Intro to ML 8/9 Cheatsheet

---

1. Splitting Data

1.1 What are X and y?

- Features are the columns we learn from to make a prediction. In Python, we store the features in X.
- The target is the column we are trying to predict. We store the target in y.

age	active?	work_from_home?	target
25	yes	no	Dog
35	yes	no	Dog
20	no	yes	Dog
20	no	no	Cat
30	no	no	Cat
45	yes	no	Cat

1.2 How do I get X and y?

Let's get X and y for the dataset above. Let's imagine the data is called adopt_data. The target column is called "target" and we want to use all the other columns.

- To get X, we want to copy over all the columns from the dataset except the target column.
- To get y, we want to specifically select and use the target column.

```python
X = adopt_data.drop(columns=["target"])
y = adopt_data["target"]
```

1.3 Golden Rule: Separating Training and Testing Data

We use a function called `train_test_split` to separate our dataset into training and testing sets. When we use `train_test_split`, we need to pass in (tell the function):

- Which dataset we want to split (represented by `X` and `y`)
- How much of the data should be in the testing set (`test_size`)

From this function, we get 4 variables: `X_train`, `X_test`, `y_train`, and `y_test`. `X_train` and `y_train` make up the training set and `X_test` and `y_test` make up the testing set.

```python
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

---

2. Preprocessing

2.1 What are variables?

Consider the following NBA dataset:

Player Name	Team	Position	Points Per Game	Rebounds Per Game	Assists Per Game	FG%
LeBron James	Los Angeles LAL	SF/PF	25.4	7.9	7.3	50.1%
Stephen Curry	Golden State GS	PG	29.8	4.5	6.2	47.5%
Giannis Antetokounmpo	Milwaukee MIL	PF/C	31.1	11.8	5.7	55.2%
Jayson Tatum	Boston BOS	SF/PF	27.2	8.6	4.4	46.4%
Luka Dončić	Dallas DAL	PG	32.6	9.2	8.1	48.7%

- A variable is a characteristic that can be measured or counted.
- In our dataset above, each column represents one variable.
    - For example, player name, position, points per game, field goal percentage...

2.2 Types of variables

- Variables can hold text or numbers.
- We can classify variables into two larger categories:
    - Numeric
    - Categorical
- Numeric variables have data values that are *numbers*.
    - e.g., price of a home, age, income, height
- Categorical variables have data values that can't be quantifiable i.e., numbers don't make sense for them. These are *words* or *categories*.
    - e.g., colours, highest education achieved, species, favourite movies, brand names

2.2.1 Types of categorical variables

Within categorical variables, we have two subcategories: nominal and ordinal variables.

Ordinal variables have a natural order to them. For example:

- T-shirt sizes
  - Small, medium, large
- Education
  - Preschool, Grade 1, Grade 2, ..., Grade 12, Undergraduate, Graduate, Doctorate

Nominal variables don't have any kind of order to them. For example:

- Types of animals
  - Cat, dog, hamster, bird
- Country
  - Canada, USA, Mexico...

2.3 Handling Missing Data (Imputer)

Real datasets often have missing values.
For example, maybe a player didn't report their height, or we don't know their college.

We can use an imputer to fill in missing values:

- Numeric data → replace with the mean, median, or a constant.
- Categorical data → replace with the most frequent value (mode) or a constant like "Unknown".

```python
from sklearn.impute import SimpleImputer
import numpy as np

# Example numeric imputer (replace missing numbers with the mean)
num_imputer = SimpleImputer(strategy='mean')

# Example categorical imputer (replace missing categories with the most frequent value)
cat_imputer = SimpleImputer(strategy='most_frequent')
```

2.4 Preprocessing numerical features

Sometimes, numbers are on very different scales.
For example, in our NBA dataset:

- Points per game: usually between 0 and 35
- Field goal %: between 0 and 100
- Rebounds: between 0 and 15

If we don't adjust these, the bigger numbers (like points) can dominate the smaller ones (like rebounds) in machine learning.

2.4.1 Two common techniques for numerical features

- **Standardization (`StandardScaler`)**
  - Turns numbers into "z-scores" so they have mean = 0 and standard deviation = 1.
  - Example: If average height is 170 cm, and you're 180 cm tall, your standardized height is about +1.

- **Normalization (`MinMaxScaler`)**
  - Shrinks all values to a fixed range, usually 0 to 1.
  - Example:
    - Points: 0 → 0, 35 → 1, 17.5 → 0.5

In scikit-learn:

```python
from sklearn.preprocessing import StandardScaler, MinMaxScaler

scaler_standard = StandardScaler()
scaler_normalize = MinMaxScaler()

X_train_scaled = scaler_standard.fit_transform(X_train)
```

Note that the code above assumes that all of the columns in X_train are *numerical* features! We will show you what to do if your data contains both numerical and categorical columns in Section 2.6.

2.5 Preprocessing categorical features

Machine learning models can't directly understand words like "PG" (for Point Guard) or "Los Angeles Lakers".
We need to turn categories into numbers.

2.5.1 Two common techniques

1. **One-Hot Encoding (OneHotEncoder)**
    a. Makes a new column for each category.
    b. Example (Position):
        i. PG → [1, 0, 0]
        ii. SF/PF → [0, 1, 0]
        iii. PF/C → [0, 0, 1]
2. **Label Encoding (LabelEncoder)**
    a. Assigns each category a number.
    b. Example (Team):
        i. Los Angeles LAL → 0
        ii. Golden State GS → 1
        iii. Milwaukee MIL → 2

```python
from sklearn.preprocessing import OneHotEncoder

encoder = OneHotEncoder()
X_train_encoded = encoder.fit_transform(X_train)
```

Note that the code above assumes that all of the columns in X_train are *nominal* features! We will show you what to do if your data contains both numerical and categorical columns in Section 2.6.

2.6 Preprocessing ordered features

If categories have an **order**, we should respect it.

For example:

- T-shirt sizes: `Small < Medium < Large`

If we one-hot encode, the model won't know that Large > Medium > Small.
Instead, we use **Ordinal Encoding**.

```python
from sklearn.preprocessing import OrdinalEncoder

ordered_tshirt_sizes = ['Small', 'Medium', 'Large']
encoder = OrdinalEncoder(categories=[ordered_tshirt_sizes])
X_encoded = encoder.fit_transform(X_ordered)
```

Then:

- Small → 0
- Medium → 1
- Large → 2

2.6.1 What if we have more than one ordinal feature?

The **first list** in `categories=[...]` corresponds to the **first column** in your data.
The **second list** corresponds to the **second column**, and so on.
Each column gets encoded according to its own order.

```python
from sklearn.preprocessing import OrdinalEncoder
import pandas as pd
```

```
# Sample dataset with 2 ordered categorical columns
data = pd.DataFrame({
    'TshirtSize': ['Small', 'Large', 'Medium', 'Small'],
    'Education': ['High School', 'College', 'Elementary', 'Middle
School']
})


# Define the custom order for each column
tshirt_order = ['Small', 'Medium', 'Large']
education_order = ['Elementary', 'Middle School', 'High School',
'College']

# Apply OrdinalEncoder with different orders per column
encoder = OrdinalEncoder(categories=[tshirt_order, education_order])

X_encoded = encoder.fit_transform(data)
```

2.7 Using a `ColumnTransformer`

In real datasets, we have **both numeric and categorical features**.
We need to preprocess them differently.

Example:

- Numeric: Points, Rebounds, Assists, FG%
- Categorical: Team, Position

`ColumnTransformer` lets us apply different transformations to different columns all at once.

Here's an example combining:

- **Numeric preprocessing** → impute missing values + scale
- **Categorical preprocessing** → impute missing values + one-hot encode

Python

```python
from sklearn.compose import make_column_transformer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.impute import SimpleImputer
from sklearn.pipeline import make_pipeline

numerical_features = ['Points Per Game','Rebounds Per Game','Assists Per Game','FG%']

categorical_features = ['Team','Position']

# Numeric pipeline: fill missing with mean → scale
num_pipeline = make_pipeline(
    SimpleImputer(strategy='mean'),
    StandardScaler()
)

# Categorical pipeline: fill missing with most frequent → one-hot encode
cat_pipeline = make_pipeline(
    SimpleImputer(strategy='most_frequent'),
    OneHotEncoder()
)

# Apply to the right columns
preprocessor = make_column_transformer(
    (num_pipeline, numerical_features),
    (cat_pipeline, categorical_features)
)

X_train_clean = preprocessor.fit_transform(X_train)
X_test_clean = preprocessor.transform(X_test)
```

3. Supervised Models

3.1 Classification vs. Regression

Any supervised machine learning problem is either a classification or regression problem.

- Classification: predicting one of a finite number of classes (e.g. is a credit card transaction fraudulent or not, is a picture a cat, dog, hamster, or bird, is a Pokemon legendary or not...)
- Regression: predicting a number (e.g. price of a house, amount of rainfall, rating out of 100...)

3.2 Classification models

- Tree-based models:
    - Decision tree classifier (hyperparameter: max_depth, controls the depth of the tree)
- Linear models:
    - Logistic regression (hyperparameter: C, controls how much the model learns from the data)
    - Linear SVM classifier (hyperparameter: C, controls how much the model learns from the data)
- Non-linear models:
    - SVM RBF classifier (hyperparameters: C and gamma, control how much the model learns from the data)
    - K-nearest neighbours classifier (hyperparameter: n_neighbors, controls how many neighbours the model polls to get the majority class)
- Ensemble models:
    - Random forest classifier (hyperparameters: max_depth, controls the depth of the tree, and n_estimators, controls how many trees to make and average together)

Python

```python
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier

tree_model = DecisionTreeClassifier(max_depth=5)
lr_model = LogisticRegression(C=1)
linear_svm_model = SVC(kernel="linear", C=1)
svm_rbf_model = SVC(C=1, gamma=1)
knn_model = KNeighborsClassifier(n_neighbors=5)
forest_model = RandomForestClassifier(max_depth=5, n_estimators=100)
```

3.3 Regression models

- Tree-based models:
    - Decision tree regressor (hyperparameter: max_depth, controls the depth of the tree)
- Linear models:
    - Ridge (hyperparameter: alpha, controls how much the model learns from the data)
- Non-linear models:
    - SVM RBF regressor (hyperparameters: C and gamma, control how much the model learns from the data)
    - K-nearest neighbours regressor (hyperparameter: n_neighbors, controls how many neighbours the model polls to get the majority class)
- Ensemble models:
    - Random forest regressor (hyperparameters: max_depth, controls the depth of the tree, and n_estimators, controls how many trees to make and average together)

```python
from sklearn.svm import SVR
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import Ridge
from sklearn.neighbors import KNeighborsRegressor
from sklearn.ensemble import RandomForestRegressor

tree_model = DecisionTreeRegressor(max_depth=5)
ridge_model = Ridge(alpha=1.0)
linear_svm_model = SVR(kernel="linear", C=1)
svm_rbf_model = SVR(C=1, gamma=1)
knn_model = KNeighborsRegressor(n_neighbors=5)
forest_model = RandomForestRegressor(max_depth=5, n_estimators=100)
```

3.4 How do I set up, train, and test a model?

There are three steps you will always follow to set up a model:

1. Create the model and set up hyperparameters
2. Train the model using `fit()`. Make sure to use the *training* set
3. Test the model using `score()`. Make sure to use the *testing* set

For example, if you were trying to set up, train, and test a decision tree classifier:

```python
from sklearn.tree import DecisionTreeClassifier

tree_model = DecisionTreeClassifier(max_depth=5)
tree_model.fit(X_train, y_train)
tree_model.score(X_test, y_test)
```

---

4. Evaluation Metrics

When we train a machine learning model, we need a way to measure how good it is. These measurements are called **evaluation metrics**.

Different tasks (like predicting numbers vs. predicting categories) need different metrics.

4.1 Metrics for Regression

Common metrics:

- **Mean Absolute Error (MAE)**
  Average of how far off we are, in absolute value.
  Example: If we predict 20 PPG but the real value is 25, the error is 5.

- **Mean Squared Error (MSE)**
  Like MAE, but squares the errors (big mistakes are punished more).

- **Root Mean Squared Error (RMSE)**
  Square root of MSE, so it's back in the same units as the prediction.

- **R^2 Score**
  Tells us how much of the variation in the data is explained by the model.
  1.0 = perfect, 0 = model is as good as guessing the average.

```python
from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

mae = mean_absolute_error(y_true, y_pred)
mse = mean_squared_error(y_true, y_pred)
rmse = mean_squared_error(y_true, y_pred, squared=False)
r2 = r2_score(y_true, y_pred)
```

4.2 Metrics for Classification

Classification = predicting a **label** (e.g., predicting whether a player is a Guard, Forward, or Center).

4.2.1 Basic metrics

- **Accuracy**
  % of predictions that are correct.
  Example: 8 correct out of 10 = 80% accuracy.

But accuracy alone can be misleading (e.g., if classes are imbalanced). That's where **precision, recall, and F1** come in.

Imagine we're predicting if a shot is **"Made"** or **"Missed"**.

- **Precision** = Of all the times we predicted "Made", how many were actually made?
- **Recall** = Of all the shots that were truly "Made", how many did we correctly predict as "Made"?

Sometimes we want higher precision (fewer false alarms).
Other times we want higher recall (catch more true cases).

The **F1 score** balances precision and recall.
It's useful when we care about both, not just one.

- Formula: It's the "harmonic mean" of precision and recall (you don't need to memorize the exact math).
  - Range: 0 to 1, where 1 = best.

```python
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Example: true labels (what actually happened)
y_true = ["Made", "Made", "Missed", "Made", "Missed", "Missed", "Made", "Missed", "Missed", "Made"]
```

```
# Example: predicted labels (what the model guessed)
y_pred = ["Made", "Missed", "Missed", "Made", "Missed", "Made", "Made",
"Missed", "Missed", "Made"]

# Calculate metrics
accuracy = accuracy_score(y_true, y_pred)
precision = precision_score(y_true, y_pred, pos_label="Made")
recall = recall_score(y_true, y_pred, pos_label="Made")
f1 = f1_score(y_true, y_pred, pos_label="Made")
```

4.2.2 Confusion Matrix

A confusion matrix shows **counts of predictions vs. actual outcomes**.

Example (binary classification: “Made” vs. “Missed”):

	Predicted: Made	Predicted: Missed
Actual: Made	50	10
Actual: Missed	5	35

- Top-left = correctly predicted made shots (true positives).
- Bottom-right = correctly predicted missed shots (true negatives).
- Off-diagonal cells = mistakes (false positives and false negatives).

Python
```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_true, y_pred)

ConfusionMatrixDisplay(cm).plot()
```