Data Science Applications Using F#

CSCI 642 - Advanced Programming Skills

Siddharth Bhople and Nicholas Tibbels

Abstract

This paper explores the benefits of the F# programming language for data science applications, examining its functional-first paradigm, rich data types, and integration with the .NET ecosystem. We demonstrate F#'s capabilities through an example data set analysis including data cleaning, transformation, visualization, and feature engineering using the Palmer Penguins dataset. The benefits of F#'s combination of strongly typed functional programming and seamless .NET integration makes it a compelling choice for data science applications, including financial risk modeling, statistical analysis, and production-ready machine learning systems.

1. Introduction

Data science has emerged as a critical discipline for extracting insights from complex datasets and performing statistical analysis. While Python and R dominate the data science landscape, alternative languages offer unique advantages for specific use cases [1]. F#, a functional programming language targeting the .NET platform, provides a robust language choice for data science that combines the benefits of functional programming with the infrastructure of the .NET ecosystem [2].

F# is an object-oriented programming (OOP) language developed at Microsoft Research in Cambridge, UK, with the first lines of code written in 2002 [3]. The language was designed to unify strongly typed functional programming with the .NET platform and OOP, making powerful functional programming concepts accessible within an enterprise-ready environment. Functional programming provides the benefits of efficient code execution, memory management, and type safety.

This paper examines F#'s capabilities for data science applications, demonstrating its practical use through data manipulation, visualization, and feature engineering. We analyze the language's core features, ecosystem support, and real-world applications to provide a comprehensive understanding of F# as a data science tool.

2. Background and Language Features

2.1 Historical Context and Meta Language

F# is considered to be a dialect of the ML (Meta Language) family of programming languages, which includes popular implementations such as OCaml, Standard ML, and Caml Light. Meta Language is a general use high-level functional programming language that established foundational concepts for type inference and pattern matching. During F#'s development, academics and researchers recognized the benefits of functional programming and sought to bring these advantages to the .NET platform while maintaining compatibility with object-oriented programming. In fact, F#'s development begins almost as a response to the object oriented wave that was sweeping industry and popular development languages [3].

Object oriented programming has a list of valid criticisms such as complex inheritance hierarchies, unreadable and messy code, and abstraction of simple tasks bringing unnecessary complexity to code to name a few. These criticisms as well as the benefits that come from functional programming such as easy to read code, maintained simplicity, and efficient execution lead researchers to begin to design F# to be the best of both worlds. F# was to be a strong choice for data science applications that combines the benefits of functional programming to help mitigate some of the drawbacks that come from OOP.

2.2 Core Language Features

2.2.1 Strongly Typed and Immutable

F# implements a strongly typed system with powerful type inference capabilities. The compiler can deduce types without explicit annotations, reducing boilerplate code while maintaining type safety. This is particularly advantageous in data science, where type mismatches can lead to subtle bugs that are difficult to detect at runtime.

The language is immutable by default, meaning that once a value is assigned, it cannot be changed. This design choice prevents many common programming errors, such as unintended side effects and race conditions in concurrent operations. For data science workflows that process large datasets, immutability ensures that transformations are predictable and reproducible [4].

2.2.2 Records and Discriminated Unions

F# provides rich data types, namely Records and Discriminated Unions, which are fundamental for representing structured data. Records are simple aggregates of named values that provide a lightweight way to classify given data points [2]. They support pattern matching, allowing developers to write concise code that is robust enough to handle special cases, and allow for further analysis of data sets through those special cases.

The following code shows an example of how records can be used to store data and interact with its types. The record of PenguinData which stores each of the measured parameters as fields in the type could also be written to have special cases considered. For example, penguins that match a species and have physical measurements larger than some standard could have special functions applied to them to evaluate interesting features about those penguins.

```
None
type PenguinData = {
    Species: string
    BillLength: float
    BillDepth: float
    FlipperLength: float
    BodyMass: float
}
```

Discriminated Unions enable the representation of heterogeneous data—data that can take multiple forms or types. This feature is particularly useful when working with datasets containing mixed data types or representing different experimental conditions and observations. The following code shows an example of a discriminated union that could be written about shapes.

```
None
type Shape =
    | Rectangle of width: int * height: int
    | Square of side: int
    | Circle of radius: int
```

These type constructs allow F# developers to model domain concepts precisely, making code self-documenting and easily understandable. Discriminated Unions allow for classifying data into separate types, and enforce type safeness when making methods to operate on that data. For example, if one had methods to operate on a Shape data type using the discriminated union above, the method is enforced to be able to handle any case of Shape, namely Rectangles, Squares, and Circles. If the arguments for defining one of the types is unique, a discriminated union will accurately classify and build the correct type. These properties of discriminated unions allow for efficient, type-safe operations on any data set, which is particularly useful for analysis.

Discriminated unions also show some of the strongly typed-ness and type safety of functional programming. They also help prevent one of the pitfalls of OOP in that others cannot naively inherit from this shape data type and expect all the methods to work with no additional coding. In OOP, a discriminated union is analogous to having a base class and writing child classes that

inherit from it. Here, one cannot just inherit from shape and then call methods associated with it; if a new shape type is added to the discriminated union, its methods must be updated where necessary to handle the new form data could take [2].

2.2.3 Let and Do Bindings

F# uses Let bindings to define variables and functions, and Do bindings to execute imperative code within classes or methods [2]. These bindings show the readability of F# code and how the design intent for the language was to be self-documenting and easy-to-read. In the following example, a method called calculateMean is passed in an argument of values and their average is calculated. Normally, this more complex and busy to write or read:

```
None
let calculateMean values =
    let sum = List.sum values
    let count = List.length values
    float sum / float count
```

```
Java
public float calculateMean(float[] values){
       float sum == 0;
       float count == values.length;
       for (float current: values){
              sum += current;
       }
       return sum / count;
}
```

2.2.4 Classes and Object-Oriented Features

Despite its functional-first design, F# fully supports object-oriented programming, including classes, inheritance, and interfaces [2]. This dual nature approach enables developers to leverage the benefits of how OOP provides structure and customization to various aspects of projects. Classes in F# support:

- Single inheritance with the `inherit` keyword
- Interface implementation (which does not count toward the single inheritance limit)
- Generic type definitions
- Member methods defined with the `member` keyword

The OOP allows for flexibility in how solutions operate. A fully realized inheritance structure need not be developed where functional programming can do the heavy lifting. This flexibility makes it a good choice for solving problems as the toolbox is expanded [5]. The structure that object oriented programming provides allows for creation of customized data structures in order to efficiently analyze data sets or to handle operations to prepare for analysis of those data sets.

2.3 Development Environment: Ionide

F# has code extensions and support across various IDE's and code editors, but no dedicated IDE for the language itself. Ionide is a lightweight, cross-platform IDE extension for Visual Studio Code that transforms it into a fully functional F# development environment. Key features include:

- **IntelliSense and Code Completion**: Context-aware suggestions and type information
- **F# Interactive (FSI)**: A Read-Evaluate-Print-Loop (REPL) that enables interactive experimentation—crucial for exploratory data analysis
- **Project Management**: Support for `.fsproj` and `.sln` files with integrated project explorers
- **Debugging and Linting**: Comprehensive error detection and debugging capabilities

The REPL functionality is particularly valuable for data science, as it allows practitioners to load data, test transformations, and visualize results interactively without compiling entire projects. This mirrors the workflow popularized by Jupyter notebooks in the Python ecosystem.

3. F# Ecosystem for Data Science

We begin our discussion of support for F# through packages and available libraries that allow for seamless development in tools already familiar with many developers. We discuss the packages utilized in the example penguin data set analysis.

3.1 Data Manipulation Libraries

3.1.1 Deedle

Deedle is a data manipulation library that provides DataFrames and Series structures similar to Python's pandas or R's data.frame [6]. It enables exploratory data analysis with operations for filtering, grouping, and transforming data:

```
None
let df = Frame.ReadCsv("penguins.csv")
let filtered = df |> Frame.filterRows (fun _ row ->
    row.GetAs<string>("species") = "Adelie")
```

Deedle's functional API integrates naturally with F#'s pipeline operators, creating readable data transformation chains.

3.1.2 FSharp.Stats

FSharp.Stats provides comprehensive statistical analysis capabilities, including distributions, hypothesis testing, and linear algebra operations. The library supports:

- Descriptive statistics (mean, median, variance)
- Probability distributions (normal, binomial, Poisson)
- Hypothesis tests (t-tests, ANOVA, chi-square)
- Regression analysis (linear, polynomial)

3.1.3 MathNet.Numerics

MathNet.Numerics offers advanced mathematical functionality including numerical integration, optimization, and Fast Fourier Transforms. Its capabilities extend from basic linear algebra to complex mathematical operations required in scientific computing.

3.2 Data Visualization: Plotly.NET

Plotly.NET is the de facto standard for data visualization in the F# ecosystem. Built on the Plotly.js JavaScript library, it generates interactive, publication-quality charts that can be rendered in browsers or Jupyter notebooks [8]. The library provides:

- **High-level Chart API**: Type-safe functions for common chart types (scatter, line, bar, box plots, histograms)
- **Styling Functions**: Composable functions using the `Chart.with*` naming convention
- **Interactive Features**: Zoom, pan, and hover tooltips enabled by default
- **Export Capabilities**: Static image export for publications

Example usage:

```
None
open Plotly.NET
let xData = [0.0 .. 0.1 .. 10.0]
let yData = xData |> List.map sin
Chart.Scatter(xData, yData, StyleParam.Mode.Lines_Markers)
|> Chart.withTitle "Sine Wave"
|> Chart.withXAxisStyle("X Values")
|> Chart.withYAxisStyle("Y Values")
|> Chart.show
```

Plotly.NET's functional composition model aligns with F#'s pipeline syntax, enabling intuitive chart construction and customization [7].

3.3 Machine Learning Frameworks

3.3.1 ML.NET

ML.NET is Microsoft's open-source, cross-platform machine learning framework designed for .NET developers. It supports both C# and F# and provides:

- **Automated Machine Learning (AutoML)**: Automatically selects algorithms and hyperparameters
- **Model Builder**: Visual tool for creating ML models without extensive coding
- **Comprehensive ML Tasks**: Classification, regression, clustering, anomaly detection, recommendation systems
- **Production-Ready Deployment**: Models integrate seamlessly into .NET applications

ML.NET's type-safe API prevents common errors during model training and prediction phases. The framework has demonstrated impressive performance, achieving 95% accuracy on Amazon's 9GB review dataset for sentiment analysis while other frameworks failed due to memory constraints [9].

F# integration with ML.NET requires careful attention to type definitions. Records used for data input and output must include the `[<CLIMutable>]` attribute to allow ML.NET to mutate values during prediction [7]:

```
None
[<CLIMutable>]
type IrisData = {
    [<LoadColumn(0)>] SepalLength: float32
    [<LoadColumn(1)>] SepalWidth: float32
    [<LoadColumn(2)>] PetalLength: float32
    [<LoadColumn(3)>] PetalWidth: float32
    [<LoadColumn(4)>] Label: string
}
```

3.3.2 SciSharp Stack

The SciSharp Stack provides .NET ports of popular Python machine learning libraries, including TensorFlow, Keras, and NumPy. This enables F# developers to leverage state-of-the-art deep learning models while remaining within the .NET ecosystem.

3.3.3 Jupyter Notebook Integration

.NET Interactive enables F# to run in Jupyter notebooks, providing an interactive environment for data exploration and visualization. This integration supports the iterative workflow common in data science, where hypotheses are tested and refined through rapid experimentation.

4. Case Study: Palmer Penguins Dataset Analysis

4.1 Dataset Description

The Palmer Penguins dataset, collected by the Palmer Station Long-Term Ecological Research Program in Antarctica from 2007-2009, serves as a modern alternative to the classic Iris dataset [11]. It contains approximately 344 observations of three penguin species (Adélie, Gentoo, and Chinstrap) across three islands of the Palmer Archipelago.

Key variables include:

- **Categorical**: species, island, sex, year
- **Numerical**: bill_length_mm, bill_depth_mm, flipper_length_mm, body_mass_g

This dataset is ideal for demonstrating data cleaning, transformation, and visualization techniques commonly employed in real-world data science projects.

4.2 Data Cleaning and Transformation

Data preparation is a critical phase in any data science workflow, as ensuring the data is accurately prepared for analysis or user-defined work can prevent misrepresenting and misinterpreting results. Our F# implementation addresses common data quality issues:

4.2.1 Loading Data

Using Deedle, data can be loaded efficiently:

```
None
let rawData = Frame.ReadCsv("penguins.csv", hasHeaders=true)
```

4.2.2 Handling Missing Values

Missing values are prevalent in real-world datasets. Our approach:

1. **Remove incomplete observations**: Drop rows missing critical measurements (flipper_length, body_mass)
2. **Standardize categorical variables**: Replace missing sex values with "Unknown" to maintain data integrity

F#'s pipeline operator (`|>`) creates readable data transformation chains ideal for Extract-Transform-Load (ETL) workflows. Data pipelines also improve the readability of code so that line-by-line transformations of data can be followed when debugging new code or understanding old code.

```
None
let cleanData =
    rawData
    |> Frame.dropSparseRows
    |> Frame.fillMissingWith "Unknown" "sex"
```

4.2.3 Feature Engineering

Creating derived features allows for in-depth analysis of data sets and the desired characteristics for analysis which cannot always be directly measured. This has applications in statistical applications as well as machine learning applications. We computed bill area by combining length and depth:

```
None
let dataWithFeatures =
    cleanData
    |> Frame.addCol "bill_area" (fun row ->
        row.GetAs<float>("bill_length_mm") *
        row.GetAs<float>("bill_depth_mm"))
```

This transformation combines related measurements into a single feature that may help further understanding of penguins' biology, and highlights the importance of derived features from data sets in general. Oftentimes in research, measurements are taken of variables that are readily and accurately measured, followed by calculations to retrieve some physical constant or research result, and so the availability of feature engineering is important for data science applications and statistical analysis.

4.2.4 Data Validation

F#'s type system enables compile-time validation of data transformations, reducing runtime errors and allowing for fine grain control over which data points are valid for analysis. Pattern matching can verify data constraints and help remove outliers or inoperable data:

```
None
let validateMeasurement value =
    match value with
    | x when x > 0.0 -> Some x
    | _ -> None
```

4.3 Data Visualization

Visualization can help reveal patterns and relationships that inform subsequent analysis. Data visualization is also a strong tool for presenting data to intended audiences and summarizing findings. Our example analysis utilizes multiple chart types in Plotly.NET:

4.3.1 Scatter Plot: Flipper Length vs. Body Mass

Scatter plots can show distributions of data points as they relate to two variables of interest. The following code shows using Plotly.NET's scatter plot and how functional programming can be simple and elegant:

```
None
Chart.Scatter(
    x = flipperLengths,
    y = bodyMasses,
    mode = StyleParam.Mode.Markers
)
|> Chart.withTitle "Flipper Length vs Body Mass"
|> Chart.withXAxisStyle("Flipper Length (mm)")
|> Chart.withYAxisStyle("Body Mass (g)")
```

Results: The scatter plot revealed a strong positive correlation between flipper length and body mass, suggesting that larger penguins possess proportionally longer flippers.

4.3.2 Box Plot: Bill Length by Species

Box plots effectively display distributional differences across categories. Once again, F#'s code can be easily read and understood, along with simple syntax to produce the plot:

```
None
let speciesGroups =
    cleanData
    |> Frame.groupRowsByString "species"

speciesGroups
|> Map.toList
|> List.map (fun (species, frame) ->
    Chart.BoxPlot(
        y = frame.GetColumn<float>("bill_length_mm"),
        Name = species
    ))
|> Chart.combine
```

Results: The box plot highlighted species-specific variations in bill length, with Gentoo penguins exhibiting distinctly different bill morphology compared to Adélie and Chinstrap species.

4.3.3 Histogram: Body Mass Distribution

Histograms illustrate frequency distributions The following code is very similar to our previous examples and shows how versatile F#'s packages can be when solving problems or analyzing data:

```
None
Chart.Histogram(
    x = bodyMasses,
    NBinsX = 30
)
|> Chart.withTitle "Body Mass Distribution"
```

Results: The histogram revealed a multi-modal distribution, indicating that body mass varies significantly—likely due to differences in species and sex.

4.4 Machine Learning Applications

ML.NET enables F# developers to build production-ready machine learning models without leaving the .NET ecosystem [9]. A classification model can predict penguin species based on morphological features:

```
None
let mlContext = MLContext()

let dataView = mlContext.Data.LoadFromTextFile<PenguinData>(
    path = "penguins.csv",
    hasHeader = true,
    separatorChar = ','
)

let pipeline =
    mlContext.Transforms.Conversion
        .MapValueToKey("Label", "species")
        .Append(mlContext.Transforms.Concatenate(
            "Features",
            [|"bill_length_mm"; "bill_depth_mm";
              "flipper_length_mm"; "body_mass_g"|]
        ))
        .Append(mlContext.MulticlassClassification.Trainers
            .SdcaMaximumEntropy())
        .Append(mlContext.Transforms.Conversion
            .MapKeyToValue("PredictedLabel"))

let model = pipeline.Fit(dataView)
```

The strong type system prevents data mismatches during training and prediction, a common source of errors in dynamically typed languages.

5. Applications and Real-World Use Cases

5.1 Financial Services

F#'s immutability and functional programming reduce errors in financial modeling and risk analysis. Major financial institutions use F# for:

- **Quantitative Analysis**: Pricing derivatives and managing portfolio risk
- **High-Frequency Trading**: Processing market data with low latency
- **Regulatory Compliance**: Maintaining audit trails through immutable data structures

The language's emphasis on correctness aligns with the industry standards for financial operations and analysis [10]. Programming in the financial industry is done as needed and so solutions for these problems must be efficient, simple, and correct. Financial data is easily analyzed by statistical analysis for things such as risk analysis, investment return expectations, etc.

5.2 Scientific Computing

F# mathematical libraries and type safety make it suitable for scientific research. Some areas of research that benefit from these powerful analysis tools include but are not limited to:

- **Bioinformatics**: Analyzing genomic sequences and protein structures
- **Physics Simulations**: Modeling complex physical systems
- **Climate Research**: Processing large-scale environmental datasets
- **Chemical Engineering**: Analysis of reaction rates and chemical species concentration

Utilizing some of the previously mentioned packages and development environments, researchers have the capability to quickly and efficiently analyze relevant data. This analysis can include operating on outliers or special cases of observations and visualization of data through several different types of plots using an easy to code and easy to understand language.

5.3 Web Development and Backend Services

F#'s integration with ASP.NET enables building web applications and APIs [2]. The Suave and Giraffe frameworks provide functional approaches to web development, emphasizing composability and testability. This also ensures that packages and extensions made for .NET web applications can also be compatible with F# code.

6. Advantages and Limitations

The advantages that F# provides for data science applications, along with its disadvantages and areas for improvement will be summarized:

6.1 Advantages

1. **Type Safety**: Strong static typing prevents many runtime errors, critical for production systems
2. **Immutability**: Default immutability ensures predictable, reproducible computations [4]
3. **Expressiveness**: Concise syntax reduces boilerplate, improving code maintainability
4. **Interoperability**: Seamless .NET integration provides access to extensive libraries [2]
5. **Functional Paradigm**: Natural representation of data transformation pipelines
6. **Performance**: Compiled code executes efficiently, suitable for large-scale data processing [5]

6.2 Limitations

1. **Ecosystem Maturity**: Fewer data science libraries compared to Python
2. **Community Size**: Smaller community means fewer tutorials and third-party packages
3. **Learning Curve**: Functional programming concepts may be unfamiliar to developers from imperative backgrounds
4. **Library Stability**: Some data science libraries are community-maintained and may lack enterprise support
5. **Tooling**: IDE support, while improving, lags behind mainstream languages

Despite these limitations, F# offers advantages for specific use cases, particularly where type safety, correctness, and .NET integration are priorities. As it continues to grow and develop support, it maintains itself as a viable option for most data science needs [5].

7. Future Directions

The F# ecosystem continues to evolve, with recent developments being prioritized to follow trends in the field. Machine learning and artificial intelligence are becoming increasingly sought after tools in many disciplines and areas of research, and so support for it through things such as ML.NET [9] allows F# to be a valuable language for it. Cloud computing and cloud services are also becoming increasingly utilized by marketing and web services and so analyzing retail data or user data is a good use case for F# and its data science applications. Compatibility for popular packages or applications used by researchers is also an ongoing opportunity for expansion and development in F#. In summary:

1. **Enhanced Notebook Support**: Improved .NET Interactive integration for data exploration
2. **Performance Optimizations**: Continued improvements in runtime efficiency

3. **Machine Learning Integration**: Deeper ML.NET integration and AutoML capabilities
4. **Cloud Computing**: Enhanced Azure integration for scalable data processing

8. Conclusion

F# is a modern option for data science applications that provides the benefits of both functional programming and OOP, particularly in environments where type safety, correctness, and .NET integration are valuable. Our example analysis of the Palmer Penguins dataset demonstrates that F# provides comprehensive capabilities for data cleaning, transformation, visualization, and feature engineering.

While F# may not replace Python or R as the dominant languages in data science education and research, it offers distinct advantages for enterprise applications, financial modeling, and scenarios requiring robust, maintainable code. The language's functional-first design, combined with pragmatic object oriented support, creates a powerful platform for building reliable data science applications.

Organizations that utilize existing .NET infrastructure, those in regulated industries requiring auditability, or developer teams valuing type safety and compile-time verification should strongly consider F# for their data science workflows. As the ecosystem matures and tooling improves, F# provides an excellent option for general purpose data science applications both in industry and in academia.

References

[1] "Guide - Data Science," fsharp.org. [Online]. Available: https://fsharp.org/guides/data-science/

[2] "What is F#," Microsoft Learn. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/fsharp

[3] D. Syme, "The Early History of F#," Microsoft Research, 2020. [Online]. Available: https://fsharp.org/history/hopl-final/hopl-fsharp.pdf

[4] "F# FOR DATA SCIENCE," SEMAGLE. [Online]. Available: https://semagle.com/blog/datascience/fsharp-for-data-science

[5] R. Dennyson, "Where F# Outshines Other Languages: A Deep Dive with Use Cases," Medium, 2024. [Online]. Available: https://medium.com/@robertdennyson/where-f-outshines-other-languages-a-deep-dive-with-use-cases-00f8ab187fc9

[6] "Deedle: Exploratory data library for .NET," Deedle Documentation. [Online]. Available: https://fslab.org/Deedle/

[7] M. Eland, "Machine Learning in .NET with F# and ML.NET 2.0," Matt on ML.NET, 2022. [Online]. Available: https://accessibleai.dev/post/mlnet_fsharp_regression/

[8] "Plotly.NET Documentation," plotly.net. [Online]. Available: https://plotly.net/

[9] "What is ML.NET?," Microsoft Learn. [Online]. Available: https://dotnet.microsoft.com/en-us/learn/ml-dotnet/what-is-mldotnet

[10] D. Syme, "F# in Finance and Financial Services," Microsoft Research, 2015.

[11] "Palmer Penguins Dataset," Kaggle. [Online]. Available: https://www.kaggle.com/datasets/satyajeetrai/palmer-penguins-dataset-for-eda.