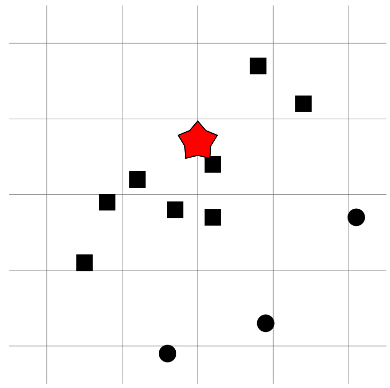

1. K-Nearest Neighbors (KNN) Review

- Pick k-nearest neighbors.
- Easy, intuitive, and effective.
- Not suitable for random data.
- Assumptions:
  - There is a pattern to discern.
  - Computationally, the algorithm calculates distance, sorts, and selects k nearest neighbors.
  - Data structures to facilitate this process include k-d trees and ball trees.
  - Example: Is the star a cat or a dog?
- k is a hyperparameter:
  - A hyperparameter is not learned or trained by the model.
  - You select the "best" k by tuning and training models on various different values of k.
  - Examples:
    - Cross-validation
    - Grid search

Cross-Validation

- What determines the split? The amount of data you have.
- Most common splits are 80/20 or 70/30.
- The more training data you have, the better the model performance.

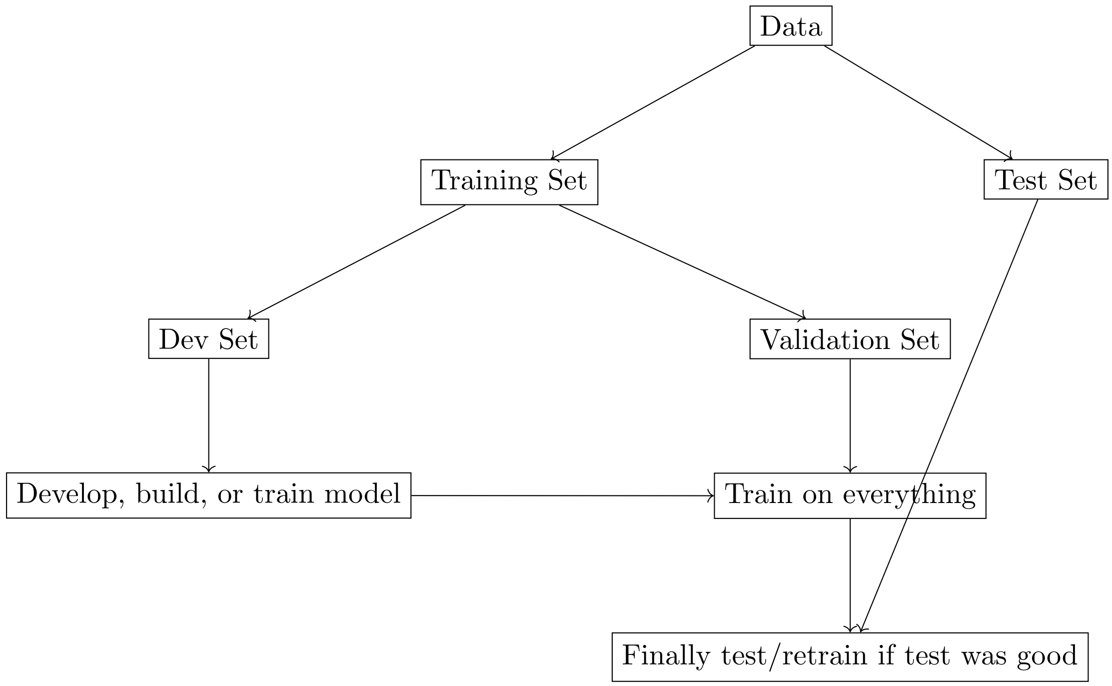

Figure 1: Model Training Process

Model Evaluation

How do we know whether a model is good? For classification models, we use the following metrics:

	+	-	
+	TP	FN	RP
-	FP	TN	RN
	PT	PN	GT

Figure 2: Confusion Matrix

- **Accuracy**: The proportion of correct predictions. $\frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$
- **Error Rate**: $1 - \text{Accuracy} = \frac{\text{FP+FN}}{\text{GT}}$
- **Precision**: Measures how well the model classifies positives. $\frac{\text{TP}}{\text{TP+FP}}$
- **Recall (Sensitivity)**: Measures how many positive instances were correctly predicted. $\frac{\text{TP}}{\text{TP+FN}}$

- **Specificity**: Measures the proportion of true negatives correctly identified. $\frac{\text{TN}}{\text{FP+TN}}$
- **F-Score**: The harmonic mean of precision and recall. Best when $B = 1$.

2. Perceptron Algorithm

- First machine learning/AI algorithm, introduced by Frank Rosenblatt in 1957.
- The goal is to find a decision boundary or hyperplane to separate the data. There can be multiple decision boundaries.
- For non-linear decision boundaries, techniques like the kernel trick can be used to create a hyperplane.

Kernel Trick: The kernel trick enables the perceptron to operate in higher-dimensional spaces by replacing the standard dot product with a kernel function. This kernel function effectively computes the dot product in a transformed, higher-dimensional space without explicitly performing the transformation. It acts as a similarity measure between points, corresponding to a dot product in that space. The trick works because the perceptron decision function is based solely on dots between data points. By substituting a kernel function, the perceptron can classify data in a higher-dimensional space, where it might become linearly separable, without the computational expense of transforming the points directly.

- Assumptions:
  - The data is linearly separable.
- Definition: Hyperplane
  - $\{(\vec{x}_1, y_1), (\vec{x}_2, y_2), ..., (\vec{x}_n, y_n)\}$ where y are labels.
  - $\mathcal{H} = \{x_i \mid w^T x + b = 0\}$ where w is the vector of weights and b is the bias and $b \in \mathbb{R}$.
  - Goal is to find w and b.
- Code example of perceptron:

```python
def perceptron(D, max_epochs=100):
    """
    Perceptron algorithm for binary classification.

    Input:
    - D: Training dataset, a list of tuples (x_i, y_i), where x_i is the feature vector
         and y_i is the label (-1 or 1).
    - max_epochs: Maximum number of iterations over the dataset.

    Output:
    - w: Learned weight vector (including bias term).
    """
```

```python
# Step 1: Initialize weight vector and bias to zero
n_features = len(D[0][0])  # Number of features in x_i
w = [0.0] * n_features  # Weight vector
b = 0.0  # Bias term

# Step 2: Initialize misclassification counter
m = 1  # Assume at least one misclassification

# Step 3: Iterate until no misclassifications or max_epochs reached
epoch = 0
while m > 0 and epoch < max_epochs:
    m = 0  # Reset misclassification counter

    # Step 5: Iterate over all training examples
    for x_i, y_i in D:
        # Step 6: Check if the example is misclassified
        # Compute the dot product w · x_i, add bias, and check if y_i * (w · x_i + b) <= 0
        if y_i * (sum(w[j] * x_i[j] for j in range(n_features)) + b) <= 0:
            # Step 7: Update weight vector and bias
            for j in range(n_features):
                w[j] += y_i * x_i[j]
            b += y_i
            m += 1  # Increment misclassification counter

    epoch += 1

# Bundle weights and bias into a single vector
w.append(b)  # Append bias as the last element of w
return w
```