Proietti

Stefano Rosini

December 2023

Contents

1 Introduction 1

2 Some Useful Functions 2

3 First Predictor 4
- 3.1 Numeric Part 4
- 3.2 Symbolic Part 5

4 Second Predictor 6
- 4.1 Numeric Part 6
- 4.2 Symbolic Part 6

5 Section 3 and 4 6

6 Last Point 7

1 Introduction

In this homework we're going to implement the Sequence predictor for a Generic Symbolic and Numeric sequence. A sequence predictor is a machine learning model that predicts some future elements of a sequence, starting from a view of past elements. In this homework we are going to implement five predictors:

1. The first one that predicts a sequence of future elements by seeing its past for both symbolic and numeric sequences;

2. The second one predicts k elements that are n steps away in the future by seeing its past (symbolic and numeric);

3. The third one is making an ensemble of predictors for trying to improve performances;

4. The fourth one who's taking $n \cdot m$ elements and predicts just one element;

5. The fifth one is to use the best model for generating a sequence of future elements by using its own predictions as new inputs.

Data for the numeric predictors were stored in two files called `train_series_out.txt` and `test_series_out.txt`. Data for symbolic predictors were generated by a function in Python called `generate_dna_sequence`.

2 Some Useful Functions

Before talking about the predictors, we're going to see a sequence of functions that were used during the project:

```python
import matplotlib.pyplot as plt
import numpy as np
from sklearn.preprocessing import OneHotEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.multioutput import MultiOutputClassifier
from sklearn.metrics import accuracy_score

from sklearn.metrics import mean_squared_error,
 mean_absolute_error, r2_score
def make_windows(series, m, k):
    X, Y = [], []
    for i in range(len(series) - m - k + 1):
        X.append(series[i : i + m])
        Y.append(series[i + m : i + m + k])
    return np.array(X, dtype=np.float32), np.array(Y, dtype=np
     .float32)

def make_windows_with_offset(series, m, k, n=0):
    """Sliding window con salto 'n' tra input e target"""
    X, Y = [], []
    offset = m + n
    for i in range(len(series) - offset - k + 1):
        X.append(series[i : i + m])
        Y.append(series[i + offset : i + offset + k])
    return np.array(X, dtype=np.float32), np.array(Y, dtype=np
     .float32)

def make_windows_with_offset_sym(series, m, k, n=0):
    X, Y = [], []
    offset = m + n
    for i in range(len(series) - offset - k + 1):
        X.append(series[i : i + m])
        Y.append(series[i + offset : i + offset + k])
    return np.array(X), np.array(Y)
```

Figure 1: Series of make_windows functions: these functions were used for building the dataset for the predictors, for numeric values.

```python
def evaluate(model, X, Y):
    preds = model.predict(X)
    rmse = np.sqrt(mean_squared_error(Y, preds))
    mae = mean_absolute_error(Y, preds)
    r2 = r2_score(Y, preds)
    return rmse, mae, r2, preds

def evaluate2(preds, Y):
    rmse = np.sqrt(mean_squared_error(Y, preds))
    mae = mean_absolute_error(Y, preds)
    r2 = r2_score(Y, preds)
    return rmse, mae, r2
```

Figure 2: Evaluation functions

```python
def generate_dna_sequence(length=10000, seed=42, noise_level=0.05):
    import random
    random.seed(seed)
    bases = ['A', 'T', 'G', 'C']
    pattern = (bases * (length // 4 + 1))[:length]

    # Aggiungi rumore casuale (5% di variazione)
    for i in range(len(pattern)):
        if random.random() < noise_level:
            pattern[i] = random.choice(bases)
    return pattern
```

Executed at 2025.08.05 12:41:13 in 13ms

Figure 3: Sequence generated for symbolic predictors

```python
def make_windows_sym(series, m, k):
    X, Y = [], []
    for i in range(len(series) - m - k + 1):
        X.append(series[i : i + m])
        Y.append(series[i + m : i + m + k])
    return np.array(X), np.array(Y)
```

Figure 4: For generating dataset for symbolic predictors

```
model = MLPRegressor(
        hidden_layer_sizes=(10, 10),
        activation='relu',
        solver='adam',
        learning_rate_init=0.01,
        max_iter=1000,
        random_state=1,
        tol=1e-4,
        n_iter_no_change=50
    )

model_sym = MultiOutputClassifier(
        MLPClassifier(
            hidden_layer_sizes = (512, 256, 128, 64),
            activation='relu',
            solver='adam',
            alpha=0.001,                # ← regolarizzazione più
             leggera
            learning_rate='adaptive',
            max_iter=50,
            random_state=1,
            early_stopping=True,
            n_iter_no_change=20,
            verbose=True
        )
    )
```

Figure 5: MLP Regressor and Classifier. For the multi-output classifier, I just needed 50 iterations with respect to the regressor.

3 First Predictor

For the symbolic and the numeric predictor we used two different ML models: MLP Regressor for the numeric ones and the Multi-Output MLP Classifier for the symbolic ones. MLP Regressor is a feedforward network that predicts continuous values and uses the Mean Squared Error as loss function. The Multi-Output MLP Classifier is used for classifying symbolic sequences and to predict discrete labels. It uses the Binary Cross Entropy loss, since data were one-hot encoded. To implement this predictor I used the sklearn library.

3.1 Numeric Part

For the numeric section, I first used the function `make_windows` for preparing data for the predictor, then I trained the MLP Regressor and, for evaluation, I used the corresponding metrics:

- RMSE: 0.10;
- MAE: 0.086;
- $R\hat{2}$: 0.97;

With these parameters I found out how well the model predicted the future data. Here I show the plot of the prediction:

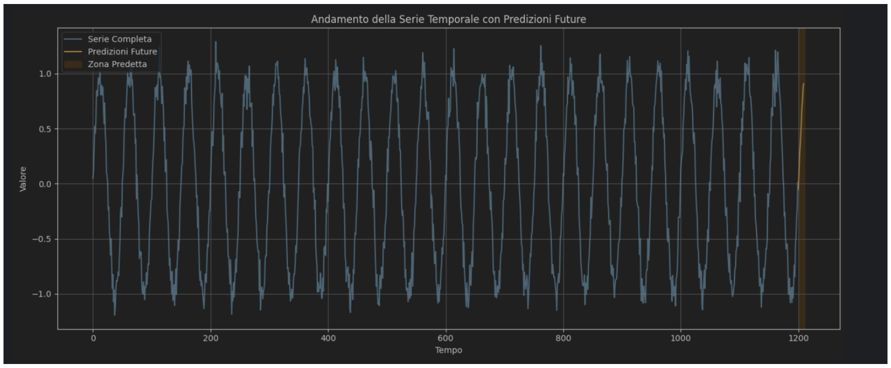

Figure 6: Prediction of the future sequence by imposing $m = 100$ and $k = 35$

3.2 Symbolic Part

For the symbolic part I first built my symbolic sequence, then split the sequence into train and testing sets, then encoded the train and test data, and finally trained the model. **Test accuracy** was about 95%, enough for correctly predicting a sequence of $k = 4$ symbolic elements. Here we show the output of the code:

```
 Test Accuracy: 0.9555

Sample input: ['C' 'A' 'T' 'G' 'C' 'A' 'T' 'G' 'C' 'A' 'T' 'G' 'C' 'T' 'T' 'G' 'C' 'A'
 'T' 'G' 'C' 'A' 'T' 'G' 'C' 'C' 'T' 'G' 'C' 'A']
True next symbols: ['T' 'G' 'C' 'A']
Predicted symbols: ['T', 'G', 'C', 'A']
```

Figure 7: Output of the testing of the MLP multi-output classifier

```
cut = int(len(sequence) * 0.8)
train_series = sequence[:cut]
test_series = sequence[cut - m - k + 1:]

X_train_raw, Y_train_raw = make_windows_sym(train_series, m, k)
X_test_raw, Y_test_raw = make_windows_sym(test_series, m, k)

# One-hot encoding input
encoder = OneHotEncoder(sparse_output=False, handle_unknown='ignore', dtype=int)
encoder.fit(np.array(sequence).reshape(-1, 1))  # su tutta la sequenza

def encode(windows):
    flat = windows.reshape(-1, 1)
    encoded = encoder.transform(flat)
    return encoded.reshape(windows.shape[0], -1)

X_train = encode(X_train_raw)
X_test = encode(X_test_raw)

# Mappying symbols into int
label_set = sorted(set(sequence))
label_to_int = {lab: i for i, lab in enumerate(label_set)}
int_to_label = {i: lab for lab, i in label_to_int.items()}

# Encoding target
Y_train = np.array([[label_to_int[s] for s in row] for row in Y_train_raw])
Y_test  = np.array([[label_to_int[s] for s in row] for row in Y_test_raw])
```

Figure 8: Encoding section — this part was done for one-hot encoding the complete sequence of symbols

4 Second Predictor

The second predictor had to predict k elements that were n steps ahead in the future. This predictor was made for both the regressor and classifier.

4.1 Numeric Part

In the numeric part I just used a different function for constructing the dataset for the predictor. Its name was `make_windows_with_offsets`: Given a series in input, an m and a k with n steps in input as well, it creates a dataset for training and testing for this specific goal.

For example, if we have this sequence of numbers:

$$[1, 2, 3, 4, 5, 6, 7, 8] \tag{1}$$

The training section will be $[1, 2, 3]$ for example, and the target will be $[5, 6]$ if $n = 1$, $k = 3$, and $m = 3$.

The whole code then is the same. Here I show the plot of the graph:

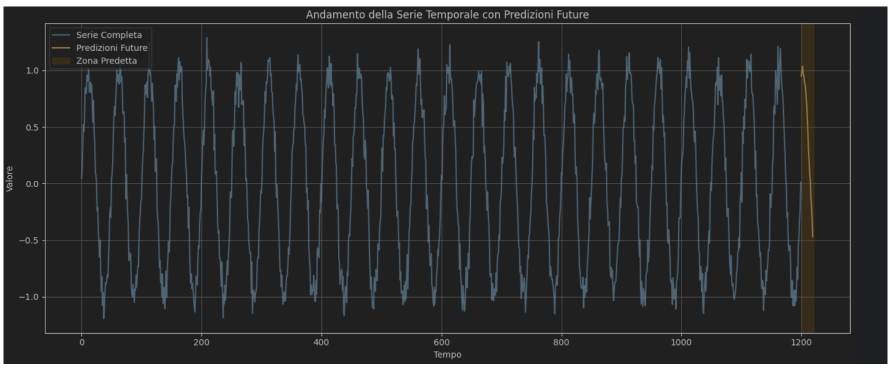

Figure 9: Output of the test model, with $m = 100$, $k = 50$, $n = 10$

4.2 Symbolic Part

For the symbolic part I applied the same concept as for the numeric part, with the exception that I used the `make_windows_with_offsets` function for the symbolic version. The code was the same as in the first point as well. Here I show the output of the testing section:

```
 Test Accuracy: 0.9659

Sample input: ['A' 'G' 'G' 'C' 'A' 'T' 'G' 'C' 'A' 'T' 'G' 'T' 'A' 'T' 'G' 'C' 'A' 'T'
 'G' 'C']
True next symbols after offset: ['T' 'G' 'C' 'A' 'C' 'G' 'C' 'A' 'T' 'G']
Predicted symbols: ['T', 'G', 'C', 'A', 'T', 'G', 'C', 'A', 'T', 'G']
```

Figure 10: Output of the testing section. $m = 20$, $k = 10$, $n = 5$

5 Section 3 and 4

For the third predictor, I implemented an ensemble of numeric predictors using the `make_windows_with_offset` function from Point 2. Each predictor learns to forecast a single step at a different future offset $(n = 1, ..., k)$, and the ensemble prediction is obtained by averaging the outputs from all these predictors.

For the fourth section, I trained a predictor that takes in a window of m input values and directly predicts k future steps in one shot. This model uses the `make_windows` function to generate training data.

In the end, I compared the ensemble and the direct multi-step predictor by evaluating their accuracy in predicting the k-th future value. The plots and metrics below illustrate the difference in performance:

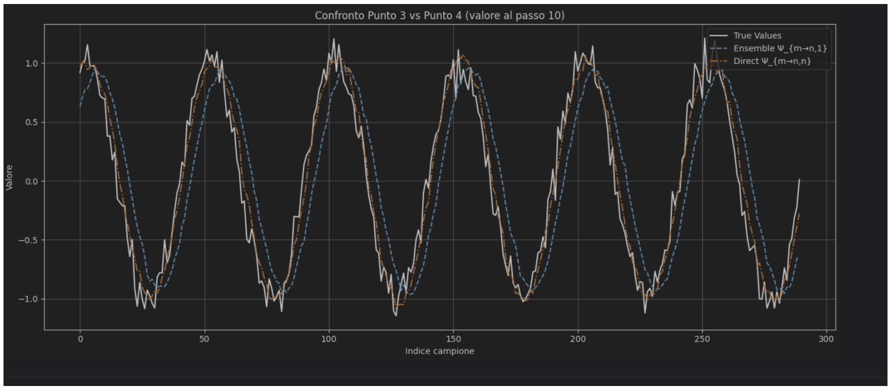

Figure 11: Comparing the curves — the direct one was closer than the ensemble's one

```
[Confronto Metriche (valore al passo 10)]
Ensemble Ψ_{m→n,1}:     RMSE=0.3985, MAE=0.3509, R2=0.7014
Direct Ψ_{m→n,n}:       RMSE=0.1086, MAE=0.0856, R2=0.9780
```

Figure 12: Indeed, we can see performance metric comparisons

6 Last Point

For the last point, I used the same multi-output MLP classifier (trained on symbolic data) as in the first section. The model was trained using the function `make_windows_sym`, with parameters $m = 30$ and $k = 4$, to predict k future symbols given a window of m past ones.

After training, I used the model as a generator by feeding it an initial sequence of m real symbols, then recursively using its own predictions to generate the next symbols in the sequence.

The code used for this symbolic sequence generation is shown in the following figure:

```
# Punto 5: Generazione

steps = 100
generated = list(test_series[:m])  # target serie to predict

print("\nGenerazione a lungo termine:")

for _ in range(steps // k):
    window = np.array(generated[-m:]).reshape(1, -1)
    encoded_window = encoder.transform(window.T).reshape(1, -1)

    pred_indices = model_sym.predict(encoded_window)[0]
    pred_symbols = [int_to_label[i] for i in pred_indices]

    generated.extend(pred_symbols)

print("\nSequenza generata (primi 100 simboli):")
print("".join(generated[m:m+100]))

true_sequence = test_series[m:m+steps]

pred_sequence = generated[m:m+steps]

correct = sum(1 for p, t in zip(pred_sequence, true_sequence) if p == t)
accuracy = correct / len(true_sequence)

print(f"Accuracy simbolica (su {steps} passi): {accuracy:.4f}")
```

Figure 13: Code used to generate the symbolic sequence.

```
Sequenza generata (primi 100 simboli):
TGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCATGCA
```

Figure 14: Sequence generated by the algorithm