Building Your First ML Models

Week 7 · CS 203: Software Tools and Techniques for AI

Prof. Nipun Batra
IIT Gandhinagar

Part 1: The Big Picture

What does it mean to "build" an ML model?

Remember Our Netflix Journey?

```
Week 1: Collected movie data (APIs, scraping)
Week 2: Cleaned and organized it (Pandas)
Week 3: Labeled movie success/failure (annotation)
Week 4: Made labeling efficient (active learning)
Week 5: Got more data (augmentation)
Week 6: Used LLMs to help (APIs)
        ↓
Week 7: NOW WE BUILD THE MODEL!
🎉
```

We finally have good data. Time to predict!

What Are We Predicting?

Our Netflix Problem:

Given movie features → Predict if it will be successful

```
INPUT (What we know)          OUTPUT (What we predict)

• Genre: Action
• Budget: $150M                → SUCCESS or FAILURE?
• Director: Nolan
• Runtime: 148 mins
```

This is called **Classification** (putting things in categories)

Two Types of Predictions

```
┌──────────────────────────────┐  ┌──────────────────────────────┐
│       CLASSIFICATION         │  │        REGRESSION            │
│                              │  │                              │
│   Predict a CATEGORY         │  │   Predict a NUMBER           │
│                              │  │                              │
│   • Success / Failure        │  │   • $500M revenue            │
│   • Spam / Not Spam          │  │   • 7.5 rating               │
│   • Cat / Dog / Bird         │  │   • 25°C temperature         │
│                              │  │                              │
│   "Which box does this go in?"  │   "How much / How many?"     │
└──────────────────────────────┘  └──────────────────────────────┘
```

Today: We'll focus on classification (predicting movie success)

The ML Workflow (Simple Version)

```
┌──────────────┐
│  Your Data   │
│(movies.csv)  │
└──────┬───────┘
       │
       ▼
┌──────────────┐
│ Split Data   │  ←── Training set + Test set
└──────┬───────┘
       │
       ▼
┌──────────────┐
│ Train Model  │  ←── Model learns patterns
└──────┬───────┘
       │
       ▼
┌──────────────┐
│ Evaluate     │  ←── How good is it?
└──────────────┘
```

Simple! But the devil is in the details...

Part 2: Starting Simple - Baseline Models

Why you should never start with deep learning

The Temptation

You: "I want to predict movie success!"

Internet: "Use a 175-billion parameter neural network!"

You: "Sounds cool! Let me try..."

3 hours later: Nothing works. GPU out of memory. Confused.

Lesson: Don't start with the fanciest tool. Start simple!

What is a Baseline?

A **baseline** is the simplest possible solution that works.

```
BASELINE EXAMPLES

Task: Predict if movie succeeds

Dumb Baseline: "Just predict the most common outcome"
               If 70% of movies succeed → always say SUCCESS
               Accuracy: 70% (for free!)

Simple Model:  Logistic Regression
               (One line of code, 80% accuracy?)

Complex Model: Deep Neural Network
               (1000 lines of code, 82% accuracy?)
```

Is that 2% worth 100x complexity?

Why Baselines Matter

Scenario 1: You build a fancy model, get 85% accuracy.

- "Wow, my model is amazing!"
- Reality: A baseline gets 84% → you only improved by 1%
- All that complexity for almost nothing

Scenario 2: You build a fancy model, get 85% accuracy.

- Baseline gets 60% → you improved by 25%!
- That complexity was worth it!

Baselines give you a reference point. Without one, you can't know if your model is actually good.

The Simplest Baseline: "Just Guess"

```python
# The dumbest model possible
def dumb_predictor(movie):
    return "SUCCESS"  # Always predict success

# If 70% of movies succeed, this gets 70% accuracy!
```

This is called a "Majority Class Classifier"

```python
from sklearn.dummy import DummyClassifier

# Create the dumbest possible classifier
baseline = DummyClassifier(strategy='most_frequent')
baseline.fit(X_train, y_train)

accuracy = baseline.score(X_test, y_test)
print(f"Dumb baseline accuracy: {accuracy:.1%}")
```

Any real model must beat this!

Baseline Model 1: Logistic Regression

Think of it as: A weighing scale for features

```
Feature                 Weight     Value      Contribution
─────────────────       ──────     ─────      ────────────
Budget ($M)             +0.3       150        +45
Star Power              +0.5       8          +4
Is Sequel               +0.2       1          +0.2
Is January Release      -0.4       0          0
                                              ─────
                                   Total:     +49.2

If Total > 0 → Predict SUCCESS
If Total < 0 → Predict FAILURE
```

It just adds up weighted features!

Logistic Regression in Code

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(
    features, labels, test_size=0.2, random_state=42
)

# Create and train the model (2 lines!)
model = LogisticRegression()
model.fit(X_train, y_train)

# Evaluate
accuracy = model.score(X_test, y_test)
print(f"Logistic Regression accuracy: {accuracy:.1%}")
```

That's it! A working ML model in 4 lines.

Baseline Model 2: Decision Tree

Think of it as: A flowchart of yes/no questions

```
                Is Budget > $100M?
               /                  \
             YES                   NO
             /                      \
      Is Sequel?              Has Star Actor?
      /        \               /          \
    YES        NO            YES          NO
    /           \            /             \
SUCCESS      Is Summer?   FAILURE      SUCCESS
              /     \
            YES     NO
            /        \
       SUCCESS    FAILURE
```

Humans can actually read and understand this!

Decision Tree in Code

```python
from sklearn.tree import DecisionTreeClassifier

# Create and train
tree = DecisionTreeClassifier(max_depth=5)  # Don't go too deep!
tree.fit(X_train, y_train)

# Evaluate
accuracy = tree.score(X_test, y_test)
print(f"Decision Tree accuracy: {accuracy:.1%}")
```

You can even visualize it:

```python
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

plt.figure(figsize=(20, 10))
plot_tree(tree, feature_names=feature_names, filled=True)
plt.show()
```

Which Baseline to Use?

Your Situation	Recommended Baseline
Just starting	**Logistic Regression** - fast, simple, often works well
Need to explain to your boss	**Decision Tree** - you can see the rules
Mixed data (numbers + categories)	**Random Forest** - handles everything
Want best performance	**AutoML** - we'll learn this soon!

Baseline Model 3: Random Forest

Think of it as: Asking 100 decision trees and taking a vote

```
Tree 1: "I think SUCCESS" ──┐
Tree 2: "I think FAILURE"   │
Tree 3: "I think SUCCESS"   │
Tree 4: "I think SUCCESS"   ├──► VOTE: SUCCESS wins!
Tree 5: "I think FAILURE"   │          (3 vs 2)
...                         │
Tree 100: "I think SUCCESS"─┘
```

Wisdom of crowds: Many weak learners → One strong learner

Random Forest in Code

```
from sklearn.ensemble import RandomForestClassifier

# Create and train
forest = RandomForestClassifier(n_estimators=100, random_state=42)
forest.fit(X_train, y_train)

# Evaluate
accuracy = forest.score(X_test, y_test)
print(f"Random Forest accuracy: {accuracy:.1%}")
```

Often the best simple model! Very hard to beat.

Part 3: Cross-Validation

How to really know if your model is good

The Problem with One Test Set

You split your data once: **80% training, 20% test**

Your model gets 85% on the test set. Great... right?

But wait:

- What if you got "lucky" with that split?
- What if the test set happened to be easy?
- What if all the hard examples ended up in training?

One test set = one roll of the dice. We need something more reliable.

The Solution: Cross-Validation

Idea: Test on EVERY part of your data (not just 20%)

```
┌──────────────────────────────────────────────────────────┐
│                 5-FOLD CROSS-VALIDATION                  │
├──────────────────────────────────────────────────────────┤
│                                                          │
│  Fold 1: [TEST][Train][Train][Train][Train]  → Accuracy: 82%
│  Fold 2: [Train][TEST][Train][Train][Train]  → Accuracy: 85%
│  Fold 3: [Train][Train][TEST][Train][Train]  → Accuracy: 84%
│  Fold 4: [Train][Train][Train][TEST][Train]  → Accuracy: 81%
│  Fold 5: [Train][Train][Train][Train][TEST]  → Accuracy: 83%
│                                                          │
│  Average: 83% ± 1.5%                                     │
│                                                          │
└──────────────────────────────────────────────────────────┘
```

Now we know: "My model gets ~83% accuracy, give or take 1.5%"

Cross-Validation: Visual Intuition

Think of it like a **rotating exam schedule**:

```
          Split 1     Split 2     Split 3     Split 4     Split 5
         ┌────────┐  ┌────────┐  ┌────────┐  ┌────────┐  ┌────────┐
Student  │ █ █ █ █│  │ ░ █ █ █│  │ ░ ░ █ █│  │ ░ ░ ░ █│  │ ░ ░ ░ ░│
   A     │(exam)  │  │(study) │  │(study) │  │(study) │  │(study) │
         └────────┘  └────────┘  └────────┘  └────────┘  └────────┘

         ┌────────┐  ┌────────┐  ┌────────┐  ┌────────┐  ┌────────┐
Student  │ ░ ░ ░ ░│  │ █ █ █ █│  │ ░ ░ ░ ░│  │ ░ ░ ░ ░│  │ ░ ░ ░ ░│
   B     │(study) │  │(exam)  │  │(study) │  │(study) │  │(study) │
         └────────┘  └────────┘  └────────┘  └────────┘  └────────┘

█ = Test (exam)     ░ = Train (study)
```

Every data point gets tested exactly once!

Cross-Validation in Code

```python
from sklearn.model_selection import cross_val_score

# Create model
model = RandomForestClassifier(n_estimators=100)

# Run 5-fold cross-validation
scores = cross_val_score(model, X, y, cv=5)

print(f"Scores for each fold: {scores}")
print(f"Average accuracy: {scores.mean():.1%}")
print(f"Standard deviation: {scores.std():.1%}")
```

Output:

```
Scores for each fold: [0.82, 0.85, 0.84, 0.81, 0.83]
Average accuracy: 83.0%
Standard deviation: 1.5%
```

Why Cross-Validation Matters

Model	Single Test	5-Fold CV
Logistic Regression	78%	76% ± 2%
Decision Tree	82%	75% ± 5% ← High variance!
Random Forest	84%	83% ± 1% ← Most stable!

Insights:

- Decision Tree looked good on one test, but it's unstable (±5%!)
- Random Forest is not only accurate but **consistent**
- Cross-validation reveals the truth!

Part 4: Hyperparameter Tuning

Making your model better with the right settings

What Are Hyperparameters?

Parameters: Values the model learns from data (weights, biases)

Hyperparameters: Values YOU choose before training

Example - Random Forest:

- `n_estimators` : How many trees? (10? 100? 500?)
- `max_depth` : How deep can each tree grow? (3? 10? unlimited?)
- `min_samples_split` : Minimum samples to split a node?

```python
# These are hyperparameters - YOU choose them
model = RandomForestClassifier(
    n_estimators=100,    # ← hyperparameter
    max_depth=10,        # ← hyperparameter
    min_samples_split=5  # ← hyperparameter
)
```

Why Hyperparameters Matter

Same model, different hyperparameters → **very different results**

n_estimators	max_depth	Accuracy
10	3	72%
100	5	79%
100	10	82%
500	None	84%

The right hyperparameters can improve your model by 10%+

But how do you find the right values?

Strategy 1: Grid Search

Idea: Try every combination and pick the best

```python
from sklearn.model_selection import GridSearchCV

# Define what to try
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 20, None]
}

# Try all combinations with cross-validation
grid_search = GridSearchCV(
    RandomForestClassifier(),
    param_grid,
    cv=5  # Use 5-fold CV for each combination
)
grid_search.fit(X, y)

print(f"Best params: {grid_search.best_params_}")
print(f"Best score: {grid_search.best_score_:.1%}")
```

Grid Search: The Problem

3 hyperparameters × 4 values each = 64 combinations

Each combination needs 5-fold CV = **320 model trainings!**

Hyperparameters	Values each	Combinations
2	3	9
3	4	64
4	5	625
5	5	3,125

Grid search doesn't scale well.

Strategy 2: Random Search

Idea: Don't try everything - randomly sample combinations

```
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint

# Define ranges to sample from
param_dist = {
    'n_estimators': randint(50, 500),
    'max_depth': randint(3, 30),
    'min_samples_split': randint(2, 20)
}

# Try 20 random combinations
random_search = RandomizedSearchCV(
    RandomForestClassifier(),
    param_dist,
    n_iter=20,  # Only try 20 combinations
    cv=5
)
random_search.fit(X, y)
```


Surprisingly effective! Often finds good solutions faster than grid search.

Hyperparameter Tuning Tips

Start with defaults - sklearn's defaults are usually reasonable

Tune the important ones first:

- Random Forest: `n_estimators`, `max_depth`
- Decision Tree: `max_depth`, `min_samples_split`
- Logistic Regression: `C` (regularization strength)

Use cross-validation - always! Never tune on test set.

Don't over-tune - spending days for +0.5% accuracy is usually not worth it

Or... just use AutoML (coming up next!)

Part 5: AutoML - Let the Computer Do It

The lazy (smart) way to build models

The Problem with Manual ML

The typical manual workflow:

1. Try Logistic Regression... okay
2. Try Decision Tree... not great
3. Try Random Forest... better
4. Try XGBoost... hmm, similar
5. Try Neural Network... takes forever
6. Tune hyperparameters for each one...
7. Try different feature combinations...
8. Repeat steps 1-7 many times...

Time spent: 3 days. Hair remaining: None.

There has to be a better way!

Enter AutoML

AutoML = Automatic Machine Learning

You: "Here's my data. Give me the best model."

AutoML: "On it! Let me try 50 different models, tune their parameters, combine the best ones, and give you a super-ensemble."

You: *goes to get coffee*

AutoML: "Done! Here's a model with 87% accuracy."

This is not magic. It just automates what experts do manually.

AutoGluon: AutoML Made Easy

AutoGluon (by Amazon) is one of the best AutoML tools.

What it does automatically:

1. Handles missing values
2. Encodes categorical features
3. Trains multiple model types (Random Forest, XGBoost, LightGBM, Neural Nets...)
4. Tunes hyperparameters
5. Creates an ensemble of the best models
6. Uses cross-validation internally

All with 3 lines of code!

AutoGluon in 3 Lines of Code

```python
from autogluon.tabular import TabularPredictor

# Step 1: Create the predictor
predictor = TabularPredictor(label='success')

# Step 2: Train on your data (that's it!)
predictor.fit(train_data)

# Step 3: Make predictions
predictions = predictor.predict(test_data)
```

Seriously. That's the entire code.

What Happens Inside AutoGluon?

Input: Your CSV file

↓ **Step 1**: Analyze data types (numbers, text, dates)

↓ **Step 2**: Preprocess features automatically

↓ **Step 3**: Train 10+ different model types

↓ **Step 4**: Cross-validate each model

↓ **Step 5**: Stack models together (ensemble)

Output: One super-model that combines the best of all

AutoGluon Leaderboard

After training, you can see how each model performed:

```
predictor.leaderboard(test_data)
```

```
                model  score_val  fit_time
0   WeightedEnsemble_L2     0.87      120s
1             CatBoost     0.85       45s
2             LightGBM     0.84       30s
3              XGBoost     0.83       50s
4         RandomForest     0.82       25s
5   NeuralNetFastAI     0.80       90s
6     LogisticRegression  0.76        5s
```

The ensemble combines the best models!

When to Use AutoML

Great for:

- Quick prototyping ("Is ML even useful for this?")
- Competitions (Kaggle)
- When you don't have ML expertise
- Setting a strong baseline to beat

Be careful:

- Takes a long time to train (10 mins to hours)
- Uses lots of memory
- Hard to explain ("Why did it predict this?")
- Model might be too big for production

AutoGluon with Time Limit

Don't have all day? Set a time limit:

```python
predictor = TabularPredictor(label='success')

# Only train for 5 minutes
predictor.fit(train_data, time_limit=300)  # 300 seconds = 5 mins
```

More time = Better models (usually)

Time Limit	What AutoGluon Can Do
1 minute	Quick baselines (RF, LR)
5 minutes	Good models (+ XGBoost, LightGBM)
30 minutes	Great models (+ Neural Nets, tuning)
2+ hours	Best possible (full tuning, stacking)

Part 6: Transfer Learning

Standing on the shoulders of giants

The Problem with Training from Scratch

Scenario: You want to classify movie posters (images)

	Train from Scratch	Use Pretrained Model
Data needed	1 million images	1,000 images
Hardware	10 GPUs for a week	1 GPU for an hour
Expertise	ML PhD	Basic Python
Cost	$10,000+	~$1

The choice is obvious!

Transfer Learning: The Analogy

Someone who has never played any sport:

- Learning tennis takes 6 months
- Starts from zero

Someone who plays badminton:

- Learning tennis takes 2 months
- Already knows: hand-eye coordination, racket grip, court movement
- Just needs to learn: different swing, ball bounce

The badminton player transfers their skills!

Same idea in ML: Use knowledge from one task for another.

How Transfer Learning Works for Images

Google trained a model on **14 MILLION images** (ImageNet).

What it learned (from simple to complex):

Layer	What it Learned	Examples
1 (bottom)	Edges, lines	horizontal, vertical, diagonal
2	Textures	fur, metal, wood
3	Shapes	circles, squares, curves
4	Parts	eyes, wheels, leaves
5 (top)	Objects	cats, cars, trees

Lower layers = Universal (useful for any image task)
Higher layers = Task-specific (cats vs dogs vs cars)

Transfer Learning Strategy

Step 1: Take a pretrained model (trained on millions of images)

Step 2: Remove the last layer (the "head")

- Original: predicts 1000 ImageNet categories
- We don't need "cat", "dog", "airplane"

Step 3: Add our own head

- New layer: predicts OUR categories
- Movie poster → "Action", "Comedy", "Drama"

Step 4: Train only the new head (freeze everything else)

- Very fast! (minutes instead of days)

Transfer Learning Visualized

```
PRETRAINED MODEL              YOUR NEW MODEL
(from Google)                 (for movies)

┌─────────────┐               ┌──────────────┐
│ Cat/Dog/Car │  ──REMOVE──►  │ Action/Comedy│  ← NEW (train this)
│   (1000)    │               │    /Drama    │
└─────────────┘               └──────────────┘
      ↑                              ↑
┌─────────────┐               ┌──────────────┐
│   Shapes    │  ───KEEP───►  │    Shapes    │  ← FROZEN
└─────────────┘               └──────────────┘
      ↑                              ↑
┌─────────────┐               ┌──────────────┐
│    Edges    │  ───KEEP───►  │    Edges     │  ← FROZEN
└─────────────┘               └──────────────┘
```

Only train the top layer. Keep everything else frozen.

Transfer Learning for Text (LLMs)

Same idea works for text!

BERT (by Google) was trained on ALL of Wikipedia + Books.

What it learned:

- Grammar and syntax
- Word meanings and relationships
- Common knowledge ("Paris is in France")
- Context understanding

Your task: Classify movie reviews as Positive/Negative

Transfer: Use BERT's language understanding, just teach it your specific task.

Fine-Tuning: A Deeper Transfer

Feature Extraction: Freeze pretrained layers, only train new head

Fine-Tuning: Also slightly update the pretrained layers

	Feature Extraction	Fine-Tuning
Head	Train	Train
Top layers	Frozen	Train slowly
Bottom layers	Frozen	Frozen
Speed	Fast	Slower
Data needed	Less	More
Accuracy	Good	Better

Start with feature extraction. Fine-tune only if you need more accuracy and have enough data.

When to Use Transfer Learning

Data Type	Use Transfer Learning?	Recommended Models
Images	Yes!	ResNet, EfficientNet, ViT
Text	Yes!	BERT, RoBERTa, or LLM APIs
Audio	Yes!	Whisper, Wav2Vec
Tabular	Rarely	Use AutoML instead

Rule of thumb:

- Images, text, audio → **Transfer learning**
- Tabular data (spreadsheets) → **AutoML (AutoGluon)**

Transfer Learning Example Code

```python
from transformers import pipeline

# Load a pretrained sentiment classifier
classifier = pipeline("sentiment-analysis")

# Use it immediately - no training needed!
reviews = [
    "This movie was absolutely fantastic!",
    "Worst film I've ever seen.",
    "It was okay, nothing special."
]

for review in reviews:
    result = classifier(review)
```

Output:

```
This movie was absolutely fant... → POSITIVE
Worst film I've ever seen.... → NEGATIVE
It was okay, nothing special.... → NEGATIVE
```

Part 7: Putting It All Together

A complete workflow

The Complete Workflow

Step 1: Understand your data

- What type? (tabular, images, text)
- How much? (100 samples vs 1 million)

Step 2: Create a baseline

- Tabular → Logistic Regression or Random Forest
- Images/Text → Pretrained model (transfer learning)

Step 3: Evaluate with cross-validation

- Get reliable accuracy estimates
- Understand variance in performance

Step 4: Improve

- Tune hyperparameters

Netflix Movie Prediction: Full Example

```python
import pandas as pd
from sklearn.model_selection import cross_val_score
from sklearn.ensemble import RandomForestClassifier
from autogluon.tabular import TabularPredictor

# Load our movie data
movies = pd.read_csv('movies.csv')

# Baseline: Random Forest with cross-validation
rf = RandomForestClassifier(n_estimators=100)
baseline_scores = cross_val_score(rf, X, y, cv=5)
print(f"Baseline (RF): {baseline_scores.mean():.1%} ± {baseline_scores.std():.1%}")

# AutoML: Let AutoGluon do its magic
predictor = TabularPredictor(label='success')
predictor.fit(movies, time_limit=300)
print(predictor.leaderboard())
```

What Good Accuracy Looks Like

Model	Accuracy
Random guessing	50%
Majority class baseline	60%
Simple model (Logistic Reg)	72%
Better model (Random Forest)	78%
AutoML (AutoGluon)	82%
State-of-the-art	85%

Key questions to ask:

- Did you beat random guessing?
- Did you beat majority class?
- Is the improvement worth the complexity?

Note: 82% might be amazing for some problems and terrible for others. Context matters!

Key Takeaways

1. **Always start with a baseline**
    - Simple models are your reference point
    - You can't know if fancy is better without simple first
2. **Use cross-validation**
    - One test set can be misleading
    - 5-fold CV gives reliable estimates
3. **Tune hyperparameters** (or use AutoML)
    - Grid search, random search, or AutoGluon
    - Can improve accuracy by 10%+
4. **Use transfer learning for images/text**
    - Don't train from scratch
    - Pretrained models save time and work better

Common Mistakes to Avoid

- Starting with deep learning before trying simple models
- Evaluating on only one train/test split
- Tuning hyperparameters on the test set (this is cheating!)
- Training image/text models from scratch with small data
- Ignoring the baseline ("My model gets 80%!" ...vs what?)
- Over-engineering for tiny improvements (+0.5% isn't worth 10x complexity)

Next Week Preview

Week 8: Model Evaluation & Deployment

- Confusion matrices (understanding errors)
- Precision, Recall, F1 (beyond accuracy)
- When accuracy is misleading
- Deploying your model to production

You've built the model. Now how do you know it's REALLY good?

Lab Preview

This week's hands-on exercises:

1. **Build baselines**: Compare Logistic Regression, Decision Tree, Random Forest
2. **Cross-validate**: Use 5-fold CV to get reliable estimates
3. **Tune hyperparameters**: Use GridSearchCV and RandomizedSearchCV
4. **Try AutoGluon**: Let it find the best model for Netflix data
5. **Transfer learning demo**: Use a pretrained model for text classification

All code will be provided. Focus on understanding!

Questions?

Today's key concepts:

- Baseline models (start simple!)
- Cross-validation (reliable evaluation)
- Hyperparameter tuning (GridSearch, RandomSearch)
- AutoML (AutoGluon)
- Transfer learning (for images/text)

Remember: Simple first, complex only if needed!