**Project Name**

**"Euclid: Predictive Analytics Platform"**

Inspired by the Euclid computer from the movie *Pi*, the name reflects a sophisticated, data-driven platform capable of unraveling financial trends and patterns.

**Frontend Module: Euclid Dashboard**

The frontend of **Euclid** is implemented using **Streamlit**, an open-source Python framework, to deliver an interactive, real-time dashboard for financial analytics and stock predictions. Below is an in-depth breakdown of its components, features, and functionality.

**Key Features and Functional Flow**

**1. Dashboard Framework and Configuration**

The dashboard is designed for accessibility and responsiveness:

* **Streamlit's UI Framework** powers the frontend.
* **Customizable Parameters**: Users can select data sources, models, and forecast parameters.
* **CSS Styling** enhances the visual appeal.

**Code Snippet:**

st.set\_page\_config(

page\_title="Financial Prediction Dashboard",

page\_icon="📈",

layout="wide",

initial\_sidebar\_state="expanded"

)

**Highlights:**

* Wide layout ensures all widgets, charts, and tables are easily viewable.
* Sidebar provides a structured way to navigate and configure settings.

**2. Data Fetching and Caching**

**Euclid** efficiently fetches and stores data using:

* **API Integration**: Communicates with backend endpoints to retrieve country lists, symbols, and model predictions.
* **Caching Layer**: st.cache\_data and DataCache classes optimize repeated API calls by storing data locally.

**DataCache Class:**

class DataCache:

"""Handles caching of API responses and data on the frontend side."""

def \_\_init\_\_(self, cache\_dir: Path):

self.cache\_dir = cache\_dir

self.cache\_duration = 3600 # 1 hour

def get(self, key: str) -> Optional[Dict]:

"""Retrieve cached data if available and valid."""

...

* DataCache enables temporary storage of symbols, avoiding redundant API calls.
* Cached files are stored in .cache directory for quick lookup.

**3. Interactive Sidebar for Configuration**

The sidebar allows users to configure:

* **Data Selection**: Asset type (stocks, indices, ETFs) and country filtering.
* **Model Parameters**:
  + Prediction Period
  + Confidence Interval
  + Train-Test Split
  + Seasonality Period

**Code Snippet:**

country\_choice = st.selectbox(

'Select Country',

['All'] + countries,

format\_func=lambda x: 'Global (No Country Specified)' if x == 'All' else x

)

csv\_choice = st.selectbox(

'Select Asset Type',

CSV\_FILES,

format\_func=lambda x: x.replace('.csv', '').title()

)

**Highlights:**

* Dynamic dropdown menus populate options based on available backend data.
* Input validation ensures appropriate ranges for prediction parameters.

**4. Symbol Selection and Model Training**

Users can:

* Select a stock symbol from the dropdown.
* Trigger model training by clicking the **"Begin Prediction"** button.

**Code Snippet:**

if st.button('Begin Prediction', key='predict'):

response = requests.post(

f"{API\_BASE\_URL}/train\_model",

json={

"symbol": symbol\_choice,

"period": prediction\_period,

"confidence": confidence\_interval,

"train\_split": train\_test\_split / 100,

"seasonality": seasonality\_period

},

timeout=120

)

**Highlights:**

* Parameters are sent to the backend for processing.
* Training progress is visually indicated with a spinner.

**5. Dynamic Tabs for Results**

Results are displayed in five tabs, each catering to specific analysis areas.

**Tab 1: Model Performance**

* Displays a comparison of trained models using metrics (RMSE, MAE, etc.).
* Interactive bar plots highlight performance differences.

metrics\_df = display\_model\_metrics(result['results'])

st.dataframe(metrics\_df, use\_container\_width=True)

st.plotly\_chart(plot\_model\_comparison(result['results']), use\_container\_width=True)

**Tab 2: Forecast Visualization**

* Integrates backend-generated plots for forecasted prices.
* Visualizes training, test, and forecast data.

html\_content = requests.get(f"http://127.0.0.1:5000{result['plot\_url']}").text

st.components.v1.html(html\_content, height=600)

**Tab 3: Training Summary**

* Key training details are presented, including:
  + Number of data points
  + Train-test split
  + Seasonal strength

**Tab 4: Company Info**

Fetches company-specific details using the Yahoo Finance API.

company\_data = fetch\_company\_info(symbol)

info\_df = pd.DataFrame([{"Field": k, "Value": v} for k, v in company\_data.items()])

st.table(info\_df)

**Tab 5: News & Sentiment**

* Fetches and analyzes recent news articles for the selected symbol.
* Sentiment analysis highlights the emotional tone of articles.

news\_articles = stock.news

for article in news\_articles:

sentiment = analyze\_sentiment(article.get('summary', ''))

...

**6. Advanced Visualizations**

**Plotly** is used to create interactive charts:

* **Model Comparison**: Groups metrics across models.
* **Forecast Charts**: Combines actual vs. forecasted prices for visual clarity.

fig.add\_trace(go.Scatter(

x=test\_data.index,

y=result\_dict['forecast'],

name=f'{model\_name} Forecast',

line=dict(dash='dash')

))

**7. Error Handling**

Logs errors and provides clear feedback:

* Displays API connectivity issues.
* Catches model training failures.

try:

response.raise\_for\_status()

except Exception as e:

st.error(f"An error occurred: {str(e)}")

**8. Footer**

A neatly designed footer provides project acknowledgments and timestamped updates.

st.markdown(

f"<div style='text-align: center'>"

f"<p>Made with ❤ | Data provided by Yahoo Finance | Last updated: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}</p>"

f"</div>",

unsafe\_allow\_html=True

)

**Backend Module: Euclid API**

The backend of **Euclid** is implemented using **Flask**, a lightweight and flexible web framework, to provide API endpoints for fetching data, training models, and delivering analytics. Below is an in-depth explanation of its architecture, features, and functionality.

**Key Features and Functional Flow**

**1. API Design**

The backend exposes multiple endpoints to interact with financial datasets and predictive models. Key routes include:

* /api/health: Health check for the backend service.
* /api/csv/<filename>: Fetch stock symbols from CSV files.
* /api/countries: Retrieve available countries for filtering.
* /api/train\_model: Train and evaluate predictive models for a given symbol.

**Example Endpoint:**

@app.route("/api/health", methods=["GET"])

def health\_check():

"""API endpoint for health checking."""

try:

return jsonify({

"status": "healthy",

"timestamp": time.time(),

"uptime": time.time() - getattr(app, 'start\_time', 0)

})

except Exception as e:

logger.error(f"Error in health check: {str(e)}")

return jsonify({"status": "error", "error": str(e)}), 500

**2. Global Data Management**

The backend uses a global in-memory dictionary (DATA\_DICT) to store financial data loaded from CSV files. This ensures fast data retrieval without repeated file I/O operations.

**Loading Data into Memory:**

def load\_csv\_into\_memory():

"""Reads CSV files from disk (LOCAL\_DIR) into global DATA\_DICT once."""

global DATA\_DICT

DATA\_DICT = {}

for csv\_file in app.config['CSV\_FILES']:

file\_path = os.path.join(app.config['LOCAL\_DIR'], csv\_file)

if os.path.exists(file\_path):

df = pd.read\_csv(file\_path)

if 'country' in df.columns:

df['country'].fillna('None', inplace=True)

DATA\_DICT[csv\_file] = df

* CSV files are loaded during app initialization or when manually triggered.

**3. Data Download and Periodic Updates**

The backend periodically downloads updated financial datasets from a remote repository. This is achieved through:

* **Safe Downloading**: Handles errors gracefully during file downloads.
* **Background Task**: A separate thread continuously updates the dataset every hour.

**Background Task:**

def periodic\_download():

"""Background task for periodic downloads."""

while True:

try:

logger.info("Starting periodic download...")

download\_csv\_files()

logger.info("Periodic download completed.")

time.sleep(3600) # Wait for 1 hour

except Exception as e:

logger.error(f"Error in periodic download: {str(e)}")

time.sleep(300) # Wait for 5 minutes before retry

**File Downloading:**

def safe\_download\_file(url, local\_path):

"""Safely download a file with error handling."""

try:

response = requests.get(url, timeout=30)

response.raise\_for\_status()

with open(local\_path, "wb") as file:

file.write(response.content)

return True

except Exception as e:

logger.error(f"Error downloading {url}: {str(e)}")

return False

**4. Model Training and Evaluation**

The /api/train\_model endpoint handles:

* Fetching historical stock data from Yahoo Finance using the yfinance library.
* Training various time-series models (ARIMA, SARIMA, Holt-Winters, etc.).
* Evaluating model performance using metrics like RMSE, MSE, and R².

**Key Steps in Training:**

1. **Fetch Data**: Retrieve historical stock prices.
2. **Scale Data**: Normalize values for better model performance.
3. **Split Data**: Divide into training and testing sets.
4. **Train Models**: Fit multiple models to the training data.
5. **Evaluate Models**: Compute metrics for each model and determine the best one.

**Code Snippet:**

def train\_and\_evaluate\_models(prices, train\_split=0.8, seasonality=12, period=30):

"""Train and evaluate multiple time series models using Statsmodels."""

scaler = MinMaxScaler()

scaled\_prices = scaler.fit\_transform(prices.values.reshape(-1, 1)).flatten()

# Train-test split

train\_size = int(len(scaled\_prices) \* train\_split)

train = scaled\_prices[:train\_size]

test = scaled\_prices[train\_size:]

results = {}

best\_model = None

best\_rmse = float('inf')

# ARIMA Model

arima\_model = ARIMA(train, order=(1, 1, 1)).fit()

arima\_forecast = arima\_model.forecast(steps=len(test))

metrics = calculate\_metrics(test, arima\_forecast)

results['ARIMA'] = metrics

if metrics['rmse'] < best\_rmse:

best\_rmse, best\_model = metrics['rmse'], 'ARIMA'

# Repeat for other models...

**5. Visualization Generation**

Generates interactive plots for model forecasts using **Plotly**:

* Training Data
* Test Data
* Predictions for each model

**Code Snippet:**

fig.add\_trace(go.Scatter(

x=train\_data.index,

y=train\_data.values,

name='Training Data',

line=dict(color='black', width=2)

))

fig.add\_trace(go.Scatter(

x=test\_data.index,

y=test\_data.values,

name='Actual Test Data',

line=dict(color='blue', width=2)

))

**6. Error Handling and Retry Logic**

The backend employs **Tenacity** for robust retry mechanisms during:

* Data fetching (e.g., from Yahoo Finance).
* File downloads.

**Retry Logic:**

@retry(

stop=stop\_after\_attempt(3),

wait=wait\_exponential(multiplier=1, min=4, max=10),

retry=retry\_if\_exception\_type((requests.exceptions.RequestException, ValueError))

)

def fetch\_ticker\_data(symbol, period="max"):

"""Fetch ticker data with retry logic."""

ticker = yf.Ticker(symbol)

history = ticker.history(period=period)

if history.empty:

raise ValueError(f"No data available for symbol {symbol}")

return history

**7. Metrics Calculation**

Defines reusable functions to compute metrics for model evaluation.

**Code Snippet:**

def calculate\_metrics(true\_values, predictions):

"""Calculate various performance metrics."""

return {

'mse': mean\_squared\_error(true\_values, predictions),

'rmse': np.sqrt(mean\_squared\_error(true\_values, predictions)),

'mae': mean\_absolute\_error(true\_values, predictions),

'r2': r2\_score(true\_values, predictions),

'mape': np.mean(np.abs((true\_values - predictions) / true\_values)) \* 100

}

**8. Initialization**

On startup:

1. CSV files are downloaded (if not available).
2. Data is loaded into memory.
3. Background tasks are initialized.

**Code Snippet:**

def init\_app(app):

"""Initialize the Flask application."""

app.start\_time = time.time()

download\_csv\_files()

thread = threading.Thread(target=periodic\_download, daemon=True)

thread.start()