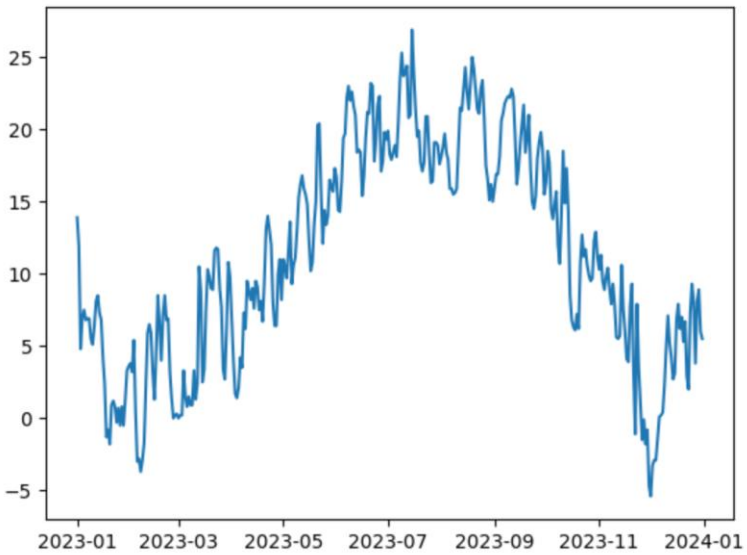
Определение задачи

Цель проекта: Построение модели для прогнозирования суточной температуры на основе исторических данных о погодных условиях.
Тип задачи: Регрессия в рамках анализа временных рядов.
Особенности данных:

- Многомерный временной ряд с 15 признаками (температура, влажность, направление ветра и др.).
- Данные охватывают период с 2000 по 2024 год с ежедневной частотой.
- Основной целевой переменной является температура (`temperature`).

```
plt.plot(data[data.time.dt.year == 2023]["time"], data[data.time.dt.year == 2023]["temperature"])
```

```
[<matplotlib.lines.Line2D at 0x7c166d8a05b0>]
```



Подготовка данных

1. **Загрузка и предобработка:**
    - Преобразование временных меток в формат `datetime`.
    - Агрегация данных по дням с усреднением значений.
    - Удаление шума и округление значений до одного знака после запятой.
2. **Инженерия признаков:**
    - Создание оконных признаков для будущих моделей: каждая строка содержит данные за текущий день и предыдущие 7 дней.
    - Использование 5 ключевых признаков: `temperature`, `relative_humidity`, `wind_direction`, `wind_speed_10m (km/h)`, `dew_point`.

3. **Нормализация:**
    - Применение `MinMaxScaler` для масштабирования данных в диапазон `[-1, 1]`.
4. **Разделение данных:**
    - Разделение на тренировочные (95%) и тестовые (5%) наборы.
    - Преобразование данных в тензоры PyTorch.

```python
def prepare_dataframe_for_lstm(df, n_steps):
    df = dc(df)
    df.set_index("time", inplace = True)

    for i in range(1, n_steps + 1):
        df[f"temp(t-{i})"] = df["temperature"].shift(i)
        df[f"rel_humid(t-{i})"] = df["relative_humidity"].shift(i)
        df[f"wind_d(t-{i})"] = df["wind_direction"].shift(i)
        df[f"wind_spd(t-{i})"] = df["wind_speed_10m (km/h)"].shift(i)
        df[f"dew_p(t-{i})"] = df["dew_point"].shift(i)

    df.dropna(inplace = True)

    return df

lookback = 7
shifted_df = prepare_dataframe_for_lstm(data, lookback)
shifted_df
```

time	temperature	relative_humidity	wind_direction	wind_speed_10m (km/h)	dew_point	temp(t-1)	rel_humid(t-1)	wind_d(t-1)	wind_spd(t-1)	dew_p(t-1)	...	temp(t-6)	rel_humid(t-6)	wind_d(t-6)	wind_spd(t-6)	dew_p(t-6)	temp(
2000-01-08	4.3	86.6	216.0	14.3	2.2	5.0	93.7	224.1	11.9	4.0	...	3.3	94.0	260.9	14.0	2.4	0.
2000-01-09	4.4	92.8	275.2	7.7	3.3	4.3	86.6	216.0	14.3	2.2	...	5.4	86.8	236.6	17.6	3.3	3.
2000-	[redacted]	91.7	277.2	7.7	0.8	4.4	92.8	275.2	7.7	3.3	...	5.4	88.5	224.7	15.8	3.6	5.

🧠 Модели для обучения

Описание модели LSTM

Модель **LSTM (Long Short-Term Memory)** — это разновидность рекуррентных нейронных сетей (RNN), предназначенная для работы с временными рядами и последовательными данными. Она решает проблему исчезающего градиента, свойственную классическим RNN, за счет механизмов забывания и запоминания информации в долгосрочной перспективе.

Архитектура модели

Представленная модель **LSTM** включает в себя:

- **Входной слой** с параметром `input_size`, определяющим количество входных признаков.

- **LSTM-слой**, состоящий из `num_stacked_layers` слоев и `hidden_size` скрытых нейронов. Он принимает входные последовательности и передает их через многослойную LSTM.

- **Полносвязный слой (`fc`)**, который преобразует выходное состояние последнего временного шага в конечный прогноз.

Во время прямого прохода:

1. **Инициализируются нулевые скрытые состояния (`h0, c0`)**, необходимые для обработки последовательности.
2. **Проход через LSTM-слой**, который обрабатывает входную последовательность и возвращает выходные состояния.
3. **Используется только последнее состояние (`out[:, -1, :]`)**, так как задача регрессии предполагает прогноз на основе последнего временного шага.
4. **Выходной слой (`fc`) предсказывает целевое значение** (например, температуру, осадки и др.).

Плюсы и минусы LSTM

✅**Плюсы**:

- Подходит для задач временных рядов, так как эффективно учитывает долгосрочные зависимости.
- Устойчивость к исчезающему градиенту благодаря механизмам `input`, `forget` и `output` gate.
- Хорошо работает при наличии нелинейных зависимостей в данных.

❌**Минусы**:

- Длительное время обучения по сравнению с простыми методами, такими как линейная регрессия.
- Требует большого количества данных для корректного обучения.
- Сложнее интерпретировать по сравнению с традиционными статистическими методами..

```
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_stacked_layers):
        super().__init__()
        self.hidden_size = hidden_size
        self.num_stacked_layers = num_stacked_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_stacked_layers,
                            batch_first = True)
        self.fc = nn.Linear(hidden_size, 1)


    def forward(self, x):
        batch_size = x.size(0)
        h0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device)
        c0 = torch.zeros(self.num_stacked_layers, batch_size, self.hidden_size).to(device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        return out

model = LSTM(X.shape[2], 4, 1)
model.to(device)
```

Описание модели Transformer

Модель **Transformer** — это архитектура, основанная на механизме **self-attention**, которая изначально была разработана для задач обработки естественного языка, но нашла широкое применение в прогнозировании временных рядов, обработке изображений и других областях.

Архитектура модели

Представленная **Transformer-модель** включает в себя следующие компоненты:

1. **Линейное преобразование входных данных (`encoder`)**
    - Исходные признаки (размерность `input_dim=5`) проецируются в пространство размерностью `d_model=64`.
    - Это необходимо, так как Transformer работает с фиксированной размерностью эмбеддингов.
2. **Позиционное кодирование (`PositionalEncoding`)**
    - В отличие от LSTM, Transformer не имеет встроенного механизма обработки последовательности, поэтому используется **синусоидальное позиционное кодирование**, которое позволяет учитывать порядок элементов во входной последовательности.
3. **Многоголовое внимание (`Multi-Head Attention`)**
    - Внутри `nn.TransformerEncoderLayer` используется механизм **self-attention**, который позволяет модели учитывать зависимости между

различными временными шагами.

- Используется `nhead=4` (количество голов в механизме внимания).

4. **Каскад энкодеров (`nn.TransformerEncoder`)**
    - Включает `num_layers=2` слоев, каждый из которых состоит из механизма **self-attention** и полносвязных слоев с нормализацией.
    - Позволяет модели захватывать сложные паттерны в данных.
5. **Выходной линейный слой (`decoder`)**
    - После прохода через трансформер используется только **последний временной шаг** для предсказания целевого значения.

Плюсы и минусы Transformer

✅**Плюсы**:

- Эффективно моделирует **долгосрочные зависимости** благодаря self-attention.
- Лучше параллелизуется по сравнению с LSTM (не требует последовательной обработки).
- Хорошо подходит для **многомерных временных рядов**, так как может одновременно учитывать зависимости между различными признаками.

❌**Минусы**:

- **Высокая вычислительная сложность**, особенно для длинных последовательностей (квадратичная сложность по отношению к длине последовательности).
- Требует большого количества данных для хорошей генерализации.
- Временные паттерны могут быть хуже уловлены без дополнительной обработки (например, в чистом виде Transformer не использует рекуррентные механизмы).

```python
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, dropout=0.1, max_len=5000):
        super(PositionalEncoding, self).__init__()
        self.dropout = nn.Dropout(p=dropout)

        # Initialize positional encoding matrix
        pe = torch.zeros(max_len, d_model)  # Shape: [max_len, d_model]
        position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)  # [max_len, 1]

        # Compute the divisor term for sinusoidal functions
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() *
            (-math.log(10000.0) / d_model)  # Use math.log instead of np.log
        )  # Shape: [d_model // 2]

        # Fill even indices with sine, odd indices with cosine
        pe[:, 0::2] = torch.sin(position * div_term)  # Even positions
        pe[:, 1::2] = torch.cos(position * div_term)  # Odd positions

        # Reshape for batch-first input: [1, max_len, d_model]
        pe = pe.unsqueeze(0)  # Add batch dimension (no transpose needed)
        self.register_buffer('pe', pe)

    def forward(self, x):
        """
        Args:
            x: Input tensor of shape [batch_size, seq_len, d_model]
        Returns:
            x + positional encoding (same shape as input)
        """
        # Add positional encoding to the input
        # Slice positional encoding to match the sequence length of `x`
        x = x + self.pe[:, :x.size(1), :]  # pe[:, :seq_len, :]
        return self.dropout(x)
```

```python
class TransformerModel(nn.Module):
    def __init__(self, input_dim=5, d_model=64, nhead=4, num_layers=2, dropout=0.2):
        super(TransformerModel, self).__init__()

        self.encoder = nn.Linear(input_dim, d_model)

        self.pos_encoder = PositionalEncoding(d_model, dropout)

        encoder_layers = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dropout=dropout,
            batch_first=False  # Set to False (default) since we permute dimensions
        )
        self.transformer_encoder = nn.TransformerEncoder(encoder_layers, num_layers)

        self.decoder = nn.Linear(d_model, 1)

    def forward(self, x):

        x = self.encoder(x)  # Output shape: [batch_size, 7, d_model]

        x = self.pos_encoder(x)  # Shape remains [batch_size, 7, d_model]

        x = x.permute(1, 0, 2)  # New shape: [7, batch_size, d_model]

        x = self.transformer_encoder(x)  # Output shape: [7, batch_size, d_model]

        x = x.permute(1, 0, 2)  # New shape: [batch_size, 7, d_model]

        x = self.decoder(x[:, -1, :])  # Output shape: [batch_size, 1]

        return x

# Initialize model with input_dim=5 (matches your X.shape[2])
transformer_model = TransformerModel(input_dim=5).to(device)
```

Оценка качества моделей

Для оценки точности предсказаний используемых моделей применяется набор метрик регрессии. Код ниже позволяет вычислить основные показатели, оценивающие, насколько хорошо модель предсказывает значения временных рядов:

```python
from sklearn.metrics import (
    mean_absolute_error,
    mean_squared_error,
    r2_score,
    explained_variance_score,
    mean_absolute_percentage_error
)

# Assuming you have:
# test_predictions = np.array([...])  # Your model's predictions
# new_y_test = np.array([...])        # Ground truth values

def print_regression_metrics(y_true, y_pred):
    """Prints comprehensive regression metrics."""
    print("Regression Metrics:")
    print("-------------------")
    print(f"1. MAE (Mean Absolute Error): {mean_absolute_error(y_true, y_pred):.4f}")
    print(f"2. MSE (Mean Squared Error): {mean_squared_error(y_true, y_pred):.4f}")
    print(f"3. RMSE (Root Mean Squared Error): {np.sqrt(mean_squared_error(y_true, y_pred)):.4f}")
    print(f"4. R² Score (Coefficient of Determination): {r2_score(y_true, y_pred):.4f}")
    print(f"5. Explained Variance Score: {explained_variance_score(y_true, y_pred):.4f}")
    print(f"6. MAPE (Mean Absolute Percentage Error): {mean_absolute_percentage_error(y_true, y_pred):.4f}")
    print(f"7. Max Error: {np.max(np.abs(y_true - y_pred)):.4f}")
```

Описание метрик

1. **MAE (Mean Absolute Error)** – средняя абсолютная ошибка предсказаний. Показывает среднее отклонение прогнозов от фактических значений.

2. **MSE (Mean Squared Error)** – среднеквадратическая ошибка, более чувствительна к выбросам из-за квадратичного масштаба.

3. **RMSE (Root Mean Squared Error)** – корень из MSE, который сохраняет единицы измерения целевой переменной.

4. **R² Score (Коэффициент детерминации)** – показатель, отражающий долю объясненной дисперсии. Чем ближе к 1, тем лучше модель объясняет данные.

5. **Explained Variance Score** – аналог R², но с меньшей чувствительностью к смещению модели.

6. **MAPE (Mean Absolute Percentage Error)** – средний абсолютный процент ошибки, удобен для понимания относительной точности модели.

7. **Max Error** – максимальная ошибка между предсказанным и фактическим значением, полезна для оценки худшего случая.

Отчет по результатам моделей

1. Введение

В ходе эксперимента были протестированы три модели для задачи прогнозирования:

- **Transformer**
- **LSTM**
- **Линейная регрессия**

Модели оценивались по стандартным метрикам регрессии, включая MAE, MSE, RMSE, R^2, Explained Variance Score, MAPE и Max Error.

Модель	MAE	MSE	RMSE	R^2 Score	Explained Variance	MAPE	Max Error
Transformer	1.8769	5.7605	2.4001	0.8834	0.8863	0.3998	9.4440
LSTM	1.8171	5.3328	2.3093	0.8921	0.8923	0.3693	6.9076
Linear Model	1.5875	4.1120	2.0278	0.9168	0.9179	0.2922	8.6744

3. Анализ

1. **Линейная регрессия показала лучшие результаты** по большинству метрик, включая наименьший MAE (1.5875) и MSE (4.1120), а также наибольший R^2 (0.9168).
2. **LSTM немного лучше Transformer'а**, особенно по R^2 (0.8921 против 0.8834) и MAPE (0.3693 против 0.3998).
3. **Нейросетевые модели пока уступают линейной**, что может быть связано с недостатком данных или чрезмерной сложностью моделей для данной задачи.
4. **Максимальная ошибка у Transformer'а самая высокая** (9.4440), что может говорить о нестабильности предсказаний на отдельных примерах.

4. Выводы и рекомендации

- **Для данной задачи линейная регрессия оказывается наиболее эффективной.** Это может говорить о том, что временные зависимости не слишком сложны, и более сложные модели не дают значительного выигрыша.
- **Нейросетевые модели могут показать лучшие результаты при увеличении объема данных.** Если будет доступно больше данных, стоит попробовать дообучить LSTM/Transformer.
- **Стоит проверить простую стратегию копирования предыдущего дня.** Если она дает результаты, сопоставимые с лучшей моделью, значит, сложные модели могут быть избыточны.

В целом, несмотря на лучшие результаты линейной модели, нейросетевые подходы остаются перспективными при наличии большего объема данных и более сложных зависимостей.