Configuration Manual

MSc Research Project

Evaluating the performance of cryptocurrency trading signal providers on social media platforms.

Mohib Ur Rehman

Student ID: x23256541

School of Computing

National College of Ireland

Supervisor: Furqan Rustom

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Mohib Ur Rehman		
Student ID:	x23256541		
Programme:	Master in Data Analytics	**Year**	2024-2025
Module:	Research Project		
Lecturer:	Furqan Rustom		
Submission Due Date:	12 December 2024		
Project Title:	Evaluating the performance of cryptocurrency trading signal providers on social media platforms		
Word Count:	1268 **Page Count:** 24		

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Mohib Rehman

Date: 12 December 2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	Yes
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	yes
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	yes

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	

Date:	
Penalty Applied (if applicable):	

Configuration Manual

Mohib Ur Rehman
Student ID: x23256541

1 Introduction

This configuration manual walks through the different stages of code development for the evaluation of trading signal providers research project. These stages include:

- Data Collection
- Data Filtration
- Data Preprocessing
- Feature Engineering
- Model Implementation

2 System Configuration

2.1 Hardware Specifications

Figure 1 shows the hardware configuration which was used for this research project. The project uses a 10th Gen Intel(R) Core™ i5-10310U @ 1.70 GHz processor and 32 GB (31.8 GB usable) installed DDR4 RAM. The type of system used is a 64-bit operating system.

Figure 1, Hardware Specification

2.1 Software Specifications and libraries

Visual Studio Code is used as the Integrated Development Environment. Python 3.9 is used as the programming language for the implementation of this project.
The libraries used and their version is shared in figure 2.

```
ccxt==4.4.2
certif6==2024.8.3
cffi==1.10.
charset-normalize ==3.4.0
contourp ==1.3.0
cryptograph ==40.0.
cycle ==0.12. 1
factor-analyze ==0.5.1
fonttool ==4.54.
frozenlis ==1.5.0
idna==3.10
importlib-resource ==6.4.5
jobli ==1.4.2
kiwisolve ==1.4.7
matplotli ==3.9.2
multidic ==6.1.0
numpy==2.0.2
packagin ==24.1
panda ==2.2.3
pillo ==11.0.
propcach0==0.2.0
pycare ==4.4.0
pycparse ==2.22
pyparsin ==3.2.0
python-dateuti ==2.9.0.post
pytz==2024.      0
request ==2.32.
scikit-lear ==1.5.2
scipy==1.13.
seabor ==0.13.
six==1.162
threadpoolct ==3.5.0
typing-extension ==4.12.
tzdat ==2024.      2
urllib ==2.2.3
yarl==1.17.
zipp==3.20.
       2
```

Figure 2, requirements.txt

Data Collection

This research utilizes the SNscrape Python library to collect text messages from Telegram. For each trading signal provider, 3,000 messages are scraped up to the current point in time. The implementation is detailed in the Scraper.py file in figure 3, which employs the snstelegram module from the SNscrape library. Data from five Telegram signal providers is gathered using their respective Telegram URLs and stored in the file located at ./data/telegram_channel_data.csv.

```python
import snscrape.modules.telegram as snstelegram
from typing import List, Dict
import csv
import datetime

You, 8 seconds ago | 1 author (You)
class TelegramScraper:
    def __init__(self):
        pass

    def scrape_telegram_channels(self, channels: List[str], limit: int = 100) -> Dict[str, List[Dict]]:
        scraped_data = {}

        for channel_url in channels:
            channel_name = channel_url.split('/')[-1]
            scraped_data[channel_name] = []

            scraper = snstelegram.TelegramChannelScraper(channel_name)

            for i, item in enumerate(scraper.get_items()):
                if i >= limit:
                    break

                post_data = {
                    'date': item.date,
                    'content': item.content,
                    'url': item.url,
                    'outlinks': item.outlinks,
                }

                if hasattr(item, 'replyTo'):
                    post_data['reply_to'] = item.replyTo
                if hasattr(item, 'viewCount'):
                    post_data['view_count'] = item.viewCount

                scraped_data[channel_name].append(post_data)

        return scraped_data
```

```python
    def create_csv_from_scraped_data(self, scraped_data: Dict[str, List[Dict]], output_file: str):
        with open(output_file, 'w', newline='', encoding='utf-8') as csvfile:
            (variable) writer: DictWriter[str] me', 'Content', 'Status']
                                               ldnames=fieldnames)
            See Real World Examples From GitHub
            writer.writeheader()

            for channel, posts in scraped_data.items():
                for post in posts:
                    date = post['date'].strftime('%Y-%m-%d')
                    time = post['date'].strftime('%H:%M:%S')
                    writer.writerow({
                        'Channel': channel,
                        'Date': date,
                        'Time': time,        You, 2 months ago • Updating NER model
                        'Content': post['content'],
                        'Status': ''
                    })

    def run(self, channels: List[str], output_file: str, limit: int = 3000):
        scraped_results = self.scrape_telegram_channels(channels, limit)
        self.create_csv_from_scraped_data(scraped_results, output_file)
        print(f"CSV file '{output_file}' has been created successfully.")

if __name__ == "__main__":
    channels_to_scrape = [
        "https://t.me/s/wallstreetqueenofficial",
        "https://t.me/CryptoSignals_Orge",
        "https://t.me/BinanceKillersVipChannel",
        "https://t.me/cryptoclubpumpsignal",
        "https://t.me/WolfxCrypto_VIP"
    ]
    output_file = './data/telegram_channel_data2.csv'
    limit = 3000

    scraper = TelegramScraper()
    scraper.run(channels_to_scrape, output_file, limit)
```

Figure 3, Telegram Scraper

Data Filtration

In the first phase of data filtration, carried out in the classifier_train.py script, raw data collected from Telegram is filtered to exclude messages that do not contain trading signals. This process begins by labeling the data using the is_trading_signal function in figure 4. The labeled data is then saved as ./data/telegram_channel_data.csv.

Next, preprocessing is performed using the pre_process_text function to prepare the data for training. Once preprocessing is complete, the refined data is passed to the train_model function in figure 5, where a machine learning algorithm is trained on the labeled dataset. This training process leverages custom feature extraction methods provided by the Feature_extractor class in figure 6.

Finally, the trained model's weights are saved in the model directory using the save_model function, enabling its application to new datasets in subsequent phases.

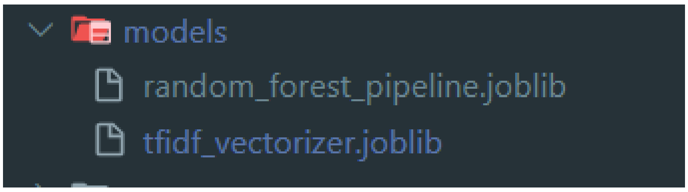

figure 3.1, Model file directory

```python
import os
import re
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, accuracy_score
from sklearn.pipeline import Pipeline
from sklearn.base import BaseEstimator, TransformerMixin
import joblib
from scipy.sparse import csr_matrix, hstack
from sklearn.model_selection import cross_val_score
from sklearn.metrics import precision_recall_fscore_support
import seaborn as sns
import matplotlib.pyplot as plt
You, 12 seconds ago | 1 author (You)
class TradingSignalClassifier:
    def __init__(self, data_path='./data/telegram_channel_data.csv', model_path='./models/random_forest_pipeline2.joblib'):
        self.data_path = data_path
        self.model_path = model_path
        self.pipeline = None

    @staticmethod
    def preprocess_text(text):
        text = str(text).lower()
        return re.sub(r'[^a-zA-Z\s]', '', text)

    @staticmethod
    def is_trading_signal(text):
        text = str(text).lower()
        if any(phrase in text for phrase in ['long', 'short', 'buy', 'sell']):
            if any(phrase in text for phrase in ['entry','enter','entries','buy','buying']):
                if any(phrase in text for phrase in ['target','targets','take-profit','tp','tp1']):
                    if any(phrase in text for phrase in ['stop','stoploss','sl','stop-loss']):
                        return True
        return False

    def load_and_prepare_data(self):
        df = pd.read_csv(self.data_path)
        df['processed_content'] = df['Content'].apply(self.preprocess_text)
        df['is_signal'] = df['Content'].apply(self.is_trading_signal)
        return df['processed_content'], df['is_signal']
```

Figure 4, Trading signal classifier overview

```python
def train_model(self):
    X, y = self.load_and_prepare_data()
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    self.pipeline = Pipeline([
        ('tfidf', TfidfVectorizer(max_features=1000, stop_words='english', ngram_range=(1, 2))),
        ('features', FeatureExtractor()),
        ('clf', RandomForestClassifier(n_estimators=200, random_state=42, class_weight='balanced'))
    ])

    cv_scores = cross_val_score(self.pipeline, X_train, y_train, cv=5)
    print(f"Cross-validation scores: {cv_scores}")
    print(f"Mean CV score: {cv_scores.mean():.3f} (+/- {cv_scores.std() * 2:.3f})")

    self.pipeline.fit(X_train, y_train)

    y_pred = self.pipeline.predict(X_test)

    precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred, average='binary')

    print("\nRandom Forest Classifier Results:")
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.3f}")
    print(f"Precision: {precision:.3f}")
    print(f"Recall: {recall:.3f}")
    print(f"F1-score: {f1:.3f}")
    print("\nClassification Report:")
    print(classification_report(y_test, y_pred))

def save_model(self):
    if self.pipeline is None:
        raise ValueError("Model has not been trained yet.")

    model_dir = os.path.dirname(self.model_path)
    os.makedirs(model_dir, exist_ok=True)

    joblib.dump(self.pipeline, self.model_path)
    print(f"Model pipeline saved as '{self.model_path}'")

    vectorizer_path = os.path.join(model_dir, 'tfidf_vectorizer.joblib')
    joblib.dump(self.pipeline.named_steps['tfidf'], vectorizer_path)
    print(f"TF-IDF vectorizer saved as '{vectorizer_path}'")
```

Figure 5, Random forest classifier training

```python
class FeatureExtractor(BaseEstimator, TransformerMixin):
    """Custom feature extractor for the trading signal classifier."""
    def __init__(self):
        self.keywords = ['long', 'short', 'buy', 'sell', 'entry', 'target', 'stop', 'sl', 'leverage', 'lev', 'profit', 'tp', 'tp1',
        'take-profit', 'stoploss', 'stop-loss', 'enter','enteries', 'take-profits']

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_dense = X.toarray()
        features = []
        for text in X_dense:
            text_str = ' '.join(map(str, text))
            feature_dict = {
                'keyword_count': sum(keyword in text_str.lower() for keyword in self.keywords),
                'has_price': bool(re.search(r'\$?\d+(\.\d+)?', text_str)),
                'has_percentage': '%' in text_str,
                'has_trading_pair': bool(re.search(r'\w+/\w+', text_str)),
                'has_entry': any(phrase in text_str.lower() for phrase in ['entry','enter','entries','buy','buying']),
                'has_target': any(phrase in text_str.lower() for phrase in ['target','targets','take-profit','tp','tp1']),
                'has_stop_loss': any(phrase in text_str.lower() for phrase in ['stop','stoploss','sl','stop-loss']),
                'has_leverage': any(phrase in text_str.lower() for phrase in ['leverage','lev','cross','x']),
            }
            features.append(list(feature_dict.values()))

        feature_matrix = csr_matrix(features)
        return hstack([X, feature_matrix])
```

Figure 6, Custom Feature Extractor

```python
def main():
    classifier = TradingSignalClassifier()
    classifier.train_model()
    classifier.save_model()

    example_messages = [
        "COIN NAME: #BTC/USDTLONG SET-UP Leverage: 5-10x Entry: 60000 - 59000$ Targets: 61000 - 62000 - 63000$ Stop-loss:
        58000$",
        "Bitcoin hits new all-time high!",
        "COIN NAME: #EOS/USDTLONG SET-UPTwo Targets done nicely within one day😲Target 1: 0.795$✅Target 2: 0.815$✅80% Profits
        booked with 10x Lev🤑Two Targets destroyed perfectly within one day🔥Whether the market goes up or down, our VIP members
        book profits in every direction. So don't miss your Profits and Join our premium and making continuous profits💯.
        Message me now @wallstreetqueenadminCheers to all VIP members🍾🍾VIP ONLY (WALLSTREET QUEEN OFFICIAL)",
        "Coin: #ETH/USDT Short Set-Up Entry: 2500$ Targets: 2400 - 2300 - 2200$ Stop-loss: 2600$",
        "#Market Sentiment💸Bitcoin Fear and Greed Index is 63 - Greed",
        "BREAKING: The #Bitcoin Hash Rate has just hit a new ATH!"
    ]

    print("\nTesting the model:")
    for message in example_messages:
        print(f"\nMessage: {message}")
        prediction = classifier.predict(message)
        print(f"Random Forest Prediction: {prediction}")


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

Figure 6.1, Main Function for trading signal classifier

Implementing the saved model on the actual data:

The code presented in Figure 7, corresponds to the `tradingsignalclassifier.py` Python file. It begins with the same `Feature_extractor` class utilized in the previous module for custom feature extraction. The trained model is then loaded using the `load_model` function, as shown in Figure 8.

Subsequently, predictions are made on the input CSV file, adding a new boolean column to each row. In this column, a value of `0` indicates a non trading signal, while `1` denotes the presence of a trading signal.

Finally, all rows containing trading signals are filtered using the `create_filtered_csv` function, also depicted in Figure 9. The resulting filtered data is saved as a new CSV file named `filtered_telegram_channel_data.csv` in the same `data` folder.

```python
import pandas as pd
import joblib
import os
import re
from sklearn.base import BaseEstimator, TransformerMixin
from scipy.sparse import csr_matrix, hstack

You, 2 weeks ago | 1 author (You)
class FeatureExtractor(BaseEstimator, TransformerMixin):
    def __init__(self):
        self.keywords = ['long', 'short', 'buy', 'sell', 'entry', 'target', 'stop', 'sl', 'leverage', 'lev', 'profit', 'tp', 'tp1', 'take-profit', 'stoploss', 'stop-loss', 'enter','enteries',
        'take-profits']

    def fit(self, X, y=None):
        return self

    def transform(self, X):
        X_dense = X.toarray()

        features = []
        for text in X_dense:
            text_str = ' '.join(map(str, text))
            feature_dict = {
                'keyword_count': sum(keyword in text_str.lower() for keyword in self.keywords),
                'has_price': bool(re.search(r'\$?\d+(\.\d+)?', text_str)),
                'has_percentage': '%' in text_str,
                'has_trading_pair': bool(re.search(r'\w+/\w+', text_str)),
                'has_entry': any(phrase in text_str.lower() for phrase in ['entry','enter','entries','buy','buying']),      You, 2 weeks ago • Uncommitted changes
                'has_target': any(phrase in text_str.lower() for phrase in ['target','targets','take-profit','tp','tp1']),
                'has_stop_loss': any(phrase in text_str.lower() for phrase in ['stop','stoploss','sl','stop-loss']),
                'has_leverage': any(phrase in text_str.lower() for phrase in ['leverage','lev','cross','x']),
            }
            features.append(list(feature_dict.values()))

        feature_matrix = csr_matrix(features)

        return hstack([X, feature_matrix])
```

Figure 7, Custom Feature Extractor

```python
def load_model(self):
    pipeline_path = os.path.join(self.model_dir, 'random_forest_pipeline2.joblib')
    vectorizer_path = os.path.join(self.model_dir, 'tfidf_vectorizer.joblib')

    if not os.path.exists(pipeline_path) or not os.path.exists(vectorizer_path):
        raise FileNotFoundError(f"Model files not found in '{self.model_dir}'")

    self.pipeline = joblib.load(pipeline_path)
    self.vectorizer = joblib.load(vectorizer_path)
    print("Model pipeline and vectorizer loaded successfully.")

def classify_signals(self, csv_file):
    if self.pipeline is None or self.vectorizer is None:
        raise ValueError("Model or vectorizer not loaded. Call load_model() first.")
    df = pd.read_csv(csv_file)
    df['processed_content'] = df['Content'].apply(self.preprocess_text)
    predictions = self.pipeline.predict(df['processed_content'])
    df['prediction'] = predictions

    return df

def preprocess_text(self, text):
    text = str(text).lower()
    return re.sub(r'[^a-zA-Z\s]', '', text)

def predict(self, message):
    if self.pipeline is None:
        raise ValueError("Model not loaded. Call load_model() first.")
    processed_message = self.preprocess_text(message)
    prediction = self.pipeline.predict([processed_message])
    return "Trading signal" if prediction[0] else "Not a trading signal"
def create_filtered_csv(self, input_csv, output_csv):
    results = self.classify_signals(input_csv)
    filtered_results = results[results['prediction'] == True]
    filtered_results = filtered_results.drop(columns=['processed_content','prediction'])
    filtered_results.to_csv(output_csv, index=False)
    print(f"Filtered results saved to {output_csv}")
```

Figure 8, Model Implementation and csv generation

```python
if __name__ == "__main__":
    classifier = TradingSignalClassifier()
    results = classifier.classify_signals('./data/telegram_channel_data2.csv')
    print("Classification results:")
    print(results[['Content', 'prediction']])
    print(results['prediction'].value_counts())
    filtered_csv_path = './data/filtered_telegram_channel_data.csv'
    classifier.create_filtered_csv('./data/telegram_channel_data.csv', filtered_csv_path)
    print(f"Created filtered CSV {filtered_csv_path}")
```

Figure 9, Model prediction and saving prediction

Data Preprocessing

The Python file `extractionPipeline.py` contains three classes dedicated to data preprocessing using regular expressions.

1) CryptoSignalFormatter

This class addresses formatting issues in text messages that arise after removing Unicode special symbols and characters.

- The `split_camel_case` function (Figure 10) splits text where words are joined together in camel case format.
- The `format_take_profit_targets` function (Figure 11) standardizes all variations of how profit targets are written into a consistent format.
- The `format_signal` function (Figure N) organizes key entities such as Coin Name, Targets, Stop Loss, and others, creating logical separations between them.
- Finally, the `final_cleanup` function (Figure N) removes unnecessary characters and eliminates double spaces in the text for a cleaner output.

```python
class CryptoSignalFormatter:
    def __init__(self):
        self.patterns = {
            'numbers': re.compile(r'\d+\.\d+'),
            'percentages': re.compile(r'\d+(?:-\d+)?%'),
        }

    def split_camel_case(self, text):
        """Split text on capital letters while preserving acronyms"""
        if text.isupper():
            return text

        parts = []
        current_word = ""

        for i, char in enumerate(text):
            if char.isupper() and i > 0:
                if current_word:
                    parts.append(current_word)
                current_word = char
            else:
                current_word += char

        if current_word:        You, 2 months ago • Updated Pipeline
            parts.append(current_word)

        return ' '.join(parts)

    def format_take_profit_targets(self, text):
        """Format Take-Profit Targets section with proper number handling"""
        numbers = re.findall(r'\d+\.\d+', text)
        if not numbers:
            return text

        formatted_items = []
        for i, number in enumerate(numbers, 1):
            formatted_items.append(f"{i}) {number}")

        return " ".join(formatted_items)
```

Figure 10, camelcase Split function

```python
def format_signal(self, text):
    """Main formatting function that preserves all symbols"""
    # Split text into sections while preserving important delimiters
    sections = re.split(r'((?:Exchanges|Signal Type|Leverage|Entry Targets|Take-Profit Targets|Stop Targets):)', text)

    formatted_parts = []
    for i, section in enumerate(sections):
        if not section:
            continue
        if section.endswith(':'):
            if i > 0:
                formatted_parts.append(' ')
            formatted_parts.append(section)
            continue

        # Process content
        content = section.strip()
        prev_header = sections[i-1] if i > 0 and sections[i-1].endswith(':') else ""

        # Special section handling
        if prev_header == "Take-Profit Targets:":
            content = self.format_take_profit_targets(content)
        elif prev_header == "Stop Targets:":
            content = re.sub(r'(\d+)-(\d+)%', r'\1-\2%', content)
        else:
            content = re.sub(r'(\d+\.?\d*)([A-Za-z])', r'\1 \2', content)
            words = re.findall(r'[A-Z][a-z]+|[A-Z]{2,}(?=[A-Z][a-z]|\d|\W|$)|[A-Z]{2,}|[a-z]+|\d+\.?\d*|[^\w\s]', content)
            content = ' '.join(words)

        formatted_parts.append(content)

    # Join all parts and cleanup
    result = ' '.join(part for part in formatted_parts if part)
    result = self._final_cleanup(result)

    return result
```

Figure 11, Formatting Targets

```python
def _final_cleanup(self, text):
    """Cleanup while preserving all symbols"""
    text = re.sub(r'\s+', ' ', text)
    text = re.sub(r'\s*:\s*', ': ', text)
    text = re.sub(r'\s*\(\s*', ' (', text)
    text = re.sub(r'\s*\)\s*', ') ', text)
    text = re.sub(r'#\s+', '# ', text)
    text = re.sub(r'([A-Z]+)\s*/\s*([A-Z]+)', r'\1/\2', text)
    text = re.sub(r'(\d+)\s*-\s*(\d+)\s*%', r'\1-\2%', text)
    text = re.sub(r'(\d+)\s+\.\s*(\d+)', r'\1.\2', text)
    text = re.sub(r'\s*x\s*', 'x', text)

    return text.strip()
```

Figure 12, Cleaning up text

2) SignalParser

The `SignalParser` class, depicted in Figure 13, implements two utility functions: `find_numbers_after_keyword` and `find_numbers_before_keyword`. These functions address the challenge of variations in how different trading signal providers write targets.

The solution involves identifying specific keywords in the text, such as "targets," and extracting the numbers that follow or precede these keywords. However, this approach may also capture unrelated numbers, such as target indexes or stop-loss values. These functions are designed to help parse and differentiate relevant numerical data from unrelated figures within the text as shown in figure 14.

```python
class SignalParser:
    def __init__(self):
        self.pair_suffix = "USDT"

    def find_coin(self, text: str) -> Tuple[str, str]:
        """Find coin name by looking for USDT pattern"""
        cleaned_text = text
        pairs = re.findall(r'#?\s*([A-Z0-9\s]+)\s*/\s*USDT', text)
        pairs = [pair.replace(" ", "") for pair in pairs]
        if pairs:
            return pairs[0], self.pair_suffix
        return "", self.pair_suffix

    def find_numbers_after_keyword(self, text: str, keyword: str, max_numbers: int = 2) -> List[float]:
        """Find numbers that appear after a keyword, looking forward only."""
        numbers = []
        match = re.search(rf'{keyword}', text, re.IGNORECASE)
        if match:
            text_after = text[match.end():]
            number_matches = re.finditer(r'(\d+\.?\d*)', text_after)
            for i, num_match in enumerate(number_matches):
                if i >= max_numbers:
                    break
                try:
                    num = float(num_match.group(1))
                    if 0 < num < 1000000:
                        numbers.append(num)
                except ValueError:
                    continue

                end_pos = num_match.end()
                if end_pos < len(text_after):
                    next_char = text_after[end_pos]
                    if next_char not in '.-' and not next_char.isspace():
                        break

        return numbers
```

Figure 13, Signal parser utility functions

```python
def find_numbers_before_keyword(self, text: str, keyword: str, max_numbers: int = 2) -> List[float]:
    """Find numbers that appear before a keyword, looking backward only."""
    numbers = []
    match = re.search(rf'{keyword}', text, re.IGNORECASE)
    if match:
        text_before = text[:match.start()+1]
        number_matches = re.finditer(r'(\d+\.?\d*)', text_before)
        for i, num_match in enumerate(number_matches):
            if i >= max_numbers:
                break
            try:
                num = float(num_match.group(1))
                if 0 < num < 1000000:
                    numbers.append(num)
            except ValueError:
                continue

            end_pos = num_match.end()
            if end_pos < len(text_before):
                next_char = text_before[end_pos]
                if next_char not in '.-' and not next_char.isspace():
                    break

    return numbers

def handle_stop_loss(self, text: str) -> bool:
    """Handle stop loss"""
    # Look for stop loss after the keyword "STOP" or "SL"
    stop_loss_match = re.search(r'(?:STOP|Stop|stop|SL|sl)', text, re.IGNORECASE)
    if stop_loss_match:
        # Look at text after the stop loss keyword
        text_after_stop = text[stop_loss_match.end():stop_loss_match.end()+30]
        if '%' in text_after_stop:
            return True
    return False

def find_all_targets(self, text: str) -> List[float]:
    """Find all valid numbers that appear after the first occurrence of 'target'"""
    targets = []
    match = re.search(r'target', text, re.IGNORECASE)
    if match:
        text_after = text[match.end():]
        numbers = re.findall(r'(\d+\.?\d*)', text_after)
        for num in numbers:
            try:
                num_float = float(num)
                if 0 < num_float < 1000000:
                    targets.append(num_float)
            except ValueError:
                continue

    return sorted(list(set(targets)))
```

Figure 14, Finding all targets

3) SignalValidator

The `SignalValidator` class in figure 15, is responsible for logically verifying and cleaning the data processed by the previous class. It applies several validation checks to ensure the integrity of the trading signal data:

- Stop Loss Validation: Ensures that the stop-loss value is less than the first target.
- Entry Point Validation: Verifies that the entry point is also less than the first target.
- clean_stoploss Function (Figure 16): Identifies the actual stop-loss value, even if it is mentioned as a percentage in some messages, and converts it into a usable format.
- clean_targets Function (Figure 17): Ensures that the target values are in increasing order and removes any outliers in the target list that fall outside the standard deviation of the array.

This class helps refine the processed data, ensuring consistency and accuracy before further analysis.

```python
class SignalValidator:
    def __init__(self):
        self.max_leverage = 125  # Maximum allowed leverage
        self.min_targets = 3     # Minimum number of targets
        self.max_targets = 10     # Maximum number of targets to consider

    def clean_entry_prices(self, entry: List[float]) -> List[float]:
        """Clean and validate entry prices"""
        # Remove any entries that are clearly wrong (like 1.0 that sometimes appears)
        cleaned = [price for price in entry if price > 1.0]
        # Sort entries
        cleaned.sort()
        # If we have no valid entries, return the original first entry
        return cleaned if cleaned else [entry[0]]

    def clean_leverage(self, leverage: List[float]) -> float:
        """Clean and validate leverage values"""
        # Filter out obviously wrong values (like signal IDs or entry prices)
        valid_leverage = [x for x in leverage if 1 <= x <= self.max_leverage]

        if not valid_leverage:
            # If no valid leverage found, return a default value
            return 1.0

        # Take the first valid leverage value
        return valid_leverage[0]

    def count_decimal_places(self, number):
        return len(str(number).split('.')[1]) if '.' in str(number) else 0
```

Figure 15, Signal Validator

```python
def clean_stop_loss(self, stop_loss: List[float], entry: List[float], direction: str, stop_loss_percentage: bool) -> float:
    """Clean and validate stop loss"""
    # Remove percentage values (like 5.0, 10.0)
    if stop_loss_percentage:
        if direction == "LONG":
            if len(stop_loss) == 1:
                stop_loss_value = [entry[0] * (1 -  (stop_loss[0] / 10000))]
            else:
                stop_loss_value = [entry[0] * (1 -  (stop_loss[0] / 10000)),entry[0] * (1 -  (stop_loss[1] / 10000))]
        else:
            if len(stop_loss) == 1:
                stop_loss_value = [entry[0] * (1 + (stop_loss[0] / 10000))]
            else:
                stop_loss_value = [entry[0] * (1 + (stop_loss[0] / 10000)),entry[0] * (1 +  (stop_loss[1] / 10000))]

        return statistics.mean([float(format(x, f".{self.count_decimal_places(entry[0])}f")) for x in stop_loss_value])

    if not stop_loss:
        # If no valid stop loss, calculate a default one based on direction
        avg_entry = statistics.mean(entry)
        if direction == "LONG":
            return avg_entry * 0.95  # 5% below entry for longs
        else:
            return avg_entry * 1.05  # 5% above entry for shorts

    # For longs, stop should be below entry
    # For shorts, stop should be above entry
    avg_entry = statistics.mean(entry)
    valid_stops = [stop for stop in stop_loss if
                   (direction == "LONG" and stop < avg_entry) or
                   (direction == "SHORT" and stop > avg_entry)]

    return valid_stops[0] if valid_stops else stop_loss[0]
```

figure 16, clean stoploss function

```python
def clean_targets(self, targets: List[float], entry: List[float], direction: str) -> List[float]:
    if not targets or not entry:
        return []

    avg_entry = statistics.mean(entry)

    # Step 1: Initial analysis of the target list
    target_stats = {
        'mean': statistics.mean(targets),
        'std': statistics.stdev(targets) if len(targets) > 1 else 0,
        'min': min(targets),
        'max': max(targets)
    }
     Ctrl+L to chat, Ctrl+K to generate


    # Step 2: Remove likely indices
    valid_targets = [
        t for t in targets
        if not self.is_likely_index(t, targets, target_stats)
    ]

    if not valid_targets:
        return self._generate_default_targets(avg_entry, direction)

    # Step 3: Apply direction-based filtering
    if direction == "LONG":
        valid_targets = [t for t in valid_targets if t > avg_entry]
    else:
        valid_targets = [t for t in valid_targets if t < avg_entry]

    # Step 4: Sort targets
    valid_targets.sort(reverse=(direction == "SHORT"))

    # Step 5: Limit number of targets
    valid_targets = valid_targets[:self.max_targets]

    # If no valid targets remain after filtering, generate defaults
    if not valid_targets:
        return self._generate_default_targets(avg_entry, direction)

    return valid_targets
```

Figure 17, clean targets function

All these classes collectively form a pipeline-based approach, with each class serving a distinct purpose. The `SignalPipeline` class, depicted in Figure 18, integrates these components in a specific sequence to ensure efficient and orderly data processing.

The `main` function in Figure 19 demonstrates the implementation of this pipeline. It processes data from the `filtered_telegram_channel_data.csv` file, applies the pipeline steps, and outputs the refined results to a file named `./data/extracted_signals.csv`. This structured workflow ensures the accurate extraction and validation of trading signals.

```python
class SignalPipeline:
    """Combined pipeline that processes crypto signals"""

    def __init__(self):
        self.formatter = CryptoSignalFormatter()
        self.parser = SignalParser()
        self.validator = SignalValidator()

    def process_signal(self, raw_text: str) -> SignalInfo:
        """Process a crypto signal through the complete pipeline"""
        try:
            # Step 1: Format the text
            formatted_text = self.formatter.format_signal(raw_text)

            # Step 2: Parse the formatted text
            parsed_info = self.parser.parse_signal(formatted_text)

            # Step 3: Validate the signal
            validated_signal = self.validator.validate_signal(parsed_info)

            # Create final result with both parsed info and formatted text
            return SignalInfo(
                coin=validated_signal.coin,
                pair=validated_signal.pair,
                entry=validated_signal.entry,
                leverage=validated_signal.leverage,
                stop_loss=validated_signal.stop_loss,
                direction=validated_signal.direction,
                stop_loss_percentage=validated_signal.stop_loss_percentage,
                targets=validated_signal.targets,
                formatted_text=formatted_text
            )
        except Exception as e:
            print(f"Error processing signal: {e}")
            return None
```

Figure 18, Signal pipeline architechture

```python
if __name__ == "__main__":
    pipeline = SignalPipeline()
    df = pd.read_csv('./data/filtered_telegram_channel_data2.csv')
    df2 = pd.DataFrame()
    df2['Channel'] = df['Channel']
    df2['Date'] = df['Date']
    df2['Time'] = df['Time']
    df2['Content'] = df['Content']
    df2['Coin'] = None
    df2['Pair'] = None
    df2['Entry'] = None
    df2['Direction'] = None
    df2['Leverage'] = None
    df2['Stop Loss'] = None
    df2['Targets'] = None
    for index, row in df.iterrows():
        result = pipeline.process_signal(row['Content'])
        if result:
            df2.loc[index, 'Coin'] = result.coin
            df2.loc[index, 'Pair'] = result.pair
            df2.loc[index, 'Entry'] = str(result.entry)
            df2.loc[index, 'Direction'] = result.direction
            df2.loc[index, 'Leverage'] = result.leverage
            df2.loc[index, 'Stop Loss'] = result.stop_loss
            df2.loc[index, 'Targets'] = str(result.targets)

    df2.to_csv('./data/extracted_signals.csv', index=False)
```

Figure 19, main function

BackTesting

The backtesting module, implemented in feature_engineering.py, gathers historical price data for each cryptocurrency mentioned in the text messages as shown in Figure 20. Using this data, several key factors are utilized to create feature-engineered columns, as illustrated in Figure 21. These columns provide insights and metrics essential for analyzing trading performance. The resulting analysis is then saved in a file named `./data/trading_analysis_results.csv`, enabling further evaluation and refinement of the trading strategies.

```python
You, 1 second ago | 1 author (You)
import ast
import pandas as pd
import numpy as np
import ccxt
import time
from datetime import datetime, timedelta
from scipy import stats
You, 1 second ago | 1 author (You)
class Backtesting:
    def __init__(self, exchange_id='binance'):
        """Initialize exchange connection"""
        self.exchange = getattr(ccxt, exchange_id)()

    def fetch_ohlcv_data(self, symbol, start_date, timeframe='4h', days=20):
        """
        Fetch historical OHLCV data from exchange
        """
        try:
            # Convert timestamp to milliseconds
            start_ts = int(datetime.strptime(start_date, '%Y-%m-%d').timestamp() * 1000)

            # Fetch OHLCV data
            ohlcv = self.exchange.fetch_ohlcv(
                symbol=symbol,
                timeframe=timeframe,
                since=start_ts,
                limit=6*days  # Maximum candles
            )

            # Convert to DataFrame
            df = pd.DataFrame(ohlcv, columns=['timestamp', 'open', 'high', 'low', 'close', 'volume'])
            df['timestamp'] = pd.to_datetime(df['timestamp'], unit='ms')
            return df

        except Exception as e:
            print(f"Error fetching data for {symbol}: {str(e)}")
            return None
```

figure 20, OHLCV Historical data collection

```python
def analyze_signal(self, signal_row, price_data):
    """
    Analyze a single trading signal using price data
    """
    metrics = {}

    entry_price = np.mean(signal_row['Entry'])
    targets = signal_row['Targets']
    stoploss = signal_row['Stop Loss']
    direction = signal_row['Direction']

    channel = signal_row['Channel']
    leverage = signal_row['Leverage']


    # Filter price data after signal
    signal_time = pd.to_datetime(signal_row['timestamp'])
    price_data = price_data[price_data['timestamp'] >= signal_time].copy()
    if len(price_data) == 0:
        return None
     # Entry check
    signal_price = price_data.iloc[0]['open']

    if signal_row['Direction'] == 'LONG':
        if(signal_price >= max(signal_row['Entry'])):
            metrics['entry_check'] = True
            metrics['targets_discard'] = [t for t in targets if t <= signal_price]
        else:
            metrics['entry_check'] = False
            metrics['targets_discard'] = []
    else:
        if(signal_price <= min(signal_row['Entry'])):
            metrics['entry_check'] = True
            metrics['targets_discard'] = [t for t in targets if t >= signal_price]
        else:
            metrics['entry_check'] = False
            metrics['targets_discard'] = []
    actual_targets = [t for t in targets if t not in metrics['targets_discard']]
    # Calculate metrics
    if direction == 'LONG':
        prices_reached = []
        target_hits = []
        hit_durations = []
        stoploss_hit = False
        stoploss_duration = None

        for idx, row in price_data.iterrows():
            # Check stoploss
            if row['low'] <= stoploss and not stoploss_hit:
                stoploss_hit = True
                stoploss_duration = round((row['timestamp'] - signal_time).total_seconds() / 3600)
                price_data = price_data.iloc[:idx+1]
                break
```

Figure 21, Feature Engineering

```python
            # Check targets
            for target in actual_targets:
                if row['high'] >= target and target not in target_hits:
                    target_hits.append(target)
                    hit_durations.append(round((row['timestamp'] - signal_time).total_seconds() / 3600))
            if(len(target_hits) == len(targets)):
                price_data = price_data.iloc[:idx+1]
                break
        max_price = price_data['high'].max()
        max_profit = ((max_price - entry_price) / entry_price) * 100
        min_price = price_data['low'].min()
        max_loss = ((min_price - entry_price) / entry_price) * 100

    else:  # SHORT
        prices_reached = []
        target_hits = []
        hit_durations = []
        stoploss_hit = False
        stoploss_duration = None

        for idx, row in price_data.iterrows():
            # Check stoploss
            if row['high'] >= stoploss and not stoploss_hit:
                stoploss_hit = True
                stoploss_duration = round((row['timestamp'] - signal_time).total_seconds() / 3600)
                price_data = price_data.iloc[:idx+1]
                break

            # Check targets
            for target in actual_targets:
                if row['low'] <= target and target not in target_hits:
                    target_hits.append(target)
                    hit_durations.append(round((row['timestamp'] - signal_time).total_seconds() / 3600))
            if(len(target_hits) == len(targets)):
                price_data = price_data.iloc[:idx+1]
                break
        min_price = price_data['low'].min()
        max_profit = ((entry_price - min_price) / entry_price) * 100
        max_price = price_data['high'].max()
        max_loss = ((max_price - entry_price) / entry_price) * 100

    indicators = self.calculate_indicators(signal_row['Coin'],price_data)
    # Calculate metrics
    metrics['percentage_hit'] = (len(target_hits) / len(targets)) * 100
    metrics['stoploss_hit'] = stoploss_hit
    metrics['stoploss_duration'] = stoploss_duration
    metrics['max_profit'] = max_profit
    metrics['max_loss'] = max_loss
    metrics['target_hit_duration'] = self.rank_durations(hit_durations)
    metrics['ma_trend_strength'] = indicators if signal_row['Direction']=='LONG' else -indicators
```

Figure 21, Feature Engineering

```python
# Target distribution
target_distances = [(actual_targets[i+1] - actual_targets[i])/actual_targets[i] * 100
                    for i in range(len(actual_targets)-1)]
metrics['targets_distribution'] = np.std(target_distances) if target_distances else 0

# Risk/Reward ratios
risk = abs(entry_price - stoploss)
metrics['average_risk_reward_ratio_per_target'] = np.mean([(abs(t - entry_price)/risk) for t in targets])

# Calculate returns for Sharpe ratio
returns = price_data['close'].pct_change().dropna()
if len(returns) > 0:
    risk_free_rate = 0.02  # Assuming 2% annual risk-free rate
    periods_per_year = 252 # Trading days in a year
    avg_return = returns.mean() * periods_per_year
    volatility = np.std(returns,ddof=1) * np.sqrt(periods_per_year)
    sharpe_ratio = (avg_return-risk_free_rate)/volatility
    metrics['sharpe_ratio'] = sharpe_ratio
else:
    metrics['sharpe_ratio'] = None

# Max drawdown
metrics['max_drawdown'] = (max_price - min_price)/max_price


# Trade duration
metrics['trade_duration'] = round((price_data.iloc[-1]['timestamp'] -
                            signal_time).total_seconds() / 3600)

metrics['Channel'] = channel
metrics['Leverage'] = leverage

return metrics
```

Figure 21, Feature Engineering

```python
def main():
    # Read signals
    signals_df = pd.read_csv('./data/extracted_signals2.csv')
    signals_df1 = signals_df[signals_df['Channel']=='CryptoSignals_Orge'].dropna()

    signals_df2 = signals_df[signals_df['Channel']=='BinanceKillersVipChannel'].dropna()

    signals_df3 = signals_df[signals_df['Channel']=='wallstreetqueenofficial'].dropna()

    signals_df4 = signals_df[signals_df['Channel']=='cryptoclubpumpsignal'].dropna()

    signals_df5 = signals_df[signals_df['Channel']=='WolfxCrypto_VIP'].dropna()

    signals_df = pd.concat([signals_df1,signals_df2,signals_df3,signals_df4,signals_df5])

    signals_df['Entry'] = signals_df['Entry'].apply(ast.literal_eval)
    signals_df['Targets'] = signals_df['Targets'].apply(ast.literal_eval)
    # Drop rows where Targets array is contains less than 3 targets
    signals_df = signals_df[signals_df['Targets'].apply(len) > 3]
    signals_df['timestamp'] = pd.to_datetime(signals_df['Date'] + ' ' + signals_df['Time'])


    analyzer = Backtesting()
    results_df = analyzer.analyze_signals(signals_df)
    results_df = results_df.dropna()
    results_df.to_csv('trading_analysis_results.csv', index=False)
    print("Analysis complete. Results saved to trading_analysis_results.csv")

    return results_df

if __name__ == "__main__":
    main()        You, 4 weeks ago • Finalized Framework
```

Figure 22, Saving results in csv

Model Implementation

The final phase of this implementation builds upon the feature-engineered data produced by the backtesting module. This code in featureengineering.py as shown in figure 23, involves applying clustering algorithms to categorize trading signals into one of three different groups:

a) "Bad" trading signals.
b) "Average" trading signals.
c) "Good" trading signals.

The clustering process utilizes the engineered features, which may include metrics such as price movements, volatility, profit-to-risk ratios, and other key performance indicators derived during the backtesting stage. These features are used to group the trading signals based on their performance patterns and characteristics.

By predicting these clustered values, the implementation provides a straightforward and interpretable classification of the trading signals. This categorization helps users or systems quickly identify the quality of signals, facilitating better decision-making in trading strategies. The results of this clustering phase can be further analyzed to refine signal generation or inform future trading methodologies.

```
You, 22 hours ago | 1 author (You)
import numpy as np
import pandas as pd
from sklearn.metrics import silhouette_score
from sklearn.mixture import GaussianMixture
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.decomposition import PCA
from sklearn.cluster import DBSCAN, AgglomerativeClustering, KMeans, SpectralClustering
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
import seaborn as sns


def weighted_analysis(df):
    weights = {
        'Percentage_Hit': 2.5,
        'Stoploss_Hit': 1.5,
        'Max_Profit': 1,
        'Max_Loss': 1.0,
        'Targets_Distribution': 1.0,
        'Sharpe_Ratio': 1.0,
        'Trade_Duration': 1.0,
    }

    features = list(weights.keys())
    X = df[features].copy()

    # Standardize features
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X)

    # Apply weights
    weight_matrix = np.array([weights[feature] for feature in features])
    X_weighted = X_scaled * weight_matrix

    # KMeans clustering
    df_result = df.copy()

    kmeans = KMeans(n_clusters=3, random_state=42, n_init='auto')
    df_result['kmeans_cluster'] = kmeans.fit_predict(X_weighted)

    # Spectral clustering
    df_result['spectral_cluster'] = SpectralClustering(
        n_clusters=3,
        random_state=42,
        affinity='nearest_neighbors', #'nearest_neighbors',
        gamma=0.02 #0.01-0.1
    ).fit_predict(X_weighted)
```

Figure 23, Model Implementation

```python
# Silhouette scores
df_result['kmeans_silhouette_score'] = silhouette_score(X_weighted, df_result['kmeans_cluster'])
df_result['spectral_silhouette_score'] = silhouette_score(X_weighted, df_result['spectral_cluster'])
df_result['gmm_silhouette_score'] = silhouette_score(X_weighted, df_result['gmm_cluster'])

# Compute performance metrics
performance_metrics_kmeans = df_result.groupby('kmeans_cluster').agg({
    'Percentage_Hit': 'mean'
})
performance_metrics_spectral = df_result.groupby('spectral_cluster').agg({
    'Percentage_Hit': 'mean'
})
performance_metrics_gmm = df_result.groupby('gmm_cluster').agg({
    'Percentage_Hit': 'mean'
})

composite_scores_kmeans = performance_metrics_kmeans['Percentage_Hit']
composite_scores_spectral = performance_metrics_spectral['Percentage_Hit']
composite_scores_gmm = performance_metrics_gmm['Percentage_Hit']

df_result['kmeans_signal_type'] = df_result['kmeans_cluster'].map(lambda x: 'Good' if x == composite_scores_kmeans.idxmax() else 'Bad' if x == composite_scores_kmeans.idxmin() else 'Average')
df_result['spectral_signal_type'] = df_result['spectral_cluster'].map(lambda x: 'Good' if x == composite_scores_spectral.idxmax() else 'Bad' if x == composite_scores_spectral.idxmin() else 'Average')
df_result['gmm_signal_type'] = df_result['gmm_cluster'].map(lambda x: 'Good' if x == composite_scores_gmm.idxmax() else 'Bad' if x == composite_scores_gmm.idxmin() else 'Average')

return df_result
```

figure 24, Cluster Named Mapping