5

Model Evaluation

Test-driven Machine Learning

5.1
Evaluation and **Validation**.

Why do we need to evaluate models?

- To avoid overfitting/underfitting
- To ensure a model's prediction are reliable
- To evaluate how a model may work in real-world scenarios
- Aid engineers to make decisions about model deployment

How do we evaluate models?

- Any evaluation we make has to be an objective one
- There are several metrics to evaluate a model, for example:
  - Accuracy
  - Precision
  - Recall
  - F1-Score

5.2
Overfitting / Underfitting.

Understanding Overfitting

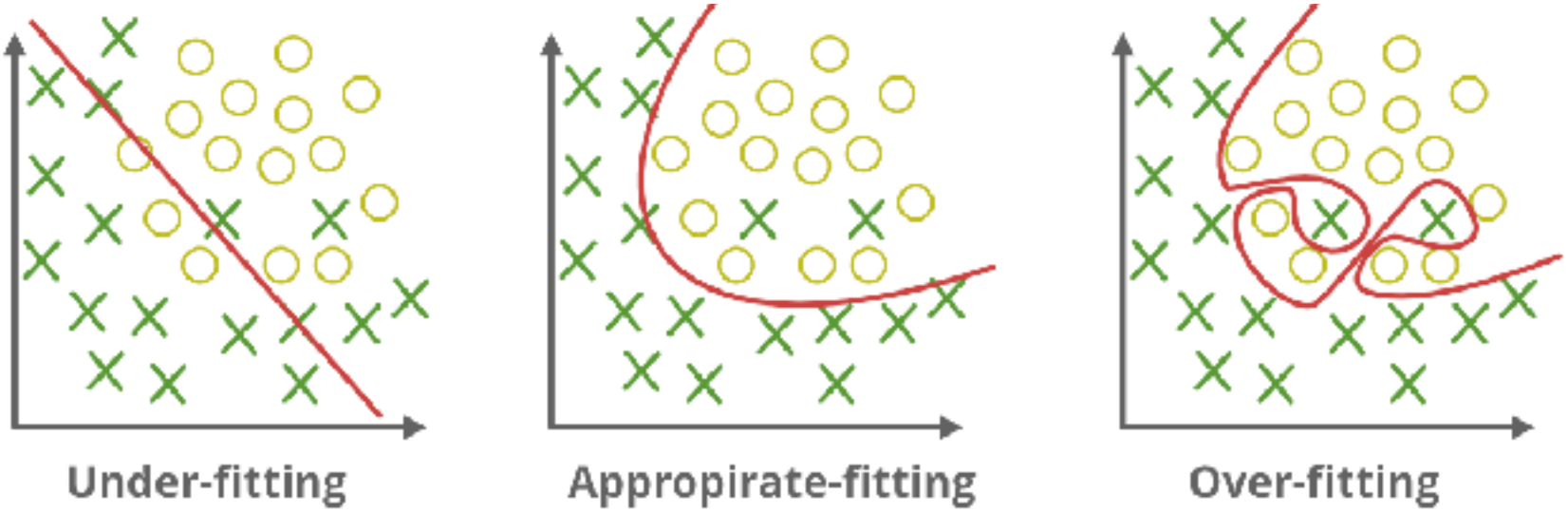
- Overfitting is when model memorizes training data
- This is the opposite of learning a pattern and generalizing
- We can identify if our model is overfitted if:
  - There is high performance on training data
  - There is poor performance on test data

Understanding Underfitting

- Underfitting happens when the model is too simple
- The underlying data patterns have not been captured
- We can identify if our model has underfitting if:
  - Low performance on training data
  - Low performance on test data

The Perfect Fit

- A perfect fit can be achieved
  - By first constructing good evaluation metrics to give us feedback on model performance
  - Then tuning hyper-parameters till the performance improves across the board


Under-fitting
Appropirate-fitting
Over-fitting

5.3
Classification
Performance **Metrics.**

Accuracy

- Accuracy is calculated by dividing the **total correct predictions** with the **total number of predictions**
- Ideal Usage
  - When class distributions are balanced
  - Ex: Determining the winner in a sports match

```
# Python Implementation

from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_true, y_pred)
```

Precision

- Precision counts the **true positives** out of all the items **predicted to be positive**
- Ideal Usage
  - When the cost of false positive is too high
  - Ex: Email spam detection

```
# Python Implementation

from sklearn.metrics import precision_score
precision = precision_score(y_true, y_pred)
```

Recall

- Recall counts how many of the true positive items were correctly classified
- Ideal Usage
  - When the missing a positive is too costly
  - Ex: Disease diagnosis

```
# Python Implementation

from sklearn.metrics import recall_score
recall = recall_score(y_true, y_pred)
```

F1-Score

- The average of precision and recall
- Ideal Usage
  - When a balance between precision and recall is required, especially when there is class imbalance
  - Ex: Content Moderation System

```
# Python Implementation

from sklearn.metrics import f1_score
f1 = f1_score(y_true, y_pred)
```

Python Implementation

```
from sklearn.linear_model import LogisticRegression
from sklearn.datasets import load_breast_cancer

# Load a sample dataset
X, y = load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

# Train the model
clf = LogisticRegression(max_iter=5000)
clf.fit(X_train, y_train)

# Make predictions on test data
predictions = clf.predict(X_test)
```

5.4
Regression
Performance **Metrics.**

Mean Absolute Error (MAE)

- The average of the absolute differences between the predicted values and actual values
- Ideal Usage: When you want to understand the magnitude of error without regard to direction, especially in contexts where large errors aren't more significant than small ones

```
# Python Implementation

from sklearn.metrics import mean_absolute_error
mae = mean_absolute_error(y_true, y_pred)
```

Mean Squared Error (MAE)

- The average of the squared differences between the predicted values and actual values
- Ideal Usage: When larger errors are particularly undesirable and should be penalized more. Such as stock market predictions

```
# Python Implementation

from sklearn.metrics import mean_squared_error
mse = mean_squared_error(y_true, y_pred)
```

Root Mean Squared Error (RMSE)

- The square root of MSE, offering error magnitude in the same units as the predicted values
- Ideal Usage: When you want to interpret the error in the original unit and penalize larger errors. Such as predicting the price of houses

```
# Python Implementation

import numpy as np
rmse = np.sqrt(mse)  # assuming mse is already calculated
```

R-squared (Coefficient of Determination)

- Represents the proportion of variance for the dependent variable that's explained by independent variables
- Ideal Usage: When you want to understand the proportion of the dataset's variability captured by the model. Such as how much variance in exam scores are explained by hours studied

```
# Python Implementation

from sklearn.metrics import r2_score
r2 = r2_score(y_true, y_pred)
```

Adjusted R-squared

- Modifies R-squared to account for the number of predictors in the model, penalizing excessive use of features
- Example: Predicting a car's fuel efficiency based on its attributes. Using too many irrelevant attributes might inflate R-squared, but adjusted R-squared will provide a more tempered view

```
n = len(y_true)  # number of samples
p = X.shape[1]   # number of predictors/features
adjusted_r2 = 1 - (1 - r2) * (n - 1) / (n - p - 1)  # assuming r2 is calculated from the previous slide
```

Mean Bias Deviation (MBD)

- The average difference between the predicted and actual values, indicating the direction of the error
- Ideal Usage: When you're interested in the direction of the error (overestimation vs. underestimation). Helps vary it over or under the limits set

```
# Python Implementation

mbd = np.mean((y_true - y_pred) / y_true) * 100  # Percentage error
```

END.