BOSTON UNIVERSITY

MUTUAL FUND STYLE CLASSIFICATION FROM PROSPECTUS

MF 815 Spring 2022

Abstract

Applying Natural Language Processing (NLP) to learn and make predictions about the investment style of a mutual fund.

Taiga Schwarz, Xuyu Cai, Caleb Qi

Section I. Executive Summary

Our goal is to apply Natural Language Processing (NLP) to mutual fund summary texts to be able to predict which investment style a fund uses. Having a robust and accurate way of doing this without having to manually read through each one will save valuable time, and thus money.

We have two available data sets on which we conduct our analysis:

1. A collection of 545 different mutual fund summaries.
2. The labels that correspond to each of the mutual fund summaries. The specific label that we will be looking at is the "Investment Strategy" label. The possible classes are the following: "Balanced Fund (Low Risk)", "Fixed Income Long Only (Low Risk)", "Equity Long Only (Low Risk)", "Commodities Fund (Low Risk)", and "(Long Short Funds (High Risk)".

The "Commodities Fund (Low Risk)" has only 1 sample, and so we simply drop this type of fund from our data set. This leaves us with a total of 4 different investment styles. The "Long Short" funds have only 4 samples, which makes it severely imbalanced compared to the others. We cover our data cleaning and feature extraction processing in further in the Methodology section.

Overall, we are quite successful in predicting the investment style given a mutual fund summary. We employ two different algorithms: one using Random Forest, and another using a combination of 4 Convolutional Neural Networks (one for each class) with a voting scheme. The latter algorithm performs slightly better on a test set, however the algorithm involving Random Forest is similarly robust and less complex. Moving forward in our paper, we will refer to the entire algorithm using Random Forest as the base model as the **RF Method**, and the entire algorithm using CNN as the base model as the **CNN Method**. Our full methodology will be covered thoroughly in the Section III Methodology section. Before that however, we will first directly show our final results.

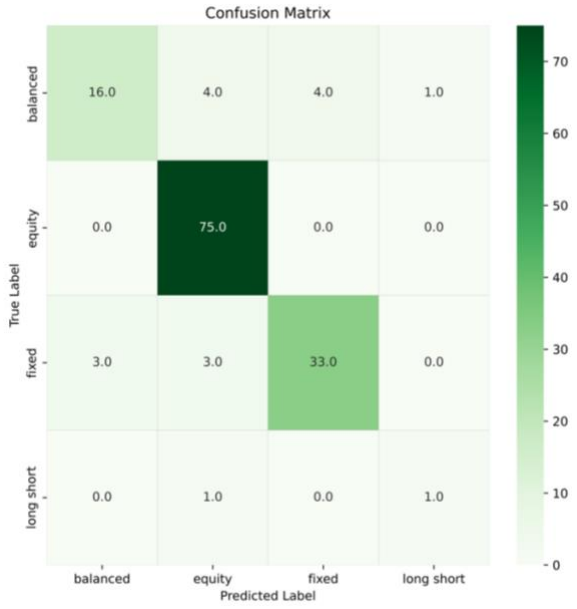
Section II. Final Results

Background:

Our out-of-sample test set is a result of doing a 70-30 stratified train-test split. This just means that we set 70% of our data as our training data and set aside 30% as our test set, where the "stratified" means that we make sure the distribution of fund types in the train and in the test are kept the same. Now, we go to our results.

CNN Method – Best Model:

The out-of-sample performance of our **best performing method**, the **CNN Method**, is below:



	precision	recall	f1-score	support
0	0.84	0.64	0.73	25
1	0.90	1.00	0.95	75
2	0.89	0.85	0.87	39
3	0.50	0.50	0.50	2
accuracy			0.89	141
macro avg	0.78	0.75	0.76	141
weighted avg	0.88	0.89	0.88	141

f1-score: 0.8812253253902085
overall accuracy: 0.8865248226950354

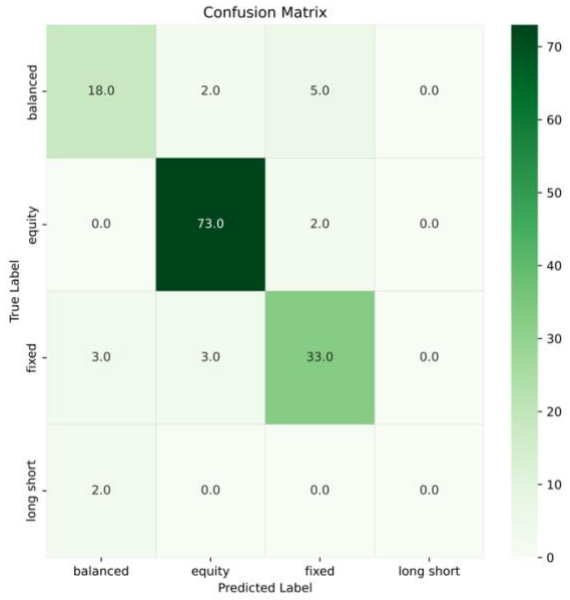
Key Points:

- The **overall accuracy** is **88.7%**, and the **weighted average of the F1-scores** is **88.1%**.
- The method clearly performs better the more data we have of each fund type to train on.
    - For example, the "Equity" type fund has the most samples in our overall data, and here we see that our method gives an F1-score of 95%, with 100% of the "Equity" class in the test set being correctly classified (*recall*) and 90% of the predictions being accurate (*precision*).

- The method correctly classifies 1 out of 2 of the “Long Short” type funds in the test set, despite the significant class imbalance.
    - We utilize an acceptance-threshold we set for the “Long Short” predictions to address the class imbalance issue. We will cover this in detail in Section III (Methodologies).

RF Method – 2nd Best Model:

The out-of-sample performance of our next best performing method, the **RF Method**, is below:



	precision	recall	f1-score	support
0	0.78	0.72	0.75	25
1	0.94	0.97	0.95	75
2	0.82	0.85	0.84	39
3	0.00	0.00	0.00	2
accuracy			0.88	141
macro avg	0.64	0.63	0.63	141
weighted avg	0.86	0.88	0.87	141

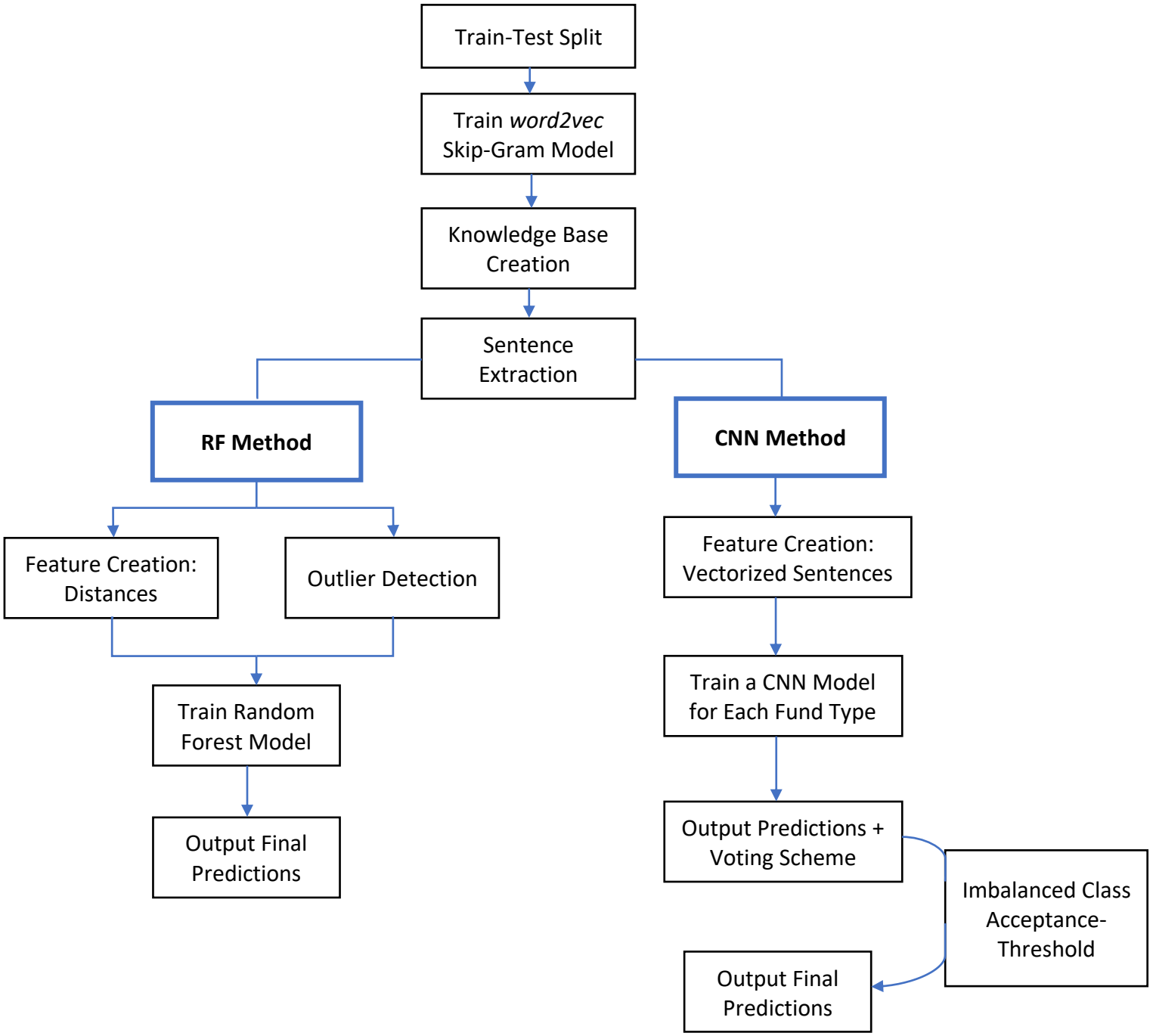
f1-score: 0.8716376307233551
overall accuracy: 0.8794326241134752

Key Points:

- The **overall accuracy** is **87.9%**, and the **weighted average of the F1-scores** is **87.2%** (not too different from the CNN Method).
- The method also performs better on classes with more data in our overall data set.
- Interestingly, the performance on the “Balanced” class is slightly better than in Method 2—but it does worse on the “Equity” class.
- None of the “Long Short” funds get classified correctly.
    - We employ outlier detection in the RF Method algorithm, and as we can see it is not very successful. We will cover this more in detail as well in Section III.

Section III. Methodology

The following diagram shows the steps in our methodology:



We now describe each section of our methodology in detail below.

Train-Test Split:

There is a clear class imbalance in the "Investment Strategy" class data (see Appendix, Section A). We have only 4 "Long Short Funds (High Risk)" and only 1 "Commodities Fund (Low Risk)". When we do our train-test split, we must make sure to include some "Long Short Funds (High Risk)" examples in the train set and the rest in the test set. For the "Commodities Fund (Low Risk)", we only have one example so we could drop it.

We do a 70-30 train-test split with stratified sampling using the Train Test Split method provided by the **sklearn** library. As mentioned earlier in this paper, the stratified sampling is so that we could preserve the proportions of the class sizes between the train and test set. This is important because we do not have a large sample size for any of the classes, and so when doing a train-test split we want to prevent the very real possibility of not having enough of one of the classes in the train set to train on. We first conduct the 70-30 train test with stratified sampling on the data without the "Long Short" funds, and then afterwards put two of the "Long Short" funds in the train and the other two in the test. The class distribution between the train and test are now proportional (see Appendix, Section B).

Train *word2vec* Skip-Gram Model:

The *word2vec* skip-gram model is a deep learning model that allows to create for each word a unique vector. These vectors are built such that two words that are used in the same context (context here is defined as the 3 to 5 words before and the 3 to 5 words after—a parameter we can tune) are close in terms of spatial distance. We train the *word2vec* skip-gram model on all of our fund summaries. To start with, we create a **Tokenizer** function that cleans and tokenizes the summaries. The tokenizer function removes the stop-words (unimportant words such as "a", "the", etc.) and the tokens with non-letter values, and then returns a list of the remaining tokens. One distinction we make is that we decide to keep hyphenated words such as "long-term", as we believe these words may hold important information.

We then process our model input. We start by building a dictionary that contains our vocabulary and the number of occurrences of each word. We also specify some parameters, such as the maximum vocabulary size, minimum occurrence, skip window, etc. (see the Appendix, Section C). Next, we create data that is simply the full tokenized corpus where we filter out the words that are not in our vocabulary and we replace the actual words by their unique ID, which in this case is the index of the word. Finally, we translate each word index to its one-hot representation. This final dictionary is called the **word embedding dictionary**.

After finishing the input processing, we create and compile the **Autoencoder** to train the skip-gram model. When the autoencoder has been trained, we can now use the matrix and bias of the encoder (the input + hidden layer) to build our vectorized word representations.

Knowledge Base Creation:

We now need a method to describe each "Investment Style" fund type so that we could connect the information in the summaries to these investment styles. One way of describing a fund type is through creating a **knowledge base** for each fund type, which we can now do using our word embedding dictionary. A knowledge base is simply an explicitly formulated set of words that describes the domain of the investment style. For our project, we create 4 knowledge bases, one corresponding to each type of fund.

We first define a small set of **key words** for each type of fund that closely represents the type of fund, and then choose the neighboring words to those key words to expand on the sets. When defining our key words, we choose the words that are *most important* to the fund type. We decide that the "most important" should be interpreted as the most commonly occurring tokens for each fund type. However, we do not want to naively choose the most commonly occurring words because some words such as "investments" are common to every fund type and thus will not add any distinctive information. We want the words to be more exclusive to the type of fund so that our knowledge bases end up being more unique to each type of fund.

Our algorithm of choosing the key words is as follows:

1. Find the top N most commonly occurring words within the fund summaries for each investment style and store each of these into a list.

2. Find the union of the set of distinct words between each pair of the lists. For example, for the unique “Fixed Income” words, we would find the distinct “Fixed Income” words between the “Fixed Income” and “Equity” list, then between “Fixed Income” and the “Balanced” list, and so on. Taking each of these sets of distinct “Fixed Income” words, we would then take their union to get our set of key words for “Fixed Income”.

Sentence Extraction + Feature Creation:

Using the knowledge bases, the objective is now to score each sentence according to their distance to the knowledge bases. We then extract the N number of sentences that are closest in distance to each knowledge base. We define a function that takes a fund summary, the knowledge base, and some hyper-parameters (see Appendix, Section D), and returns the N sentences that are closest to the knowledge base in terms of spatial distance and also the minimum distance between any of the sentences and the knowledge base. We compute the spatial distance by finding the cosine distance between the barycenter of the sentence and the barycenter of the knowledge base.

As a result, we have **two possible inputs for our modeling: (1) the minimum distance between the sentences and the knowledge base, (2) the vector representation of the most representative sentences**. This is the point at which our methodology splits between the **RF Method** and the **CNN Method**.

RF Method:

The RF Method utilizes the distances themselves as inputs (see the Appendix, Section E for an example). For a given fund summary, we have four different inputs, with each representing the summary’s distance to each knowledge base. The Random Forest model is trained using these distances to make its classifications.

RF Method – Outlier Detection:

In order to deal with the class imbalance issue caused by the “Long Short” funds, we compute the **Local Outlier Factor** for the “Long Short” distances. This method uses a nearest-

neighbors approach to identify the outliers. An additional feature is then added as an input, where the value is a 0 if it is not predicted to be an outlier and a 1 if it is.

When implemented, the method did not do well predicting outliers, even when trying various values of the hyperparameter for the number of nearest neighbors to use.

RF Method – Modeling:

First, we scale the input data using **StandardScaler** of the **sklearn** library. We then use a grid search cross-validation using **GridSearchCV** to optimize the Random Forest hyper-parameters (see the Appendix, Section F). Finally, we train our model using the preprocessed input data. The final results of the predictions are covered in Section II.

CNN Method:

The CNN Method utilizes the vector representation of the most representative sentences as model inputs. However, further data pre-processing needs to occur before we can use the data for our convolutional neural networks.

CNN Method – Embedding Matrix Creation:

We once again use our *word2vec* algorithm, as we would be able to preserve the contextual information of the words. We create an embedding matrix with the vector representations of the words for each fund type, which we would then pass each through a CNN model. First, we set the hyperparameters of the *word2vec* preprocessing: num_words = 2500, maxlen = 150, and word_dimension = 50 (see Appendix, Section G for descriptions). Next, we assign a unique index to each word in the vocabulary of the corpus and transform the texts to a list of word indices. We then use the **pad_sequence** function provided by **keras** to truncate or lengthen each text by adding zeros such that they all have the same length of 150. Finally, we create our embedding matrices, which are (2500, 50) arrays filled with the vector representations of the words in each of the 4 fund type vocabularies.

CNN Method – Modeling:

We train 4 different CNN models (one for each fund type), with each CNN model predicting whether or not the text belongs to that fund type. This means that we would also

have to split our 4 class labels into 4 separate binary labels. For example, for predicting whether or not a given text is “Fixed Income” or not, we would establish our labels to be 0 = “not Fixed Income” and 1 = “Fixed Income”. See the Appendix, Section G for further details of the CNN training.

CNN Method – Voting Scheme + Acceptance-Threshold for “Long-Short”:

Each of these models would then be used in a voting scheme, in which for a given fund, each model predicts a probability that the fund belongs to its own class, and we choose the model that gives the highest probability prediction. Now one thing we notice is that due to the lack of data, the "Long Short" predicted probabilities are small relative to the others and thus never get chosen. To fix this, we set a low acceptance-threshold for the "Long Short" class predicted probabilities such that if the "Long Short" probability is above some threshold, the vote would automatically go to predicting "Long Short" for that fund. See the Appendix, Section G for details on the threshold and a visual of the voting scheme.

After implementing this method, we see that it succeeds in making a correct "Long Short" prediction in the out-of-sample set, while also not negatively impacting the performance of the other predictions to any degree of significance.

Section IV. Contributions

First of all, we would like to thank Professor Hao Xing for his instruction, and for the hard work of the TA’s behind the scenes. Next, we list below our individual contributions to this project in terms of the coding.

Xuyu Cai conducted the train-test split methodology and the *word2vec* skip-gram model training. He also assisted with the Random Forest modeling.

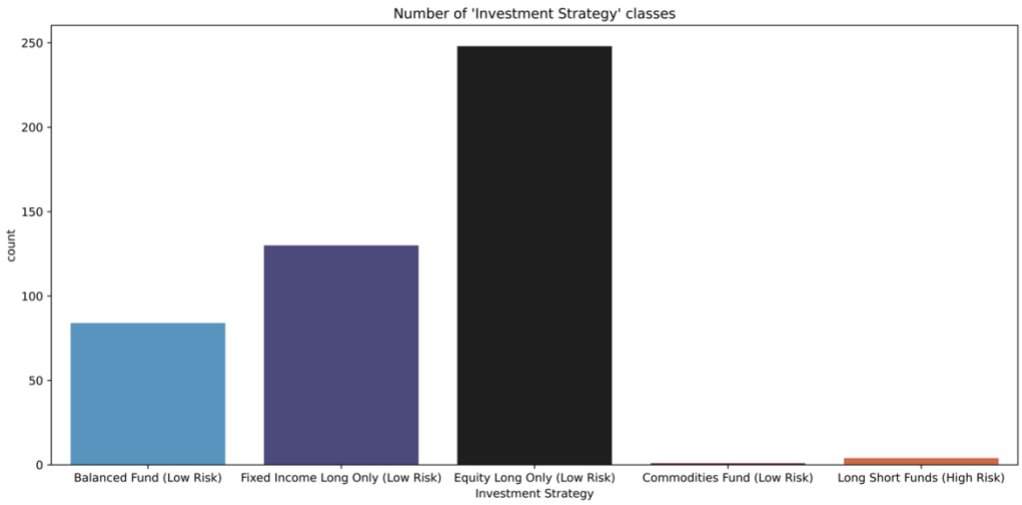
Caleb Qi established the key word logic and created the knowledge bases. He also assisted with the sentence extraction.

Taiga Schwarz conducted the feature creation and data preprocessing for both the RF Method and the CNN Method. He also worked on the RF modeling and also designed the CNN modeling + voting scheme algorithm.

Appendix

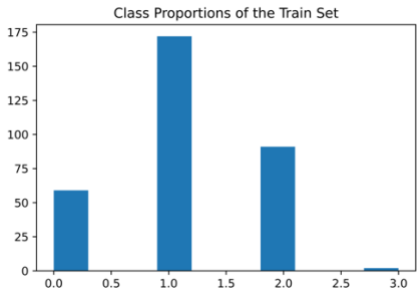
A. Data:

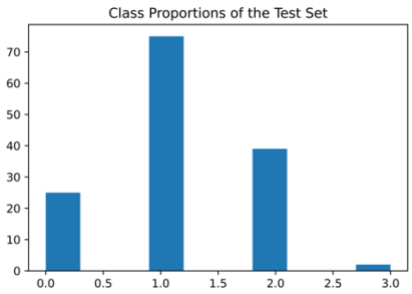
The distribution of the “Investment Strategy” classes is:



B. Train-Test Split:

The distribution of the “Investment Strategy” classes in the train and test set, respectively:





C. *word2vec* Skip-Gram Model:

The *word2vec* dictionary-creation parameters:

```
# Training Parameters
batch_size = 128 # The model will be trained batch per batch and one batch contains 128 rows
num_epochs = 2 # The model will go through all the data twice

# Word2Vec Parameters
embedding_size = 50 # Dimension of the embedding vector
max_vocabulary_size = 5000 # Total number of different words in the vocabulary
min_occurrence = 10 # Remove all words that do not appear at least 10 times
skip_window = 3 # How many words to consider left and right
num_skips = 4 # How many times to reuse an input to generate a label
```

D. Sentence Extraction

The sentence extraction function + hyperparameters is below:

```
# Takes a summary, the knowledge base and some hyper parameters and returns the "num_sent" sentences
# of the summary that are closer to the the knowledge base in term of spatial distances.
def extract_sentence_distance(summary, knowledge, n_closer, n_reject, num_sent):
    # Split the summary into sentences.
    sentences = sent_tokenize(summary)
    sentence_scores = []
    # Loop over the sentences.
    for j, sentence in enumerate(sentences):
        # we tokenize and clean the sentence
        tokens = tokenizer(sentence)

        sentence_barycenter = np.zeros(embedding_size)
        effective_len = 0
        # Compute the barycenter of the sentence
        for token in tokens :
            try :
                sentence_barycenter += np.array(word2vec[token])
                effective_len += 1
            except KeyError :
                pass
            except :
                raise

        # Reject sentences with less than n_reject words in our word2vec map
        if effective_len <= n_reject :
            sentence_scores.append(1)

        else :
            sentence_barycenter = sentence_barycenter/effective_len
            # Compute the distance sentence_barycenter -> words in our knowledge base
            barycenter_distance = [cosine(sentence_barycenter, word2vec[key_word]) for key_word in knowledge]
            barycenter_distance.sort()
            # Create the score of the sentence by averaging the "n_closer" smallest distances
            score = np.mean(barycenter_distance[:n_closer])
            sentence_scores.append(score)
    # Select the "num_sent" sentences that have the smallest score (smallest distance score with the knowledge base)
    sentence_scores, sentences = zip(*sorted(zip(sentence_scores, sentences)))
    top_sentences = sentences[:num_sent]
    return ' '.join(top_sentences), min(sentence_scores)
```

E. RF Method: Inputs

Raw input data:

	fixed_distance	balanced_distance	equity_distance	longshort_distance	outliers
0	0.186404	0.316491	0.336669	0.262801	0
1	0.228248	0.298263	0.318796	0.290218	0
2	0.204048	0.282583	0.342987	0.267531	1
3	0.204048	0.323756	0.331914	0.269578	1
4	0.213445	0.373294	0.326716	0.302242	0
...	...	...	...	...	...
319	0.206269	0.330636	0.330479	0.246755	0
320	0.240902	0.332864	0.317398	0.308867	0
321	0.191929	0.290371	0.328612	0.260897	0
322	0.209250	0.298263	0.326934	0.237544	0
323	0.252557	0.275764	0.328756	0.258655	0

After scaling using StandardScaler:

	fixed_distance	balanced_distance	equity_distance	longshort_distance	outliers
0	-1.100156	-0.120109	0.756362	-0.443301	0
1	-0.292666	-0.525140	-0.318844	0.484874	0
2	-0.759664	-0.873576	1.136450	-0.283154	1
3	-0.759664	0.041331	0.470328	-0.213847	1
4	-0.578324	1.142138	0.157586	0.891934	0
...	...	...	...	...	...
319	-0.716809	0.194208	0.384003	-0.986508	0
320	-0.048493	0.243717	-0.402930	1.116221	0
321	-0.993538	-0.700518	0.271681	-0.507761	0
322	-0.659294	-0.525140	0.170706	-1.298321	0
323	0.176430	-1.025106	0.280333	-0.583654	0

F. RF Method: Optimal Hyper-Parameters

Optimal hyper-parameters as determined by GridSearchCV:

```
{'criterion': 'entropy', 'max_depth': 10, 'max_features': 'auto', 'n_estimators': 50}
```

G.CNN Method:

The *word2vec* pre-processing hyper-parameters:

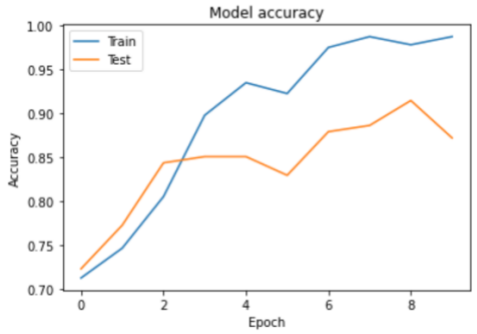
- num_words = The size of the vocabulary used. We only consider the 2500 most common words. The other words are removed from the texts.
- maxlen = The number of words considered for each document. We cut or lengthen the texts by adding 0's to the texts such that all have maxlen words.
- word_dimension = The dimension of our word vectors.

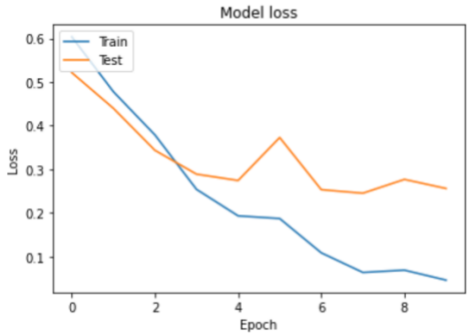
Convolutional Neural Network Training:

Fixed Income Model:

```
# Plot training & validation accuracy
plt.plot(CNN_history_fixed.history['accuracy'])
plt.plot(CNN_history_fixed.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plot training & validation loss values
plt.plot(CNN_history_fixed.history['loss'])
plt.plot(CNN_history_fixed.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

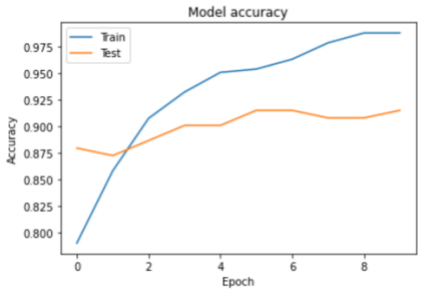


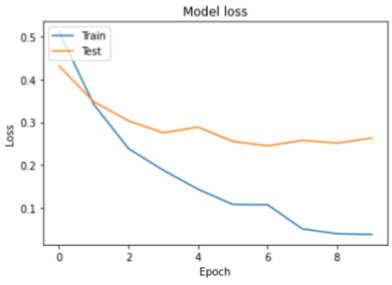


Balanced Fund Model:

```
# Plot training & validation accuracy
plt.plot(CNN_history_balanced.history['accuracy'])
plt.plot(CNN_history_balanced.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plot training & validation loss values
plt.plot(CNN_history_balanced.history['loss'])
plt.plot(CNN_history_balanced.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

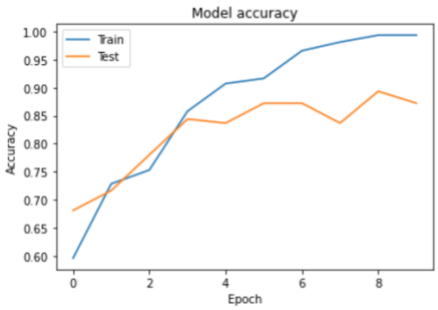


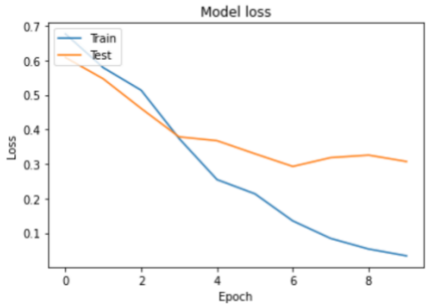


Equity Fund Model:

```
# Plot training & validation accuracy
plt.plot(CNN_history_equity.history['accuracy'])
plt.plot(CNN_history_equity.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plot training & validation loss values
plt.plot(CNN_history_equity.history['loss'])
plt.plot(CNN_history_equity.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

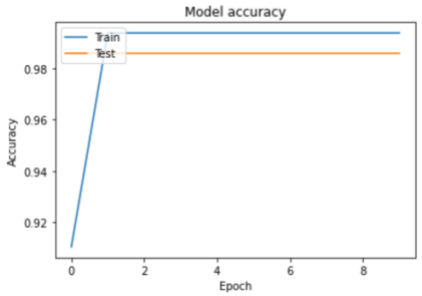


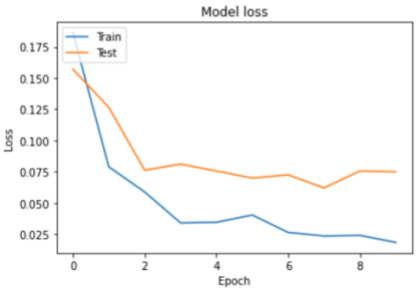


Long Short Fund Model:

```
# Plot training & validation accuracy
plt.plot(CNN_history_longshort.history['accuracy'])
plt.plot(CNN_history_longshort.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```

```
# Plot training & validation loss values
plt.plot(CNN_history_longshort.history['loss'])
plt.plot(CNN_history_longshort.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()
```





Voting Scheme + Acceptance Threshold:

The acceptance threshold we choose for choosing “Long Short” is > 1.3%. The code is below:

```
CNN_preds = pd.DataFrame()
CNN_preds['fixed_pred_prob'] = y_pred_CNN_fixed.flatten()
CNN_preds['balanced_pred_prob'] = y_pred_CNN_balanced.flatten()
CNN_preds['equity_pred_prob'] = y_pred_CNN_equity.flatten()
CNN_preds['longshort_pred_prob'] = y_pred_CNN_longshort.flatten()
N = CNN_preds.shape[0]
pred_class = np.zeros(N)
for i in range(N):
    if (max(CNN_preds.iloc[i,:]) == CNN_preds.iloc[i, 3]) or (CNN_preds.iloc[i, 3] > 0.013):
        pred_class[i] = 3  # longshort
    elif max(CNN_preds.iloc[i,:]) == CNN_preds.iloc[i, 0]:
        pred_class[i] = 2  # fixed income
    elif max(CNN_preds.iloc[i,:]) == CNN_preds.iloc[i, 1]:
        pred_class[i] = 0  # balanced
    elif max(CNN_preds.iloc[i,:]) == CNN_preds.iloc[i, 2]:
        pred_class[i] = 1  # equity

CNN_preds['pred_class'] = pred_class.astype(int)
```

The voting scheme can be visualized by the following dataframe:

	fixed_pred_prob	balanced_pred_prob	equity_pred_prob	longshort_pred_prob	pred_class
0	0.020143	0.040010	0.935864	0.000584	1
1	0.908441	0.128655	0.032022	0.001053	2
2	0.000229	0.000148	0.988057	0.000501	1
3	0.000059	0.006215	0.999722	0.000682	1
4	0.000645	0.000490	0.595848	0.000063	1
...	...	...	...	...	...
136	0.002134	0.001713	0.992294	0.000564	1
137	0.990072	0.022940	0.001719	0.000191	2
138	0.000251	0.002571	0.532466	0.001681	1
139	0.684197	0.039924	0.002039	0.014330	3
140	0.758382	0.074645	0.934612	0.002157	1

These are the predictions each CNN model makes on the test set. Each row is a fund summary in the test set. Each column except for the last contains the predicted probability made by each model type, for example the first column contains the predicted probability that the given fund is a “Fixed Income” type fund. The last column, “pred_class”, is the result of the voting scheme, with 0 = “Balanced”, 1 = “Equity”, 2 = “Fixed Income”, and 3 = “Long Short”.