Climate Risk Assessment: Practical Application of the BERT Model

1. Introduction

The impact of climate change on society, economy, and the environment has garnered significant attention, and accurately assessing the associated risks and positive effects is crucial. For instance, keywords such as “carbon tax,” “renewable energy,” and “flood” may have varying risk levels and positive values in different contexts. Traditional methods primarily rely on expert judgment or statistical analysis, making it difficult to efficiently process large-scale text data and capture complex semantic relationships.

In recent years, LLM and pre-trained language models (such as BERT) have shown excellent performance in text understanding and multi-label tasks. However, the application of NLP technologies in climate change research remains limited, particularly in assessing keyword risks and positive impacts.

This study uses the BERT model to assess the risk and positive scores of 25 climate change-related keywords and explores its potential value in policy-making, risk management, and public communication. The innovations of this study include: (1) applying BERT to this task; (2) designing a training method that simultaneously predicts risk and positive scores; (3) evaluating the model’s performance through error analysis and visualization.

The structure of this paper is as follows: Section 2 introduces data preparation and preprocessing, Section 3 describes model training and evaluation, Section 4 presents experimental results, Section 5 discusses model limitations, and Section 6 summarizes research findings and outlines future work.

1.1 Quantifying Climate Change

This article quantifies climate change-related terms through the dimensions of climate risk (risk score) and climate positivity (positive score) to better understand the role of different keywords in policy, risk management, and public perception.

A) Climate Risk Dimension

This dimension measures the potential negative impact or risk level of a term within the context of climate change.

- **High-risk terms (score 8-10)**: Terms related to natural disasters, extreme weather events, or policy uncertainty, such as “flood,” “hurricane,” and “policy uncertainty.” These risks may lead to significant socio-economic losses.

- **Moderate-risk terms (score 5-7)**: Terms related to emissions or regulations, such as “carbon emission” and “climate regulation.” These risks reflect the regulatory strength of policies and markets.
- **Low-risk terms (score 0-4)**: Terms related to positive actions or technologies, such as “renewable energy” and “emission reduction target.” These risks have minimal environmental impact and may even help reduce risks.

B) Climate Positivity Dimension

This dimension measures the positive contribution or proactivity of a term in addressing climate change.

- **High positivity terms (score 8-10)**: Terms related to emission reduction, renewable energy, or adaptation measures, such as “renewable energy,” “emission reduction target,” and “climate adaptation.” These terms represent the core of climate policies and sustainable development.
- **Moderate positivity terms (score 5-7)**: Terms related to technologies or policy tools, such as “low-carbon technology” and “carbon pricing.”
- **Low positivity terms (score 0-4)**: Terms related to risks or issues, such as “flood” and “hurricane.”

In the future, the scoring system can be adjusted in combination with expert opinions to improve the accuracy and applicability of the assessments.

In [1]:
```
risk_scores = {
    "flood": 10, "hurricane": 10, "drought": 10, "extreme heat": 9,
    "storm": 9, "policy uncertainty": 8, "carbon emission": 7,
    "greenhouse gas": 7, "emission intensity": 6, "carbon dioxide": 6,
    "climate regulation": 6, "carbon tax": 5, "emission trading": 5,
    "carbon pricing": 5,"emission reduction target": 4,
    "climate resilience": 3, "infrastructure upgrade": 3,
    "emergency plan": 3, "climate adaptation": 3, "adaptation plan": 3,
    "renewable energy": 2, "solar power": 2, "wind power": 2,
    "low-carbon technology": 2, "energy transition": 2
}

positive_scores = {
    "renewable energy": 10, "solar power": 10, "wind power": 10,
    "low-carbon technology": 9, "energy transition": 9,
    "emission reduction target": 9, "climate adaptation": 8,
    "adaptation plan": 8, "climate resilience": 8,
    "infrastructure upgrade": 8, "emergency plan": 8,
    "carbon tax": 7, "emission trading": 7, "carbon pricing": 7,
    "climate regulation": 6, "carbon emission": 4, "greenhouse gas": 4,
    "emission intensity": 3, "carbon dioxide": 3, "flood": 2, "hurricane"
    "drought": 2, "extreme heat": 2, "storm": 2, "policy uncertainty": 1
}
```

In [2]:
```
keywords = list(risk_scores.keys())
```

2. Data Preparation and Preprocessing

2.1 Creating the Dataset

First, a dataset is created that includes keywords, risk scores (risk_label), and positive scores (positive_label). The scores are then normalized to a range between 0 and 1 (by dividing by 10). The dataset is subsequently converted into a format compatible with the Hugging Face datasets library using Dataset.from_dict() for later model training or analysis.

In [3]:
```
from datasets import Dataset
import numpy as np

data = {
    "text": keywords,
    "risk_label": [risk_scores.get(kw, 0) / 10.0 for kw in keywords],
    "positive_label": [positive_scores.get(kw, 0) / 10.0 for kw in keywor
}
dataset = Dataset.from_dict(data)
```

2.2 Loading the Pre-trained Model and Tokenizer

Next, the pre-trained BERT model (bert-base-uncased) is loaded and configured for a multi-label classification task (num_labels=2, for predicting risk and positive scores). At the same time, the corresponding BERT tokenizer (AutoTokenizer) is loaded to convert the text into a format that the model can accept as input.

In [4]:
```
from transformers import AutoModelForSequenceClassification, AutoTokenize
model_name = "bert-base-uncased"
model = AutoModelForSequenceClassification.from_pretrained(model_name, nu
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
Some weights of BertForSequenceClassification were not initialized from th
e model checkpoint at bert-base-uncased and are newly initialized: ['class
ifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to u
se it for predictions and inference.
```

2.3 Data Preprocessing Function

Next, a preprocessing function, preprocess_function, is defined to encode the text data, process it into fixed-length inputs, and combine the risk and positive labels into a new label field before returning.

In [5]:
```
def preprocess_function(examples):
    encodings = tokenizer(examples["text"], truncation=True,
                          padding="max_length", max_length=128)
    encodings["labels"] = np.array(list(zip(examples["risk_label"],
                                            examples["positive_label"])),
                                   dtype=np.float32)
    return encodings
```

2.4 Encoding the Dataset

Next, the map method is used to preprocess the dataset in batches by calling the preprocess_function for each data entry. Then, the original text and label columns are removed. Finally, the dataset is split into 80% training set and 20% test set.

In [6]:
```
encoded_dataset = dataset.map(preprocess_function, batched=True)
```

```
Map:   0%|          | 0/25 [00:00<?, ? examples/s]
```

In [7]:
```
encoded_dataset = encoded_dataset.remove_columns(["text", "risk_label",
                                                  "positive_label"])
```

In [8]:
```
encoded_dataset = encoded_dataset.train_test_split(test_size=0.2)
```

2.5 Custom Trainer for Handling Two Labels (Risk and Positive Scores)

Then, a custom trainer is defined to optimize the model training by overriding the loss calculation method. Specifically, it calculates the difference between the model's predictions and the actual labels (using Mean Squared Error, MSE) and returns the loss value for training.

In [9]:
```
from transformers import TrainingArguments, Trainer
import torch

class CustomTrainer(Trainer):
    def compute_loss(self, model, inputs, return_outputs=False, **kwargs)
        labels = inputs["labels"]
        if isinstance(labels, np.ndarray):  # 兼容 NumPy 格式
            labels = torch.tensor(labels, dtype=torch.float32).to(model.d

        model_inputs = {k: v for k, v in inputs.items() if k != "labels"}
        outputs = model(**model_inputs)
        logits = outputs.logits
        loss_fct = torch.nn.MSELoss()
        loss = loss_fct(logits, labels)
        return (loss, outputs) if return_outputs else loss
```

3. Model Training and Evaluation

3.1 Defining Evaluation Metrics

A function is designed to calculate the Mean Squared Error (MSE) between the model's predictions and the true labels, and return a dictionary containing the MSE value.

In [10]:
```
def compute_metrics(eval_pred):
    predictions, labels = eval_pred
    if isinstance(predictions, tuple):
        predictions = predictions[0]

    mse = np.mean((predictions - labels) ** 2)
    return {"mse": mse}
```

3.2 Setting Training Parameters

Then, the training parameters, training_args, are defined to set training configurations such as output directory, evaluation strategy, learning rate, batch size, number of training epochs, weight decay, logging, and saving strategies. Specifically, it will evaluate and save the model after each training epoch and log every 10 steps.

In [11]:
```
training_args = TrainingArguments(
    output_dir="./results",
    eval_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=20,
    weight_decay=0.01,
    logging_strategy="steps",
    logging_steps=10,
    save_strategy="epoch",
)
```

3.3 Initializing the Trainer

A CustomTrainer instance is created, passing the model, training parameters, training set, test set, and the function for calculating evaluation metrics. This instance will be used for training and evaluating the model.

In [12]:
```
trainer = CustomTrainer(
    model=model,
    args=training_args,
    train_dataset=encoded_dataset["train"],
    eval_dataset=encoded_dataset["test"],
    compute_metrics=compute_metrics,
)
```

3.4 Training the Model

In [13]:
```
trainer.train()
trainer.save_model("./trained_model")
```

[60/60 00:46, Epoch 20/20]

Epoch	Training Loss	Validation Loss	Mse
1	No log	0.314457	0.314457
2	No log	0.150936	0.150936
3	No log	0.083095	0.083095
4	0.256200	0.084059	0.084059
5	0.256200	0.070650	0.070650
6	0.256200	0.073462	0.073462
7	0.084900	0.061905	0.061905
8	0.084900	0.030397	0.030397
9	0.084900	0.036832	0.036832
10	0.045900	0.030561	0.030561
11	0.045900	0.017708	0.017708
12	0.045900	0.017047	0.017047
13	0.045900	0.018596	0.018596
14	0.031800	0.017361	0.017361
15	0.031800	0.015835	0.015835
16	0.031800	0.015627	0.015627
17	0.038900	0.015775	0.015775
18	0.038900	0.015989	0.015989
19	0.038900	0.016263	0.016263
20	0.027700	0.016346	0.016346

Table 1: Training Loss, Validation Loss, and MSE during Model Training

This table shows the training loss, validation loss, and Mean Squared Error (MSE) during the 20 epochs of model training. It can be observed that the validation loss reached its minimum value of 0.015627 at epoch 16, indicating that the model performed best at this point.

4. Experimental Results and Analysis

4.1 Function for Predicting New Text

A function, predict_scores, is designed to encode input text, pass it through the model for prediction, retrieve the logits from the model's output, and return the risk score and positive score.

In [14]:
```
def predict_scores(text, model, tokenizer):
    inputs = tokenizer(text, return_tensors="pt", truncation=True,
```

```
                        padding="max_length", max_length=128)
    inputs = {key: val.to(model.device) for key, val in inputs.items()}

    with torch.no_grad():
        outputs = model(**inputs)
        logits = outputs.logits

    if logits.shape[-1] != 2:
        raise ValueError(f"Expected 2 output values (risk, positive),
                           but got {logits.shape[-1]}")

    risk_score, positive_score = logits[0].cpu().numpy()

    return float(risk_score), float(positive_score)
```

4.2 Testing and Validation of a Single Keyword

The text “flood” is predicted, and the output risk score and positive score are compared with the expected scores. This helps to validate the model’s accuracy in evaluating individual keywords.

In [15]:
```
text = "flood"
risk_score, positive_score = predict_scores(text, model, tokenizer)
print(f"Text: {text}")
print(f"Predicted Risk Score: {risk_score * 10:.2f}
            (Expected: {risk_scores[text]})")
print(f"Predicted Positive Score: {positive_score * 10:.2f}
            (Expected: {positive_scores[text]})")
```

```
Text: flood
Predicted Risk Score: 9.06 (Expected: 10)
Predicted Positive Score: 1.24 (Expected: 2)
```

4.3 Prediction Errors between Predicted and True Values

Three functions are designed to perform the following key tasks:

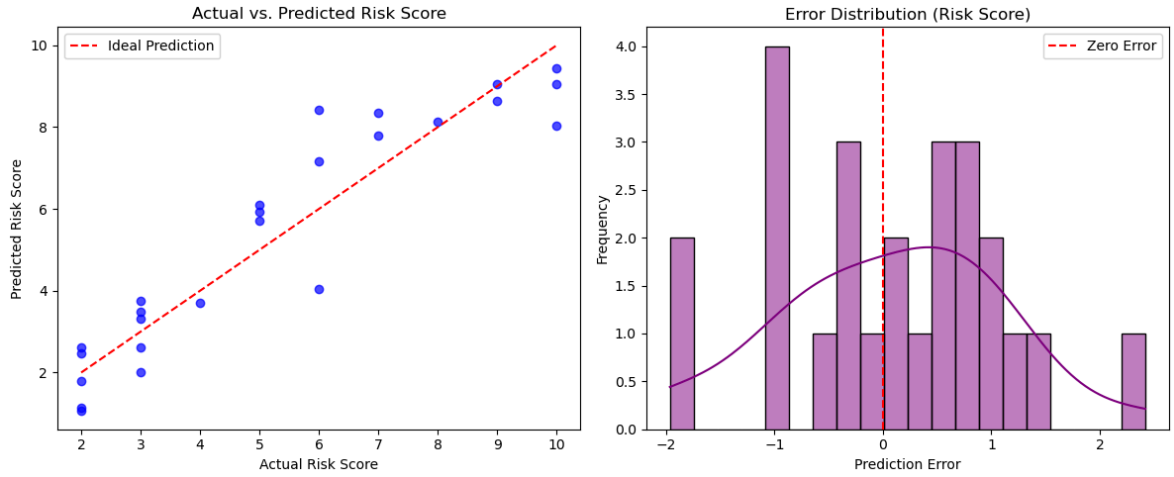
1. get_all_predictions: This function is used to obtain the predicted values for all keywords and record the true values.
2. compute_errors: This function calculates the error between the actual values and the predicted values.
3. plot_results: This function visualizes the prediction results, including a scatter plot of the actual vs. predicted values and a histogram of the error distribution.

In [16]:
```
import torch
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from tqdm import tqdm

def get_all_predictions(keywords, model, tokenizer):
    predicted_risk_scores = []
    predicted_positive_scores = []
    actual_risk_scores = []
    actual_positive_scores = []
```

```
    for keyword in tqdm(keywords, desc="Predicting"):
        risk_pred, positive_pred = predict_scores(keyword, model, tokeniz
        actual_risk_scores.append(risk_scores[keyword])
        actual_positive_scores.append(positive_scores[keyword])
        predicted_risk_scores.append(risk_pred * 10)
        predicted_positive_scores.append(positive_pred * 10)

    return actual_risk_scores, predicted_risk_scores,
           actual_positive_scores, predicted_positive_scores

def compute_errors(actual, predicted):
    errors = np.array(predicted) - np.array(actual)
    return errors

def plot_results(actual, predicted, score_type="Risk Score"):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.scatter(actual, predicted, alpha=0.7, color="blue")
    plt.plot([min(actual), max(actual)], [min(actual), max(actual)],
             "--", color="red", label="Ideal Prediction")
    plt.xlabel(f"Actual {score_type}")
    plt.ylabel(f"Predicted {score_type}")
    plt.title(f"Actual vs. Predicted {score_type}")
    plt.legend()

    plt.subplot(1, 2, 2)
    errors = compute_errors(actual, predicted)
    sns.histplot(errors, bins=20, kde=True, color="purple")
    plt.axvline(x=0, color="red", linestyle="--", label="Zero Error")
    plt.xlabel("Prediction Error")
    plt.ylabel("Frequency")
    plt.title(f"Error Distribution ({score_type})")
    plt.legend()
    plt.tight_layout()
    plt.show()
```

In [17]:
```
actual_risk, predicted_risk, actual_positive, predicted_positive = get_al
plot_results(actual_risk, predicted_risk, "Risk Score")
plot_results(actual_positive, predicted_positive, "Positive Score")
```

```
Predicting: 100%|██████████| 25/25 [00:00<00:00, 64.27it/s]
```



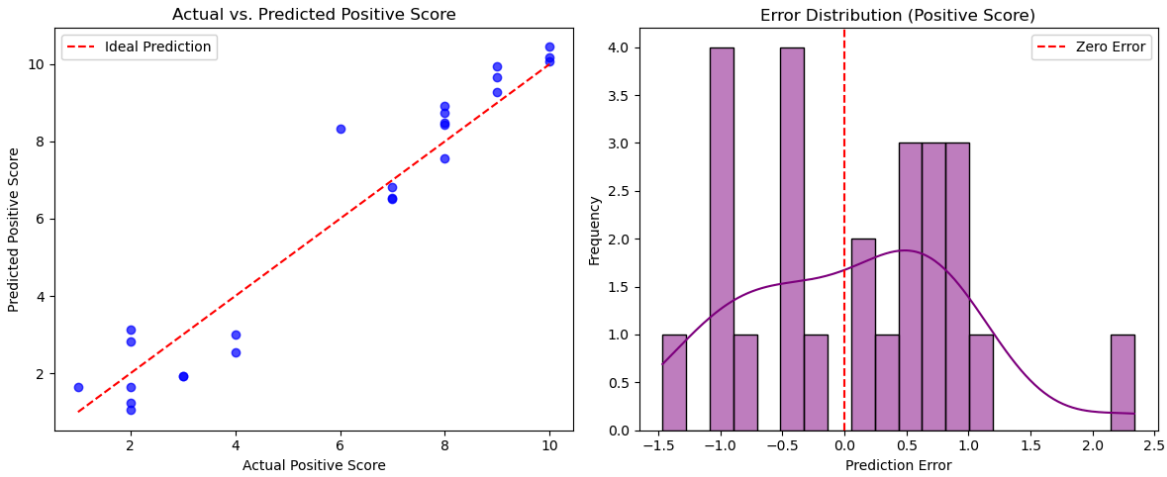


Figure 1: Comparison of Actual and Predicted Risk Scores
The left image shows a scatter plot of the actual vs. predicted risk scores, with the red line representing the ideal prediction. The right image displays the distribution of prediction errors, with the purple curve representing the kernel density estimation and the red line indicating the zero error line.

Ideally, all data points should closely align along the red dashed line (y=x). However, the actual results show some deviations, with more dispersed points indicating larger model errors. Despite this, the overall trend remains positive. In the error distribution plot, the errors fluctuate around 0, indicating that the model has some bias and still has room for improvement. To enhance the model's performance, several strategies can be considered: increasing the training data, adjusting the optimizer (e.g., using MSELoss), normalizing the data, and increasing the number of training epochs to improve the model's generalization ability.

4.4 Testing New Text

The model is used to predict the risk score and positive score for the new text, "Our city is investing in renewable energy and climate adaptation." The predicted scores are then compared with the expected scores for the keywords "renewable energy" and "climate adaptation." This helps evaluate the model's performance in handling new and complex sentences.

In [95]:
```
new_text = "Our city is investing in renewable energy and climate adaptat
risk_score, positive_score = predict_scores(new_text, model, tokenizer)
print(f"Text: {new_text}")
print(f"""Predicted Risk Score: {risk_score * 10:.2f}
        (Expected: renewable energy({risk_scores['renewable energy']}),
         climate adaptation({risk_scores['climate adaptation']}))""")
print(f"""Predicted Positive Score: {positive_score * 10:.2f}
        (Expected: renewable energy({positive_scores['renewable energy']}
        climate adaptation({positive_scores['climate adaptation']}))""")
```

```
Text: Our city is investing in renewable energy and climate adaptation.
Predicted Risk Score: 4.62
        (Expected: renewable energy(2),
         climate adaptation(3))
Predicted Positive Score: 7.10
        (Expected: renewable energy(10),
        climate adaptation(8))
```

This indicates that the risk score for the text is close to the scores for “renewable energy” and “climate adaptation”, while the positive score is also high, similar to the scores for “renewable energy” and “climate adaptation”. This suggests that the model correctly assesses the text as having a low risk and high positive contribution related to these keywords.

4.5 Semantically Similar Text Testing

To validate the model’s generalization ability, we can test a piece of text that does not directly contain the training keywords but has a similar meaning. For example:

In [93]:
```
new_text = "Our city is investing in clean energy initiatives " \
           "and disaster preparedness."
risk_score, positive_score = predict_scores(new_text, model, tokenizer)
print(f"Text: {new_text}")
print(f"""Predicted Risk Score: {risk_score * 10:.2f}
        (Expected: renewable energy({risk_scores['renewable energy']}),
        climate adaptation({risk_scores['climate adaptation']}))""")
print(f"""Predicted Positive Score: {positive_score * 10:.2f}
        (Expected: renewable energy({positive_scores['renewable energy']}
        climate adaptation({positive_scores['climate adaptation']}))""")
```

```
Text: Our city is investing in clean energy initiatives and disaster prepa
redness.
Predicted Risk Score: 5.35
        (Expected: renewable energy(2),
        climate adaptation(3))
Predicted Positive Score: 6.67
        (Expected: renewable energy(10),
        climate adaptation(8))
```

This indicates that the model is able to recognize the meaning of “clean energy initiatives” (similar to renewable energy) and “disaster preparedness” (similar to climate adaptation), and provide reasonable scores based on these concepts.

5. Model Limitations and Suggestions for Improvement

5.1 Insufficient Data

The current dataset only contains 25 keywords, which is limited and may lead to overfitting or insufficient generalization on new data. To address this issue, the following measures will be taken in the future:

1. Expand the training dataset by extracting more text from publicly available climate change-related documents and manually labeling risk and positive scores.
2. Generate more training data through techniques such as synonym replacement and sentence rewriting. For example, “renewable energy” could be replaced with “clean energy” or “green energy” to enrich the corpus.

5.2 Using Domain-Specific Models

BERT-base-uncased is a general-purpose pre-trained model suitable for various tasks. In the future, I consider using the following domain-specific pre-trained models:

- **distilbert-base-uncased**: A lighter model suitable for tasks with smaller datasets and lower computational overhead.
- **ClimateBERT**: A model specifically designed for climate change tasks, capable of better understanding climate change-related terminology and context.
- **SciBERT**: A model tailored for the scientific domain, particularly effective in handling texts related to environmental science and excelling in processing specialized terminology.

6. Conclusion

This paper provides a comprehensive assessment of climate-related keywords through a dual-dimension analysis (risk and positivity), which not only identifies potential risks but also measures their positive impacts. Subsequently, using natural language processing (NLP) techniques and pre-trained models (such as BERT), the model was trained on keywords and their respective scores. The model evaluation results show that certain progress has been made. Given new text inputs, the model is able to provide reasonably accurate predicted scores.

The method proposed in this paper can be applied to climate risk assessment, aiding governments, businesses, and researchers in optimizing climate management and response strategies. In the future, the accuracy and applicability of risk and positivity assessments can be further improved by incorporating expert ratings, advanced NLP techniques, and large language models.