0.1 Basic Terminology

Features (X): Relevant characteristics of the problem, usually suggested by experts. d is the number of features.
Target (y): The feature we want to predict.
Example: A row of feature values (may or may not include y). n is the number of examples.
Training: The process of learning the mapping between X and y.

0.2 AI, ML, and DL

Artificial Intelligence (AI): Broad goal of making computers perform tasks that typically require human intelligence.
Machine Learning (ML): A subset of AI where systems learn patterns from data instead of being explicitly programmed.
Deep Learning (DL): A subset of ML that uses multi-layered neural networks to learn complex patterns.

0.3 Classification Steps

1. Read the data
2. Create X and y (features and target)
3. Create a classifier object
4. `fit` the classifier
5. `predict` on new examples
6. `score` the model

0.4 Parameters vs. Hyperparameters

Parameters: Values learned by the algorithm from the data during `fit` (e.g., split features/thresholds in a decision tree).
Hyperparameters: "Knobs" set *before* calling `fit` that control the learning (e.g., `max_depth` in a decision tree). Set via expert knowledge, heuristics, or optimization.

0.5 Tree Terminology

Root Node: The first condition/question to check.
Branch: Connects nodes, typically representing true or false.
Internal Node: Represents conditions within the tree.
Leaf Node: Represents the predicted class/value.
Tree Depth: The number of edges on the path from the root node to the farthest leaf node.

0.6 Decision Regression Trees

Model: Use `DecisionTreeRegressor`.
Paradigm: `fit` and `predict` are similar to classification.
Score: The `score` method returns the $\mathbf{R^2}$ **score**.
$\mathbf{R^2}$**:** Maximum is 1 (perfect predictions). Can be negative (worse than `DummyRegressor` which returns the mean of y).
Common Metric: Mean Squared Error (MSE).

```
X = regression_df.drop(["quiz2"], axis=1)
y = regression_df["quiz2"]

depth = 2
reg_model = DecisionTreeRegressor(max_depth=depth)
reg_model.fit(X, y)
regression_df["predicted_quiz2"] = reg_model.predict(X)
print("R^2 score: %0.3f" % (reg_model.score(X, y)))
```

0.7 Classification Example

```
depth = 2
model = DecisionTreeClassifier(max_depth=depth)
model.fit(X_subset.values, y)
# Score on training data
score = model.score(X_subset.values, y)
print("Error: %0.3f" % (1 - score))
```

0.8 Train, Validation, Test Split

Golden Rule: The test data cannot influence the training phase in any way.
Deployment Data: Data in the wild where y is unknown. Deployment error is the true goal.
Validation Data (Dev Set): Used for hyperparameter tuning. Locked in a "vault" until ready to evaluate.
Expected Error Hierarchy:
E-train $<$ E-validation $<$ E-test $<$ E-deployment

```
train_df, test_df = train_test_split(df, test_size=0.2,
     random_state=123)
X_train, y_train = train_df.drop(columns=["country"]),
     train_df["country"]
X_test, y_test = test_df.drop(columns=["country"]),
     test_df["country"]
```

0.9 Cross-Validation (CV)

Purpose: Evaluate how well the model generalizes to unseen data (gets validation scores).
Process: Split data into k folds (e.g., $k = 10$). Each fold gets a turn as the validation set. CV does *not* shuffle data (that's `train_test_split`).
`cross_validate`**:** More powerful than `cross_val_score`. Gives access to training and validation scores.

```
from sklearn.model_selection import cross_val_score, cross_validate
model = DecisionTreeClassifier(max_depth=4)
cv_scores = cross_val_score(model, X_train, y_train, cv=10)
# Use cross_validate to get both train and test (validation) scores
scores = cross_validate(model, X_train, y_train, cv=10,
                        return_train_score=True)
```

0.10 Underfitting (High Bias)

Cause: Model is too simple (`max_depth=1`). Fails to capture useful training patterns.
Effect: Both train and validation errors are **high**. Low gap between them.
Error Relation: E-best $<$ E-train $\leq$ E-valid

0.11 Overfitting (High Variance)

Cause: Model is too complex (`max_depth=None`). Learns unreliable, noisy training patterns.
Effect: Training error is very **low**, but there is a **big gap** between training and validation error.
Error Relation: E-train $<$ E-best $<$ E-valid
Tradeoff: As complexity ↑, E-train ↓ but (E-valid − E-train) ↑. We want to avoid both.

0.12 k-NN KNeighborsClassifier Hyperparameter Tuning

Example loop to tune k (`n_neighbors`) using cross-validation and collect scores.

```
results_dict = {...} # See full data
param_grid = {"n_neighbors": np.arange(1, 50, 5)}
for k in param_grid["n_neighbors"]:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_validate(knn, X_train, y_train,
                            return_train_score=True)
    results_dict["n_neighbors"].append(k)
    results_dict["mean_cv_score"].append(np.mean(scores["test_score"]))
    results_dict["mean_train_score"].append(np.mean(scores["train_scor
    # ... append standard deviations ...
results_df = pd.DataFrame(results_dict)
best_n_neighbours = results_df.idxmax()["mean_cv_score"]
knn = KNeighborsClassifier(n_neighbors=best_n_neighbours)
knn.fit(X_train, y_train)
print("Test accuracy: %0.3f" % (knn.score(X_test, y_test)))
```

0.13 k-NN KNeighborsRegressor Hyperparameter Tuning

Example loop to tune k (`n_neighbors`) using cross-validation and collect scores.

0.14 Pros of k-NNs for supervised learning

Easy to understand, interpret.
Simple hyperparameter $n_neighbors$ controlling the fundamental tradeoff.
Can learn very complex functions given enough data.
Lazy learning: Takes no time to fit

0.15 Cons of k-NNs for supervised learning

Can be potentially be VERY slow during prediction time, especially when the training set is very large.
Often not that great test accuracy compared to the modern approaches.
It does not work well on datasets with many features or where most feature values are 0 most of the time (sparse datasets).

0.16 Curse of dimensionality

Affects all learners but especially bad for nearest-neighbour.
k-NN usually works well when the number of dimensions d is small but things fall apart quickly as d goes up.
If there are many irrelevant attributes, k-NN is hopelessly confused because all of them contribute to finding similarity between examples.
With enough irrelevant attributes the accidental similarity swamps out meaningful similarity and k-NN is no better than random guessing.

0.17 Support Vector Machines (SVMs) with RBF kernel

The decision boundary is defined by a set of positive and negative examples and their weights together with their similarity measure.
A test example is labeled positive if on average it looks more like positive examples than the negative examples.

```
from sklearn.svm import SVC
svm = SVC(gamma=0.01)
scores = cross_validate(svm, X_train, y_train,
     return_train_score=True)
print("Mean validation score \%0.3f" \%
     (np.mean(scores["test_score"])))
```

0.18 Preprocessing

fit and transform paradigm for transformers We **fit** the transformer on the train split and then transform the train split
We apply **transform** on the test split.
TRANFORMER used to change the input representation
Imputation: Tackling missing values.
Scaling: Scaling of numeric features (e.g., MinMax or Standardization).
One-hot encoding: Tackling categorical variables.
Ordinal encoding: data with a meaningful rank or order—into integers that preserve that specific sequence.
Are we applying fit_transform on train portion and transform on validation portion in each fold?
Here you might be allowing information from the validation set to leak into the training step.
You need to apply the SAME preprocessing steps to train/validation.
With many different transformations and cross validation the code gets unwieldy very quickly.
Likely to make mistakes and "leak" information.
In these examples our test accuracies look fine, but our methodology is flawed.
Implications can be significant in practice!

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer # For handling missing
     values
# Define feature types (X_train.columns assumed)
numeric_features = ['age', 'income']
categorical_features = ['city', 'gender']
# Create transformers for different column types
numeric_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='mean')),
    ('scaler', StandardScaler())
])
categorical_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='most_frequent')),
    ('onehot', OneHotEncoder(handle_unknown='ignore'))
])
# Define the specific order for the categories in your feature
size_categories = ['Small', 'Medium', 'Large', 'Extra Large']
ordinal_transformer = Pipeline(steps=[
    ('imputer', SimpleImputer(strategy='constant',
         fill_value='Missing')), # Handles NaNs if any
    ('ordinal', OrdinalEncoder(categories=[size_categories])) #
         Apply encoding based on defined order
])
# Create the preprocessor using ColumnTransformer
preprocessor = ColumnTransformer(
    transformers=[
        ('num', numeric_transformer, numeric_features),
        ('cat', categorical_transformer, categorical_features),
        ('cat_ordinal', ordinal_transformer, ordinal_features),
        (binary_transformer, binary_features),
        ("drop", drop_features)
    ])

# Fit and transform the data
X_train_processed = preprocessor.fit_transform(X_train)
X_test_processed = preprocessor.transform(X_test)
```

0.19 Pipelines

Each transformer is applied to the specified columns and the result are concatenated horizontally, Single object for transform.