TEAM ID	PNT2022TMID41770
TOPIC	Model Performance Metries

Like any digital product, an AI product's success should be determined by its profit contribution. Business performance indicators such as operating cash flow (OCF) or the monthly recurring revenue (MRR) are suitable for measuring the product's contribution to the company's success.

But in order to quantify the accuracy of the underlying algorithm or the interaction of the AI product with the user, business indicators are not enough. This requires operational KPIs and proxy metrics for data driven decision making:

Regression metrics

Regression models have continuous output. So, we need a metric based on calculating some sort of distance between *predicted* and *ground truth.*

In order to evaluate Regression models, we'll discuss these metrics in detail:

- Mean Absolute Error (MAE),
- Mean Squared Error (MSE),
- Root Mean Squared Error (RMSE),
- R^2 (R-Squared).

Note: We'll use the Boston Housing dataset to implement regressive metrics. You can find the notebook containing all the code used in this blog here.

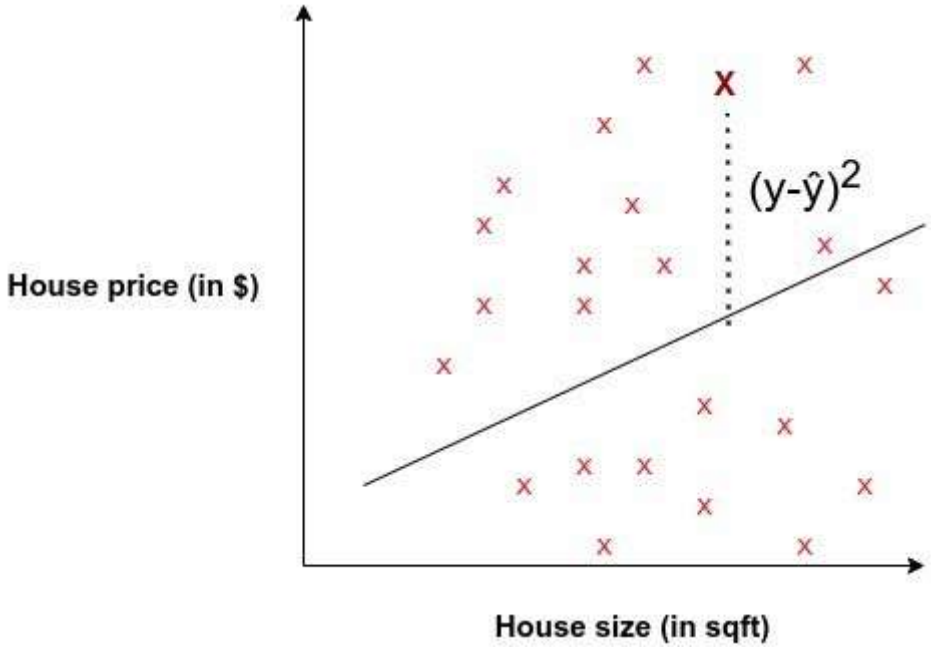
Mean Squared Error (MSE)

Mean squared error is perhaps the most popular metric used for regression problems. It essentially finds the average of the squared difference between the target value and the value predicted by the regression model.

$$MSE = \frac{1}{N}\sum_{j=1}^{N}(y_j - \check{y}_j)^2$$

Where:

- y_j: ground-truth value
- y_hat: predicted value from the regression model
- N: number of datums



Few key points related to MSE:

- It's differentiable, so it can be optimized better.
- It penalizes even small errors by squaring them, which essentially leads to an overestimation of how bad the model is.

- Error interpretation has to be done with squaring factor(scale) in mind. For example in our Boston Housing regression problem, we got MSE=21.89 which primarily corresponds to (Prices)².
- Due to the squaring factor, it's fundamentally more prone to outliers than other metrics.

This can be implemented simply using NumPy arrays in Python.

```
mse = (y-y_hat)**2

print(f"MSE: {mse.mean():0.2f} (+/- {mse.std():0.2f})")
```

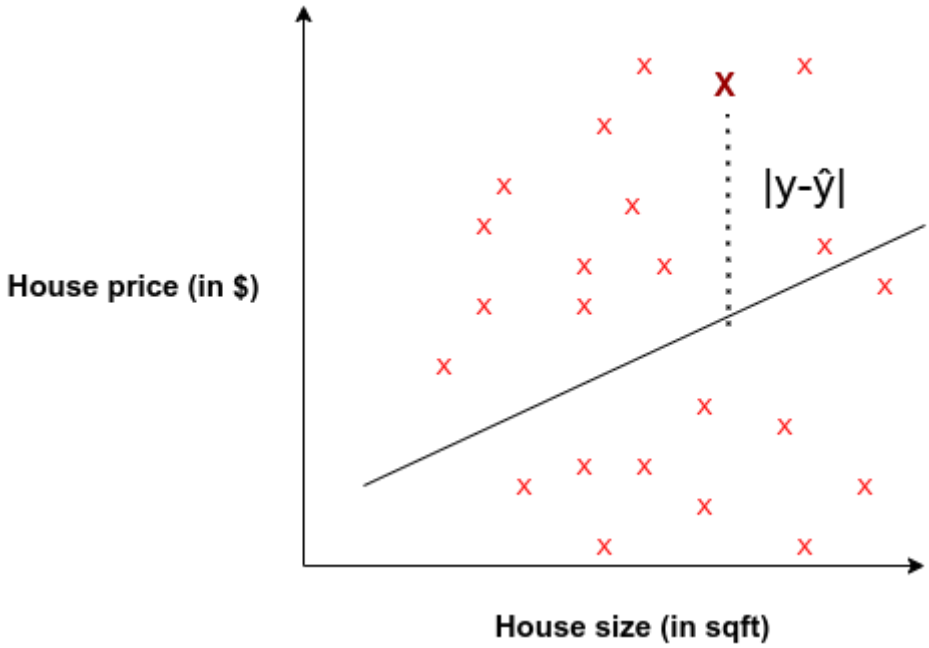
Mean Absolute Error (MAE)

Mean Absolute Error is the average of the difference between the ground truth and the predicted values. Mathematically, its represented as :

$$MAE = \frac{1}{N}\sum_{j=1}^{N}|y_j - \check{y}_j|$$

Where:

- y_j: ground-truth value
- y_hat: predicted value from the regression model
- N: number of datums



Few key points for MAE

- It's more robust towards outliers than MAE, since it doesn't exaggerate errors.
- It gives us a measure of how far the predictions were from the actual output. However, since MAE uses absolute value of the residual, it doesn't give us an idea of the direction of the error, i.e. whether we're under-predicting or over-predicting the data.
- Error interpretation needs no second thoughts, as it perfectly aligns with the original degree of the variable.
- MAE is non-differentiable as opposed to MSE, which is differentiable.

Similar to MSE, this metric is also simple to implement.

```
mae = np.abs(y-y_hat)


print(f"MAE: {mae.mean():0.2f} (+/- {mae.std():0.2f})")
```

Root Mean Squared Error (RMSE)

Root Mean Squared Error corresponds to the square root of the average of the squared difference between the target value and the value predicted by the regression model. Basically, sqrt(MSE). Mathematically it can be represented as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(y_j - \check{y}_j)^2}$$

It addresses a few downsides in MSE.

Few key points related to RMSE:

- It retains the differentiable property of MSE.
- It handles the penalization of smaller errors done by MSE by square rooting it.
- Error interpretation can be done smoothly, since the scale is now the same as the random variable.
- Since scale factors are essentially normalized, it's less prone to struggle in the case of outliers.

Implementation is similar to MSE:

```
mse = (y-y_hat)**2

rmse = np.sqrt(mse.mean())

print(f"RMSE: {rmse:0.2f}")
```

R² Coefficient of determination

R² Coefficient of determination actually works as a post metric, meaning it's a metric that's calculated using other metrics.

The point of even calculating this coefficient is to answer the question **"How much (what %) of the total variation in Y(target) is explained by the variation in X(regression line)"**

This is calculated using the sum of squared errors. Let's go through the formulation to understand it better.

Total variation in Y (Variance of Y):

$$SE(\bar{Y}) = \sum_{i=1}^{N}(y_i - \bar{y})^2$$

Percentage of variation described the regression line:

$$\frac{SE(line)}{SE(\bar{Y})}$$

Subsequently, the percentage of variation described the regression line:

$$1 - \frac{SE(line)}{SE(\bar{Y})}$$

Finally, we have our formula for the coefficient of determination, which can tell us how good or bad the fit of the regression line is:

$$coeff(R^2) = 1 - \frac{SE(line)}{SE(\bar{Y})}$$

This coefficient can be implemented simply using NumPy arrays in Python.

```
# R^2 coefficient of determination

SE_line = sum((y-y_hat)**2)

SE_mean = sum((y-y.mean())**2)

r2 = 1-(SE_line/SE_mean)

print(f"R^2 coefficient of determination:
{r2*100:0.2f}%")
```

Few intuitions related to R^2 results:

- If the sum of Squared Error of the regression line is small => R^2 will be close to 1 (Ideal), meaning the regression was able to capture 100% of the variance in the target variable.
- Conversely, if the sum of squared error of the regression line is high => R^2 will be close to 0, meaning the regression wasn't able to capture any variance in the target variable.
- You might think that the range of R^2 is (0,1) but it's actually ($-\infty$,1) because the ratio of squared errors of the regression line and mean can surpass the value 1 if the squared error of regression line is too high (>squared error of the mean).

Adjusted R^2

The Vanilla R^2 method suffers from some demons, like misleading the researcher into believing that the model is improving when the score is increasing but in reality, the learning is not happening. This can happen when a model overfits the data, in that case the variance explained will be 100% but the learning hasn't happened. To rectify this, R^2 is adjusted with the number of independent variables.

Adjusted R^2 is always lower than R^2, as it adjusts for the increasing predictors and only shows improvement if there is a real improvement.

$$R_a^2 = 1 - [(\frac{n-1}{n-k-1}).(1-R^2)]$$

Where:

- n = number of observations
- k = number of independent variables
- Ra^2 = adjusted R^2

Classification metrics

Classification problems are one of the world's most widely researched areas. Use cases are present in almost all production and industrial environments. Speech recognition, face recognition, text classification – the list is endless.

Classification models have discrete output, so we need a metric that compares discrete classes in some form. *Classification Metrics* evaluate a model's performance and tell you how good or bad the classification is, but each of them evaluates it in a different way.

MAY INTEREST YOU

24 Evaluation Metrics for Binary Classification (And When to Use Them)

So in order to evaluate Classification models, we'll discuss these metrics in detail:

- Accuracy
- Confusion Matrix (not a metric but fundamental to others)
- Precision and Recall
- F1-score
- AU-ROC

Note: We're gonna use the UCI Breast cancer dataset to implement classification metrics. You can find the notebook containing all the code used in this blog here.

Accuracy

Classification accuracy is perhaps the simplest metric to use and implement and is defined as the **number of correct predictions divided by the total number of predictions,** multiplied by 100**.**

We can implement this by comparing ground truth and predicted values in a loop or simply utilize the scikit-learn module to do the heavy lifting for us (not so heavy in the case).

CHECK ALSO

F1 Score vs ROC AUC vs Accuracy vs PR AUC: Which Evaluation Metric Should You Choose?

Start by just importing the *accuracy_score* function from the *metrics* class.

```
from sklearn.metrics import accuracy_score
```

Then, just by passing the ground truth and predicted values, you can determine the accuracy of your model:

```
print(f'Accuracy Score is
{accuracy_score(y_test,y_hat)}')
```

Confusion Matrix

Confusion Matrix is a tabular visualization of the **ground-truth labels versus model predictions**. Each row of the confusion matrix represents the instances in a predicted class and each column represents the instances in an actual class. Confusion Matrix is not exactly a performance metric but sort of a basis on which other metrics evaluate the results.

In order to understand the confusion matrix, we need to set some value for the null hypothesis as an assumption. For example, from our Breast Cancer data, let's assume our **Null Hypothesis H^0** be "*The individual has cancer*".

		Predicted	
		Has Cancer	Doesn't Have Cancer
Ground Truth	Has Cancer	TP	FP
	Doesn't Have Cancer	FN	TN

Confusion Matrix for H^0

Each cell in the confusion matrix represents an evaluation factor. Let's understand these factors one by one:

- **True Positive(TP)** signifies how many positive class samples your model predicted correctly.
- **True Negative(TN)** signifies how many negative class samples your model predicted correctly.
- **False Positive(FP)** signifies how many negative class samples your model predicted incorrectly. This factor represents **Type-I error** in statistical nomenclature. This error positioning in the confusion matrix depends on the choice of the null hypothesis.
- **False Negative(FN)** signifies how many positive class samples your model predicted incorrectly. This factor represents **Type-II** error in statistical nomenclature. This error positioning in the confusion matrix also depends on the choice of the null hypothesis.

We can calculate the cell values using the code below:

```
def find_TP(y, y_hat):

    # counts the number of true positives (y = 1, y_hat
= 1)

    return sum((y == 1) & (y_hat == 1))

def find_FN(y, y_hat):

    # counts the number of false negatives (y = 1, y_hat
= 0) Type-II error

    return sum((y == 1) & (y_hat == 0))
```

```
def find_FP(y, y_hat):

   # counts the number of false positives (y = 0, y_hat
= 1) Type-I error

   return sum((y == 0) & (y_hat == 1))

def find_TN(y, y_hat):

   # counts the number of true negatives (y = 0, y_hat
= 0)

   return sum((y == 0) & (y_hat == 0))
```

We'll look at the Confusion Matrix in two different states using two sets of hyper-parameters in the Logistic Regression Classifier.

```
from sklearn.linear_model import LogisticRegression

clf_1 = LogisticRegression(C=1.0,
class_weight={0:100,1:0.2}, dual=False,
fit_intercept=True,

                  intercept_scaling=1, l1_ratio=None,
max_iter=100,

                  multi_class='auto', n_jobs=None,
penalty='l2',

                  random_state=None, solver='lbfgs',
tol=0.0001,       verbose=0,

                  warm_start=False)

clf_2 = LogisticRegression(C=1.0,
class_weight={0:0.001,1:900}, dual=False,
fit_intercept=True,
```

```
                   intercept_scaling=1, l1_ratio=None,
max_iter=100,

                   multi_class='auto', n_jobs=None,
penalty='l2',

                   random_state=None, solver='lbfgs',
tol=0.0001, verbose=0,

                   warm_start=False)
```