COS30018 - Option C - Task 7 Report: Sentiment-Based Prediction

Anh Vu Le
104653505

I. Introduction and Executive Summary

1.1 Project Goal and Transition

Task C.7 marks a significant shift in the FinTech101 project from **Regression** (predicting the exact future price) to **Classification** (predicting the next-day price movement: UP or DOWN). The core goal is to build a robust model for the stock ticker **CBA.AX** (Commonwealth Bank of Australia) by integrating two distinct data modalities: **Technical Indicators** (from historical price data) and **Sentiment Features** (from external news sources).

1.2 Key Enhancement and Flow

The key enhancement of this task is the incorporation of market psychology through news sentiment. The end-to-end pipeline is orchestrated by `task7_runner.py` and follows this main flow:

1. **Data Collection:** Scrape news articles using `web_scraper.py`.
2. **Sentiment Analysis:** Generate daily sentiment scores (mean, volatility).
3. **Feature Engineering:** Combine 14 Technical Indicators + 2 Sentiment Features using `feature_builder.py`.
4. **Modelling:** Train 21 different classification models (XGBoost, Gradient Boosting, etc.).
5. **Evaluation:** Compare performance against a technical-only baseline to quantify sentiment's value.

1.3 Main Finding

A comprehensive comparative study showed that sentiment features add significant predictive power. The best-performing model, **Gradient Boosting (Sentiment-Only Features)**, achieved an **F1 Score of 68.8%**. This result demonstrated a substantial **+4.8% improvement** in F1 Score compared to the best Technical-Only Baseline (F1 = 65.7%), confirming the hypothesis that news sentiment captures valuable forward-looking information.

Section II. Data Collection & Preprocessing

This section details the complete methodology for acquiring, cleaning, and synchronizing the two distinct data streams required for the analysis: (1) historical stock prices and (2) textual news data.

Chunk 1: Stock Data Collection (The Temporal Backbone)

Flow: The entire project's timeline is built upon the stock market's trading calendar. This data stream acts as the "temporal backbone" for the project. The flow is initiated in the `task7_runner.py` script. This script calls the `yfinance` library to send an API request for the `CBA.AX` ticker over a specified 2-year period. The `yfinance` library conveniently handles the first layer of preprocessing: it *only* returns data for valid trading days, automatically excluding all weekends and public holidays. This clean, 505-row DataFrame, indexed by date, is then passed directly to the feature engineering stage.

Code Implementation (from `task7_runner.py`): This code chunk from the main runner script executes the collection of the stock price data.

```
    # Download historical stock data from Yahoo Finance
  ticker = 'CBA.AX'  # Commonwealth Bank of Australia
  end_date = datetime.now()
  start_date = end_date - timedelta(days=730)  # Last 2 years

  stock_df = yf.download(ticker, start=start_date, end=end_date, progress=False)
  stock_df.reset_index(inplace=True)
  stock_df.columns = [col[0] if isinstance(col, tuple) else col for col in stock_df.columns]

  print(f"\n[OK] Downloaded {len(stock_df)} days of stock data")
```

Code Explanation (by Chunk):

- **Chunk 1 (Configuration):** The `ticker`, `end_date`, and `start_date` variables are defined. `timedelta(days=730)` ensures a consistent 2-year window of data is requested.
- **Chunk 2 (API Call):** `yf.download(...)` is the core function call. It queries the Yahoo Finance API and returns a Pandas DataFrame containing the Open, High, Low, Close, and Volume data. `progress=False` is used to suppress the console output.
- **Chunk 3 (Cleaning):** `stock_df.reset_index(inplace=True)` moves the `Date` from the DataFrame's index into a regular column. The subsequent line,

`stock_df.columns = ...`, is a helper function to flatten any multi-level column headers that `yfinance` might return, ensuring clean column names like `Close` instead of `('Close', '')`.

Chunk 2: News Data Collection (The Sentiment Source)

Flow: The second data stream involves collecting the textual news data. As API limits are a major constraint, the project uses a robust web scraping approach defined in `web_scraper.py`. The flow is as follows:

1. The `WebNewsCollector` class is instantiated.
2. The main `collect_news` method is called, which orchestrates calls to multiple sub-methods (e.g., `_scrape_yahoo_finance`, `_scrape_google_finance`, `_fetch_rss_feeds`).
3. Each sub-method uses the `requests` library to fetch HTML/XML and `BeautifulSoup` to parse the content, extracting the `title`, `description`, `date`, and `source`.
4. All collected articles are aggregated into a single list, converted to a DataFrame, and then passed to the cleaning flow (Chunk 3).
5. *Note:* The main `task7_runner.py` script loads this data from a pre-collected CSV (`task7_data/news_raw/news_raw.csv`), but the `web_scraper.py` file defines the original collection logic.

Code Implementation (from `web_scraper.py`): This code chunk shows the scraping logic for a public RSS feed, which is a reliable method for collecting structured news data without an API key.

```
def _fetch_rss_feeds(self, company_name: str, max_articles: int) -> List[Dict]:
    """
    Fetch news from financial RSS feeds
    """
    articles = []

    rss_feeds = [
        'https://www.ft.com/rss/companies/banks',
        'https://feeds.bloomberg.com/markets/news.rss',
        'https://www.reuters.com/rssFeed/businessNews',
    ]

    for feed_url in rss_feeds:
        try:
            # 1. Fetch the RSS (XML) data
            response = self.session.get(feed_url, timeout=10)
            if response.status_code != 200:
                continue

            # 2. Parse the XML
            soup = BeautifulSoup(response.content, 'xml')
            items = soup.find_all('item')

            for item in items:
                # 3. Extract relevant fields
                title = item.find('title').get_text(strip=True)

                # 4. Filter by relevance
                if company_name.lower() not in title.lower():
                    continue

                description = item.find('description').get_text(strip=True) if item.find('description')
else ''
                link = item.find('link').get_text(strip=True) if item.find('link') else ''
                pub_date_str = item.find('pubDate').get_text(strip=True) if item.find('pubDate') else
''
                date = self._parse_date(pub_date_str) # (Helper function to parse date)

                # 5. Append to list
                articles.append({
                    'date': date,
                    'title': title,
                    'description': description,
                    'content': description,
                    'source': 'RSS Feed',
                    'url': link
                })
        except Exception as e:
            continue
    return articles
```

Code Explanation (by Chunk):

- **Chunk 1 (Iteration):** The code iterates through a predefined list of `rss_feeds`.
- **Chunk 2 (Fetch & Parse):** It uses `self.session.get` to download the raw XML content of the feed. `BeautifulSoup(response.content, 'xml')` is then used to parse this content, and `soup.find_all('item')` creates a list of all news articles in that feed.
- **Chunk 3 (Extraction & Filtering):** For each `item`, the code extracts the `title`, `description`, and `pubDate`. A crucial business logic step is performed: `if company_name.lower() not in title.lower(): continue`. This filters out irrelevant articles, ensuring only news related to "Commonwealth Bank" is kept.
- **Chunk 4 (Aggregation):** The cleaned, relevant data is appended as a dictionary to the `articles` list, which is returned at the end.

Chunk 3: News Text Cleaning (Sanitization)

Flow: The 99 raw articles are unusable for sentiment analysis. They are "dirty" with digital noise. This flow, defined in `news_collector.py`, sanitizes them.

1. **Input:** A raw text string (e.g., a `title` or `description`) is passed to the `_clean_text` function.
2. **Normalization:** The text is converted to lowercase for consistency (e.g., "Profit" and "profit" are treated as the same word).
3. **Sanitization Pipeline:** The text is piped through a series of regular expression (regex) filters to strip noise in a specific order:
    - First, HTML tags (e.g., `<p>`, `<strong>`) are removed.
    - Second, URLs (e.g., `http://...`) are removed.
    - Third, all non-alphanumeric characters (punctuation, symbols) are removed.
4. **Whitespace Collapsing:** All excess whitespace (tabs, newlines, multiple spaces) is

collapsed into single spaces.

5. **Output:** A clean, normalized string containing only words is returned, ready for sentiment analysis.

Code Implementation (from `news_collector.py`): The `_clean_text` method implements this sanitization pipeline.

```
def _clean_text(self, text: str) -> str:
    """
    Clean individual text field
    """
    if pd.isna(text) or not isinstance(text, str):
        return ''

    # --- Chunk 3.1: HTML Tag Removal ---
    # Use BeautifulSoup to parse and remove any HTML
    text = BeautifulSoup(text, 'html.parser').get_text()

    # --- Chunk 3.2: URL and Special Char Removal ---
    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)
    # Remove email addresses
    text = re.sub(r'\S+@\S+', '', text)
    # Remove special characters but keep basic punctuation
    text = re.sub(r'[^a-zA-Z0-9\s.,!?-]', '', text)

    # --- Chunk 3.3: Whitespace Collapsing ---
    # Remove extra whitespace, newlines, tabs
    text = ' '.join(text.split())

    return text.strip()
```

Code Explanation (by Chunk):

- **Chunk 3.1 (HTML Removal):** `BeautifulSoup(text, 'html.parser').get_text()` is a robust way to strip all HTML tags from the text, leaving only the human-readable content.
- **Chunk 3.2 (Regex Sanitization):** This chunk is a pipeline of regular expressions. `re.sub(r'http\S+...', '', ...)` finds and removes all URLs. `re.sub(r'[^a-zA-Z0-9\s.,!?-]', '', text)` is a key filter that removes any character that is *not* a letter, number, whitespace, or basic punctuation, effectively cleaning out noise.
- **Chunk 3.3 (Whitespace Collapsing):** `text.split()` breaks the string into a list of words, which discards all whitespace. `' '.join(...)` reassembles them with single spaces. This is a highly efficient way to normalize all whitespace.

Chunk 4: Time Alignment (The Critical Synchronization Step)

Flow: This is the most critical preprocessing step, addressing the temporal mismatch between the 24/7 news cycle and the Mon-Fri stock market.

1. **Problem:** A positive news article published on a Sunday must influence the *next* trading day (Monday), not the non-existent "Sunday" stock price.
2. **Solution (Flow):** We must "push" all weekend news forward to the next valid trading day.
3. **Implementation:** The `_align_to_trading_days` function (from `news_collector.py`) manages this:
    - **Detection:** It identifies the day of the week (Monday=0, Sunday=6) for each article.
    - **Offset Logic:** It applies business logic to calculate an offset. If `weekday == 5` (Saturday), the offset is +2 days. If `weekday == 6` (Sunday), the offset is +1 day.
    - **Synchronization:** It applies this offset, shifting all weekend dates to the following Monday.
4. **Output:** A DataFrame of news articles where all dates are now aligned with the trading calendar.

Code Implementation (from `news_collector.py`): This function is essential for preventing look-ahead bias.

```
def _align_to_trading_days(self, df: pd.DataFrame) -> pd.DataFrame:
    """
    Align news dates to trading days (Move weekends to next Monday)
    """
    print("  [6] Aligning to trading days...")

    df_aligned = df.copy()

    # --- Chunk 4.1: Detection ---
    # Get the day of the week (Monday=0, Sunday=6)
    df_aligned['weekday'] = df_aligned['date'].dt.dayofweek

    # --- Chunk 4.2: Offset Logic ---
    # Create an offset column, defaulting to 0
    df_aligned['days_to_add'] = 0
    # Find all Saturdays (5) and set their offset to +2 days
    df_aligned.loc[df_aligned['weekday'] == 5, 'days_to_add'] = 2
    # Find all Sundays (6) and set their offset to +1 day
    df_aligned.loc[df_aligned['weekday'] == 6, 'days_to_add'] = 1

    # --- Chunk 4.3: Synchronization ---
    # Apply the offset to the date to get the correct trading date
    df_aligned['date'] = df_aligned['date'] + pd.to_timedelta(df_aligned['days_to_add'], unit='D')

    # --- Chunk 4.4: Cleanup ---
    # Remove the temporary helper columns
    df_aligned = df_aligned.drop(columns=['weekday', 'days_to_add'])

    return df_aligned
```

Code Explanation (by Chunk):

- **Chunk 4.1 (Detection):** The flow starts by using the Pandas `.dt.dayofweek` accessor to get the weekday number for each article's date.

- **Chunk 4.2 (Offset Logic):** This is the core business rule. It uses `.loc` to find all rows where `weekday == 5` (Saturday) and assigns `2` to their `days_to_add` column. It does the same for `weekday == 6` (Sunday), assigning `1`. All other days (Mon-Fri) remain `0`.
- **Chunk 4.3 (Synchronization):** `pd.to_timedelta` is used to safely add the `days_to_add` offset to the original `date`. This operation correctly pushes all Saturday and Sunday dates forward to the following Monday, ensuring the data is temporally sound.
- **Chunk 4.4 (Cleanup):** The helper columns are dropped, leaving the DataFrame clean.

Chunk 5: Data Synchronization & Missing Data Handling

Flow: This is the final preprocessing flow where the two streams (Stock prices and News sentiment) are merged into a single dataset. This logic is handled by the `SentimentFeatureBuilder` class in `feature_builder.py`.

1. **Inputs:**
    - `stock_with_tech`: The 505-day DataFrame of stock prices (from Chunk 1).
    - `sentiment_df`: The DataFrame of news, *after* Section III's sentiment analysis and daily aggregation (e.g., ~50-60 days that *have* news).
2. **Merge:** The two DataFrames are merged using a `left` join, anchored to the `stock_with_tech` DataFrame.
3. **Problem:** This `left` join keeps all 505 trading days but creates `NaN` (Missing) values in the sentiment columns for the 450+ days that had *no* news.
4. **Business Assumption:** The flow makes a critical assumption: **No News = Neutral Sentiment**.
5. **Implementation:** This assumption is implemented by filling all `NaN` values in sentiment columns with `0`.
6. **Output:** A single, synchronized, `NaN`-free DataFrame of 505 rows, ready for modeling.

Code Implementation (from `feature_builder.py`): This code snippet shows the merge and the handling of missing data.

```
def merge_sentiment(self, sentiment_df: pd.DataFrame) -> pd.DataFrame:


    # (Stock data is loaded and technical features added first)

    stock_with_tech = self.add_technical_indicators(self.stock_df.copy())


    # Ensure date columns are matching types

    stock_with_tech['date'] = pd.to_datetime(stock_with_tech['date']).dt.date
```

```
sentiment_df['date'] = pd.to_datetime(sentiment_df['date']).dt.date

# --- Chunk 5.1: The Merge ---
# Merge on date, using 'left' to keep all trading days
merged = pd.merge(
    stock_with_tech,
    sentiment_df,
    on='date',
    how='left'  # <-- Keeps all 505 stock days
)

# --- Chunk 5.2: Missing Data Handling ---
# Apply the "No News = Neutral" assumption
sentiment_cols = [col for col in sentiment_df.columns if col != 'date']

# Fill sentiment_score with 0 (neutral)
if 'sentiment_score' in merged.columns:
    merged['sentiment_score'] = merged['sentiment_score'].fillna(0)

# Fill counts with 0
if 'article_count' in merged.columns:
    merged['article_count'] = merged['article_count'].fillna(0)

# Fill volatility metrics with 0
if 'sentiment_std' in merged.columns:
    merged['sentiment_std'] = merged['sentiment_std'].fillna(0)

return merged
```

Code Explanation (by Chunk):

- **Chunk 5.1 (The Merge):** `pd.merge(..., how='left')` is the key. It anchors the

merge to the `stock_with_tech` DataFrame (the 505 trading days) and only attaches sentiment data from `sentiment_df` where the dates match. All other rows in `stock_with_tech` are kept, but their sentiment columns are filled with `NaN`.
- **Chunk 5.2 (Missing Data Handling):** This chunk implements the "No News = Neutral" assumption. The `.fillna(0)` calls are critical: they replace all `NaN` values (days with no news) with `0`. This correctly informs the model that on those specific days, the sentiment signal was neutral (0.0), not positive or negative.

Section III. Sentiment Analysis

This section details the methodology used to convert the cleaned textual data (from Section II) into quantitative sentiment features. This process involves two main stages: (1) calculating sentiment for individual articles, and (2) aggregating these scores to a daily level to align with the stock price data.

Chunk 1: Tool Selection and Rationale

Flow: The first step in the sentiment analysis flow is selecting the right tool. The project requirements (`Tasks C.7 - Extension.pdf`) allow for experimenting with different tools. A key decision was made to *not* use complex, deep learning models like FinBERT (which is reserved for Section VI: Independent Research) in the main pipeline.

Approach & Rationale: For the primary pipeline (`task7_runner.py`), a **lexicon-based (dictionary-based)** model was chosen.

1. **Tool:** The project uses `TextBlob` (called via the `SentimentAnalyzer` class, as seen in `task7_runner.py`).
2. **Why `TextBlob`?**
    - **Speed & Efficiency:** It is extremely fast, processing 99 articles in milliseconds. This is crucial for a rapid pipeline.
    - **No GPU Required:** Unlike FinBERT, it runs on any CPU.
    - **Simplicity:** It provides a single, standardized polarity score from -1.0 (highly negative) to +1.0 (highly positive), which is ideal for feature engineering.
3. **Instantiation:** The main runner script instantiates the analyzer, explicitly choosing the `'lexicon'` method.

Code Implementation (from `task7_runner.py`): This code shows the instantiation of the chosen sentiment analysis tool.

```
# Task 7 custom modules for sentiment-based prediction

# ...

from task7_sentiment.sentiment_analyzer import SentimentAnalyzer
```

```
# ...

def main():
    # ... (Stage 2) ...
    print("\n" + "="*80)
    print("STAGE 2: SENTIMENT ANALYSIS")
    print("="*80)

    # --- Chunk 1.1: Tool Instantiation ---
    # Initialize sentiment analyzer with lexicon-based method (fast and reliable)
    analyzer = SentimentAnalyzer(primary_model='lexicon')
```

- **Code Explanation:**
  - `from task7_sentiment.sentiment_analyzer import SentimentAnalyzer`: This line imports the custom class responsible for handling sentiment analysis.
  - `analyzer = SentimentAnalyzer(primary_model='lexicon')`: This line creates an instance of the analyzer. By passing `primary_model='lexicon'`, we are configuring the system to use a fast, dictionary-based approach (which in turn calls `textblob`) rather than a heavy neural network.

Chunk 2: Article-Level Sentiment Calculation Flow

Flow: Once the tool is selected, the flow proceeds to analyze each of the 99 news articles individually.

1. **Input:** The DataFrame `news_df` containing the 99 cleaned articles (from Section II).
2. **Iteration:** The flow uses the efficient Pandas `.apply()` method to iterate over the `title` column of the DataFrame.
3. **Analysis:** For each title, the `.apply()` method passes the text (as `x`) to the `analyzer.analyze` function.
4. **Scoring:** The `analyzer` (using TextBlob) processes the text and returns a single polarity score (e.g., `0.35`).
5. **Output:** A new column, `sentiment_score`, is created in the DataFrame, storing the score for each individual article.

Code Implementation (from `task7_runner.py`): This chunk of code performs the sentiment analysis on all 99 articles.

```
# --- Chunk 2.1: Analyze each article title ---
# Apply the sentiment analyzer to the 'title' column
# The lambda function calls the analyzer for each row
news_df['sentiment_score'] = news_df['title'].apply(
    lambda x: analyzer.analyze(str(x), method='textblob')
)


# --- Chunk 2.2: Categorize for interpretability ---
# (This step is for logging/stats, not for feature engineering)
news_df['sentiment_category'] = news_df['sentiment_score'].apply(
    lambda x: 'positive' if x > 0.05 else ('negative' if x < -0.05 else 'neutral')
)
```

Code Explanation (by Chunk):

- **Chunk 2.1 (Analysis):** `news_df['title'].apply(...)` is the core of this flow.
    - `lambda x:`: This creates a small, anonymous function that runs for each title (`x`).
    - `analyzer.analyze(str(x), method='textblob')`: This is the call to the analysis engine. It takes the title string, runs TextBlob's polarity algorithm on it, and returns the float score. The result is saved in the `sentiment_score` column.
- **Chunk 2.2 (Categorization):** This second `.apply()` is used for verification and reporting. It converts the continuous score (e.g., `0.35`) into a simple category ("positive"), which is then used to print the distribution (e.g., "Positive: 65%, Negative: 7%").

Chunk 3: Daily Aggregation Flow (Critical Requirement)

Flow: This is the most important flow in Section III, as it addresses a key requirement from `Tasks C.7 - Extension.pdf`: "scores should be aggregated at a daily level."

1. **Problem:** On a given trading day (e.g., 2024-05-10), there might be 5 news articles with 5 different sentiment scores. The model, however, needs *one* set of sentiment features for that single day.

2. **Solution (Flow):** We aggregate all articles from the *same day* into a single row.
3. **Implementation:** The flow uses the Pandas `groupby()` function on the (already time-aligned) `date` column.
4. **Aggregation:** The `.agg()` function is then called to compute multiple statistics for all articles within that group (day):
    - **`mean`**: The average sentiment score (e.g., `sentiment_mean`). This is the primary feature.
    - **`std`**: The standard deviation of scores (e.g., `sentiment_std`). This measures sentiment *volatility* or *disagreement* (a high `std` means some news was very positive and some very negative).
    - **`count`**: The number of articles (`article_count`).
    - **`positive_ratio`**: The percentage of positive articles.
5. **Output:** A new DataFrame, `daily_sentiment`, where each row represents a single, unique trading day with its computed sentiment features.

Code Implementation (from `task7_runner.py`): This code block aggregates the article-level scores into daily features.

```
# --- Chunk 3.1: Ensure date format ---
# Aggregate sentiment by day
news_df['date'] = pd.to_datetime(news_df['date'])


# --- Chunk 3.2: Group and Aggregate ---
daily_sentiment = news_df.groupby(news_df['date'].dt.date).agg({
    'sentiment_score': ['mean', 'std', 'count'],  # Mean, volatility, article count
    'sentiment_category': lambda x: (x == 'positive').sum() / len(x) if len(x) > 0 else 0  #
Positive ratio
}).reset_index()
```

Code Explanation (by Chunk):

- **Chunk 3.1 (Date Format):** Ensures the `date` column is a `datetime` object, which is required for the `.groupby()` operation.
- **Chunk 3.2 (Group & Agg):** This is the core logic.
    - `news_df.groupby(news_df['date'].dt.date)`: This groups all 99 articles by their unique date.
    - `.agg({ ... })`: This dictionary tells Pandas what to compute for each group.
    - `'sentiment_score': ['mean', 'std', 'count']`: This is the key instruction. It tells Pandas to take the `sentiment_score` column for that day's articles and calculate its mean, standard deviation, and count.

    ◦ `lambda x: ...`: This calculates the ratio of positive articles for that day.

Chunk 4: Sentiment Feature Creation

Flow: The output from Chunk 3 (`daily_sentiment`) has messy, multi-level column names (e.g., `('sentiment_score', 'mean')`). The flow must clean this up to create usable feature columns.

1. **Input:** The `daily_sentiment` DataFrame with multi-level columns.
2. **Flattening:** The column names are flattened into simple, single-level names.
3. **Feature Selection:** As defined in `task7_runner.py`, only the most important aggregated features are selected for the model: `sentiment_mean` and `sentiment_std`.
4. **Output:** A clean DataFrame with columns `date`, `sentiment_mean`, `sentiment_std`, etc., ready to be merged (in Chunk 5) with the stock data.

Code Implementation (from `task7_runner.py`): This code flattens the column names and identifies the final features.

```
# --- Chunk 4.1: Flatten multi-level columns ---
daily_sentiment.columns = ['date', 'sentiment_mean', 'sentiment_std',
                           'article_count', 'positive_ratio']


# --- Chunk 4.2: Handle potential NaNs ---
# Fill std with 0 for single-article days (std is NaN for 1 value)
daily_sentiment['sentiment_std'] = daily_sentiment['sentiment_std'].fillna(0)


# ... (Later in Stage 3) ...


# --- Chunk 4.3: Define Sentiment Feature List ---
# Identify feature columns by prefix
sentiment_features = [c for c in full_df.columns if c.startswith('sentiment_')]
# (This list will resolve to ['sentiment_mean', 'sentiment_std', ...])
```

Code Explanation (by Chunk):

- **Chunk 4.1 (Flattening):** `daily_sentiment.columns = [...]` is a direct way to rename the complex `('sentiment_score', 'mean')` headers to the simple `sentiment_mean`, which is required for the model.

- **Chunk 4.2 (Handle NaNs):** This is a subtle but important data cleaning step. If a day had only *one* article, its standard deviation (`std`) is `NaN` (mathematically undefined). `fillna(0)` correctly sets the sentiment volatility to 0 for those days, as there was no disagreement.
- **Chunk 4.3 (Feature List):** The line `sentiment_features = [c for c in ...]` programmatically selects all columns that start with `sentiment_`, resulting in the final list of features (`sentiment_mean`, `sentiment_std`, etc.) that will be fed to the model.

Chunk 5: Handling News-less Days (Merge & FillNA)

Flow: This final flow connects the sentiment data (now aggregated by day) to the stock data backbone.

1. **Problem:** The `stock_df` has 505 trading days. The `daily_sentiment` DataFrame only has ~50-60 rows (days with news). We must create sentiment features for the 450+ "news-less" days.
2. **Merge:** The flow, executed in `task7_runner.py` (Stage 3), performs a `left` merge, anchoring on the 505-day `stock_df`.
3. **Result:** This merge creates `NaN` (Missing) values in `sentiment_mean` and `sentiment_std` for all days that had no news.
4. **Business Assumption:** A critical assumption is made: **No News = Neutral Sentiment**.
5. **Implementation:** The flow uses `.fillna(0)` to replace all `NaN` values in the sentiment columns with `0`.
6. **Output:** A single, 505-row DataFrame where every trading day has a valid sentiment score (either the calculated score or 0 for neutral).

Code Implementation (from `task7_runner.py`): This code (from Stage 3 of the runner) shows the merge and the handling of missing data.

```
# --- Chunk 5.1: The Merge ---
# Merge stock data with sentiment data on date
# Left join ensures we keep all stock days (even without news)
stock_df['date'] = pd.to_datetime(stock_df['Date']).dt.date
daily_sentiment['date'] = pd.to_datetime(daily_sentiment['date']).dt.date


full_df = stock_df.merge(daily_sentiment, on='date', how='left')
```

```
# --- Chunk 5.2: Missing Data Handling ---
# Fill missing sentiment values (days without news) with neutral values
# This is important: missing news doesn't mean negative sentiment!
full_df['sentiment_mean'] = full_df['sentiment_mean'].fillna(0)  # Neutral sentiment
full_df['sentiment_std'] = full_df['sentiment_std'].fillna(0)  # No volatility
full_df['article_count'] = full_df['article_count'].fillna(0)  # No articles
full_df['positive_ratio'] = full_df['positive_ratio'].fillna(0.5) # 50% positive (neutral)
```

Code Explanation (by Chunk):

- **Chunk 5.1 (The Merge):** `how='left'` is the most important parameter. It preserves all 505 rows from `stock_df` (the left table) and only attaches data from `daily_sentiment` (the right table) where the dates match.
- **Chunk 5.2 (Missing Data Handling):** This chunk implements the "No News = Neutral" assumption. `full_df['sentiment_mean'].fillna(0)` replaces all `NaN` values with `0`. This is a crucial step that ensures the model has a complete dataset and correctly interprets days without news as having a neutral (0.0) sentiment signal.

IV. Feature Engineering & Modelling

This section covers the creation of the final features, the definition of the classification target, the preprocessing pipeline, and the comprehensive model training flow.

Chunk 1: Technical Feature Engineering

Flow: The feature engineering flow begins in Stage 3 of `task7_runner.py`. After loading the 505-day `stock_df`, the script *manually* calculates 14 different technical indicators. This is done to create a rich set of features that capture price momentum, trend, and volatility. These calculations are performed directly using Pandas' rolling and math functions.

Code Implementation (from `task7_runner.py`): This code snippet shows the direct, manual calculation of these technical features.

```
# (Quoted from: task7_runner.py, STAGE 3)

    # --- 1. RETURNS: Measure price changes ---
    stock_df['return_1d'] = stock_df['Close'].pct_change()
    stock_df['return_5d'] = stock_df['Close'].pct_change(5)

    # --- 2. VOLATILITY: Measure price instability ---
    stock_df['volatility_20d'] = stock_df['return_1d'].rolling(20).std()

    # --- 3. MOVING AVERAGES: Smooth price trends ---
    stock_df['ma_20'] = stock_df['Close'].rolling(20).mean()
    stock_df['ma_50'] = stock_df['Close'].rolling(50).mean()

    # --- 4. RSI (Relative Strength Index): Momentum oscillator ---
    delta = stock_df['Close'].diff()
    gain = (delta.where(delta > 0, 0)).rolling(14).mean()
    loss = (-delta.where(delta < 0, 0)).rolling(14).mean()
    rs = gain / loss
    stock_df['rsi'] = 100 - (100 / (1 + rs))
    # --- 5. MACD (Moving Average Convergence Divergence) ---
    exp1 = stock_df['Close'].ewm(span=12, adjust=False).mean()
    exp2 = stock_df['Close'].ewm(span=26, adjust=False).mean()
    stock_df['macd'] = exp1 - exp2
    stock_df['macd_signal'] = stock_df['macd'].ewm(span=9, adjust=False).mean()
```

Code Explanation:

- **Chunks 1-3:** These use standard Pandas functions. `.pct_change(5)` calculates the 5-day return, and `.rolling(20).std()` calculates the 20-day rolling volatility.
- **Chunk 4 (RSI):** This implements the standard RSI formula by calculating the average `gain` and `loss` over a 14-day window.
- **Chunk 5 (MACD):** This calculates the 12-day and 26-day Exponential Moving Averages (EMA) and finds their difference (`macd`) and the 9-day EMA of that difference (`macd_signal`).

Chunk 2: Target Variable Creation (The Core Task)

Flow: This is the most important step for Task C.7. Instead of predicting the *price* (a regression problem), we are predicting the *direction* (a classification problem). The flow is simple but critical:

1. Take the `Close` price column.
2. Use `.shift(-1)` to "look" one day into the future and pull that price onto the current day's row.
3. Compare that future price to the current day's price.
4. If `Future_Close > Current_Close`, set the target to `1` (UP). Otherwise, set it to `0` (DOWN).
5. This process creates `NaN`s (e.g., on the last row), which are then dropped.

Code Implementation (from `task7_runner.py`): This single line of code transforms the project from regression to classification.

```
# (Quoted from: task7_runner.py, STAGE 3)


    # --- Create binary target variable: 1 if price goes UP tomorrow, 0 if DOWN ---
    # shift(-1) looks at next day's price (future prediction target)
    full_df['target'] = (full_df['Close'].shift(-1) > full_df['Close']).astype(int)
    # Remove rows with NaN (caused by rolling windows and shift operations)
    full_df = full_df.dropna()
```

Code Explanation:

- `full_df['Close'].shift(-1)`: This is the key. It pulls tomorrow's closing price onto today's row.
- `(...) > full_df['Close']`: This boolean comparison returns `True` (if the price went up) or `False`.
- `.astype(int)`: This converts `True` to `1` and `False` to `0`, creating our binary target.
- `full_df = full_df.dropna()`: This cleans the DataFrame, removing the first 50 rows (due to the 50-day MA) and the very last row (due to the `.shift(-1)`).

Chunk 3: Preprocessing Pipeline (SMOTE & Scaling)

Flow: Before training, the data (now 456 samples) must be preprocessed. This flow is defined in the `SentimentClassifierTrainer` class within `classifier_models.py`.

1. **Temporal Split:** First, the data is split into training (80%) and testing (20%) sets *temporally* (no shuffling) in `task7_runner.py`.
2. **Scaling:** The `preprocess_data` function is called. It fits a `StandardScaler` to the *training data only* to learn its mean/std. It then uses this *fitted* scaler to transform both the train and test sets.
3. **Balancing (SMOTE):** The training set is imbalanced (149 DOWN vs 215 UP). To fix this, `SMOTE` (Synthetic Minority Over-sampling Technique) is applied *only to the scaled training set*. It synthetically creates new "DOWN" samples until the classes are balanced (215 UP vs 215 DOWN).
4. **Output:** The flow returns the `(X_train_scaled_balanced, y_train_balanced, X_test_scaled)`.

Code Implementation (from `classifier_models.py`): This function shows the complete, correct preprocessing pipeline.

```
  # (Quoted from: classifier_models.py)


    def preprocess_data(self, X_train: pd.DataFrame, y_train: pd.Series,
                        X_test: pd.DataFrame = None,
                        experiment_name: str = 'default') -> Tuple:
```

```
# --- Chunk 3.1: Scaling ---
print("  [1] Scaling features...")
scaler = StandardScaler()

# Fit on TRAIN, transform TRAIN
X_train_scaled = scaler.fit_transform(X_train)
# Transform TEST using TRAIN's scaler
X_test_scaled = scaler.transform(X_test) if X_test is not None else None

self.scalers[experiment_name] = scaler # Store the scaler

# --- Chunk 3.2: SMOTE (on training data ONLY) ---
if self.use_smote:
    print("  [2] Applying SMOTE for class balance...")
    unique, counts = np.unique(y_train, return_counts=True)
    print(f"      Before SMOTE: {dict(zip(unique, counts))}")

    smote = SMOTE(random_state=self.random_state)
    X_train_resampled, y_train_resampled = smote.fit_resample(
        X_train_scaled, y_train
    )

    unique, counts = np.unique(y_train_resampled, return_counts=True)
    print(f"      After SMOTE: {dict(zip(unique, counts))}")

    X_train_processed = X_train_resampled
    y_train_processed = y_train_resampled
# ... (else statement)
return X_train_processed, y_train_processed, X_test_scaled
```

Code Explanation:

- `scaler.fit_transform(X_train)`: Correctly fits *and* transforms the training data.
- `scaler.transform(X_test)`: Correctly applies the *same* transformation (using the training data's mean/std) to the test data. This prevents data leakage.
- `smote.fit_resample(X_train_scaled, y_train)`: This is the crucial SMOTE step. It is applied *after* scaling and *only* on the training set.

Chunk 4: The 3x7 Experimental Design (Training Flow)

Flow: This project runs a comprehensive 3x7 factorial experiment (21 models total), orchestrated by two scripts: `task7_runner.py` and `task7_extended_models.py`.

1. **`task7_runner.py` (Stage 4):** This script trains the first 9 models. It loops through 3 feature sets (Technical, Sentiment, Combined) and 3 algorithms (Logistic, RF, XGBoost).
2. **`task7_extended_models.py`:** This script is then run. It loads the *same* preprocessed data and trains the remaining 12 models: 4 new algorithms (SVM-Linear, SVM-RBF, Gradient Boosting, MLP) on the same 3 feature sets.
3. **`run_all.py`:** This script simply calls the two runner scripts above in sequence to execute the full 21-model experiment.
4. **Output:** All 21 models are saved as `.pkl` files in the `task7_models/` directory, and their results are merged into `evaluation_metrics.json`.

Code Implementation (from `task7_runner.py` & `task7_extended_models.py`): First, the 3x3 loop in `task7_runner.py` defines the 3 feature sets and trains the 9 base models.

```
# (Quoted from: task7_runner.py, STAGE 4)


  # Define three feature sets for comparison
  feature_sets = {
    'technical_only': technical_features,
    'sentiment_only': sentiment_features,
    'combined': technical_features + sentiment_features
  }
```

```
    # Three classification algorithms
    algorithms = ['logistic', 'random_forest', 'xgboost']

    trainer = SentimentClassifierTrainer(use_smote=True, random_state=42)

    # Loop through each feature set
    for feat_name, feat_cols in feature_sets.items():
        # ... (Get X_train, X_test, y_train, y_test for this set) ...
        X_train_proc, y_train_proc, X_test_proc = trainer.preprocess_data(...)

        # Loop through each algorithm
        for algo in algorithms:
            model_name = f"{feat_name}_{algo}"
            model = trainer.train_model(X_train_proc, y_train_proc, model_type=algo, ...)
            # ... (Save model and store predictions) ...
```

Second, `task7_extended_models.py` calls its internal functions to train the other 12 models.

```
# (Quoted from: task7_extended_models.py, main())

    # 1. Train SVM models (6 models: 2 kernels × 3 feature sets)
    svm_results, svm_models = train_svm_models(full_df, tech_feats, sent_feats, y,
split_idx)
    all_new_results.extend(svm_results)

    # 2. Train Gradient Boosting models (3 models: 1 algorithm × 3 feature sets)
    gb_results, gb_models = train_gradientboosting_models(full_df, tech_feats,
sent_feats, y, split_idx)
    all_new_results.extend(gb_results)

    # 3. Train MLP Neural Network models (3 models: 1 architecture × 3 feature sets)
    mlp_results, mlp_models = train_mlp_models(full_df, tech_feats, sent_feats, y,
split_idx)
    all_new_results.extend(mlp_results)


    # Merge with existing results (9 from main + 12 from extended = 21 total)
    update_evaluation_results(all_new_results)
```

Code Explanation:

- The nested `for` loops in `task7_runner.py` clearly show the 3x3 experimental design.
- `task7_extended_models.py` systematically calls its `train_...` functions to train the advanced models on the *exact same* 3 feature sets.
- `update_evaluation_results` (in `task7_extended_models.py`) merges the JSON results, creating one master file with all 21 models for comparison.

V. Evaluations

This section concisely covers how the 21 trained models were evaluated and, most importantly, how the value of sentiment was assessed.

Chunk 1: Comprehensive Metrics Calculation

Flow: The evaluation flow (Stage 5 in `task7_runner.py`) iterates through every prediction made by the 9 base models. The `task7_extended_models.py` script does the same for its 12 models.

1. For each model, it retrieves the true labels (`y_test`) and the model's predictions (`y_pred`).
2. It passes these arrays to a suite of functions from `sklearn.metrics` (e.g., `accuracy_score`, `precision_score`, `recall_score`, `f1_score`).
3. These metrics are stored in a dictionary.
4. Finally, `update_evaluation_results` (in `task7_extended_models.py`) merges all 21 dictionaries and saves them as `evaluation_metrics.json` and `model_comparison.csv`.

Code Implementation (from `task7_runner.py`): This code from Stage 5 shows the comprehensive metrics calculation.

```
# (Quoted from: task7_runner.py, STAGE 5)


  all_results = []  # Store all evaluation results
```

```
# Evaluate each model on test set
for model_name, preds in all_predictions.items():
    y_test = preds['y_test']  # True labels
    y_pred = preds['y_pred']  # Predicted labels

    # Calculate all required metrics for Task C.7
    metrics = {
        'model': model_name,
        'feature_set': preds['feature_set'],
        'algorithm': preds['algorithm'],
        'accuracy': accuracy_score(y_test, y_pred),
        'precision': precision_score(y_test, y_pred, zero_division=0),
        'recall': recall_score(y_test, y_pred, zero_division=0),
        'f1': f1_score(y_test, y_pred, zero_division=0)
    }

    # Confusion matrix: [[TN, FP], [FN, TP]]
    cm = confusion_matrix(y_test, y_pred)
    metrics['confusion_matrix'] = cm.tolist()

    all_results.append(metrics)
```

Code Explanation:

- This block systematically calculates the four key classification metrics (Accuracy, Precision, Recall, F1).
- **Precision (`TP/(TP+FP)`)** answers: "Of all the days we predicted UP, what percentage was actually UP?"
- **Recall (`TP/(TP+FN)`)** answers: "Of all the days that actually went UP, what percentage did we catch?"
- **F1 Score** is the harmonic mean of Precision and Recall, providing a single, balanced score for comparison.

Chunk 2: The Baseline Comparison (The "So What?")

Flow: This is the most important evaluation step, as it directly answers the task's central question: "does sentiment add value?" This flow is executed at the end of `task7_runner.py`.

1. The code filters the `all_results` list to find the *best* model (by F1 score) from the `technical_only` group. This is the **Baseline**.
2. It does the same to find the best model from the `sentiment_only` and `combined` groups.
3. It then prints a direct comparison of these three "champion" models.
4. Finally, it calculates the percentage improvement of the best sentiment-based model over the technical baseline, providing a clear, quantitative answer. *Note: `task7_advanced_evaluation.py` also runs this logic in its `create_summary_report`.*

Code Implementation (from `task7_runner.py`): This block provides the final conclusion for the report.

```
# (Quoted from: task7_runner.py, BASELINE COMPARISON)

 # Best technical-only model (BASELINE - no sentiment features)
 tech_results = [r for r in all_results if r['feature_set'] == 'technical_only']
 best_tech = max(tech_results, key=lambda x: x['f1']) if tech_results else None

 # Best sentiment-only model
 sent_results = [r for r in all_results if r['feature_set'] == 'sentiment_only']
 best_sent = max(sent_results, key=lambda x: x['f1']) if sent_results else None

 # ... (Find best_comb) ...

 # Calculate percentage improvement (sentiment vs baseline)
 if best_tech and best_sent:
     # ... (Print F1 scores) ...
     imp_f1 = (best_sent['f1'] - best_tech['f1']) / best_tech['f1'] * 100
     print(f"F1 Score Improvement: {imp_f1:+.1f}%")
```

```
        if best_sent['f1'] > best_tech['f1']:
            print("\n✅ CONCLUSION: Sentiment features ADD SIGNIFICANT VALUE!")
```

Code Explanation:

- `best_tech = max(..., key=lambda x: x['f1'])`: This line efficiently finds the dictionary in the `tech_results` list that has the highest F1 score. This is our baseline to beat.
- `imp_f1 = ...`: This line calculates the percentage change, directly quantifying the value added by sentiment.
- The final `if` statement provides the project's definitive conclusion.

Chunk 3: Visualization & Reporting Flow

Flow: The final step in the evaluation flow is orchestrated by `task7_advanced_evaluation.py`. This script is designed to be run *after* all 21 models have been trained.

1. **Load:** It loads the master `evaluation_metrics.json` (containing all 21 results) and all 21 `.pkl` model files.
2. **Generate Plots:** It calls its internal plotting functions to generate the three key visualizations:
    - `plot_confusion_matrices`: Creates a 2x3 grid of heatmaps for the Top 6 models.
    - `plot_model_comparison`: Creates a 4-in-1 dashboard comparing all 21 models, their feature sets, and algorithms.
    - `plot_feature_importance`: Extracts and plots the feature importances from all tree-based models (RF, XGBoost, Gradient Boosting).
3. **Generate Reports:** It generates two `.txt` files:
    - `classification_reports.txt`: A detailed, multi-page report with metrics for all 21 models.
    - `executive_summary.txt`: A high-level summary of the findings (as seen in `01_executive_summary.md`).
4. **Orchestration:** The `run_all.py` script simply calls this file at the very end to complete the pipeline.

Code Implementation (from `task7_advanced_evaluation.py`): The `main()` function of this script shows the entire visualization and reporting flow.

```
# (Quoted from: task7_advanced_evaluation.py)

def main():
    # ...
    print("\n[LOAD] Loading evaluation results...")
    with open('task7_results/evaluation_metrics.json', 'r') as f:
        all_results = json.load(f)

    print("\n[LOAD] Loading trained models...")
    # ... (Loop to load all .pkl files into 'models' dict) ...

    # --- Chunk 3.1: Generate Visualizations ---
    print("\n" + "="*80)
    print("GENERATING VISUALIZATIONS")
    print("="*80)

    plot_confusion_matrices(all_results)
    plot_model_comparison(all_results)

    if models:
        plot_feature_importance(models,
                                ['return_1d', ..., 'sentiment_std']) # Pass all 16 feature names

    # --- Chunk 3.2: Generate Text Reports ---
    print("\n" + "="*80)
    print("GENERATING REPORTS")
    print("="*80)

    generate_classification_reports(all_results)
    create_summary_report(all_results)
    # ... (Print summary of outputs) ...


if __name__ == '__main__':
    main()
```

Code Explanation:

- **Chunk 3.1:** This block calls the three key plotting functions. `plot_confusion_matrices(all_results)` takes the full results, finds the top 6, and plots them. `plot_feature_importance` is passed the *models* themselves (which were loaded from the `.pkl` files) so it can access their `.feature_importances_` attribute.
- **Chunk 3.2:** This block calls the two report-writing functions, which save the final text-based analysis to the `task7_results/` directory, completing the evaluation.

Section VI. Independent Research Component

To fulfill the independent research requirement, this project went beyond the baseline `TextBlob` model to explore advanced, domain-specific sentiment techniques. These methods are defined in `finbert_tuner.py` and, while not part of the final 21-model experiment, they provide a clear path for future performance improvements.

Chunk 1: Financial Lexicon Enhancement

Flow: The first enhancement was to create a `FinancialLexiconEnhancer` (in `finbert_tuner.py`). This tool *augments* a base sentiment score (like TextBlob's) by scanning the text for domain-specific keywords relevant to Australian banking.

The flow is:

1. A default lexicon is built with terms like 'profit upgrade' (+0.9) or 'APRA investigation' (-0.7).
2. The text is scanned for these terms to create a `lexicon_score`.
3. This score is combined with the `base_sentiment` using a weighted average.

Code Implementation (from `finbert_tuner.py`): This function shows the weighted average logic, giving 70% weight to the base model and 30% to the lexicon adjustment.

```
# (Quoted from: finbert_tuner.py)


  def enhance_sentiment(self, text: str, base_sentiment: float) -> Tuple[float, Dict]:
```

```
# ... (Finds matching terms and calculates lexicon_score) ...

if matched_scores:
    lexicon_score = np.mean(matched_scores)
else:
    lexicon_score = 0.0

# --- Chunk 1.1: Weighted Average ---
# Combine scores (weighted average)
# Give more weight to base sentiment (70%), lexicon adds adjustment (30%)
enhanced_sentiment = 0.7 * base_sentiment + 0.3 * lexicon_score

# Ensure within bounds [-1, 1]
enhanced_sentiment = np.clip(enhanced_sentiment, -1.0, 1.0)

# ... (Return enhanced_sentiment and details) ...
```

Code Explanation:

- **Chunk 1.1 (Weighted Average):** This logic (`0.7 * base_sentiment + 0.3 * lexicon_score`) provides a simple way to "nudge" the generic sentiment score in the correct direction based on highly specific financial terms, making the model more domain-aware.

Chunk 2: FinBERT Fine-Tuning Strategy

Flow: The second and most advanced technique is the `FinBERTFineTuner` class. The flow is designed to create a "specialist" model:

1. **Load:** It loads the powerful, pre-trained `ProsusAI/finbert` model.
2. **Justification:** This base model is trained on *general* financial news (mostly US/EU). To improve its accuracy for *our* specific task, we must fine-tune it on *our* data (CBA/Australian news).
3. **Auto-Label:** Since we have no human-labeled data, the `prepare_training_data` function uses the pre-trained FinBERT model to create initial "weak" labels for our 99

articles.

4. **Fine-Tune:** The `fine_tune` function then re-trains the model on this small, domain-specific dataset, making it an expert on Australian banking terminology.

Code Implementation (from `finbert_tuner.py`): The `fine_tune` method uses the Hugging Face `Trainer` to re-train the model.

```
# (Quoted from: finbert_tuner.py)

 def fine_tune(self, train_df: pd.DataFrame, val_df: pd.DataFrame = None,
          epochs: int = 3, batch_size: int = 8,
          learning_rate: float = 2e-5):

   # --- Chunk 2.1: Create Datasets ---
   train_dataset = self.create_dataset(train_df)
   val_dataset = self.create_dataset(val_df) if val_df is not None else None

   # --- Chunk 2.2: Define Training Arguments ---
   training_args = TrainingArguments(
     output_dir=self.output_dir,
     num_train_epochs=epochs,
     per_device_train_batch_size=batch_size,
     learning_rate=learning_rate,
     evaluation_strategy='epoch' if val_dataset else 'no',
     # ... (Other parameters) ...
   )

   # --- Chunk 2.3: Initialize Trainer ---
   trainer = Trainer(
     model=self.model,
     args=training_args,
     train_dataset=train_dataset,
     eval_dataset=val_dataset,
     # ... (Other parameters) ...
   )


   # --- Chunk 2.4: Run Fine-Tuning ---
   trainer.train()
   trainer.save_model(self.output_dir)
```

Code Explanation:

- This code defines a standard fine-tuning loop. The `TrainingArguments` (Chunk 2.2) define the parameters (e.g., `learning_rate=2e-5`), and the `Trainer` (Chunk 2.3) handles the entire re-training process, saving the new, specialized model to `self.output_dir`.

Chunk 3: Aspect-Based Sentiment

Flow: Finally, the `finbert_tuner.py` module defines a function for **Aspect-Based Sentiment**. Instead of one score for an entire article, this flow allows us to ask more specific questions. For example, `get_aspect_sentiment(text, 'profit')` will scan the text *only* for profit-related keywords (e.g., 'earnings', 'revenue') and calculate a sentiment score for just that aspect, ignoring other topics. This provides a much more granular analysis.

Section VII. Conclusion and Future Work

Chunk 1: Key Achievements

This project successfully achieved its primary objective: to transition the stock prediction system from a price-based **regression** model to a direction-based **classification** model.

The key finding, shown in the `task7_advanced_evaluation.py` summary report, is that **sentiment features provide a significant and measurable lift in predictive performance.**

- **Baseline (Technical-Only):** The best technical-only model (Gradient Boosting) achieved an F1 Score of **65.7%**.
- **Sentiment-Enhanced:** The best sentiment-only model (also Gradient Boosting) achieved an F1 Score of **68.8%**.

This represents a **+4.8% relative improvement in F1 Score**, validating the hypothesis that news sentiment captures market psychology not reflected in historical price data alone. The final 21-model experiment provides a robust, fast (14-second runtime), and fully automated pipeline for end-to-end sentiment-based trading analysis.

Chunk 2: Future Work

Based on the project's findings, two clear paths for future work emerge:

1. **Integrate Independent Research:** The most logical next step is to replace the simple `TextBlob` model (from Section III) with the fine-tuned `FinBERT` model (from Section VI). This would combine the project's best-performing algorithm (Gradient Boosting) with its most advanced sentiment analysis tool, likely pushing performance even higher.
2. **Explore Sequential Models:** The current models (XGBoost, SVM) are "static"—they do not consider the *sequence* of past days. A promising direction would be to re-introduce the LSTM/GRU architectures (from Task C.4/C.5) but adapt them for this project's *classification* task, allowing the model to learn from temporal patterns in both price and sentiment.