

# Kaggle Challenge - Machine Learning Models

## We'll cover the following ^

- 4. Create and Assess Machine Learning Models
  - Train and Evaluate Multiple Models on the Training Set
    - Comparative analysis of the models and their errors
    - Evaluation Using Cross-Validation
  - Jupyter Notebook

## 4. Create and Assess Machine Learning Models #

### Train and Evaluate Multiple Models on the Training Set #

At last! We framed the problem, we got the data, explored it, prepared the data, and wrote transformation pipelines to clean up the data for machine learning algorithms automatically. We are now ready for the most exciting part: to select and train a machine learning model.

The great news is that thanks to all the previous steps, things are going to be way simpler than you might think! Scikit-learn makes it all very easy!

#### Create a Test Set

As a first step we are going to split our data into two sets: training set and test set. We are going to train our model only on part of the data because we need to keep some of it aside in order to evaluate the quality of our model.

Creating a test set is quite simple: the most common approach is to pick some instances randomly, typically 20% of the dataset, and set them aside. The simplest function for doing this Scikit-learn's `train_test_split()`.

It is a common convention to name the feature set with  $X$  in the name,  $X_{train}$  and  $X_{test}$ , and the data with the variable to be predicted with  $y$  in the name

and `X_test`, and the data with the variable to be predicted with `y` in the name, `y_train` and `y_test`:

```
1 # Split data into train and test formate
2 from sklearn.model_selection import train_test_split
3
4 X_train, X_test, y_train, y_test = train_test_split(housing_X_prepared, housing_y_prepared,
```



```
In [42]: # Split data into train and test formate
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(housing_X_prepared, housing_y_prepared, test_size=0.2, random_state=7)
```

With the training and test data in hand, creating a model is really easy. Say we want to create a Linear Regression model. In general, this is what it looks like:

```
# Import modules
from sklearn.linear_model import LinearRegression

# Train the model on training data
model = LinearRegression()
model.fit(X_train, y_train)

# Evaluate the model on test data
print("Accuracy:", model.score(X_test, y_test)*100)
```



**And that's it! There you have a linear regression model in three lines of code!**

Now we want to create and compare multiple models, so we are going to store the results from the evaluation of each model in a variable. Since we are dealing with a regression problem, we are also going to use *RMSE* as the main performance measure to assess the quality of our models.

**RMSE (Root Mean Square Error)** is a typical performance measure for regression problems. It gives an idea of how much error the system typically makes in its predictions by measuring the differences between values predicted by the model and the actual values, actual prices vs predicted prices. It is the standard deviation of the prediction errors, a measure of how spread out these errors are from the line of best fit.

The equation for RMSE is simple: we sum the square of all the errors between predicted values and actual values, we divide by the total number of test examples and then we take the square root of the results:

$$\text{RMSE} = \sqrt{\sum \frac{(y_{\text{pred}} - y_{\text{ref}})^2}{N}}$$

Again, not to worry about implementing formulas, because we are going to measure RMSE of our regression models using Scikit-learn's `mean_squared_error` function.

One more thing to remember is that we took the log of our target variable, *SalePrice*. This means that before evaluating RMSE, we need to convert prices back to their original values. Inverse of the log means to simply take the exponential of the log values, i.e., we will simply call `np.exp()`. And since we need to get the inverse multiple times, we are going to write a function as a good coding practice, like so:

```
def inv_y(y):
    return np.exp(y)
```

Let's train our models:

```
from sklearn.metrics import mean_squared_error

from sklearn.linear_model import Lasso
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
from xgboost import XGBRegressor
import xgboost

# Invert the log-transformed value
def inv_y(transformed_y):
    return np.exp(transformed_y)

# Series to collect RMSE for the different algorithms: "algorithm name + rmse"
rmse_compare = pd.Series()
rmse_compare.index.name = 'Model'

# Series to collect accuracy scores for the different algorithms: "algorithm name + score"
scores_compare = pd.Series()
scores_compare.index.name = 'Model'

# Model 1: Linear Regression =====
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)

linear_val_predictions = linear_model.predict(X_test)
linear_val_rmse = mean_squared_error(inv_y(linear_val_predictions), inv_y(y_test))
```

```

linear_val_rmse = mean_squared_error(inv_y(linear_val_predictions), inv_y(y_test))
linear_val_rmse = np.sqrt(linear_val_rmse)
rmse_compare['LinearRegression'] = linear_val_rmse

lr_score = linear_model.score(X_test, y_test)*100
scores_compare['LinearRegression'] = lr_score

# Model 2: Decision Trees. Define the model. =====
dtree_model = DecisionTreeRegressor(random_state=5)
dtree_model.fit(X_train, y_train)

dtree_val_predictions = dtree_model.predict(X_test)
dtree_val_rmse = mean_squared_error(inv_y(dtree_val_predictions), inv_y(y_test))
dtree_val_rmse = np.sqrt(dtree_val_rmse)
rmse_compare['DecisionTree'] = dtree_val_rmse

dtree_score = dtree_model.score(X_test, y_test)*100
scores_compare['DecisionTree'] = dtree_score

# Model 3: Random Forest. Define the model. =====
rf_model = RandomForestRegressor(random_state=5)
rf_model.fit(X_train, y_train)

rf_val_predictions = rf_model.predict(X_test)
rf_val_rmse = mean_squared_error(inv_y(rf_val_predictions), inv_y(y_test))
rf_val_rmse = np.sqrt(rf_val_rmse)
rmse_compare['RandomForest'] = rf_val_rmse

rf_score = rf_model.score(X_test, y_test)*100
scores_compare['RandomForest'] = rf_score

# Model 4: Gradient Boosting Regression =====
gbr_model = GradientBoostingRegressor(n_estimators=300, learning_rate=0.05,
                                     max_depth=4, random_state=5)
gbr_model.fit(X_train, y_train)

gbr_val_predictions = gbr_model.predict(X_test)
gbr_val_rmse = mean_squared_error(inv_y(gbr_val_predictions), inv_y(y_test))
gbr_val_rmse = np.sqrt(gbr_val_rmse)
rmse_compare['GradientBoosting'] = gbr_val_rmse

gbr_score = gbr_model.score(X_test, y_test)*100
scores_compare['GradientBoosting'] = gbr_score

```

```
In [43]: from sklearn.metrics import mean_squared_error

from sklearn.linear_model import Lasso
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import ElasticNet
from sklearn.neighbors import KNeighborsRegressor
from sklearn.svm import SVR
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.ensemble import AdaBoostRegressor
from sklearn.tree import DecisionTreeRegressor
from xgboost import XGBRegressor
import xgboost

# Invert the log-transformed value
def inv_y(transformed_y):
    return np.exp(transformed_y)

# Series to collect RMSE for the different algorithms: "algorithm name + rmse"
rmse_compare = pd.Series()
rmse_compare.index.name = 'Model'

# Series to collect accuracy scores for the different algorithms: "algorithm name + score"
scores_compare = pd.Series()
scores_compare.index.name = 'Model'

# Model 1: Linear Regression =====
linear_model = LinearRegression()
linear_model.fit(X_train, y_train)

linear_val_predictions = linear_model.predict(X_test)
linear_val_rmse = mean_squared_error(inv_y(linear_val_predictions), inv_y(y_test))
linear_val_rmse = np.sqrt(linear_val_rmse)
rmse_compare['LinearRegression'] = linear_val_rmse

lr_score = linear_model.score(X_test, y_test)*100
scores_compare['LinearRegression'] = lr_score
```

```
In [44]: # Model 2: Decision Trees. Define the model. =====
dtree_model = DecisionTreeRegressor(random_state=5)
dtree_model.fit(X_train, y_train)

dtree_val_predictions = dtree_model.predict(X_test)
dtree_val_rmse = mean_squared_error(inv_y(dtree_val_predictions), inv_y(y_test))
dtree_val_rmse = np.sqrt(dtree_val_rmse)
rmse_compare['DecisionTree'] = dtree_val_rmse

dtree_score = dtree_model.score(X_test, y_test)*100
scores_compare['DecisionTree'] = dtree_score
```

```
In [45]: # Model 3: Random Forest. Define the model. =====
rf_model = RandomForestRegressor(random_state=5)
rf_model.fit(X_train, y_train)

rf_val_predictions = rf_model.predict(X_test)
rf_val_rmse = mean_squared_error(inv_y(rf_val_predictions), inv_y(y_test))
rf_val_rmse = np.sqrt(rf_val_rmse)
rmse_compare['RandomForest'] = rf_val_rmse

rf_score = rf_model.score(X_test, y_test)*100
scores_compare['RandomForest'] = rf_score
```

```
In [46]: # Model 4: Gradient Boosting Regression =====
gbr_model = GradientBoostingRegressor(n_estimators=300, learning_rate=0.05,
                                     max_depth=4, random_state=5)
gbr_model.fit(X_train, y_train)

gbr_val_predictions = gbr_model.predict(X_test)
gbr_val_rmse = mean_squared_error(inv_y(gbr_val_predictions), inv_y(y_test))
gbr_val_rmse = np.sqrt(gbr_val_rmse)
rmse_compare['GradientBoosting'] = gbr_val_rmse

gbr_score = gbr_model.score(X_test, y_test)*100
scores_compare['GradientBoosting'] = gbr_score
```

**We have trained four different models.** As you can see, training from one model to another just means that you just select a different one from Scikit-Learn's library and change a single line of code!

Comparative analysis of the models and their errors #

Now let's get the performance measures for our models in sorted order, from best to worst:

```
print('RMSE values for different algorithms:')
rmse_compare.sort_values(ascending=True).round()
```



```
print('Accuracy scores for different algorithms:')
scores_compare.sort_values(ascending = False).round(3)
```



```
In [47]: print('RMSE values for different algorithms:')
rmse_compare.sort_values(ascending=True).round()
```

RMSE values for different algorithms:

```
Out[47]: Model
LinearRegression    24637.0
GradientBoosting    27212.0
RandomForest        31491.0
DecisionTree        37872.0
dtype: float64
```

```
In [48]: print('Accuracy scores for different algorithms:')
scores_compare.sort_values(ascending = False).round(3)
```

Accuracy scores for different algorithms:

```
Out[48]: Model
LinearRegression    89.591
GradientBoosting    89.567
RandomForest        84.796
DecisionTree        72.805
dtype: float64
```

The simplest model, Linear Regression, seems to be performing the best, with predicted prices that are off by about 24K. This might or might not be an acceptable amount of deviation depending on the desired level of accuracy or the metric we are trying to optimize based on our business objective.

## General Notes

A large prediction error usually means an example of a model **underfitting** the training data. When this happens it can mean that the features do not provide enough information to make good predictions, or that the model is not powerful enough. The main ways to fix underfitting are to select a more powerful model, to feed the training algorithm with better features, or to reduce the constraints on the model.

In this case, we have trained more powerful models, capable of finding complex nonlinear relationships in the data, like a DecisionTreeRegressor as well. However, the more powerful model seems to be performing worse! The Decision Tree model is overfitting badly enough to perform even worse than the simpler Linear Regression model.

Possible solutions to deal with **overfitting** are to simplify the model, constrain it, or get more training data.

Random Forests work by training many Decision Trees on random subsets of the features, then averaging out their predictions. Building a model on top of many other models is called **Ensemble Learning**, and it is used to improve the performance of the algorithms. In fact, we can see that Random Forests are performing much better than Decision Trees.

## Evaluation Using Cross-Validation #

One way to evaluate models is to split the training set into a smaller training set and a validation set, then train the models against the smaller training set and evaluate them against the validation set. This is called cross-validation. We can use Scikit-Learn's cross-validation feature, `cross_val_score`, for this.

Let's perform a **K-fold cross-validation** on our best model: the cross-validation function randomly splits the training set into  $K$  distinct subsets or folds, then it trains and evaluates the model  $K$  times, picking a different fold for evaluation every time and training on the other 9 folds. The result is an array containing the  $K$  evaluation scores:

```
from sklearn.model_selection import cross_val_score

# Perform K fold cross-validation, where K=10
scores = cross_val_score(linear_model, X_train, y_train,
                          scoring="neg_mean_squared_error", cv=10)
linear_rmse_scores = np.sqrt(-scores)

# Display results
def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(linear_rmse_scores)
```

```
In [49]: from sklearn.model_selection import cross_val_score
scores = cross_val_score(linear_model, X_train, y_train,
                          scoring="neg_mean_squared_error", cv=10)
linear_rmse_scores = np.sqrt(-scores)

def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(linear_rmse_scores)

Scores: [0.11254073 0.13883274 0.10564025 0.12889019 0.10899275 0.11501349
0.10957026 0.11747952 0.13342401 0.11532405]
Mean: 0.11857079770059828
Standard deviation: 0.010669351420601209
```

```
In [50]: scores = cross_val_score(rf_model, X_train, y_train,
                                  scoring="neg_mean_squared_error", cv=10)
rf_rmse_scores = np.sqrt(-scores)

def display_scores(scores):
    print("Scores:", scores)
    print("Mean:", scores.mean())
    print("Standard deviation:", scores.std())

display_scores(rf_rmse_scores)

Scores: [0.13156435 0.17088973 0.12085582 0.16462507 0.13179427 0.14031469
0.14391635 0.11671486 0.15015448 0.14648264]
Mean: 0.14173122731360774
Standard deviation: 0.016548003161403323
```

From the results, we notice that cross-validation gives us the mean and standard deviation for the scores as well. But cross-validation comes at the cost of training the model several times, so it is not always the most viable choice.

✎ **Note:** In general, save your models so that you can come back to any model you want. Make sure to save the hyperparameters, the trained parameters, and also the evaluation scores. Why? Because this will allow you to easily compare scores across model types and compare the types of errors they make. This will especially be useful when the problem is complex, your notebook is huge and/or model training time is very large.

Scikit-learn models can be saved easily using the `pickle` module, or using `sklearn.externals.joblib`, which is more efficient at serializing large NumPy arrays:

```
from sklearn.externals import joblib

# Save model
joblib.dump(my_model, "my_model.pkl")





# Load saved model
my_model_loaded = joblib.load("my_model.pkl")
```



# Jupyter Notebook #

You can see the instructions running in the Jupyter Notebook below:

## How to Use a Jupyter Notebook?

- Click on “**Click to Launch**”  button to work and see the code running live in the notebook.
- You can click  to open the **Jupyter Notebook in a new tab**.
- Go to files and click *Download as* and then choose the format of the file to **download** . You can choose Notebook(.ipynb) to download the file and work locally or on your personal Jupyter Notebook.
-  The notebook **session expires after 30 minutes of inactivity**. It will reset if there is no interaction with the notebook for 30 consecutive minutes.

Your app can be found at: <https://5lrqw92v88k2y-live-app.educative.run/notebooks/MachineLearningModels.ipynb>



Click to launch app!

