Machine Learning Models

Code Examples — Supervised, Unsupervised & Reinforcement Learning

---

Minimal, runnable Python code for every model in the companion notes document — Scikit-learn, PyTorch/Keras, and Gym where relevant.

Prepared for Ashutosh — Code With Purpose

Contents

1. Supervised — Regression: Code Examples

Linear Regression

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

df = pd.read_csv("house_prices.csv")
X = df.drop(columns=["price"])
X = pd.get_dummies(X, drop_first=True)   # one-hot encode categoricals
y = df["price"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = LinearRegression()
model.fit(X_train, y_train)
preds = model.predict(X_test)

print("RMSE:", mean_squared_error(y_test, preds, squared=False))
print("R2:", r2_score(y_test, preds))
print("Coefficients:", dict(zip(X.columns, model.coef_)))
```

Polynomial Regression

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt

X = np.array([[1],[2],[3],[4],[5],[6],[7],[8],[9],[10]])
y = np.array([45000,50000,60000,80000,110000,150000,200000,300000,500000,1000000])

poly = PolynomialFeatures(degree=4)
X_poly = poly.fit_transform(X)

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

X_grid = np.arange(1, 10.1, 0.1).reshape(-1, 1)
y_pred = model.predict(poly.transform(X_grid))

plt.scatter(X, y, color="red")
plt.plot(X_grid, y_pred, color="blue")
plt.title("Polynomial Regression: Position Level vs Salary")
plt.xlabel("Position Level"); plt.ylabel("Salary")
plt.show()

print("Prediction for level 6.5:", model.predict(poly.transform([[6.5]]))[0])
```

Ridge & Lasso Regression

```
from sklearn.linear_model import Ridge, Lasso
from sklearn.model_selection import cross_val_score
import numpy as np

# X, y assumed already loaded and preprocessed (scaled numeric features)
ridge = Ridge(alpha=1.0)
lasso = Lasso(alpha=0.1)

ridge_scores = cross_val_score(ridge, X, y, cv=5, scoring="neg_root_mean_squared_error")
lasso_scores = cross_val_score(lasso, X, y, cv=5, scoring="neg_root_mean_squared_error")

print("Ridge RMSE:", -np.mean(ridge_scores))
print("Lasso RMSE:", -np.mean(lasso_scores))
```

```
lasso.fit(X, y)
zero_coef_features = [f for f, c in zip(X.columns, lasso.coef_) if c == 0]
print("Features Lasso dropped (coef = 0):", zero_coef_features)
```

Decision Tree Regressor

```
from sklearn.tree import DecisionTreeRegressor, plot_tree
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# df: bike sharing dataset with features (season, hour, weather, holiday) and target 'count'
X = df.drop(columns=["count"])
y = df["count"]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

tree = DecisionTreeRegressor(max_depth=6, min_samples_leaf=10, random_state=42)
tree.fit(X_train, y_train)
preds = tree.predict(X_test)

print("RMSE:", mean_squared_error(y_test, preds, squared=False))

plt.figure(figsize=(16, 8))
plot_tree(tree, feature_names=X.columns, filled=True, max_depth=3)
plt.show()
```

2. Supervised — Classification: Code Examples

Logistic Regression

```
from sklearn.linear_model import LogisticRegression
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report, roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

model = LogisticRegression(max_iter=1000)
model.fit(X_train_scaled, y_train)

preds = model.predict(X_test_scaled)
probs = model.predict_proba(X_test_scaled)[:, 1]

print(confusion_matrix(y_test, preds))
print(classification_report(y_test, preds))
print("ROC-AUC:", roc_auc_score(y_test, probs))

# Interpret coefficients as odds ratios
import numpy as np
odds_ratios = np.exp(model.coef_[0])
print(dict(zip(X.columns, odds_ratios)))
```

Decision Tree Classifier

```
from sklearn.tree import DecisionTreeClassifier, plot_tree
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

clf = DecisionTreeClassifier(max_depth=4, criterion="gini", random_state=42)
clf.fit(X_train, y_train)

print("Test Accuracy:", clf.score(X_test, y_test))

importances = dict(zip(X.columns, clf.feature_importances_))
print("Feature importances:", sorted(importances.items(), key=lambda x: -x[1]))

plt.figure(figsize=(14, 7))
plot_tree(clf, feature_names=X.columns, class_names=["Rejected", "Approved"], filled=True)
plt.show()
```

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import average_precision_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

param_grid = {"n_estimators": [100, 200], "max_depth": [8, 12, None]}
rf = RandomForestClassifier(class_weight="balanced", random_state=42)

grid = GridSearchCV(rf, param_grid, cv=3, scoring="average_precision")
grid.fit(X_train, y_train)

best_model = grid.best_estimator_
probs = best_model.predict_proba(X_test)[:, 1]
```

```
print("Best params:", grid.best_params_)
print("PR-AUC:", average_precision_score(y_test, probs))
```

Support Vector Machine (SVM)

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler

digits = load_digits()
X, y = digits.data, digits.target
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

param_grid = {"C": [0.1, 1, 10], "gamma": ["scale", 0.01, 0.001]}
grid = GridSearchCV(SVC(kernel="rbf"), param_grid, cv=3)
grid.fit(X_train_s, y_train)

print("Best params:", grid.best_params_)
print("Test accuracy:", grid.score(X_test_s, y_test))
```

K-Nearest Neighbors (KNN)

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import StandardScaler

# user_item_matrix: rows = users, columns = movies, values = ratings (0 if unrated)
scaler = StandardScaler()
scaled_matrix = scaler.fit_transform(user_item_matrix)

knn = NearestNeighbors(n_neighbors=6, metric="euclidean")
knn.fit(scaled_matrix)

target_user_idx = 0
distances, indices = knn.kneighbors([scaled_matrix[target_user_idx]])
similar_users = indices[0][1:]   # exclude the user themself

# Recommend movies highly rated by similar users but not yet seen by target user
unseen = np.where(user_item_matrix[target_user_idx] == 0)[0]
recommend_scores = user_item_matrix[similar_users][:, unseen].mean(axis=0)
top_movie_indices = unseen[np.argsort(-recommend_scores)[:5]]
print("Recommended movie indices:", top_movie_indices)
```

Naive Bayes

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score, confusion_matrix

# df: columns 'message' (text) and 'label' (spam/ham)
X_train, X_test, y_train, y_test = train_test_split(
    df["message"], df["label"], test_size=0.2, random_state=42)

vectorizer = TfidfVectorizer(stop_words="english", max_features=3000)
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)

model = MultinomialNB()
model.fit(X_train_vec, y_train)
preds = model.predict(X_test_vec)

print("Accuracy:", accuracy_score(y_test, preds))
```

```
print(confusion_matrix(y_test, preds))

new_msg = ["Congratulations! You've won a free prize, click here now!"]
print("Prediction:", model.predict(vectorizer.transform(new_msg)))
```

Gradient Boosting (XGBoost)

```
import xgboost as xgb
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, stratify=y, random_state=42)

model = xgb.XGBClassifier(
    n_estimators=500, learning_rate=0.05, max_depth=5,
    eval_metric="auc", early_stopping_rounds=20, random_state=42
)
model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)

probs = model.predict_proba(X_test)[:, 1]
print("ROC-AUC:", roc_auc_score(y_test, probs))

xgb.plot_importance(model, max_num_features=10, importance_type="gain")
```

Neural Network (Multi-Layer Perceptron)

```
import tensorflow as tf
from tensorflow import keras
from sklearn.model_selection import train_test_split

# X, y: image data flattened/normalized (e.g., Fashion-MNIST), y one-hot or sparse labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

model = keras.Sequential([
    keras.layers.Dense(128, activation="relu", input_shape=(X_train.shape[1],)),
    keras.layers.Dropout(0.3),
    keras.layers.Dense(64, activation="relu"),
    keras.layers.Dense(10, activation="softmax")   # 10 classes
])

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

early_stop = keras.callbacks.EarlyStopping(patience=5, restore_best_weights=True)
history = model.fit(X_train, y_train, validation_split=0.2,
                    epochs=50, batch_size=64, callbacks=[early_stop])

test_loss, test_acc = model.evaluate(X_test, y_test)
print("Test accuracy:", test_acc)
```

3. Unsupervised Learning: Code Examples

K-Means Clustering

```
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt

df = pd.read_csv("mall_customers.csv")
X = df[["Annual_Income", "Spending_Score"]]
X_scaled = StandardScaler().fit_transform(X)

# Elbow method to choose K
wcss = []
for k in range(1, 11):
    km = KMeans(n_clusters=k, n_init=10, random_state=42)
    km.fit(X_scaled)
    wcss.append(km.inertia_)
plt.plot(range(1, 11), wcss, marker="o")
plt.xlabel("K"); plt.ylabel("WCSS"); plt.title("Elbow Method")
plt.show()

# Fit final model (say K=5 chosen from the elbow plot)
kmeans = KMeans(n_clusters=5, n_init=10, random_state=42)
df["cluster"] = kmeans.fit_predict(X_scaled)

plt.scatter(X["Annual_Income"], X["Spending_Score"], c=df["cluster"], cmap="viridis")
plt.xlabel("Annual Income"); plt.ylabel("Spending Score")
plt.show()
```

Hierarchical Clustering

```
import scipy.cluster.hierarchy as sch
from sklearn.cluster import AgglomerativeClustering
from sklearn.preprocessing import StandardScaler

# rfm_df: columns Recency, Frequency, Monetary per customer
X_scaled = StandardScaler().fit_transform(rfm_df)

# Plot dendrogram
dendrogram = sch.dendrogram(sch.linkage(X_scaled, method="ward"))

model = AgglomerativeClustering(n_clusters=4, linkage="ward")
rfm_df["cluster"] = model.fit_predict(X_scaled)

print(rfm_df.groupby("cluster")[["Recency", "Frequency", "Monetary"]].mean())
```

DBSCAN

```
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

X_scaled = StandardScaler().fit_transform(traffic_features)

dbscan = DBSCAN(eps=0.5, min_samples=5)
labels = dbscan.fit_predict(X_scaled)

n_anomalies = (labels == -1).sum()
print(f"Detected {n_anomalies} anomalous points out of {len(labels)}")

# Visualize in 2D via PCA
X_2d = PCA(n_components=2).fit_transform(X_scaled)
plt.scatter(X_2d[:, 0], X_2d[:, 1], c=labels, cmap="tab10")
plt.title("DBSCAN Clusters (label -1 = anomaly)")
```

```
plt.show()
```

Principal Component Analysis (PCA)

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_wine
import matplotlib.pyplot as plt

data = load_wine()
X_scaled = StandardScaler().fit_transform(data.data)

pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

print("Explained variance ratio:", pca.explained_variance_ratio_)
print("Total variance captured:", sum(pca.explained_variance_ratio_))

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=data.target, cmap="viridis")
plt.xlabel("PC1"); plt.ylabel("PC2")
plt.title("Wine Dataset Projected to 2D via PCA")
plt.show()
```

Association Rule Learning (Apriori)

```
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
from mlxtend.preprocessing import TransactionEncoder

# transactions: list of lists, e.g. [['bread','butter'], ['milk','bread','eggs'], ...]
te = TransactionEncoder()
te_array = te.fit(transactions).transform(transactions)
basket_df = pd.DataFrame(te_array, columns=te.columns_)

frequent_itemsets = apriori(basket_df, min_support=0.02, use_colnames=True)
rules = association_rules(frequent_itemsets, metric="lift", min_threshold=1.2)

top_rules = rules.sort_values("lift", ascending=False).head(10)
print(top_rules[["antecedents", "consequents", "support", "confidence", "lift"]])
```

Autoencoders

```
import tensorflow as tf
from tensorflow import keras
import numpy as np

# X_train_clean, X_train_noisy: normalized MNIST images (28x28), noisy versions have Gaussian noise added
input_dim = 784   # 28*28 flattened

input_img = keras.Input(shape=(input_dim,))
encoded = keras.layers.Dense(128, activation="relu")(input_img)
encoded = keras.layers.Dense(32, activation="relu")(encoded)   # bottleneck
decoded = keras.layers.Dense(128, activation="relu")(encoded)
decoded = keras.layers.Dense(input_dim, activation="sigmoid")(decoded)

autoencoder = keras.Model(input_img, decoded)
autoencoder.compile(optimizer="adam", loss="mse")

autoencoder.fit(X_train_noisy, X_train_clean,
                epochs=20, batch_size=256, shuffle=True,
                validation_split=0.1)

denoised = autoencoder.predict(X_train_noisy[:5])
# Plot denoised vs noisy vs clean images with matplotlib to compare
```

4. Reinforcement Learning: Code Examples

Q-Learning

```
import numpy as np
import gym

env = gym.make("FrozenLake-v1", is_slippery=False)
n_states = env.observation_space.n
n_actions = env.action_space.n

Q = np.zeros((n_states, n_actions))
alpha, gamma, epsilon = 0.8, 0.95, 1.0
episodes = 2000

for ep in range(episodes):
    state, _ = env.reset()
    done = False
    while not done:
        if np.random.rand() < epsilon:
            action = env.action_space.sample()
        else:
            action = np.argmax(Q[state])

        next_state, reward, done, truncated, _ = env.step(action)
        Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state]) - Q[state, action])
        state = next_state
    epsilon = max(0.01, epsilon * 0.995)

print("Learned Q-table:\n", Q)
```

SARSA

```
import numpy as np
import gym

env = gym.make("CliffWalking-v0")
n_states = env.observation_space.n
n_actions = env.action_space.n

Q = np.zeros((n_states, n_actions))
alpha, gamma, epsilon = 0.5, 0.99, 0.1
episodes = 500

def choose_action(state):
    if np.random.rand() < epsilon:
        return env.action_space.sample()
    return np.argmax(Q[state])

for ep in range(episodes):
    state, _ = env.reset()
    action = choose_action(state)
    done = False
    while not done:
        next_state, reward, done, truncated, _ = env.step(action)
        next_action = choose_action(next_state)
        Q[state, action] += alpha * (reward + gamma * Q[next_state, next_action] - Q[state, action])
        state, action = next_state, next_action

print("Learned Q-table shape:", Q.shape)
```

Deep Q-Network (DQN)

```
import torch, torch.nn as nn, torch.optim as optim
import random, numpy as np
from collections import deque
import gym
```

```
env = gym.make("CartPole-v1")
state_dim = env.observation_space.shape[0]
n_actions = env.action_space.n

class QNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 64), nn.ReLU(),
            nn.Linear(64, 64), nn.ReLU(),
            nn.Linear(64, n_actions))
    def forward(self, x):
        return self.net(x)

q_net = QNet()
target_net = QNet()
target_net.load_state_dict(q_net.state_dict())
optimizer = optim.Adam(q_net.parameters(), lr=1e-3)
replay_buffer = deque(maxlen=10000)
gamma, epsilon, batch_size = 0.99, 1.0, 64

def select_action(state):
    if random.random() < epsilon:
        return env.action_space.sample()
    with torch.no_grad():
        return q_net(torch.FloatTensor(state)).argmax().item()

for episode in range(300):
    state, _ = env.reset()
    done = False
    while not done:
        action = select_action(state)
        next_state, reward, done, truncated, _ = env.step(action)
        replay_buffer.append((state, action, reward, next_state, done))
        state = next_state

        if len(replay_buffer) >= batch_size:
            batch = random.sample(replay_buffer, batch_size)
            s, a, r, s2, d = zip(*batch)
            s, s2 = torch.FloatTensor(s), torch.FloatTensor(s2)
            a, r, d = torch.LongTensor(a), torch.FloatTensor(r), torch.FloatTensor(d)

            q_vals = q_net(s).gather(1, a.unsqueeze(1)).squeeze()
            next_q = target_net(s2).max(1)[0]
            target = r + gamma * next_q * (1 - d)

            loss = nn.functional.mse_loss(q_vals, target.detach())
            optimizer.zero_grad(); loss.backward(); optimizer.step()

    epsilon = max(0.05, epsilon * 0.98)
    if episode % 10 == 0:
        target_net.load_state_dict(q_net.state_dict())
```

Policy Gradient (REINFORCE)

```
import torch, torch.nn as nn, torch.optim as optim
import gym

env = gym.make("CartPole-v1")
state_dim = env.observation_space.shape[0]
n_actions = env.action_space.n

class PolicyNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(state_dim, 64), nn.ReLU(),
            nn.Linear(64, n_actions), nn.Softmax(dim=-1))
    def forward(self, x):
        return self.net(x)
```

```
policy = PolicyNet()
optimizer = optim.Adam(policy.parameters(), lr=1e-2)
gamma = 0.99

for episode in range(500):
    state, _ = env.reset()
    log_probs, rewards = [], []
    done = False
    while not done:
        probs = policy(torch.FloatTensor(state))
        dist = torch.distributions.Categorical(probs)
        action = dist.sample()
        log_probs.append(dist.log_prob(action))

        state, reward, done, truncated, _ = env.step(action.item())
        rewards.append(reward)

    # Compute discounted returns
    returns, G = [], 0
    for r in reversed(rewards):
        G = r + gamma * G
        returns.insert(0, G)
    returns = torch.FloatTensor(returns)
    returns = (returns - returns.mean()) / (returns.std() + 1e-8)

    loss = -torch.stack([lp * G for lp, G in zip(log_probs, returns)]).sum()
    optimizer.zero_grad(); loss.backward(); optimizer.step()
```

Actor-Critic

```
import torch, torch.nn as nn, torch.optim as optim
import gym

env = gym.make("Pendulum-v1")
state_dim = env.observation_space.shape[0]
action_dim = env.action_space.shape[0]

class Actor(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(state_dim, 64), nn.ReLU(), nn.Linear(64, action_dim), nn.Tanh())
    def forward(self, x):
        return self.net(x) * 2.0   # scale to Pendulum's action range [-2, 2]

class Critic(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.Linear(state_dim, 64), nn.ReLU(), nn.Linear(64, 1))
    def forward(self, x):
        return self.net(x)

actor, critic = Actor(), Critic()
actor_opt = optim.Adam(actor.parameters(), lr=1e-4)
critic_opt = optim.Adam(critic.parameters(), lr=1e-3)
gamma = 0.99

for episode in range(300):
    state, _ = env.reset()
    done = False
    while not done:
        state_t = torch.FloatTensor(state)
        action = actor(state_t) + torch.randn(action_dim) * 0.1   # exploration noise
        next_state, reward, done, truncated, _ = env.step(action.detach().numpy())

        value = critic(state_t)
        next_value = critic(torch.FloatTensor(next_state))
        advantage = reward + gamma * next_value.item() * (1 - done) - value.item()

        critic_loss = advantage ** 2
        actor_loss = -advantage * action.sum()   # simplified continuous-action objective
```

```
critic_opt.zero_grad(); torch.tensor(critic_loss, requires_grad=True).backward(); critic_opt.step()
actor_opt.zero_grad(); actor_loss.backward(); actor_opt.step()

state = next_state
```

5. Setup Notes & Required Libraries

Library	Install Command	Used For
scikit-learn	`pip install scikit-learn`	Regression, classification, clustering, PCA
pandas / numpy	`pip install pandas numpy`	Data loading and manipulation
matplotlib	`pip install matplotlib`	Plotting charts, trees, dendrograms
xgboost	`pip install xgboost`	Gradient Boosting model
mlxtend	`pip install mlxtend`	Apriori / Association Rule Learning
scipy	`pip install scipy`	Hierarchical clustering dendrograms
tensorflow	`pip install tensorflow`	Neural Networks, Autoencoders (Keras)
torch	`pip install torch`	DQN, Policy Gradient, Actor-Critic
gym / gymnasium	`pip install gymnasium`	Reinforcement learning environments

How to Use This Guide

These snippets are intentionally minimal and assume you plug in your own dataset (X, y, or df) where noted in comments. Run each one end-to-end on a small public dataset first (as suggested in the companion notes PDF) before adapting it to your own project — that's the fastest way to actually internalize how each model behaves.