Advanced Programming 2025
Course Wrap-Up
A Journey Through Python, Machine Learning, and Parallel Computing

Lecture 14

13 Weeks of Learning

December 2025

Course Wrap-Up Agenda (90-120 min)

Part I: Foundations (25 min)

- Python Fundamentals Recap
- Object-Oriented Programming
- Functions & Data Structures

Part II: Scientific Computing (20 min)

- NumPy & Pandas
- Data Visualization
- Debugging Techniques

Part III: Machine Learning (25 min)

- Regression Techniques
- Classification Methods
- Model Evaluation

Part IV: Deep Learning (20 min)

- Neural Networks
- PyTorch & TensorFlow
- Unsupervised Learning

Part V: Advanced Topics (20 min)

- Parallel Programming
- Finance Applications

Course Learning Journey

- **Weeks 1-4:** Python fundamentals, OOP, functions, data structures
- **Weeks 5-6:** Scientific computing, NumPy, Pandas, debugging
- **Weeks 7-9:** Machine learning: regression, classification, optimization
- **Weeks 10-11:** Deep learning, PyTorch, TensorFlow, workflows
- **Weeks 12-13:** Parallel programming, finance applications

Python: Primitive Data Types

```
# Numeric types
x = 42            # int
y = 3.14          # float
z = 2 + 3j        # complex

# Boolean
flag = True       # bool

# String
name = "Python" # str

# None type
empty = None      # NoneType
```

Type Checking:

```
type(x)           # <class 'int'>
isinstance(x, int)  # True

# Type conversion
int("42")         # 42
float("3.14")     # 3.14
str(42)           # "42"
bool(0)           # False
bool(1)           # True
```

Python: Container Types

List (mutable, ordered)

```
nums = [1, 2, 3, 4]
nums.append(5)
nums[0] = 10
```

Dictionary (key-value pairs)

```
student = {"name": "Alice",
           "grade": 95}
student["age"] = 20
```

Tuple (immutable, ordered)

```
point = (10, 20)
x, y = point  # unpacking
```

Set (unique elements)

```
unique = {1, 2, 3, 3}  # {1, 2, 3}
```

Key Insight: Choose the right data structure for your use case!

Control Flow: Conditionals and Loops

Conditional Statements

```
if condition:
    # execute if True
elif other_condition:
    # alternative
else:
    # default case
```

While Loops

```
while condition:
    if done: break
    if skip: continue
```

For Loops

```
for item in collection:
    process(item)

for i, item in enumerate(lst):
    print(f"{i}: {item}")
```

List Comprehensions

```
squares = [x**2 for x in range(10)]
evens = [x for x in nums if x % 2 ==
    0]
```

Functions: Building Blocks of Programs

```python
def calculate_statistics(data, verbose=False):
    """Calculate mean and standard deviation."""
    n = len(data)
    mean = sum(data) / n
    variance = sum((x - mean)**2 for x in data) / n
    std_dev = variance ** 0.5

    if verbose:
        print(f"Mean: {mean:.2f}, Std: {std_dev:.2f}")
    return mean, std_dev

# Usage
mean, std = calculate_statistics([1, 2, 3, 4, 5], verbose=True)
```

Key Concepts:

- Default parameters, docstrings, multiple return values
- Scope: local vs global variables
- Lambda functions: `lambda x: x**2`

Object-Oriented Programming: Classes

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance

    @property
    def balance(self):
        return self._balance

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount

    def __str__(self):
        return f"{self.owner}: ${self._balance}"
```

OOP Principles:

- **Encapsulation**
- **Abstraction**
- **Inheritance**
- **Polymorphism**

Special Methods:

- `__init__`: Constructor
- `__str__`: String repr
- `__add__`: + operator

Inheritance and Polymorphism

```
class Shape:
    def area(self):
        raise NotImplementedError("Subclass must implement")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width, self.height = width, height
    def area(self):
        return self.width * self.height

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def area(self):
        return 3.14159 * self.radius ** 2

# Polymorphism: same interface, different behaviors
shapes = [Rectangle(4, 5), Circle(3)]
total = sum(s.area() for s in shapes)  # Works for any Shape!
```

Exception Handling

```python
def safe_divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Cannot divide by zero!")
        return None
    except TypeError as e:
        print(f"Type error: {e}")
        return None
    else:
        return result
    finally:
        print("Operation complete")
```

Best Practices:

- Catch specific exceptions
- Don't silence errors blindly
- Use `finally` for cleanup

Common Exceptions:

- `ValueError`
- `TypeError`
- `KeyError`
- `IndexError`

NumPy: Array Creation

```
import numpy as np

# From lists
arr = np.array([1, 2, 3, 4])

# Special arrays
zeros = np.zeros((3, 4))
ones = np.ones((2, 2))
identity = np.eye(3)

# Ranges
range_arr = np.arange(0, 10, 0.5)
linspace = np.linspace(0, 1, 100)

# Random
random = np.random.randn(3, 3)
```

Array Operations

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

c = a + b       # [5, 7, 9]
d = a * b       # [4, 10, 18]
e = a ** 2      # [1, 4, 9]

# Aggregations
np.sum(a)       # 6
np.mean(a)      # 2.0
np.std(a)       # 0.816
```

Why NumPy? 10-100x faster!

NumPy: Indexing and Slicing

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

# Basic indexing
element = arr[1, 2]             # 7
row = arr[0, :]                 # [1, 2, 3, 4]
col = arr[:, 1]                 # [2, 6, 10]

# Slicing
subarray = arr[0:2, 1:3]        # [[2, 3], [6, 7]]

# Boolean indexing
mask = arr > 5
filtered = arr[mask]            # [6, 7, 8, 9, 10, 11, 12]

# Fancy indexing
selected = arr[[0, 2]]          # rows 0 and 2
```

Pandas: DataFrame Basics

Creation & Info

```
import pandas as pd

df = pd.DataFrame({
    'name': ['Alice', 'Bob'],
    'age': [25, 30],
    'salary': [50000, 60000]
})

# From file
df = pd.read_csv('data.csv')

# Info
df.shape       # (rows, cols)
df.describe() # statistics
```

Selection

```
# Columns
names = df['name']
subset = df[['name', 'age']]

# Rows
df.loc[0]     # by label
df.iloc[0]    # by position

# Conditional
high = df[df['salary'] > 55000]

# Sorting & Grouping
df.sort_values('salary')
df.groupby('dept').mean()
```

Pandas: Data Manipulation

```
# Handle missing values
df.dropna()                    # Remove rows with NaN
df.fillna(0)                   # Fill NaN with 0
df.fillna(df.mean())           # Fill with column mean

# Create new columns
df['return'] = df['close'].pct_change()
df['log_ret'] = np.log(df['close'] / df['close'].shift(1))

# Apply functions
df['cat'] = df['return'].apply(lambda x: 'up' if x > 0 else 'down')

# Merge DataFrames
merged = pd.merge(df1, df2, on='date', how='inner')

# Pivot tables
pivot = df.pivot_table(values='return', index='year', columns='month')
```

Data Visualization: Matplotlib

```
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
y = np.sin(x)

plt.figure(figsize=(8, 5))
plt.plot(x, y, 'b-', label='sin(x)')
plt.xlabel('x')
plt.ylabel('y')
plt.title('Sine Function')
plt.legend()
plt.grid(True)
plt.savefig('plot.png')
plt.show()
```

Common Plot Types:

- `plot()` – Line plots
- `scatter()` – Scatter plots
- `hist()` – Histograms
- `bar()` – Bar charts
- `boxplot()` – Box plots

Subplots:

```
fig, axes = plt.subplots(2, 2)
axes[0,0].plot(x, y)
```

Debugging Techniques

Print Debugging

```
def buggy_function(data):
    print(f"Input: {data}")
    for i, item in enumerate(data):
        print(f"Item {i}: {item}")
        result = process(item)
    return result
```

Using pdb

```
import pdb
pdb.set_trace()  # Breakpoint
# n(ext), s(tep), c(ontinue), q(uit)
```

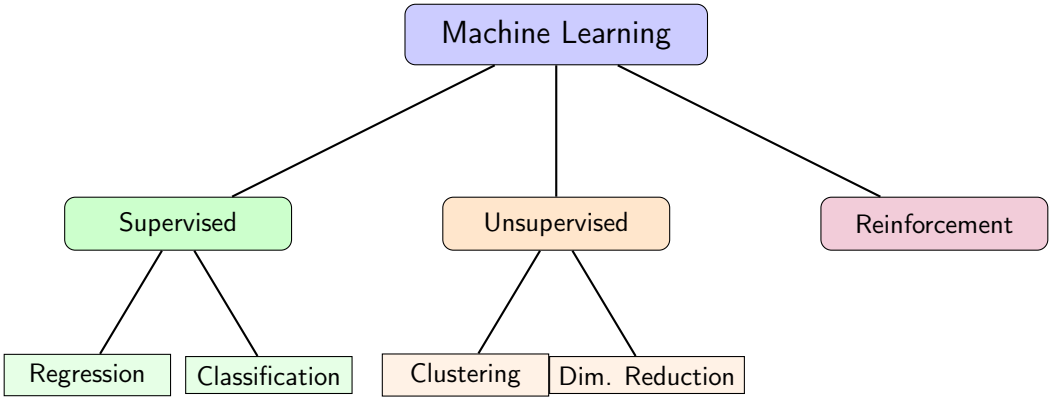
Strategies:

1. Reproduce consistently
2. Isolate the problem
3. Check with assertions
4. Binary search for bug
5. Read error messages!

Assertions

```
assert value >= 0, "Must be positive"
```

Machine Learning Overview



- **Supervised:** Learn from labeled data (regression, classification)
- **Unsupervised:** Find patterns in unlabeled data (clustering, PCA)
- **Reinforcement:** Learn through interaction and rewards

Linear Regression

Model: $\hat{y} = \beta_0 + \beta_1 x_1 + \ldots + \beta_n x_n$

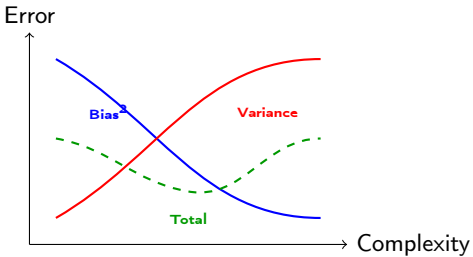
Loss (MSE): $L = \frac{1}{n} \sum (y_i - \hat{y}_i)^2$

```
from sklearn.linear_model import \
    LinearRegression

X = data[['feat1', 'feat2']]
y = data['target']

model = LinearRegression()
model.fit(X, y)

pred = model.predict(X_test)
print(f"R^2: {model.score(X, y):.3f}")
```

Polynomial Regression:

```
from sklearn.preprocessing import \
    PolynomialFeatures
from sklearn.pipeline import Pipeline

model = Pipeline([
    ('poly', PolynomialFeatures(3)),
    ('linear', LinearRegression())
])
model.fit(X, y)
```

Bias-Variance Tradeoff



- **High Bias:** Underfitting (too simple)
- **High Variance:** Overfitting (too complex)
- **Goal:** Find the sweet spot!

Total Error:

$$\text{Error} = \text{Bias}^2 + \text{Variance} + \text{Noise}$$

Regularization: Ridge & Lasso

Problem: Overfitting with many features

Ridge (L2): $L + \lambda \sum \beta_j^2$

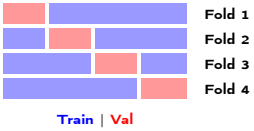
```
from sklearn.linear_model import Ridge

model = Ridge(alpha=1.0)
model.fit(X_train, y_train)

# Cross-validation
from sklearn.linear_model import RidgeCV
model = RidgeCV(alphas=[0.1,1,10])
```

Lasso (L1): $L + \lambda \sum |\beta_j|$

```
from sklearn.linear_model import Lasso

model = Lasso(alpha=0.1)
model.fit(X_train, y_train)
# Produces sparse solutions!
```

λ controls regularization strength

K-Fold Cross-Validation




```python
from sklearn.model_selection import \
    cross_val_score, KFold

kf = KFold(n_splits=5, shuffle=True)

scores = cross_val_score(
    model, X, y, cv=kf,
    scoring='neg_mean_squared_error'
)

print(f"CV MSE: {-scores.mean():.3f}")
print(f"Std: {scores.std():.3f}")
```

Gradient Descent

Update rule: $\theta_{t+1} = \theta_t - \alpha \nabla L(\theta_t)$

```
def gradient_descent(X, y, lr=0.01,
                     epochs=1000):
    theta = np.zeros(X.shape[1])
    for _ in range(epochs):
        pred = X @ theta
        grad = X.T @ (pred - y) / len(y)
        theta -= lr * grad
    return theta
```

Variants:

- **Batch:** All data (slow, accurate)
- **Stochastic:** 1 sample (fast, noisy)
- **Mini-batch:** Best of both

Key: Learning rate α

- Too high $\rightarrow$ diverge
- Too low $\rightarrow$ slow

Classification: Logistic Regression

Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$ maps to probability [0,1]

```
from sklearn.linear_model import \
    LogisticRegression

model = LogisticRegression()
model.fit(X_train, y_train)

# Probabilities
probs = model.predict_proba(X_test)

# Classes
pred = model.predict(X_test)
```

Cross-Entropy Loss:

$$L = -\frac{1}{n}\sum[y\log(\hat{p}) + (1-y)\log(1-\hat{p})]$$

Decision Rule:

$$\hat{y} = \begin{cases} 1 & \hat{p} \geq 0.5 \\ 0 & \text{else} \end{cases}$$

Classification Metrics

Confusion Matrix:

	Pred 0	Pred 1
Act 0	TN	FP
Act 1	FN	TP

Metrics:

- Accuracy: $\frac{TP+TN}{Total}$
- Precision: $\frac{TP}{TP+FP}$
- Recall: $\frac{TP}{TP+FN}$
- F1: $\frac{2PR}{P+R}$

```
from sklearn.metrics import (
    confusion_matrix,
    classification_report,
    accuracy_score
)

cm = confusion_matrix(y_true, y_pred)
print(classification_report(
    y_true, y_pred))
acc = accuracy_score(y_true, y_pred)
```

K-Nearest Neighbors (KNN)

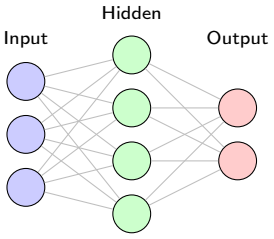
```
from sklearn.neighbors import \
    KNeighborsClassifier
from sklearn.preprocessing import \
    StandardScaler

# Scale features first!
scaler = StandardScaler()
X_sc = scaler.fit_transform(X)

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train_sc, y_train)
pred = knn.predict(X_test_sc)
```

Key Considerations:

- Feature scaling essential!
- Small k = noisy
- Large k = smooth
- Distance: Euclidean, Manhattan

Pros: Simple, no training
Cons: Slow prediction, curse of dimensionality

Neural Networks Architecture



Components:

- Neurons: weighted sum + activation
- Layers: Input $\rightarrow$ Hidden $\rightarrow$ Output
- Backpropagation: chain rule gradients

Activation Functions:

- ReLU: $\max(0, x)$
- Sigmoid: $\frac{1}{1+e^{-x}}$
- Softmax: probabilities

Building Neural Networks with Keras

```
import tensorflow as tf
from tensorflow.keras import layers, Sequential

model = Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dropout(0.2),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=10,
                    batch_size=32, validation_split=0.2)
```

PyTorch: Define and Train

Define Model:

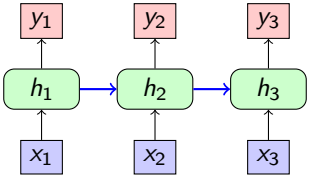
```
import torch.nn as nn

class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.fc1 = nn.Linear(784, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = nn.ReLU()(self.fc1(x))
        return self.fc2(x)
```

Training Loop:

```
model = Net()
criterion = nn.CrossEntropyLoss()
opt = torch.optim.Adam(
    model.parameters(), lr=0.001)

for epoch in range(10):
    out = model(X_batch)
    loss = criterion(out, y_batch)
    opt.zero_grad()
    loss.backward()
    opt.step()
```

Recurrent Neural Networks (RNNs)

Use Case: Sequential data (time series, text)




```
model = Sequential([
    layers.LSTM(64, return_sequences=True)
        ,
    layers.LSTM(32),
    layers.Dense(1)
])
```

Variants:

- LSTM: Long Short-Term Memory
- GRU: Gated Recurrent Unit

Principal Component Analysis (PCA)

Steps:

1. Standardize data
2. Compute covariance matrix
3. Find eigenvectors/values
4. Project onto top k components

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import \
    StandardScaler

X_sc = StandardScaler().fit_transform(X)
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_sc)
print(pca.explained_variance_ratio_)
```

Applications:

- Visualization (2-3D)
- Noise reduction
- Feature extraction
- Data compression

Key: Principal components are orthogonal directions of maximum variance

Clustering: K-Means & GMM

K-Means:

```
from sklearn.cluster import KMeans

km = KMeans(n_clusters=3)
labels = km.fit_predict(X)

# Elbow method
inertias = [KMeans(k).fit(X).inertia_
            for k in range(1, 11)]
```

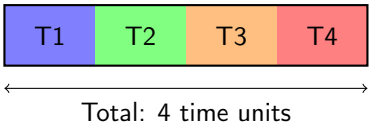
Gaussian Mixture:

```
from sklearn.mixture import \
    GaussianMixture

gmm = GaussianMixture(n_components=3)
gmm.fit(X)
probs = gmm.predict_proba(X)
labels = gmm.predict(X)
```

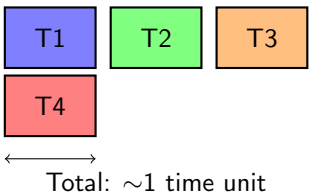
K-Means: Hard assignments, spherical clusters
GMM: Soft assignments (probabilities), elliptical clusters

Why Parallel Programming?

Sequential Execution



Parallel Execution (4 cores)



Amdahl's Law: Speedup $= \frac{1}{(1-P)+P/N}$ (P = parallel fraction, N = processors)

I/O-Bound vs CPU-Bound Tasks

I/O-Bound → Threading

- Waiting for external resources
- Network requests, file I/O
- Database queries
- GIL released during I/O wait

Examples:

- API data fetching
- Reading multiple files
- Web scraping

CPU-Bound → Multiprocessing

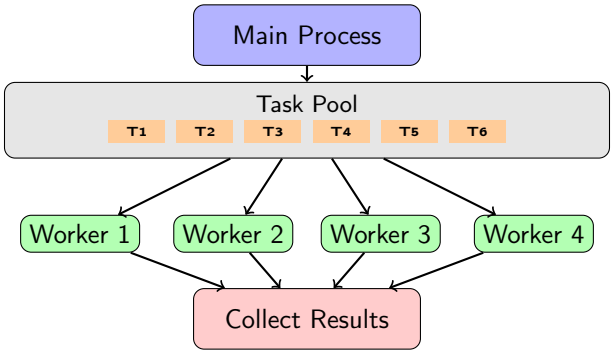
- Heavy computation
- Number crunching
- Bypasses GIL (separate processes)

Examples:

- Monte Carlo simulations
- Image processing
- Numerical optimization

Python GIL: Only one thread executes Python bytecode at a time!

Parallel Work Distribution



Pattern: Split tasks → Distribute to workers → Collect results

Threading for I/O-Bound Tasks

```python
from concurrent.futures import ThreadPoolExecutor, as_completed
import requests

def fetch_stock(symbol):
    url = f"https://api.example.com/stocks/{symbol}"
    return symbol, requests.get(url).json()

symbols = ['AAPL', 'GOOGL', 'MSFT', 'AMZN', 'META']

# Sequential: ~5 sec | Parallel: ~1 sec
with ThreadPoolExecutor(max_workers=5) as executor:
    futures = {executor.submit(fetch_stock, s): s for s in symbols}
    for future in as_completed(futures):
        sym, data = future.result()
        print(f"{sym}: {data['price']}")
```

Multiprocessing for CPU-Bound Tasks

```python
from concurrent.futures import ProcessPoolExecutor
import numpy as np

def monte_carlo_option(params):
    S0, K, r, sigma, T, n = params
    Z = np.random.standard_normal(n)
    ST = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
    payoffs = np.maximum(ST - K, 0)
    return np.exp(-r*T) * np.mean(payoffs)

params = [(100, 100, 0.05, 0.2, 1, 100000) for _ in range(8)]

with ProcessPoolExecutor() as executor:
    prices = list(executor.map(monte_carlo_option, params))

print(f"Price: {np.mean(prices):.4f} +/- {np.std(prices):.4f}")
```

Finance: Bootstrap Sharpe Ratio

```python
from concurrent.futures import ProcessPoolExecutor
import numpy as np

def bootstrap_sharpe(returns, n_boot=1000):
    sharpes = []
    for _ in range(n_boot):
        sample = np.random.choice(returns, len(returns), replace=True)
        sharpe = np.mean(sample) / np.std(sample) * np.sqrt(252)
        sharpes.append(sharpe)
    return np.percentile(sharpes, [2.5, 97.5])

# Parallel for multiple assets
with ProcessPoolExecutor() as ex:
    results = list(ex.map(bootstrap_sharpe, [ret_A, ret_B, ret_C]))
```

Use cases: Monte Carlo, VaR, portfolio optimization, parameter sweeps

Parallel Programming Best Practices

Do:

- Profile first to find bottleneck
- Match tool to task type
- Handle exceptions in workers
- Test with small data first
- Use context managers (`with`)

Don't:

- Share mutable state
- Parallelize tiny tasks
- Use threading for CPU work

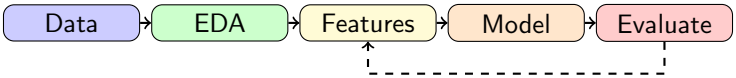
Advanced Tools:

- **joblib:** Simple parallel loops
- **dask:** Parallel pandas/arrays
- **ray:** Distributed computing
- **numba:** JIT compilation

When to Parallelize:

- Task $>$ 1 second
- Multiple independent tasks
- Clear I/O or CPU bound

End-to-End Pipeline



1. **Data:** Collect, clean, validate
2. **EDA:** Explore distributions, correlations, outliers
3. **Features:** Transform, create, select
4. **Model:** Train, tune hyperparameters
5. **Evaluate:** Test, interpret, iterate

Course Technologies Summary

Domain	Key Libraries
Core Python	Built-ins, itertools, functools
Scientific Computing	NumPy, SciPy, SymPy
Data Analysis	Pandas, Matplotlib, Seaborn
Machine Learning	scikit-learn
Deep Learning	TensorFlow/Keras, PyTorch
Parallel Computing	concurrent.futures, multiprocessing
Development	Jupyter, Git, TensorBoard

Key: Know when to use each tool!

Key Takeaways

Foundations:

- Right data structure
- Clean, readable code
- OOP principles
- Exception handling

ML/DL:

- Bias-variance tradeoff
- Cross-validation
- Feature scaling
- Regularization

Advanced:

- Threading vs multiproc
- Profile first
- Handle exceptions
- Iterate on pipeline

Algorithm Selection Guide

Problem	Algorithm	When
Regression	Linear/Ridge/Lasso	Linear, regularization needed
	Random Forest	Non-linear, feature importance
Classification	Logistic Regression	Baseline, interpretable
	KNN	Simple, non-parametric
	Neural Network	Complex patterns, lots of data
Clustering	K-Means	Spherical, fast
	GMM	Soft assignments, elliptical
Dim. Reduction	PCA	Linear, preserves variance

Resources

Books:

- Python Data Science Handbook
- Hands-On ML (Géron)
- Deep Learning (Goodfellow)

Online:

- fast.ai
- Coursera ML/DL
- scikit-learn.org

Practice:

- Kaggle competitions
- GitHub projects
- Personal projects!

"The best way to learn is by doing."

Thank You!

Questions?

Good luck with your future endeavors in
programming, data science, and machine learning!