# 💰 Esercizio: Mini-Boilerplate per Predittore Spese Personali

## 📋 Caso d'Uso Reale

**Scenario**: Vuoi creare un'app che ti aiuti a gestire il budget mensile predicendo le tue spese future basandosi sullo storico delle transazioni.

**Problema da risolvere**:

1. **Predire la spesa totale** del mese successivo (regressione)
2. **Classificare le transazioni** per categoria (classificazione)
3. **Identificare anomalie** nelle spese (es. spese insolite da investigare)

**Perché è utile**:

- Ti aiuta a pianificare il budget
- Prevede quando potresti finire i soldi
- Identifica spese sospette o inusuali
- Ti avvisa se stai spendendo troppo in una categoria

---

## 🎯 Obiettivo dell'Esercizio

Creare un **boilerplate completo e riutilizzabile** per analisi predittiva di spese personali che possa essere facilmente adattato per altri problemi simili (predizione vendite, analisi budget aziendali, ecc.).

**Tempo stimato**: 3-4 ore
**Difficoltà**: ⭐⭐⭐

---

## 📂 Struttura del Boilerplate Richiesta

```
expense-predictor/
│
├── data/
│   ├── raw/                  # Dati originali (transazioni.csv)
│   ├── processed/            # Dati preprocessati
│   └── sample_data.csv       # Dati di esempio generati
│
├── configs/
│   ├── default.yaml          # Configurazione base
│   └── config_schema.py      # Validazione config
│
├── src/
│   ├── __init__.py
│   │
│   ├── data/
│   │   ├── __init__.py
│   │   ├── generator.py      # Genera dati sintetici
│   │   ├── loader.py         # Carica e processa CSV
│   │   └── preprocessor.py   # Feature engineering
│   │
│   ├── models/
│   │   ├── __init__.py
│   │   ├── base_model.py     # Classe base
│   │   ├── expense_regressor.py
│   │   └── category_classifier.py
│   │
│   ├── training/
│   │   ├── __init__.py
│   │   └── trainer.py        # Training pipeline
│   │
│   └── utils/
│       ├── __init__.py
│       ├── logger.py         # Logging utilities
│       └── metrics.py        # Custom metrics
│
├── models/                   # Modelli salvati
│   └── .gitkeep
│
├── logs/                     # Log di training
│   └── .gitkeep
│
```

```
├── notebooks/
│   └── exploratory_analysis.ipynb
│
├── train.py                # Script training
├── predict.py              # Script predizione
├── requirements.txt
└── README.md
```

---

# 📊 Formato Dati

## Input: CSV con transazioni bancarie

csv

```
date,amount,category,merchant,day_of_week,is_weekend,month
2024-01-15,45.50,Restaurant,Pizza Place,Monday,0,1
2024-01-16,120.00,Groceries,Supermarket,Tuesday,0,1
2024-01-20,80.00,Transport,Gas Station,Saturday,1,1
2024-01-22,1200.00,Rent,Landlord,Monday,0,1
2024-01-25,25.00,Entertainment,Cinema,Thursday,0,1
...
```

**Features disponibili**:

- `date`: Data transazione
- `amount`: Importo speso (€)
- `category`: Categoria spesa (Groceries, Rent, Transport, Restaurant, Entertainment, Shopping, Healthcare, Utilities, Other)
- `merchant`: Nome esercente
- `day_of_week`: Giorno settimana
- `is_weekend`: 1 se weekend, 0 altrimenti
- `month`: Mese (1-12)

---

# 🎯 Task da Implementare

## Task 1: Data Generation Script

**File**: `src/data/generator.py`

Crea uno script che generi dati sintetici realistici:

python

```python
import pandas as pd
import numpy as np
from datetime import datetime, timedelta

def generate_synthetic_expenses(
    n_months=12,
    avg_transactions_per_month=50,
    seed=42
):
    """
    Genera transazioni sintetiche realistiche

    Pattern da includere:
    - Spese ricorrenti (es. affitto sempre il 1° del mese)
    - Variabilità stagionale (più shopping a Dicembre)
    - Più spese nei weekend per ristoranti
    - Anomalie occasionali (spese inusuali)
    """
    # TODO: Implementa la logica
    pass
```

**Requisiti**:

- Genera almeno 12 mesi di dati
- Include pattern realistici (affitto fisso, spese variabili)
- Aggiungi noise e anomalie occasionali
- Salva in `data/sample_data.csv`

---

## Task 2: Configuration Management

**File**: `configs/default.yaml`

yaml

```yaml
# Data paths
data:
  raw_path: "data/sample_data.csv"
  processed_path: "data/processed/"
  train_split: 0.7
  val_split: 0.15
  test_split: 0.15

# Feature engineering
features:
  include_day_features: true        # Day of week, is_weekend
  include_lag_features: true        # Spesa degli ultimi N giorni
  lag_days: [7, 14, 30]             # Lag periods
  include_rolling_stats: true       # Rolling mean, std
  rolling_windows: [7, 30]          # Rolling window sizes
  include_categorical: true         # One-hot encoding categorie

# Model configurations
regressor:
  model_type: "random_forest"      # o "gradient_boosting", "neural_network"
  hyperparameters:
    n_estimators: 100
    max_depth: 10
    random_state: 42

classifier:
  model_type: "random_forest"
  hyperparameters:
    n_estimators: 100
    max_depth: 8

# Training
training:
  save_best_model: true
  model_save_path: "models/"
  log_path: "logs/"
  verbose: true

# Prediction
prediction:
  confidence_threshold: 0.7        # Soglia per alert anomalie
```

```
  alert_if_exceeds_budget: true
  monthly_budget: 2000          # Budget mensile in €
```

**File**: `configs/config_schema.py`

python

```python
from dataclasses import dataclass
from typing import List
import yaml

@dataclass
class DataConfig:
    raw_path: str
    processed_path: str
    train_split: float
    val_split: float
    test_split: float

    def validate(self):
        assert self.train_split + self.val_split + self.test_split == 1.0
        # TODO: aggiungi altre validazioni

@dataclass
class Config:
    data: DataConfig
    # TODO: aggiungi altre sezioni

    @classmethod
    def from_yaml(cls, path: str):
        """Carica config da YAML"""
        # TODO: implementa
        pass
```

---

## Task 3: Data Preprocessing

**File**: `src/data/preprocessor.py`

python

```python
import pandas as pd
import numpy as np
from sklearn.preprocessing import StandardScaler, LabelEncoder

class ExpensePreprocessor:
    """
    Preprocessa dati di spese per ML
    """

    def __init__(self, config):
        self.config = config
        self.scaler = StandardScaler()
        self.label_encoder = LabelEncoder()
        self.feature_columns = None

    def create_features(self, df):
        """
        Feature engineering:
        - Estrai giorno, mese, anno da date
        - Crea lag features (spesa ultimi 7/14/30 giorni)
        - Crea rolling statistics
        - Encoding categorie
        """
        df = df.copy()

        # TODO: Implementa feature engineering
        # Esempi:
        # - df['day_of_month'] = df['date'].dt.day
        # - df['lag_7_days'] = df.groupby('category')['amount'].shift(7)
        # - df['rolling_mean_30'] = df['amount'].rolling(30).mean()

        return df

    def fit(self, df):
        """Fit su training data"""
        df_features = self.create_features(df)
        # TODO: Fit scaler e encoders
        return self

    def transform(self, df):
        """Transform new data"""
```

```
        # TODO: Applica transformations
        pass

    def save(self, path):
        """Salva preprocessor"""
        import joblib
        joblib.dump(self, path)

    @staticmethod
    def load(path):
        """Carica preprocessor salvato"""
        import joblib
        return joblib.load(path)
```

---

## Task 4: Models

**File**: `src/models/base_model.py`

python

```python
from abc import ABC, abstractmethod
import joblib

class BaseModel(ABC):
    """Classe base per tutti i modelli"""

    def __init__(self, config):
        self.config = config
        self.model = None

    @abstractmethod
    def build(self):
        """Costruisci il modello"""
        pass

    @abstractmethod
    def fit(self, X_train, y_train, X_val=None, y_val=None):
        """Training"""
        pass

    @abstractmethod
    def predict(self, X):
        """Predizione"""
        pass

    def save(self, path):
        """Salva modello"""
        joblib.dump(self.model, path)
        print(f"✓ Model saved to {path}")

    def load(self, path):
        """Carica modello"""
        self.model = joblib.load(path)
        print(f"✓ Model loaded from {path}")
```

**File**: `src/models/expense_regressor.py`

python

```python
from .base_model import BaseModel
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.neural_network import MLPRegressor

class ExpenseRegressor(BaseModel):
    """
    Modello per predire spesa totale mensile
    """

    def build(self):
        model_type = self.config.regressor.model_type
        params = self.config.regressor.hyperparameters

        models = {
            'random_forest': RandomForestRegressor(**params),
            'gradient_boosting': GradientBoostingRegressor(**params),
            'neural_network': MLPRegressor(**params)
        }

        self.model = models.get(model_type)
        if self.model is None:
            raise ValueError(f"Unknown model type: {model_type}")

        return self

    def fit(self, X_train, y_train, X_val=None, y_val=None):
        """Train the regressor"""
        self.model.fit(X_train, y_train)

        # Validation score
        if X_val is not None and y_val is not None:
            score = self.model.score(X_val, y_val)
            print(f"Validation R² Score: {score:.4f}")

        return self

    def predict(self, X):
        return self.model.predict(X)
```

**File**: `src/models/category_classifier.py`

python

```python
from .base_model import BaseModel
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report

class CategoryClassifier(BaseModel):
    """
    Classifica transazioni per categoria
    """

    def build(self):
        # TODO: Implementa come ExpenseRegressor ma per classificazione
        pass

    def fit(self, X_train, y_train, X_val=None, y_val=None):
        # TODO: Implementa
        # Stampa classification report su validation set
        pass

    def predict(self, X):
        # TODO: Implementa
        pass

    def predict_proba(self, X):
        """Restituisce probabilità per ogni classe"""
        return self.model.predict_proba(X)
```

---

### Task 5: Training Pipeline

**File**: `src/training/trainer.py`

python

```python
import pandas as pd
import numpy as np
from pathlib import Path
import logging

class ExpenseTrainer:
    """
    Gestisce il training dei modelli
    """

    def __init__(self, config):
        self.config = config
        self.logger = self._setup_logger()
        self.history = {
            'train_metrics': [],
            'val_metrics': []
        }

    def _setup_logger(self):
        """Setup logging"""
        logger = logging.getLogger('ExpenseTrainer')
        logger.setLevel(logging.INFO)

        # File handler
        log_file = Path(self.config.training.log_path) / 'training.log'
        log_file.parent.mkdir(exist_ok=True)

        handler = logging.FileHandler(log_file)
        formatter = logging.Formatter(
            '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
        )
        handler.setFormatter(formatter)
        logger.addHandler(handler)

        # Console handler
        console = logging.StreamHandler()
        console.setFormatter(formatter)
        logger.addHandler(console)

        return logger
```

```python
def train_regressor(self, X_train, y_train, X_val, y_val):
    """
    Train expense amount predictor
    """
    self.logger.info("Training expense regressor...")

    from src.models.expense_regressor import ExpenseRegressor

    model = ExpenseRegressor(self.config)
    model.build()
    model.fit(X_train, y_train, X_val, y_val)

    # Evaluate
    from sklearn.metrics import mean_absolute_error, mean_squared_error, r2_score

    y_pred = model.predict(X_val)
    mae = mean_absolute_error(y_val, y_pred)
    rmse = np.sqrt(mean_squared_error(y_val, y_pred))
    r2 = r2_score(y_val, y_pred)

    self.logger.info(f"Validation Metrics:")
    self.logger.info(f"  MAE:  €{mae:.2f}")
    self.logger.info(f"  RMSE: €{rmse:.2f}")
    self.logger.info(f"  R²:   {r2:.4f}")

    # Save
    if self.config.training.save_best_model:
        model_path = Path(self.config.training.model_save_path) / 'regressor.pkl'
        model_path.parent.mkdir(exist_ok=True)
        model.save(model_path)

    return model

def train_classifier(self, X_train, y_train, X_val, y_val):
    """
    Train category classifier
    """
    self.logger.info("Training category classifier...")

    # TODO: Implementa simile a train_regressor
    # ma per classificazione
```

```
        pass

    def train_all(self, data):
        """
        Train both models
        """
        # TODO: Split data, preprocess, train both models
        pass
```

---

## Task 6: Main Scripts

**File**: `train.py`

python

```python
import argparse
from pathlib import Path
from src.data.generator import generate_synthetic_expenses
from src.data.loader import load_expense_data
from src.data.preprocessor import ExpensePreprocessor
from src.training.trainer import ExpenseTrainer
from configs.config_schema import Config

def main(args):
    print("=" * 60)
    print(" 💰 EXPENSE PREDICTOR - Training Pipeline")
    print("=" * 60)

    # Load config
    config = Config.from_yaml(args.config)
    print(f"\n✓ Loaded configuration from {args.config}")

    # Generate or load data
    data_path = Path(config.data.raw_path)
    if not data_path.exists():
        print(f"\n⚠ Data not found. Generating synthetic data...")
        df = generate_synthetic_expenses(n_months=12)
        df.to_csv(data_path, index=False)
        print(f"✓ Generated {len(df)} transactions")
    else:
        print(f"\n✓ Loading data from {data_path}")
        df = pd.read_csv(data_path)

    print(f"  Total transactions: {len(df)}")
    print(f"  Date range: {df['date'].min()} to {df['date'].max()}")
    print(f"  Total amount: €{df['amount'].sum():,.2f}")

    # Preprocess
    print("\n 📊 Preprocessing data...")
    preprocessor = ExpensePreprocessor(config)
    # TODO: Implementa preprocessing

    # Train
    print("\n 🎯 Training models...")
    trainer = ExpenseTrainer(config)
    # TODO: Implementa training
```

```
    print("\n" + "=" * 60)
    print("✅ Training completed!")
    print("=" * 60)

if __name__ == '__main__':
    parser = argparse.ArgumentParser(
        description='Train expense prediction models'
    )
    parser.add_argument(
        '--config',
        type=str,
        default='configs/default.yaml',
        help='Path to config file'
    )
    args = parser.parse_args()

    main(args)
```

**File**: `predict.py`

python

```python
import argparse
import pandas as pd
from pathlib import Path
from src.data.preprocessor import ExpensePreprocessor
from src.models.expense_regressor import ExpenseRegressor

def main(args):
    print("=" * 60)
    print("🔮 EXPENSE PREDICTOR - Prediction")
    print("=" * 60)

    # Load preprocessor
    print("\n📊 Loading preprocessor...")
    preprocessor = ExpensePreprocessor.load(args.preprocessor)

    # Load model
    print("🤖 Loading model...")
    model = ExpenseRegressor(None)  # Config not needed for inference
    model.load(args.model)

    # Load new data
    print(f"📂 Loading data from {args.input}...")
    df = pd.read_csv(args.input)

    # Preprocess
    X = preprocessor.transform(df)

    # Predict
    print("\n🔮 Making predictions...")
    predictions = model.predict(X)

    # Add predictions to dataframe
    df['predicted_amount'] = predictions

    # Summary
    print("\n📈 Prediction Summary:")
    print(f"  Total predicted expenses: €{predictions.sum():,.2f}")
    print(f"  Average per transaction:  €{predictions.mean():.2f}")
    print(f"  Max predicted amount:     €{predictions.max():.2f}")

    # Save
```

```
    output_path = Path(args.output)
    df.to_csv(output_path, index=False)
    print(f"\n✓ Predictions saved to {output_path}")

    # Check budget
    monthly_budget = 2000  # TODO: prendi da config
    if predictions.sum() > monthly_budget:
        print(f"\n⚠️  WARNING: Predicted expenses (€{predictions.sum():.2f}) "
              f"exceed budget (€{monthly_budget:.2f})")

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('--input', required=True, help='Input CSV file')
    parser.add_argument('--output', default='predictions.csv', help='Output file')
    parser.add_argument('--model', default='models/regressor.pkl', help='Model path')
    parser.add_argument('--preprocessor', default='models/preprocessor.pkl', help='Preprocessor path')
    args = parser.parse_args()

    main(args)
```

---

# 🎯 Funzionalità Bonus (Opzionali)

## 1. Anomaly Detection

Aggiungi un modulo che identifica transazioni anomale:

python

```python
# src/models/anomaly_detector.py
from sklearn.ensemble import IsolationForest

class AnomalyDetector:
    """Detect unusual transactions"""

    def fit(self, X):
        self.model = IsolationForest(contamination=0.1)
        self.model.fit(X)

    def predict(self, X):
        """Returns -1 for anomalies, 1 for normal"""
        return self.model.predict(X)
```

## 2. Budget Alert System

python

```
def check_budget_alerts(predictions, budget, config):
    """
    Controlla se le predizioni superano il budget
    Invia alert per categoria
    """
    pass
```

## 3. Visualization Dashboard

Crea un notebook Jupyter con:

- Grafico spese per categoria
- Trend temporale
- Predicted vs Actual comparison
- Budget tracking

## 4. Web API con FastAPI

python

```python
# api.py
from fastapi import FastAPI, UploadFile

app = FastAPI()

@app.post("/predict")
async def predict_expenses(file: UploadFile):
    # Carica CSV, preprocessa, predici
    pass
```

---

# ✅ Criteri di Valutazione

### Funzionalità (40%)

- ✅ Genera dati sintetici realistici
- ✅ Preprocessing completo con feature engineering
- ✅ Training pipeline funzionante
- ✅ Script train.py e predict.py funzionanti
- ✅ Salvataggio e caricamento modelli

### Architettura Boilerplate (30%)

- ✅ Struttura directory pulita e organizzata
- ✅ Modularità del codice
- ✅ Configuration management con YAML
- ✅ Separazione responsabilità (data/models/training)
- ✅ Pattern riutilizzabili

### Qualità Codice (20%)

- ✅ Type hints
- ✅ Docstring complete
- ✅ Gestione errori
- ✅ Logging appropriato
- ✅ PEP 8 compliance

### Documentation (10%)

- ✅ README completo con:
  - Descrizione progetto
  - Setup instructions
  - Usage examples
  - Struttura progetto
- ✅ Commenti nel codice
- ✅ Esempi d'uso

---

## 📝 Deliverable Finali

1. **Repository GitHub** con tutta la struttura
2. **README.md** completo
3. **requirements.txt** con dipendenze
4. **Demo**: Screenshot o video che mostra:
    - Training completo
    - Predizione su nuovi dati
    - Log e metriche
5. **Report breve** (1-2 pagine) che spiega:
    - Scelte architetturali
    - Feature engineering
    - Performance modelli
    - Come estendere il boilerplate

---

## 🚀 Come Iniziare

1. **Setup iniziale**:

bash

```
mkdir expense-predictor
cd expense-predictor
git init
# Crea struttura directory
```

2. **Dependencies**:

bash

```
pip install pandas numpy scikit-learn pyyaml joblib matplotlib
```

3. **Workflow suggerito**:
    - Giorno 1: Setup struttura + data generation
    - Giorno 2: Preprocessing + feature engineering
    - Giorno 3: Models + training pipeline
    - Giorno 4: Scripts + testing + documentation
4. **Testing mentre sviluppi**:
    - Testa ogni componente separatamente
    - Usa dati sintetici piccoli prima di scalare
    - Verifica che train.py e predict.py funzionino end-to-end

---

# 💡 Estensioni Future (Post-Esercizio)

Una volta completato il boilerplate base, puoi:

1. Aggiungere support per multiple utenti
2. Implementare time-series forecasting (ARIMA, Prophet)
3. Creare mobile app con predizioni real-time
4. Integrare con API bancarie reali
5. Deploy su cloud con automatic retraining

**Il bello del boilerplate è che puoi riutilizzarlo per problemi simili!** 🎉

---

# ❓ FAQ

**Q: Posso usare librerie oltre a quelle base?**
A: Sì, ma documenta perché le hai scelte.

**Q: Devo implementare tutti i bonus?**
A: No, i bonus sono opzionali per chi vuole sfide extra.

**Q: Quanto deve essere accurato il modello?**
A: Non è critico - focus su architettura boilerplate, non su performance ML.

**Q: Posso usare dati reali invece di sintetici?**
A: Sì! Usa pure le tue transazioni bancarie (anonimizzale se condividi).

---

**Buon lavoro!** 💪