numerical attributes to the `SimpleImputer` object's `fit` method.
- The imputer has simply computed the median of each attribute and stored the result in its `statistics_` instance variable.
- Now you can use this "trained" `imputer` to transform the training set by replacing missing values with the learned medians:

```python
X = imputer.transform(housing_num)
```

- **Scikit-Learn Design**:
  - Consistency: All objects share a consistent and simple interface:
    * Estimators: Any object that can estimate some parameters based on a dataset is called an estimator (e.g., an imputer is an estimator).
    * Transformers: Some estimators (such as an imputer) can also transform a dataset; these are called transformers.
    * Predictors: Finally, some estimators, given a dataset, are capable of making predictions; they are called predictors.
  - Inspection: All the estimator's hyperparameters are accessible directly via public instance variables (e.g., `imputer.strategy`), and all the estimator's learned parameters are accessible via public instance variables with an underscore suffix (e.g., `imputer.statistics_`).
  - Nonproliferation of classes: Datasets are represented as NumPy arrays or SciPy sparse matrices, instead of homemade classes. Hyperparameters are just regular Python strings or numbers.
  - Composition: Existing building blocks are reused as much as possible. For example, it is easy to create a Pipeline estimator from an arbitrary sequence of transformers followed by a final estimator, as we will see.
  - Sensible defaults: Scikit-Learn provides reasonable default values for most parameters, making it easy to quickly create a baseline working system.

4.2 Handling Text and Categorical Features

- To convert categorical attributes to numerical attributes, use `OrdinalEncoder` and then `OneHotEncoder` classes.
- The output after encoding is a SciPy sparse matrix, instead of a NumPy array. This is very useful when you have categorical attributes with thousands of categories. After one-hot encoding, we get a matrix with thousands of columns, and the matrix is full of 0s except for a single 1 per row.
- If you want to convert it to a (dense) NumPy array, just call the toarray() method.

4.3 Custom Transformers

- You can create custom transformers of your own by inheriting the `BaseEstimator` and the `TransformerMixin` classes:

```python
from sklearn.base import BaseEstimator, TransformerMixin

rooms_ix, bedrooms_ix, population_ix, households_ix = 3, 4, 5, 6

class CombinedAttributesAdder(BaseEstimator, TransformerMixin):
    def __init__(self, add_bedrooms_per_room = True): # no *args or **kargs
        self.add_bedrooms_per_room = add_bedrooms_per_room
    def fit(self, X, y=None):
        return self  # nothing else to do
    def transform(self, X, y=None):
        rooms_per_household = X[:, rooms_ix] / X[:, households_ix]
        population_per_household = X[:, population_ix] / X[:, households_ix]
        if self.add_bedrooms_per_room:
            bedrooms_per_room = X[:, bedrooms_ix] / X[:, rooms_ix]
            return np.c_[X, rooms_per_household, population_per_household,
                         bedrooms_per_room]

        else:
            return np.c_[X, rooms_per_household, population_per_household]

attr_adder = CombinedAttributesAdder(add_bedrooms_per_room=False)
housing_extra_attribs = attr_adder.transform(housing.values)
```

4.4 Feature Scaling

- One of the most important transformations you need to apply to your data is feature scaling.
- There are two common ways to get all attributes to have the same scale: min-max scaling and standardization.

- Min-max scaling (many people call this normalization) is the simplest: 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 different: first it subtracts the mean value (so standardized values always have a zero mean), and then it divides by the standard deviation so that the resulting distribution has unit variance. Unlike min-max scaling, standardization does not bound values to a specific range, which may be a problem for some algorithms (e.g., neural networks often expect an input value ranging from 0 to 1). However, standardization is much less affected by outliers.

4.5 Transformation Pipelines

- 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.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler

num_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy="median")),
        ('attribs_adder', CombinedAttributesAdder()),
        ('std_scaler', StandardScaler()),
    ])

housing_num_tr = num_pipeline.fit_transform(housing_num)
```

- So far, we have handled the categorical columns and the numerical columns separately. It would be more convenient to have a single transformer able to handle all columns, applying the appropriate transformations to each column. For this, use `ColumnTransformer`.

```
from sklearn.compose import ColumnTransformer

num_attribs = list(housing_num)
cat_attribs = ["ocean_proximity"]

full_pipeline = ColumnTransformer([
        ("num", num_pipeline, num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

housing_prepared = full_pipeline.fit_transform(housing)
```

5. Select and Train a Model

5.1 Training and Evaluating on the Training Set

- You can train Linear Regression as follows:

```
from sklearn.linear_model import LinearRegression

lin_reg = LinearRegression()
lin_reg.fit(housing_prepared, housing_labels)
```

- We can measure the performance of the model using RMSE.
- You can train other models like Decision Trees or Random Forests.

5.2 Better Evaluation Using Cross-Validation

- Scikit-Learn's K-fold cross-validation feature: The following code randomly splits the training set into 10 distinct subsets called folds, then it trains and evaluates the Decision Tree model 10 times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the 10 evaluation scores:

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(tree_reg, housing_prepared, housing_labels,
                         scoring="neg_mean_squared_error", cv=10)
tree_rmse_scores = np.sqrt(-scores)
```

- You should save every model you experiment with so that you can come back easily to any model you want. Make sure you save both the hyperparameters and the trained parameters, as well as the cross-validation scores and perhaps the actual predictions as well.
- You can easily save Scikit-Learn models by using Python's pickle module or by using the joblib library.

```
import joblib

joblib.dump(my_model, "my_model.pkl")
# and later...
my_model_loaded = joblib.load("my_model.pkl")
```

6. Fine-Tune Your Model

6.1 Grid Search

- All you need to do is tell it which hyperparameters you want it to experiment with and what values to try out, and it will use cross-validation to evaluate all the possible combinations of hyperparameter values.

```python
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)
```

- You can call the methods `best_params_` and `best_estimator_` on the grid search object to get the best parameters and best estimator respectively.

6.2 Randomized Search

- The grid search approach is fine when you are exploring relatively few combinations, like in the previous example, but when the hyperparameter search space is large, it is often preferable to use RandomizedSearchCV instead.
- This class can be used in much the same way as the GridSearchCV class, but instead of trying out all possible combinations, it evaluates a given number of random combinations by selecting a random value for each hyperparameter at every iteration.

6.3 Ensemble Methods

- Another way to fine-tune your system is to try to combine the models that perform best. The group (or “ensemble”) will often perform better than the best individual model (just like Random Forests perform better than the individual Decision Trees they rely on), especially if the individual models make very different types of errors.

6.4 Analyse the Best Models and Their Errors

- You will often gain good insights on the problem by inspecting the best models.
- You should also look at the specific errors that your system makes, then try to understand why it makes them and what could fix the problem (adding extra features or getting rid of uninformative ones, cleaning up outliers, etc.).

6.5 Evaluate Your System on the Test Set

- Run your `full_pipeline` to transform the data (call `transform()`, not `fit_transform()` —you do not want to fit the test set), and evaluate the final model on the test set.

7. Launch, Monitor and Maintain Your System

- You can launch your model within a website.
- You can wrap the model within a dedicated web service that your web application can query through a REST API.
- Another popular strategy is to deploy your model on the cloud, for example on Google Cloud AI Platform (formerly known as Google Cloud ML Engine): just save your model using joblib and upload it to Google Cloud Storage (GCS), then head over to Google Cloud AI Platform and create a new model version, pointing it to the GCS file.
- But deployment is not the end of the story. You also need to write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
- You should probably automate the whole process as much as possible. Here are a few things you can automate:
  - Collect fresh data regularly and label it (e.g., using human raters).
  - Write a script to train the model and fine-tune the hyperparameters automatically. This script could run automatically, for example every day or every week, depending on your needs.
  - Write another script that will evaluate both the new model and the previous model on the updated test set, and deploy the model to production if the performance has not decreased (if it did, make sure you investigate why).
- You should also make sure you evaluate the model's input data quality.
- Finally, make sure you keep backups of every model you create and have the process and tools in place to roll back to a previous model quickly, in case the new model starts failing badly for some reason.
- Having backups also makes it possible to easily compare new models with previous ones.
- As you 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.