TECHNIEK

Leveraging LLMs for Code Conversion in Finance: Best Practices and Challenges

This article explores how LLMs can be leveraged for code conversion in finance.

BACKGROUND

In the finance sector, there is a trend to convert models and code from one language to another, amongst others due to the following reasons:

- Productivity gains: enhancing and improving current workflows with new implementations that can further automate tasks, can lead to efficiency gains.
- Improve maintainability: existing codebases can become difficult to maintain and finding developers with expertise in languages that have become less popular can be challenging.
- Performance boost: due to increasing demands on the existing systems a performance boost might be required.
- Quality boost: the quality of existing code and models might not meet modern standards.

With advancements in natural language processing (NLP), large language models (LLMs), such as GPT-4, have shown promise in aiding code conversion tasks. The LLMs perform well in the direct translation of relatively simple code (e.g. scripts). This article explores how LLMs can be leveraged for code conversion in finance, focusing on three key areas: handling non-direct translations, addressing data integration challenges, and dealing with lack of unit tests.

Throughout our discussion, we'll highlight best practices and potential limitations to consider when employing LLMs in your code conversion projects. For non-direct translations, a code example is given. More examples are given in the accompanying working paper[1].

NON-DIRECT TRANSLATIONS

A challenge in code conversion is translating code that has no direct equivalent in the target language. This often occurs when dealing with language-specific features or specialized libraries. Consider this SAS code that performs linear regression with forward selection of explanatory variables:

```
Proc Reg data=CleanedData outest=ADJRSQ_summary;
    Title 'Adjusted R-Squared - Training data';
    model TARGET_AMT=
          log_INCOME
          No_Income
          [Some variables omitted for brevity]
          URBANCITY_HU /selection = FORWARD AIC VIF BIC MSE groupnames= 'EDUCATION' 'JOB' 'CAR_TYPE';

    output out=ADJRSQ_out pred=yhat residual=resid ucl=ucl lcl=lcl cookd=cook
    covratio=cov dffits=dfits press=prss;
run;
```

Exhibit 1: SAS code for linear regression with forward variable selection.

Translating this to Python requires implementing the forward selection algorithm and handling the various output statistics. Here's how an LLM might approach this translation:

```python
def forward_selection_custom(X, y) -> list[str]:
    initial_features = []
    remaining_features = list(X.columns)
    best_features = []

    while remaining_features:
        criterion_candidates = []
        for feature in remaining_features:
            X_selected = X[initial_features + [feature]]
            model = sm.OLS(y, X_selected).fit()
            criterion_candidates.append((feature, model.aic))

        best_candidate = sorted(criterion_candidates, key=lambda x: x[1])[0]

        if best_candidate[1] < model.aic:
            # Code omitted for brevity
            ...
        else:
            break

    return best_features

def run_regression(X, y):
    selected_features = forward_selection_custom(X, y)
    X_selected = X[selected_features]
    model = sm.OLS(y, X_selected).fit()
    return model, selected_features
```

Exhibit 2: LLM translated python code for regression with forward variable selection.

When considering the quality of the translation, some observations can be made:

1. **The good**: the LLM has done a decent job of implementing its own forward selection regression algorithm, without the input prompt specifying that this was required.
2. **The bad**: There is no guarantee that the regression and forward selection algorithm that are being used will lead to the same outcome. For the regression algorithm, there might be implementation differences that cause (numerical) difference. Similarly, there could be differences in the implementation of optimization measures that could cause different variables to

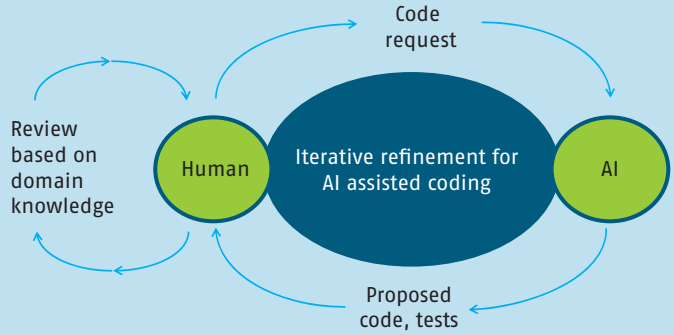
B. Jochems is partner at Risk at Work and helps clients in the areas of quantitative finance, data science and IT.

1 – See Jochems (2024), Code conversion using LLMs, working paper.

enter. Both are issues that could or could not really matter, depending on the specific application.

3. **The ugly**: the SAS code has a particularity that isn't included in the translation. This is the part that specifies "groupnames ...". This essentially tells SAS that these are categorical variables (meaning that they have values in a few categories instead of numerical values) and how the model should deal with those. This feature is completely missing in the Python code.

Best Practice: Iterative Refinement
When dealing with non-direct translations like this SAS to Python conversion, it is best practice to use the LLM-generated code as a starting point. Then, iteratively refine the code with domain expertise. In this case, you might need to adjust the forward selection algorithm to more closely match SAS's implementation, deal with categorical variables or add additional diagnostic statistics that are important for your specific use case.



In addition, LLMs can also generate tests using so-called mocks. What this does is essentially replacing part of the code with pre-generated outcomes. This is for example especially useful for testing if the regression model implementation differs between SAS and Python, without the results being influenced by the outcomes of the variable selection.

Limitation: Domain-Specific Knowledge and Edge Cases
LLMs may struggle with highly specialized financial models or proprietary libraries. In our SAS example, the LLM didn't fully implement all the options specified in the original code. With further iterative refinement, this can be improved.

Moreover, even though it may seem that LLMs generate good test cases, they could also be subtly wrong, even when the code looks good at first glance. It's important to review and supplement the generated tests with domain-specific test cases that reflect real-world usage of your models.

By combining LLM-generated code with rigorous testing and domain expertise, you can ensure that your converted code not only replicates the functionality of the original but also keeps the robustness required for financial applications.

CHALLENGES OF INTEGRATING CODE INTO EXISTING SYSTEMS

Beyond function conversions, integrating new code into pre-existing architectures presents additional challenges, especially in finance where systems often use specialized frameworks like Object-Relational Mappers (ORMs).

A common challenge arises when moving from systems that handle data with tables or dataframes (such as R or SQL) to those using ORMs (e.g., SQLAlchemy for Python or Entity Framework for C#). LLMs may convert the logic but might not account for database schema details or query optimizations crucial for performance.

Best Practice: Context Awareness
To improve the translation, we can provide the LLM with context about our ORM setup, model relationships, and project conventions. With this context, the LLM could produce a more appropriate translation.

Limitation: Performance Considerations
While the context-aware translation is more aligned with the project's structure, it's crucial to note that ORMs can sometimes generate suboptimal SQL, especially for complex queries. For instance, if this query is performance-critical, one might need to add indexing hints or partitioning strategies that are specific to your database system.

UNIT TESTING

Unit testing, i.e., the act of testing small components of functionality in isolation, is fundamental to ensuring high-quality implementations, helping to pinpoint functionality issues and document expected behaviour. However, in practice, many financial models brought to production often lack comprehensive unit tests. LLMs can play a crucial role in addressing this gap.

LLMs can assist in generating unit tests for both the original code and the target language implementation. This capability is particularly valuable when dealing with models developed in Excel, SAS, R, or Python that lack existing unit tests.

When converting code, LLMs can not only translate the logic but also generate corresponding unit tests to ensure the results remain consistent across both languages. By auto-generating these functional tests, LLMs reduce the manual overhead needed for verifying that the converted code remains consistent with the original version.

Best Practice: Comprehensive Testing
When using LLMs for code conversion, it's crucial to generate unit tests for both the original and converted code. This approach helps to ensure that the functionality stays consistent across languages. Tolerance-based testing can be used to account for minor discrepancies in floating-point arithmetic between languages.

Limitation: Test Coverage
While LLMs can generate basic test cases, they may not cover all edge cases or complex scenarios specific to your financial models. It's important to review and supplement the generated tests with domain-specific test cases that reflect real-world usage of your models. These tests can be generated manually, or be generated through additional prompting.

CONCLUSION

LLMs present a powerful tool for accelerating code conversion in finance, offering solutions for common problems in practice, such as unit testing, non-direct translations, and data integration challenges. However, their effective use requires a balanced approach that combines automated conversion with human expertise. By following best practices such as iterative refinement, comprehensive testing and providing context to LLMs, financial institutions can leverage these tools to modernize their technology stack more efficiently. At the same time, it's crucial to be aware of limitations around test coverage, domain-specific knowledge, and performance optimization.

As LLM technology continues to evolve, its role in code conversion and software development is likely to expand, offering even greater possibilities for streamlining financial technology operations. However, the key to success will always lie in combining the power of AI with human expertise and domain knowledge. ■