Machine Learning Project

Data

- In a pipeline, each component pulls in a large amount of data, processes it, and spits out the result in another data store. Then, sometime later, the next component in the pipeline pulls in this data and spits out its output.

Performance Measure

- A standard measure is the root mean square error (RMSE).
  - RMSE describes how much error the model makes in its predictions.
  - $RMSE(X, h) \ = \sqrt{\frac{1}{m}\sum_{i=1}^{m} h((X^{i}) - y^{i})^{2}}$
    - m = number of instances in the dataset
    - X^i = vector of all the feature values (minus the label) of the ith instance
    - y^i is the label (desired output value for the ith instance)
    - X = matrix containing all the feature values (minus the labels) of all instances in the dataset; one row per instance
    - h = the hypothesis, the system’s prediction function, h(X(i)) = ŷ(i)
    - RMSE(X,h) = the cost function measured on the set of examples using the hypothesis h
- If there are many outliers, then the mean absolute error (MAE) is better.
  - $MAE(X, h) \ = \ \frac{1}{m}\sum_{i=1}^{m} h(X^{i}) - y^{i}$
  - Both the RMSE and the MAE measure the distance between two vectors: the prediction vectors and the target vectors.

Taking a Look at the Data

- We can use **data.info()**, **data.head()**, and **data.describe()** to look at the dataset's structure.
  - describe() displays each attribute's count, mean, min, and max.
- Check whether each attribute is numerical, categorical, or another object type.
- We can use Pyplot’s plot functions to see the data's appearance.

Creating a Test Set

- The training set will comprise 80% of the data, while the test set will contain 20%.

- We can use Scikit-Learn's **train_test_split()** function to split the data.
    - Parameters:
        - Arrays: can be lists, numpy arrays, scipy-sparse matrices, or pandas data frames
        - Test_size: should be between 0.0 and 1.0, which represents the size of the test set
        - Random_state: sets the random generator seed
- Instead of using random sampling, we can use stratified sampling to ensure that the sampling population is representative of the target population.
    - In stratified sampling, the population is divided into homogeneous subgroups called strata, and the right number of instances are sampled from each stratum to guarantee that the test set is representative of the overall population
    - Scikit-Learn's **StratifiedShuffleSplit()** function can create a test and training set that employs stratified sampling.
    - Or, we can simply specify the "stratify" argument in the train_test_split() function with whichever strata or category we want to use.

Looking for Correlations

- We can compute the standard correlation coefficient with the **data.corr()** method.
- The coefficient ranges from -1 to 1, which describes the strength of the linear relationship.
- Pandas's **scatter_matrix()** function plots every numerical attribute against each other.
- It may also be a good idea to combine some attributes, which can give us more useful information.

Preparing the Data for ML Algorithms

- Always keep a clean training set (by copying the original version of the set).
- Clear the data of any missing values using Pandas DataFrame's **dropna(), drop(),** or **fillna()** methods.
- If the feature is categorical, then replace any missing values with the most frequent category.
- We can use imputation to set the missing values to some value (zero, the mean, median, etc.)
    - We can use Scikit-Learn's **SimpleImputer()** function to do this for us.
    - Once we have the imputer, we can use its **fit()** method to fit the imputer instance to the training data.
    - Then, we can use the imputer's **transform()** method to replace any missing values in the training data to the strategy (i.e., the median).
    - Some other imputers include the KNNImputer and the IterativeImputer.

Scikit-Learn’s Design Principles

- All objects share a consistent and simple interface:
    - Estimators
        - Estimate parameters based on a dataset (e.g., a SimpleImputer)
        - The estimation is performed by the fit() method with the dataset and the labels as the parameters.
        - The strategy is a hyperparameter.
    - Transformers
        - Some estimators can transform a dataset, performed by the transform() method.
        - Generally, transformers rely on learned parameters, such as in the case for a SimpleImputer.
    - Predictors
        - Some estimators can make predictions.
        - A LinearRegression model is a predictor.
        - A predictor has the predict() method and a score() method that measures the quality of the predictions.
- Each estimator’s hyperparamerers are accessible via public instance variables (e.g., imputer.strategy). The learned parameters can be accessed similarly (e.g., imputer.statistics_).
- Datasets are represented as NumPy arrays or SciPy-sparse matrices. Hypermarameters are either strings or numbers.

Handling Text and Categorical Attributes

- We can convert categories from text to numbers using encoders.
    - Scikit-Learn’s **OrdinalEncoder()** function can do this for us.
    - However, if we have multiple categories, two nearby values will be more similar than two distant values.
    - Scikit-Learn’s **OneHotEncoder()** allows us to set each attribute to 0 or 1, called one-hot encoding.
    - Then, we can use the encoder’s fit and transform methods to train the categorical dataset.
- When you fit any Scikit-Learn estimator using a DataFrame, the estimator stores the column names in the **feature_names_in_ attribute**.
- Transformers also provide a **get_feature_names_out()** method that you can use to build a DataFrame around the transformer’s output.

Feature Scaling and Transformation

- We must ensure that the dataset’s scaling (e.g., if one attribute’s values range from 6-40000 and another’s range from 0-15) does not affect its predictions. We can standardize or normalize the dataset.
    - Normalization/Min-max scaling

        - For each attribute, the values are shifted and rescaled so that they end up ranging from 0 to 1.
        - This is performed by subtracting the min value and dividing by the difference between the min and the max.
        - Scikit-Learn provides the **MinMaxScaler()** function to perform this process.
    - Standardization
        - First, it subtracts the mean value (so standardized values have a zero mean). It divides the result by the standard deviation (so standardized values have a standard deviation equal to 1).
        - Standardization does not restrict values to a specific range, but it is much less affected by outliers.
        - We can use Scikit-Learn's **StandardScaler()** function.
    - As always, we can transform any set once we have our scaler. However, only fit or fit_transform the training set.
- We also have to handle distributions that have a heavy tail or skew.
    - Scalers will squash most values into small ranges. To avoid this, we shrink the tail by making the distribution roughly symmetrical.
    - We can replace the feature with its square root or its logarithm.
    - We can also "bucketize" the feature, which means chopping its distribution into roughly equal-sized buckets and replacing each feature value with the index of the bucket it belongs to, like its percentile.
- Furthermore, we have to handle features that have a multimodal distribution.
    - We can bucketize the feature by treating the buckets as categories by using a one-hot encoder.
    - We can also use a radial basis function (RDF) to add a feature representing the similarity between the feature and the mode. We can use Scikit-Learn's **rbf_kernel()** function for this.
- Lastly, we may have to transform the target values as well.
    - We can use the logarithmic function of the target to handle heavy tails, but then, the model would predict the log of our targeted value.
    - Scikit-Learn's **inverse_transform()** function allows us to compute the inverse of their transformation.
    - We can also use the **TransformedTargetRegressor()** function to accomplish the same task.
- We can build custom transformers using Scikit-Learn's **FunctionTransformer()** function.

Transformation Pipelines

- Pipelines can help us transform datasets in the right order. We can use Scikit-Learn's **Pipeline()** function.
    - The pipeline takes in a list of name/estimator tuples. The estimators must all be transformers (i.e., they have a fit_transform() method.)
    - We can also use the **make_pipeline()** function to make the pipeline.

    - The **ColumnTransformer()** function allows us to handle categorical columns and numerical columns at the same time.

Selecting and Training a Model

- Choose a model depending on the task. For example, we can use the **LinearRegression()** function for a linear regression model.
- Use the function's fit() and **predict()** methods.
- Then, we can evaluate the model using performance measures.
    - The **mean_squared_error()** function employs the RMSE measure. A lower value signifies that the model predicts well.
    - We can use the **DecisionTreeRegressor()** function to find complex non-linear relationships in the data.

Cross-Validation

- We can use Scikit-Learn's **cross_val_score()** function to perform k-fold cross-validation (randomly splits the training set into n nonoverlapping subsets called folds, then it trains and evaluates the decision tree model n times, picking a different fold for evaluation every time and using the other n-1 folds for training.)
    - Note that this returns the utility function, so to get the cost function, simply convert the score to negative.
    - If the training error is low but the validation error is high, then the model is overfitting.
- We can use the **RandomForestRegressor()** function instead of a single tree.
    - Random forests train many decision trees on random subsets of the features and averaging out their predictions.
    - These models consist of many other models called ensembles.

Fine-Tuning the Model

- We can experiment with different hyperparameter combinations to find the best one.
    - The **GridSearchCV()** function will try different combinations based on the parameters you pass into the function. Get the best parameters with the **best_params_** value.
    - The **RandomizedSearchCV()** function evaluates a fixed number of combinations, selecting a random value for each hyperparameter. This function is preferable when the dataset is too large.

Analyzing the Best Models

- We can use the **SelectFromModel()** function to drop the least useful features in the dataset automatically. We can train this model using the fitting function and look at its **feature_importances_** attribute. When we call transform, the function drops the other features.

Evaluating the System on the Test Set

- Now, we can finally run the model on the test set.
- We simply call the model's predict() method. We can always check the RMSE using the mean_squared_error() function.
- We can compute a 95% confidence interval using Scipy's **stats.t.interval()** method.

Launching, Monitoring, and Maintaining the System

- We can store our final model using joblib's **dump()** method. The model can later be loaded using the **load()** method and use it to make further predictions.
- We can upload the model to a website or a cloud system, like Google's Vertex AI.
- We also need to check how the model performs over time. This can be done by using performance measures or using human raters.
- Lastly, we need to collect new data and possibly retrain the model so its accuracy does not decay.