Mutual Fund Style Classification

Muya Liu, Xuan Liu

1 Introduction

Mutual funds are a cornerstone of modern portfolio management, offering diversified investment strategies tailored to varying risk and return profiles. Accurately identifying and classifying a fund’s investment style—such as equity, fixed income, or balanced—is essential for investors, advisors, and regulatory bodies.

In this project, we propose a hybrid classification pipeline that leverages both AI models and a custom Skip-gram-based deep learning framework. Our approach not only automates the classification of mutual fund summaries but also integrates interpretability by grounding predictions in semantic evidence extracted from fund documentation.

2 Data preprocessing

2.1 Data sources

This project uses the following two data sources for the fund investment strategy classification task:

- MutualFundSummary/XXX.txt

Each text file in this folder contains two parts, separated by <head_breaker>: the first part is the fund name, and the second part is the fund summary information. The summary information is used as the input feature of the OpenAI model.

- MutualFundLabels.csv

This csv file contains the real investment strategy label corresponding to each fund, which serves as the supervision signal (label) of the classification model. We use it as the ground truth of the fund type for the evaluation of the results after model training.

2.2 Label processing

The original label distribution is as follows:

Table 1: Investment Strategies distribution

Investment Strategy	count	Investment Strategy	count
Equity Long Only (Low Risk)	248	Long Short Funds (High Risk)	4
Fixed Income Long Only (Low Risk)	130	Commodities Fund (Low Risk)	1
Balanced Fund (Low Risk)	84		

In order to focus on the mainstream categories, we removed the categories with very small sample sizes, namely Long Short Funds (High Risk) and Commodities Fund (Low Risk) during the preprocessing stage. Finally we retained the following three categories and encoded using numeric labels.

3 RAG algorithm with OpenAI

3.1 Langchain initialization

The following table shows the OpenAI tools we need to use in our process.

Table 2: Overview of the functions needed

Function / Class	Source Module	Purpose
DirectoryLoader	langchain.document_loaders	Loads text files from a directory and returns them as Document objects.
RecursiveCharacter TextSplitter	langchain.text_splitter	Recursively splits long text into meaningful chunks.
OpenAIEmbeddings	langchain_openai	Converts text into embeddings for similarity search.
ChatOpenAI	langchain_openai	Creates a ChatGPT-like model interface for Q&A.
Chroma	langchain_chroma	Creates a database for storing and retrieving embeddings.
RetrievalQA	langchain.chains	Builds a retrieval-augmented QA pipeline.
PromptTemplate	langchain.prompts	Defines prompt formats for controlling model output.

We use OpenAI and RAG algorithm to predict investment strategy in the following steps:

(1) Use DirectoryLoader to read fund summary text from the MutualFundSummary/ directory

(2) Use RecursiveCharacterTextSplitter to split long summaries into small chunks.

(3) Use OpenAIEmbeddings to convert each text chunk into a vector.

(4) Use Chroma to store all vectors to support subsequent retrieval based on semantic similarity.

(5) Set up the prompt template & RAG questions, then ask openai to provide results.

3.2 RAG algorithm

We use the following prompt template for classification tasks:

```
Prompt Template:
Classify the fund type from context into 3 categories:
  1 - Equity Long Only (Low Risk)
  2 - Fixed Income Long Only (Low Risk)
  3 - Balanced Fund (Low Risk)
If you don't know the answer, just say that you don't know, don't try to
make up an answer, and report 0 as a category.
Respond ONLY with the numeric code and confidence score in this format:
<number>,<confidence score>

{context}
Question: {question}
Helpful Answer:
Question:
"Which numeric category (1/2/3) does {fund_name}! belong to?"
```

This prompt combines Langchain document retrieval and context analysis to help improve classification performance and control hallucination risks.

3.3 classification report

The figure and table shows the confusion matrix and classification report for prediction. We can see that OpenAI's classification effect is very good, with a precision of 0.9 for each category.

Figure 1: Confusion matrix

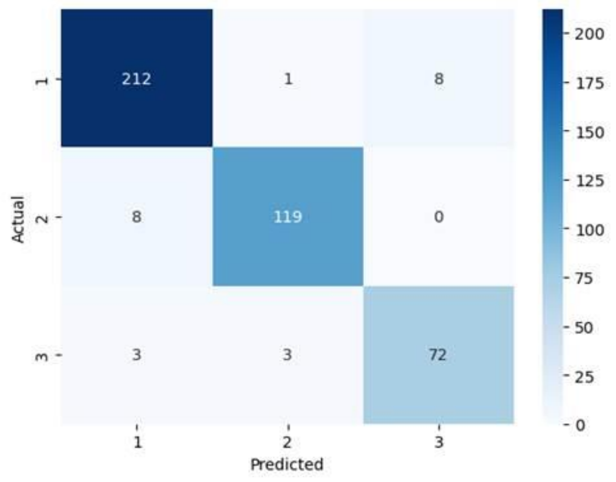


Table 3: Classification report

	precision	recall	f1-score
1	0.95	0.96	0.95
2	0.97	0.94	0.95
3	0.9	0.92	0.91
accuracy			0.95
macro avg	0.94	0.94	0.94
weighted avg	0.05	0.95	0.95

3.4 classification comments

We selected the 23 funds with inconsistent classifications and manually determined which category they should belong to based on the supporting evidence provided by the OpenAI model. We summarized 3 main possible reasons for the differences (see Appendix 7 - key points in supporting evidences.xlsx for all comparisons):

- provide A and B in supporting evidence
- benchmark index that measures the investment return of stocks in the United States
- The supporting evidence lists the specific shares in the fund. We need to check the webpage to determine which category it belongs to.

4 NLP application in skip-gram model

In this section, we use the classification results from RAG as the "ground truth" to train our skip-gram model and classified them into certain categories.

4.1 dataset split

To ensure robust model evaluation and effective hyperparameter tuning, we divide the entire dataset into three distinct subsets. First, we split the data into a training set and a test set using a 4:1 ratio. The test set is reserved for final model evaluation. Next, we further split the training set into a new training subset and a validation subset, also in a 4:1 ratio, to facilitate hyperparameter tuning.

It is important to note that, since the Skip-gram model requires the construction of word2vec embeddings based on the entire corpus, we apply text vectorization to the full dataset rather than any individual split. Therefore, we only retain and store the indices corresponding to each of the three subsets to guide subsequent data partitioning and model training.

4.2 skip-gram model and word embedding dictionary

In this section, we implement the Skip-gram model to generate word embeddings tailored to our mutual fund prospectus dataset. The process consists of several key modules:

1. Text Preprocessing and Tokenization: We define a tokenizer() function to clean the input text by converting it to lowercase, removing punctuation, stop words, and non-alphabetic characters and applied this function to all the supporting evidence.

2. Parameter Configuration: We need to define several parameters that will be used throughout the embedding learning process:

Training Parameters:

- batch_size — number of samples in each training batch
- num_epochs — number of complete passes through the training dataset

Word2Vec Parameters

- embedding_size — dimensionality of the word embedding vectors
- max_vocabulary_size — maximum number of unique words to keep in the vocabulary
- min_occurrence — minimum number of times a word must appear to be included in the vocabulary
- skip_window — window size on either side of the center word to consider as context
- num_skips — number of context words to sample for each center word

These parameters govern both the training process and the structure of the Skip-gram model and were selected to balance training efficiency and representation quality

3. Vocabulary Construction and one-hot encoding: We retained the most frequent words in the corpus and built a word-to-index dictionary. The reverse dictionary id2word is also created for interpretation. We also convert the corpus into a sequence of word IDs using the constructed vocabulary. To prepare the data for training, we implement to_one_hot() function to transforms each word ID into a one-hot encoded vector.

4. Skip-gram Batch Generation: A custom batch_generator() function is implemented to dynamically generate training samples. For each center word, context words within a defined window are randomly selected to form (input, output) pairs, which are then one-hot encoded and yielded as batches.

5. Autoencoder-based Embedding Learning: We design a batch_generator() function to generate training samples for the skip-gram model. For each center word, we randomly select context words within a defined window to form (input, output) word pairs. These pairs are then converted to one-hot vectors and returned as training batches.

6. Word Vector Extraction and Storage: After training, each word in the vocabulary is passed through the encoder to generate its embedding. These embeddings are stored in a word2vec dictionary and can be saved to disk for reuse in downstream tasks.

The procedure above helped us to create a dictionary which includes every words in the supporting evidence.

4.3 knowledge base and distance measure

In order to better categorize investment strategies, we created 3 lists of words which is related to each categories separately.

Table 4: Knowledge base of each categories

Investment Strategy	Knowledge Base
Equity Long Only (Low Risk)	['stock','stocks', 'equity','equities', 'growth', 'capital', 'shares','long-term', 'investment', 'dividend', 'market', 'value']
Fixed Income Long Only (Low Risk)	['bond', 'duration', 'yield', 'credit', 'interest','rate', 'fixed', 'income', 'debt', 'government', 'corporate','security','securities', 'high-yield', 'maturity', 'coupon', 'treasury', 'municipal', 'bonds']
Balanced Fund (Low Risk)	['allocation', 'diversified', 'equity', 'bond', 'mix', 'balanced']

We implement a function get_n_closer() that retrieves the most semantically similar words to a given keyword using cosine similarity over the trained word embeddings. Using this, we design a function create_knowledge_base() that takes a list of seed keywords and expands it by retrieving their top-N closest neighbors in the embedding space. This allows us to automatically enrich the domain vocabulary for each class by including words with similar semantics.

We then apply this function to each class-specific keyword list to create three comprehensive knowledge bases: one for Equity, one for Fixed Income, and one for Balanced funds. These knowledge bases are sets of words that serve as semantic prototypes for each investment style.

Finally, we print the contents of each knowledge base along with the top related words for manual inspection and verification. These curated word sets serve as the foundation for semantic similarity computation and feature extraction in the downstream classification task.

To extract fund-relevant content guided by the knowledge base, we design two approaches:

1. **Semantic Distance-Based Sentence Extraction:** We implement a function extract_sentence_distance() that computes a semantic score for each sentence in a fund summary by:

- Tokenizing the sentence and computing its barycenter vector using word embeddings.
- Measuring the cosine distance between the sentence vector and all words in the relevant knowledge base.
- Averaging the top-N smallest distances to assign a semantic score.
- Selecting the top-K most semantically similar sentences as features.

2. **Keyword Match-Based Sentence Extraction:** We also define a simpler function extract_sentence_match() that:

- Counts how many knowledge base words appear in each sentence.
- Selects the top-K sentences with the highest keyword match count.

4.4 data preprocessing

Before implementing deep learning models, we first perform a comprehensive feature preprocessing procedure to structure the input data appropriately:

1. Match supporting evidence with knowledge bases: For each fund summary in the training and validation sets, we compare the extracted supporting evidence with the three knowledge bases (Equity, Fixed Income, Balanced), resulting in the creation of three new columns: equity_match, fixed_income_match, and balanced_match.

2. Combine Sentence-Level Features: We concatenate the equity_match, fixed_income_match, and balanced_match sentence fields into a unified match_all column, forming the textual basis for model input.

3. Visualize Document Lengths: We analyze the distribution of token counts across documents in match_all to inform the selection of an appropriate maximum input length (maxlen).

4. Define Preprocessing Parameters: We set the vocabulary size to 2,500 (num_words = 2500), input length to 150 tokens (maxlen = 150), and word embedding dimension to 50 (word_dimension = 50).

5. Split Data and One-Hot Encode Labels: Using predefined indices, we divide the dataset into training and validation subsets, and one-hot encode the class labels for multi-class classification.

6. Tokenize Text: We fit a Keras Tokenizer on the training set and generate integer index sequences for the three categories of matched sentences.

7. Sequence Padding: Each token sequence is padded or truncated to the fixed maxlen value to ensure uniform input size.

8. Feature Construction: We concatenate the padded sequences from the three text categories (equity, fixed income, balanced) to form the final input feature matrix for each sample.

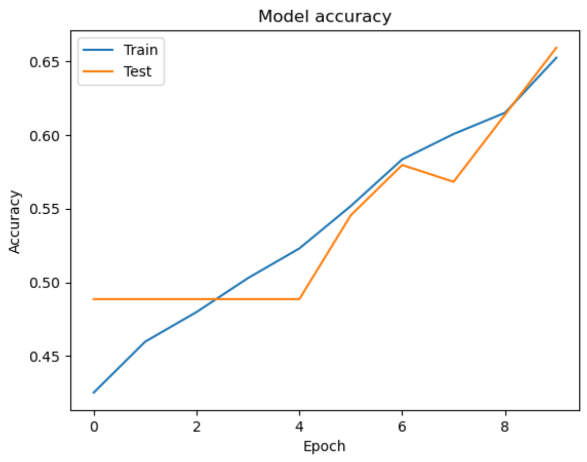
9. Build Embedding Matrix: We initialize an embedding matrix using the pre-trained word2vec vectors. For each word in the tokenizer's vocabulary, if a vector is available in the embedding dictionary, it is inserted into the matrix at the corresponding index.

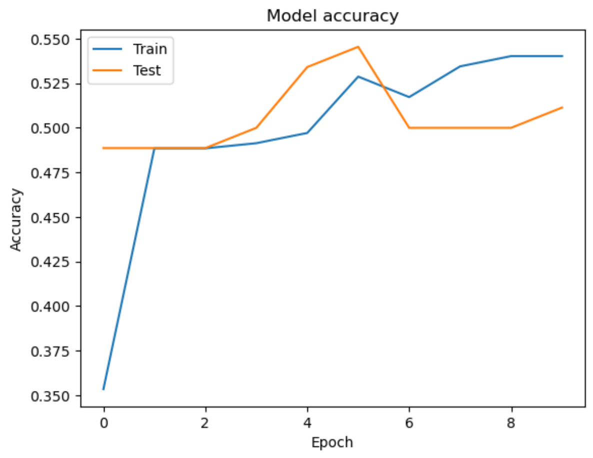
This preprocessing pipeline ensures that the input to classification models is numerical rather than textual, and that it is consistent, dense, and semantically informed.

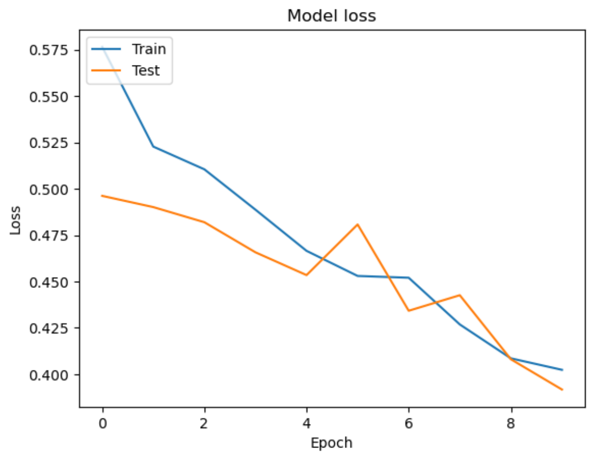
4.5 classification algorithm

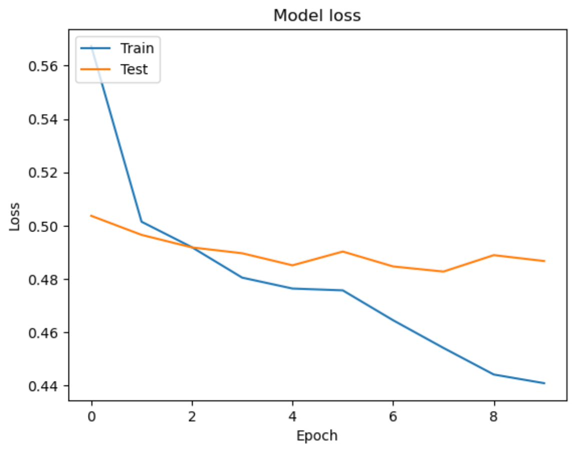
To classify fund strategies, we explore and compare both CNN and RNN architectures. After evaluating their performance on the validation set, we find that the CNN model outperforms the RNN model in terms of both accuracy and training efficiency. Based on this result, we select CNN as the primary deep learning model for downstream parameter tuning.

Figure 2: Comparison of CNN (left) and RNN (right) model









At the same time, we also experiment with a wide range of classical machine learning algorithms, including Logistic Regression, Random Forest, Support Vector Machine, XGBoost, and Gradient Boosting. Each model is trained and evaluated using the original input features. We select the models with higher accuracies into the next steps.

Table 5: accuracy and f1-score of each machine learning model

Model	Accuracy	F1 Score	Model	Accuracy	F1 Score
LogisticRegression	0.579545	0.484612	BaggingClassifier	0.659091	0.474324
RidgeClassifier	0.511364	0.426592	XGBClassifier	0.659091	0.47791
SGDClassifier	0.545455	0.417211	LGBMClassifier	0.670455	0.488095
RandomForestClassifier	0.647727	0.466655	CatBoostClassifier	0.647727	0.461988
GradientBoostingClassifier	0.647727	0.489935	MLPClassifier	0.522727	0.444216
AdaBoostClassifier	0.545455	0.322969	SVC	0.625	0.414024
ExtraTreesClassifier	0.659091	0.485345			

4.5 parameter tunning on validation

We use ParameterGrid to loop through each hyper parameter combination and applied the model to validation set to find the best parameter.

Table 6: Hyper Parameter of each model

MODEL	HYPER-PARAMETER
CNN	`'filters1': [64, 128] 'filters2': [32, 64]`
	`'dense_units': [64, 128] 'dropout_rate': [0.3, 0.5]`
	`'optimizer': ['adam', 'rmsprop'] 'batch_size': [32] 'epochs': [10]`
LIGHTGBM	`'n_estimators': [100, 200, 300,500] 'learning_rate': [0.001,0.01, 0.05, 0.1]`
	`'max_depth': [3, 5, 7, 9]`
XGBOOST	`'n_estimators': [100, 200, 300,500] 'learning_rate': [0.001,0.01, 0.05, 0.1]`
	`'max_depth': [3, 5, 7, 9]`
BAGGING	`'n_estimators': [100, 200, 300,500] 'max_samples': [0.5, 0.75, 1.0]`
	`'max_features': [0.5, 0.75, 1.0] 'bootstrap': [True, False]`

The final results of the parameter tunning are as follows:

Table 7: Parameter tunning results

Model	params	accuracy	f1_score
CNN	{'batch_size': 32, 'dense_units': 128, 'dropout_rate': 0.3, 'epochs': 10, 'filters1': 128, 'filters2': 32, 'optimizer': 'rmsprop'}	0.772727	0.567281
lightgbm	{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 300}	0.704545	0.541554
XGBoost	{'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 500}	0.715909	0.544563
bagging	{'bootstrap': True, 'max_features': 0.5, 'max_samples': 0.75, 'n_estimators': 100}	0.681818	0.508123

4.6 prediction on test data

We begin by applying the same preprocessing procedure in section4.4 to the whole Train dataset and test set, ensuring consistency in input features. This includes both the semantic distance and keyword match-based sentence extraction based on the knowledge base.

Next, we retrain each of the four selected models — the CNN model with optimal parameters, as well as the top-performing traditional classifiers — on the full training data (the combination of the original training and validation sets). The final results are summarized in the table below:

Table 8: model performance on test set

Model	Accuracy	F1 Score
CNN	0.624	0.492
XGBoost	0.56	0.416
Bagging	0.642	0.452
LightGBM	0.486	0.164

Reference

1 Mutual fund chat_4.ipynb

2 NLP_app_2024.ipynb