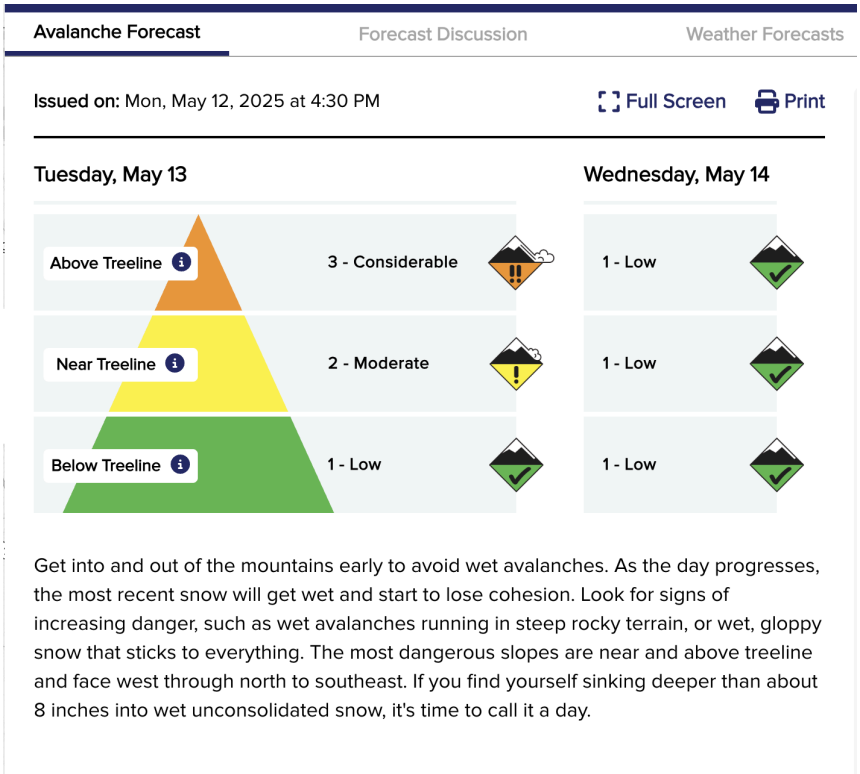
Avalanche Forecast Deep Learning

Sam Vredenburgh

Summary:
As an avid backcountry skier, the Colorado Avalanche Information Center (CAIC) has been an invaluable resource, providing avalanche danger forecasts across the state at all aspects and elevations. For my final project I wanted to incorporate the text forecast summaries that appear on the front page of each forecast zone, along with the forecasted danger levels at each elevation tier corresponding to the forecast summary text. The text is typically 1-2 paragraphs, the elevation tiers are always **Below Treeline**, **Near Treeline** and **Above Treeline**, and the danger level for each elevation tier is a categorical scale as follows:

1 - Low
2 - Moderate
3 - Considerable
4 - High
5 - Extreme

There is an additional forecast level designated as **Early Season**, typically used in the Fall before substantial snowfall. Below is an example forecast with associated danger levels per elevation tier.



My main goal was to use NLP techniques and different neural network architectures to build a multi-output multi-class classification model that, given the text forecast summary, can classify

the danger level for each elevation tier. I believe deep learning will be integral to building a meaningful model because natural language is messy and unstructured, and there are many useful open source and past projects leveraging NLP and neural network tools and techniques for text classification tasks such as this project.

Data Wrangling:

My first step was reaching out to the CAIC (https://avalanche.state.co.us/) to request historical forecast data that included the forecast summary text along with the danger levels for each elevation tier. The contacts at CAIC were receptive to the idea and open to providing the information I need, however because they had a busy Spring forecasting, It took over a month and many follow up emails to finally receive any data. The raw data file was in json, and the text was unparsed html. I spent a bit of time removing the json errors stemming from links, <a> tags, that caused issues with the hrefs having double quotes. After that I was able to pull in the json and create a dataframe. The file contained 3334 forecasts, which was whittled down to 3177 after removing all instances that had any danger levels with 'noRating' values. I then used BeautifulSoup to parse the html text summaries into a new column. I went through another round of removing empty strings in this clean summary column, resulting in a new count of 2885 rows of data. I later manually pulled another 56 summaries and danger levels from the CAIC website, and after concatenating those with the original data was left with a total of 2941 instances that I exported in a new csv file.

EDA:

I created a new notebook for exploratory data analysis and to further refine the raw forecast data with text preprocessing. After looking into the value_counts for each elevation tier column, it was clear there were 0 instances of the **Extreme** danger level 5, although some instances of **Early Season**. I decided to use all of the Early Season instances and keep the total number of target classes at 5, and map the string values to integers:

- 0 - earlySeason
- 1 - low
- 2 - moderate
- 3 - considerable
- 4 - high

After mapping the output levels, I moved on to the summary text. I used the 'nltk' package 'stopwords' to create a new column with the stopwords and punctuation removed, and all words to lowercase. From there I performed some visual EDA including a wordcloud visual, different plots for the most used words across all summaries, and corresponding to the different danger levels. I created a histogram of the distribution of summary lengths, and finally performed a TF-IDF analysis to discover the most informative words across all of the summaries.

Modeling:

After the EDA and text parsing process, I removed all rows with any null values and was left with a total of 2939 instances. My plan had been to use both Bag of Words/N-grams approach and

sequence models for the classification task, however the rule of thumb from the book stated to look at the ratio of the number of samples to the mean sample length and if it was less than 1500 to use Bag-of-bigrams. That ratio for my data was very small at around 36, but I decided to continue with all different kinds of models anyway as a learning experience. Before getting into the deep learning models I used sklearn's DummyClassifier to set a baseline for the tier level class prediction accuracy levels that I aimed to beat.

I used this basic outline to ramp up model complexity:

- bag of Words
- bi-grams
- tf-idf
- Bidirectional LSTM
- word embeddings
- TransformerEncoder
- Pre-trained Transformer

The reason for this is partly due to how the text classification topics were covered in the textbook using the IMBD dataset with sentiment analysis as a classification task.

One of the most confusing parts of my project was the text preprocessing involved. I needed to leverage different vectorizer parameters from tensorflow to make sure my text was in the correct format for the neural networks and consisted of the correct shapes. Similar to the textbook, my first models with dense layers used multi-hot encoding, and eventually with bigram and TF-IDF. The more complex models with embeddings used 'int' as the output mode to use the embedding models or trained embeddings. These layers trained on more epochs as well. All of the models below were created with the keras functional API and used three separate output layers, one for each elevation tier, all with 5 units (number of classes) and softmax activation functions.

```
inputs = keras.Input(shape=(input_dim,))
x = layers.Dense(16, activation="relu")(inputs)
x = layers.Dense(16, activation='relu')(x)
x = layers.Dense(16, activation='relu')(x)

output_below = layers.Dense(num_classes, activation='softmax', name='below')(x)
output_treeline = layers.Dense(num_classes, activation='softmax', name='treeline')(x)
output_above = layers.Dense(num_classes, activation='softmax', name='above')(x)

model = keras.Model(inputs=inputs, outputs=[output_below, output_treeline, output_above])
```

I will go through each model below, referencing the text preprocessing, layers and outcome.

Bag of Words - Basic Dense Layers:

- Text Preprocessing:
  - TextVectorization layer
  - multi-hot encoding
- Layers:
  - 2 Dense layers of 16 neurons each
- Outcome:
  - **Best val_above_accuracy: 0.7891**
  - **Best val_below_accuracy: 0.8146**
  - **Best val_treeline_accuracy: 0.8061**

Bag of Words - Extra Layer:

- Text Preprocessing:
  - TextVectorization layer
  - multi-hot encoding
- Layers:
  - 3 Dense layers of 16 neurons each
- Outcome:
  - **Best val_above_accuracy: 0.7823**
  - **Best val_below_accuracy: 0.8231**
  - **Best val_treeline_accuracy: 0.8180**

Bigram:

- Text Preprocessing:
  - TextVectorization layer
  - Multi-hot
  - Ngrams = 2
- Layers:
  - 2 Dense layers of 16 neurons each
- Outcome:
  - **Best val_above_accuracy: 0.7959**
  - **Best val_below_accuracy: 0.8333**
  - **Best val_treeline_accuracy: 0.7959**

Bigram with TF-IDF:

- Text Preprocessing:
  - TextVectorization layer
  - TF-IDF output
  - Ngrams = 2
- Layers:
  - 2 Dense layers of 16 neurons each

- Outcome:
    - **Best val_above_accuracy: 0.7823**
    - **Best val_below_accuracy: 0.8350**
    - **Best val_treeline_accuracy: 0.8027**

Embedding with Bidirectional LSTM:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - Embedding Layer
    - Bidirectional LSTM with 32 neurons
    - 50% Dropout layer
- Outcome:
    - **Best val_above_accuracy: 0.7517**
    - **Best val_below_accuracy: 0.8061**
    - **Best val_treeline_accuracy: 0.7840**

Embedding with Bidirectional LSTM - with Masking:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - Embedding Layer
    - Bidirectional LSTM with 32 neurons
    - 50% Dropout layer
    - mask_zero = True
- Outcome:
    - **Best val_above_accuracy: 0.7755**
    - **Best val_below_accuracy: 0.8180**
    - **Best val_treeline_accuracy: 0.7976**

GloVe Word Embedding:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - Embedding Layer - with Embedding Matrix
    - Bidirectional LSTM with 32 neurons
    - 50% Dropout layer
- Outcome:

        - **Best val_above_accuracy: 0.7432**
        - **Best val_below_accuracy: 0.7942**
        - **Best val_treeline_accuracy: 0.7789**

Word2Vec:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - Embedding Layer - with Embedding Matrix
    - Bidirectional LSTM with 32 neurons
    - 50% Dropout layer
- Outcome:
    - **Best val_above_accuracy: 0.7704**
    - **Best val_below_accuracy: 0.8146**
    - **Best val_treeline_accuracy: 0.8078**

Word2Vec with Training:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - Embedding Layer - with Embedding Matrix
    - Bidirectional LSTM with 32 neurons
    - 50% Dropout layer
    - Embedding layer trainable = True
    - Embedding layer mask_zero = True
- Outcome:
    - **Best val_above_accuracy: 0.7942**
    - **Best val_below_accuracy: 0.8282**
    - **Best val_treeline_accuracy: 0.8095**

Transformer Encoder:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - Embedding Layer
    - Transformer Encoder layer (from Textbook):
        - MultiHeadAttention
        - Normalization layers

        - Dense layers
    - GlobalMaxPooling1D
    - 50% Dropout layer
- Outcome:
    - **Best val_above_accuracy: 0.7840**
    - **Best val_below_accuracy: 0.7942**
    - **Best val_treeline_accuracy: 0.7976**

Transformer Encoder with Positional Embedding:

- Text Preprocessing:
    - TextVectorization layer
    - ‘int’ output mode
    - Sequence length = 126
- Layers:
    - PositionalEmbedding Layer (from Textbook)
    - Transformer Encoder layer (from Textbook):
        - MultiHeadAttention
        - Normalization layers
        - Dense layers
    - GlobalMaxPooling1D
    - 50% Dropout layer
- Outcome:
    - **Best val_above_accuracy: 0.7925**
    - **Best val_below_accuracy: 0.8180**
    - **Best val_treeline_accuracy: 0.8044**

Pretrained Transformer (DistilBert):

- Text Preprocessing:
    - DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
- Layers:
    - DistilBert model
    - 50% dropout
- Outcome:
    - **Best val_tf.nn.softmax_accuracy: 0.5000**
    - **Best val_tf.nn.softmax_1_accuracy: 0.4201**
    - **Best val_tf.nn.softmax_2_accuracy: 0.4354**

Bigram with TF-IDF - More Layers:

- Text Preprocessing:
    - TextVectorization layer
    - TF-IDF output
    - Ngrams = 2
- Layers:
    - 2 Dense layers of 16 neurons each

    - Dense layer of 32 neurons
    - Dense layer of 64 neurons
    - ‘rmsprop’ optimizer
- Outcome:
    - **`Best val_above_accuracy: 0.7993`**
    - **`Best val_below_accuracy: 0.8333`**
    - **`Best val_treeline_accuracy: 0.8027`**

Conclusion:

I found that almost all of the models besides the most complex, DistilBert, had around 80% accuracy for each elevation tier. It seemed like no matter what I did to try and breach that threshold, I could only reach a few percentage points higher, like with the last Bigram TF-IDF model. I used this last model to predict the current CAIC forecast, inputting raw text to the vectorizer and predicting the levels. The model predicted:

Below treeline danger: 2
Treeline treeline danger: 1
Above treeline danger: 1

With the actual levels being:

Below treeline danger: 1
Treeline treeline danger: 1
Above treeline danger: 2

Overall, these models performed much better than the baseline that I was aiming to beat, however there was not much difference in the outcome from all these different techniques, using bag of words vs sequence models or adding complexity with positional embedding and multi-head transformers. I think the main cause for this is the fact that my dataset was not very large. I only had around 3000 instances to train on from the CAIC vs 50000 in the IMBD example from the textbook. If I was able to continue this work I would request much more data, including examples with the **Extreme** danger level and across many years. I think being able to process many more instances across the danger levels could improve model performance, especially with the more complex sequence-based models and pre-trained transformers.

References:

I researched the DistilBERT model through huggingface (https://huggingface.co/docs/transformers/model_doc/distilbert), although I was admittedly a bit lost in the full implementation. I worked along with the textbook, Deep Learning with Python by Francois Chollet, very closely to build these models as referenced above. For the positional embedding and transformer encoder layers I took most of the class definitions from the class notebook and made it work with my data and text preprocessing.