Outsiders

Beating the Bookies with Neural Networks

by Tom Bidewell, James Brett, Nicholas Evans, and Joseph McInerny.

Abstract

Descriptions of horses and horse races written by bookies were collected over three months from the betting website RacingPost. We analyzed the data collected through implicit tokenization, via two different methods: one, feeding sentences into the TF*IDF algorithm, followed by training using a Multilayer Perceptron using the ReLU activation function; two, using HuggingFace's BERTModel class. Following this, we used several ranking methods to predict the place of each horse in a given race, which in turn we then used to train our model. We evaluated the results using NCDG and our results were promising, indicating that it may be possible to predict horse races using data available online.

Introduction

In 2020, there were 4.5 billion GBP worth of bets placed on horse racing each year in the United Kingdom alone. The horse gambling industry is worth 402.3 billion USD, and is expected to double in the next 7 years. Betting companies primarily set their bets in two different ways: parimutuel and odds setting. As the old adage goes, "the house always wins": no matter which horse wins and which pool takes the money, the owners of the betting houses always take their cut for providing the service. This is especially true for odds setting systems, which are run by bookmakers, or "bookies". These experts set the odds for each race based on (in theory) the perceived strength and past performances of both the horse and its jockey.

The aim of this project is to use data available on the betting website *Racing Post ("Horse Racing Cards, Results & Betting | Racing Post")* to see if a combination of numerical and textual information provided by the bookies to their users can be harnessed to create a predictive model that can predict the winners of horse races across the globe, from the UK to the USA to Hong Kong (Silverman).

Using the tools and strategies learned in our various modules on machine learning, natural language processing, and linguistics, we put together the following plan for building this predictive model:

1. Tokenizing and embedding the data collected;
2. Putting our data through a multi-layered perceptron,with appropriate activation function;
3. Selecting learning and ranking strategies
4. Finally, analysing our results to see what conclusions we can draw.

Theory

Many approaches were considered for each step in the process of building the project. This section will outline in detail the Machine Learning concepts and principles behind each step in the project, along with evaluating the potential approaches for each step.

TF*IDF

TF*IDF stands for “Term Frequency*Inverse Document Frequency”. Frequently used in Natural Language Processing tasks, TF*IDF evaluates the occurrences of words relative to the number of documents in a given corpus. In doing so, it captures words uniquely common in a given document, and gives less weight to words frequently occurring across documents in the entire corpus.

$$\mathrm{tf}(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}},$$

tf (term frequency) function w/ parameters *t* for terms and *d* for documents. The function *f* represents the frequency of term *t* in doc. *d.*

$$\mathrm{idf}(t,D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

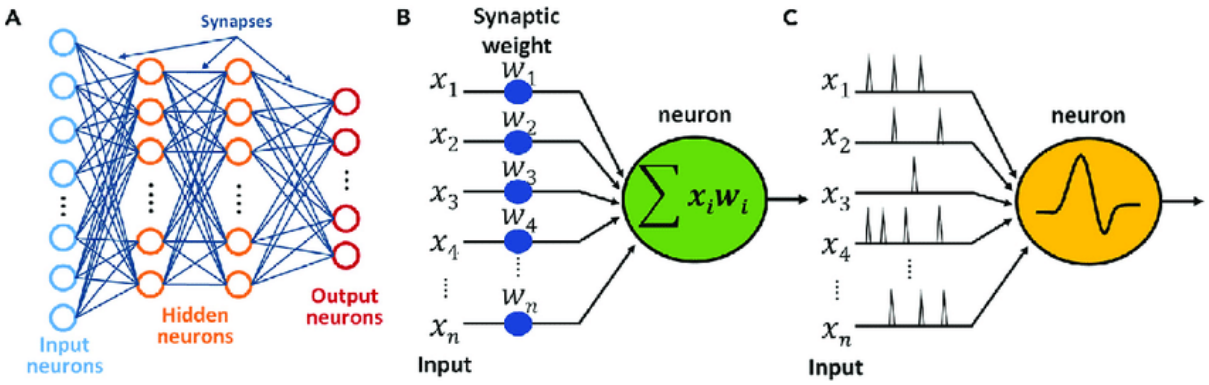
idf (inverse doc freq), w/ parameters term *t* and document set D. N represents the count of the documents.

For example, common and essential words in languages such as articles and determiners (in English) will be given lower weights by this function because they appear universally across

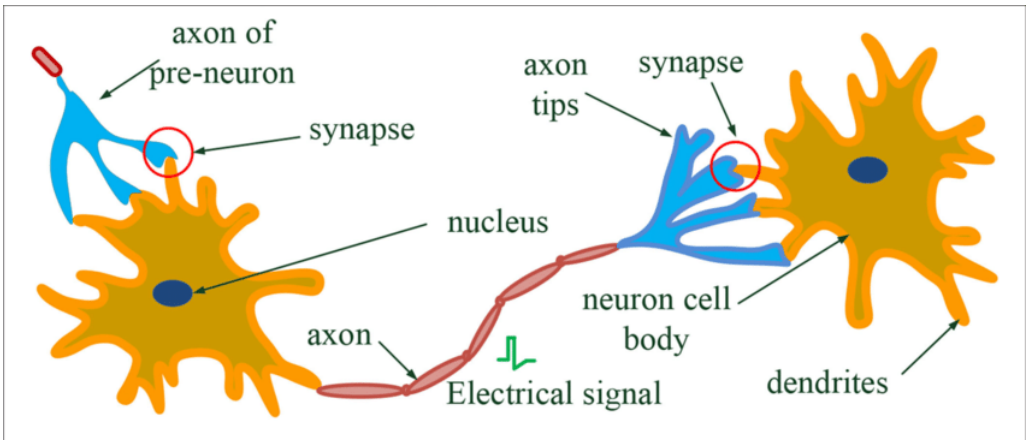
documents, although their relative frequency in each document will be taken into account. For our purposes, this provides us with our embeddings and accomplishes implicit tokenization.

Neural Networks

Neural Networks, or Artificial Neural Networks (NNs or ANNs) are a computational model based on and inspired by the internal architecture of human brains. Like human brains, NNs are composed of (in this case, artificial) neurons and connections (synapses) between these neurons. The electrical signal from natural neuron to neuron is modeled by activation functions in the artificial model.



Diagrams of artificial neurons (above) and biological neurons (below)



From this essential architectural concept, there are now many different types of NNs. For the purpose of prefacing the following theoretical bases of our project, we will outline the basic functionality of feedforward neural networks (FNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs).

FNNs are the original form of NN. They consist of an input layer, one or more hidden layers, and an output layer. They are monodirectional and non-sequential: the input layer takes in a set of data simultaneously, and this data flows through the hidden layers without cycles. We will be utilizing MLPs (Multi-Layered Perceptrons), which are FNNs composed of the typical input and output layers of a NN, an activation function, plus multiple hidden layers. MLPs are a standard and relatively simple form of neural network commonly used for ML and NLP tasks.

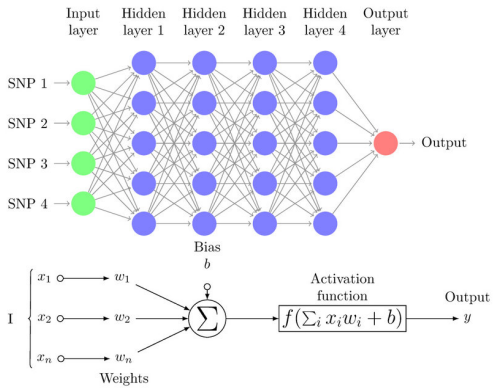


Diagram at left shows a MLP with 4 hidden layers.

There are many activation functions that can be selected for neural networks, but we will briefly go over Sigmoid and ReLU.

$$f(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases} \quad f'(x) = \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x < 0. \end{cases}$$

ReLU function.

ReLU, or Rectified Linear Unit, is an activation function that shifts any negative values to zero in order to account for the vanishing gradient problem, where negative values cause the weights of various features to remain unchanged throughout the training process. sigmoid function.

Sigmoid activation functions are used for accounting for the rising gradient problem. Sigmoid functions essentially tighten the upper bounds of the weights in transition from neuron to neuron. Unfortunately this also means that they can fall victim to the vanishing gradient problem.

CNNs are neural networks that use filters called convolutional layers to filter their input before sending the data forward in the form of feature maps to the typical hidden layers and output. These feature maps are, as the name suggests, designed to capture certain features of the input data set. Theoretically, feature maps could be useful for highlighting the most significant statistics in determining the winner of a horse race.

RNNs, on the other hand, are cyclical and sequential by nature. The cyclical aspect comes in the form of recurrent connections, based on a hidden state. A hidden state acts as the memory of the network; it stores information about previously treated data and is constantly updated to inform future decisions. This hidden state allows RNNs to handle sequences of varying lengths, making it a versatile tool capable of dealing with many types of data, especially natural language.

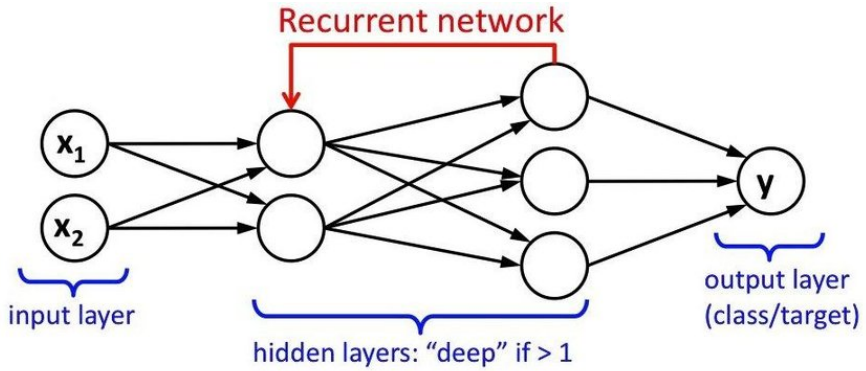


Diagram of a simple RNN.

BERT

BERT (Bidirectional Encoder Representations from Transformers) is a group of language models created by researchers at Google and published to the world in 2018. In the following years, BERT became one of the highest standards of NLP and language modelling available.

At its core, BERT is made up of layers of transformers, a type of seq2seq model, which in turn are encoding models based off of LSTM (Long Short-Term Memory), which in turn are built on RNNs. BERT was initially trained on two corpuses of 800 million words and 2.5 billion words (Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova). These massive training sets along with the merits of the transformer model in NLP make BERT incredibly effective.

Ranking

The results of a horse race are always formatted in a ranked list of the relative finishes of each horse. The nature of these outcomes means that the task of predicting their outcome necessitates the use of Learning to Rank, or LTR, approaches of classifying the output of our data. This section will give an overview of each of the potential strategies under this category.

Pointwise

Pointwise ranking is a method used to rank items individually based on their features and relevance. Unlike pairwise and listwise ranking methods, which consider interactions between items, pointwise ranking treats each item as an independent entity and assigns a score or rank based on its individual characteristics. In this way, pointwise ranking methods focus on predicting the relevance and quality of each item individually. The relationships or dependencies between items in the ranking process are not considered; the ranking is based solely on the predicted scores or relevance values assigned to each item.

In our context, pointwise ranking would assign a predictive score to each horse based on its features and past performances. The horses would then be ranked individually based on their predicted scores, without considering the relationships or interactions between them: they would simply be compiled into a list. The implementation of this strategy would be straightforward and relatively simple, but falls into several shortcomings. Not only would it fail to model the real-world events on the track, but pointwise tends to be inaccurate thanks to its atomised approach: the results of a race are not based solely on the merit of each horse, but rather the relative performance of each horse in comparison to the others. A pointwise approach doesn't take into account the nature of this relative performance.

Pairwise

Pairwise methods are a class of LTR techniques used in information retrieval and recommendation systems. Pairwise methods focus on ranking items or instances based on their relative preferences or relevance. The training data consists of pairs of instances along with their corresponding labels indicating the preference or order between them, with the objective of developing a ranking function that can accurately rank new instances based on their comparisons. Essentially, we would compute scores for each horse, and then compare them, one by one, to all the other horses in the race. Based on their performance in these one-on-one comparisons, they would then be ranked for the race.

For our purposes, these methods offer an approach to rank horses based on their relative performance rather than directly predicting their finishing positions. This allows the ranking process to be more nuanced and flexible, and accounts for variations and uncertainties in horse racing outcomes. As noted previously, the finishing position of a horse

does not necessarily depend on its performance by its own standards, but rather its performance compared to the other horses in the race.

Listwise

Listwise ranking methods have the advantage of considering the entire list of items and their interdependencies when learning the ranking function. This wider perspective allows for more accurate and comprehensive ranking predictions compared to pairwise or pointwise methods.

$$\sum_{i=1}^{m} L(y^{(i)}, z^{(i)})$$

where L=listwise loss function, y=list of scores, and z=rank results; in our case, we use NDCG as a loss function.

In the context of horse race outcomes, listwise ranking can take into account the complex interactions among multiple horses, their features, and the overall race dynamics to produce a ranked list of horses that reflects their likelihood of success.

Implementing listwise ranking for horse race outcomes, however, requires careful consideration of various factors, such as the quality and relevance of features, availability of training data, and the choice of the appropriate listwise learning algorithm. The success of listwise ranking models in horse racing prediction relies on the availability of reliable and comprehensive data and the ability of the model to capture the intricate dynamics of racing. A potential shortfall for us when using this method would be the quality and quantity of our data. If the data we've collected isn't sufficient to The results of our models are already heavily reliant on the quality of the data we've collected, and if the bookies' information isn't of sufficient quality we could find ourselves at a disadvantage.

Evaluation

Normalised Discounted Cumulative Gain (nDCG)

Normalised Discounted Cumulative Gain is a method commonly used to evaluate ranking models from 0 to 1 (continuous). In NLP, it is often used to measure the accuracy of document retrieval in a search engine. It takes the list of documents and assigns each of them a relevance score. It then compares these relevancy scores with the gold classification. It evaluates the model by determining how accurate its predictions are.

Wang et Al (Wang et al.) showed that the nDCG scores converge to 1 over a data set as it increases in size. This appears worrying as many uses of nDCG evaluation are applied to large data sets. Fortunately Wang was also able to prove that nDCG can still be used effectively to distinguish between the effectiveness of different ranking algorithms. Due to this nDCG is a suited evaluation model for our task to evaluate different ranking models abilities to rank horses in a horse race.

The DCG is calculated using the following formula:

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{\log_2(i+1)}$$

Here p represents the number of horses in the race. *rel(i)* represents the relevance of a particular horse. DCG usually sorts relevancy where the most relevant document is the document with the highest relevancy score. In our case where the relevancy is represented by the horse's position in the race, the relevancy is calculated as *1 / rel(i)*. This way for example the horse predicted at position 1 will have a higher relevancy score than the horse at position 2 and so on (1/1 > 1/2) . This will then allow us to sort the list where the horse that came first is the most relevant.

The Ideal Discounted Cumulative Gain serves as a tool for normalising DCG scores. It normalises DCG scores by comparing them with the ‘ideal’ i.e., horses are correctly positioned in our case. This ideal list of positions is represented by REL which is a list where *REL(i)* would be *1/1* and *REL(p)* would be *p / 1* in increasing order.

IDCG:

$$\mathrm{IDCG_p} = \sum_{i=1}^{|REL_p|} \frac{rel_i}{\log_2(i+1)}$$

The formula to calculate the normalised value for a race is therefore as follows:

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p},$$

The mean NDCG score is then taken across all races to provide an evaluation of the ranking model.

An Illustration of the Calculation

To illustrate this method with an example, I will take a 4 horse race where the horses are named Joey, Tom, Niko and James. The example data is as follows:

Horse Name	Ideal Position	Predicted Position
Joey	1	2
Tom	2	1
James	3	4
Niko	4	3

In order to understand why the predicted position will be inferior to the ideal position we can define the denominator and the numerator. The numerator is the relevance of the horse's position and the denominator is the position based discounting (of the list).
So, a desirable score will have the following:
High relevance / low position
This is to maximise the value of the sum. The full calculation example is below.

$$IDCG = \frac{2^{\frac{1}{1}}-1}{\log_2(0+2)} + \frac{2^{\frac{1}{2}}-1}{\log_2(1+2)} + \frac{2^{\frac{1}{3}}-1}{\log_2(2+2)} + \frac{2^{\frac{1}{4}}-1}{\log_2(3+2)}$$

$$IDCG \approx 1 + 0.261 + 0.130 + 0.081 \approx 1.472$$

$$DCG = \frac{2^{\frac{1}{2}}-1}{\log_2 0+2} + \frac{2^{\frac{1}{1}}-1}{\log_2 1+2} + \frac{2^{\frac{1}{4}}-1}{\log_2 2+2} + \frac{2^{\frac{1}{3}}-1}{\log_2 3+2}$$

$$DCG \approx 0.414 + 0.632 + 0.095 + 0.118 \approx 1.259$$

$$NDCG \approx DCG/IDCG = 1.259/1.472 \approx 0.856$$

Methods

Data

Data Collection and Processing

We gathered data for the project by making a web scraper for collecting predictions or results data from the Racing Post website. Here's an overview of the key components:

1. ScraperClass:
   This main class manages the scraping process and subclasses the pred_writing and ResultsScraper classes. The constructor initialises the object with a specified date and boolean flags for scraping results or predictions.
   a. scrape(self):
      This method initiates the scraping process based on the specified flags. It calls the appropriate methods from the respective subclasses to scrape and write data to a CSV file.

2. ResultsScraper:
   Handles the scraping and writing of results data. Methods include:
   a. scrape_website(self, url, date) for extracting relevant information from webpages;
   b. write_to_csv(self, url, date) for writing results data to a CSV file;
   c. add_to_csv(self, url, date) for appending additional data, and;
   d. get_place_urls(self) to retrieve race location URLs.

e. pred_writing: Focuses on scraping and writing predictions data. The uploading_to_csv(self) method initiates the scraping process by retrieving race location URLs using getUrls(self, url). It passes the URLs to the predData class for writing data to a CSV file.

3. predData:
   Extracts predictions data from race pages using various methods for scraping jockey, trainer, horse information, and comments. It also includes methods for writing the scraped data to a CSV file.

The code was designed to run on a Replit server using BeautifulSoup, Google Chrome and Selenium. BeautifulSoup is a python library for web scraping, and Selenium is a widely used web testing framework that can be employed for web scraping purposes. In this specific implementation, Selenium was used to navigate the Racing Post website, simulate user actions such as clicking on elements, and extract the necessary data for both predictions and results.

Data Points

We have gathered a wide range of data points from the Racing Post website, employing our web scraping techniques earlier. These data points serve as the foundation for our analyses and predictions. Thay are described as following:

Date: The date of the race.
Time: The time of the race.
Race Location: The location where the race is taking place.
Going: The ground condition or track surface for the race (e.g., Soft, Good, Firm).
Horse Name: The name of the horse participating in the race.
Horse Age: The age of the horse in miles per hour.
Horse Top Speed: The highest recorded speed of the horse.
Horse RPRS Ranking: The ranking of the horse based on Racing Post Rating Standard (RPRS).
Horse ORS Ranking: The ranking of the horse based on Official Rating Standard (ORS).
Horse Weight: The weight of the horse in kilograms.

Prediction: The predicted outcome or performance of the horse in the for a comment from an expert.
Jockey Last 14: The performance of the jockey in the last 14 days.
Trainer Last 14: The performance of the trainer in the last 14 days.

Embeddings and Model Implementation

TFIDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic used to evaluate the importance of a word in a collection of documents. Here's a general description of the implementation of TF-IDF using the provided code:

1. TF-IDF Transformation: The add_tf_idf function implements the TF-IDF transformation using the TfidfVectorizer class from the scikit-learn library. It fits and transforms the "Prediction" column, generating TF-IDF representations for each document (prediction sentence).
2. Feature Selection: The TF-IDF values for each word in the vocabulary are summed across all documents to obtain the combined TF-IDF values. The n words with the highest TF-IDF values are selected for feature representation.
3. Bag-of-Words (BOW) Encoding: The CountVectorizer class from scikit-learn is used to perform BOW encoding using the selected highest TF-IDF words as the vocabulary. It transforms the "Prediction" column into a matrix of word counts.
4. DataFrame Update: The BOW-encoded features are concatenated with the original DataFrame using pd.concat(). The resulting DataFrame (df) includes the TF-IDF and BOW features.
5. Data Export: The final DataFrame is saved as a CSV file named "tf_idf_data.csv".

Overall, the implementation performs TF-IDF encoding to represent the prediction sentences as numerical features, which can be used for further analysis or modelling tasks.

BERT

We implement BERT embeddings by utilising the BERTModel class from the Hugging Face transformers library. Here's a brief description of the code:

1. Tokenization: The input text is tokenized using the specified tokenizer (self.tokenizer) to obtain a list of tokens.
2. Input Formatting: Special tokens such as [CLS], [SEP], and padding tokens ([PAD]) are added to the token list to create a standardised input sequence.
3. BERT Model: The BERT model (self.bert) is initialised with the specified configuration (config). It takes the tokenized input (input_ids) and attention mask (attention_mask) as input and returns the contextualised representations.
4. Additional Layers: The code adds additional layers on top of BERT to incorporate other features or customise the output. For example, numerical features are concatenated with the BERT embeddings using torch.cat().
5. Model Output: The output of the model is obtained by passing the concatenated features through the additional layers. The final output can be used for different downstream tasks, such as regression or classification.

In our code we showcase two examples: BERT_Pointwise for pointwise prediction tasks and BERT_Pairwise for pairwise comparison tasks. Both classes inherit from BertPreTrainedModel, which is a base class for fine-tuning BERT models.

Overall, our code demonstrates the integration of BERT embeddings into a larger model architecture, allowing for the combination of textual and numerical features for various natural language processing tasks.

Models Used

Pointwise

Method for Pointwise Regression Model:
The Pointwise Regression model is designed to predict the position of a horse based on its embeddings and is used in both BERT and TFIDF. The method can be outlined as follows:

1. Data Preprocessing:

   The horse embeddings are loaded from a CSV file and stored in a DataFrame. Irrelevant columns such as 'horse_race_id' and 'Race_Id' are dropped.
2. Dataset Preparation:

   A custom dataset, called PointwiseDataset, is created by subclassing torch.utils.data.Dataset. The dataset takes the preprocessed DataFrame as input. It implements the len method to return the dataset length and the getitem method to retrieve a sample, extracting input features and target values.
3. Train-Validation-Test Split:

   The dataset is split into train, validation, and test sets using an 80-20 ratio. The resulting sets are converted into instances of the PointwiseDataset class.
4. Model Architecture: The Pointwise model is defined as a subclass of nn.Module. It consists of two linear layers: linear_1 and linear_2. The linear_1 layer takes input features of size emb_size and outputs hidden_layer_size. The ReLU activation function is applied. The linear_2 layer takes the output of linear_1 and produces a single output for regression.
5. Training Loop:

   The model is trained using the Adam optimizer and Mean Squared Error (MSE) loss function. The training loop iterates over a specified number of epochs. During each epoch:
   a. The model is set to training mode.
   b. The training data is fed to the model in batches using a DataLoader.
   c. Gradients are zeroed, predictions are obtained, loss is computed, and gradients are back propagated.
   d. The average loss for the epoch is calculated.
6. Validation Loop:

   The model is set to evaluation mode. The validation loop calculates the validation loss by iterating over the validation data using a DataLoader. The average validation loss is calculated and printed for each epoch.
7. Testing Loop:

   The model is set to evaluation mode. The testing loop calculates the test loss by iterating over the test data using a DataLoader. The average test loss is calculated and printed.

Pairwise

The pairwise classification method aims to predict the winning horse in every race by considering all possible combinations of pairs of horses. This approach involves several steps to preprocess the data, train the model, and evaluate its performance. The method can be summarised as follows:

1. Data Loading and Preparation:
   The horse race data is loaded into a DataFrame. Unnecessary columns are dropped, and a defaultdict is initialised to store the classification results. Unique race IDs in the data are identified.

2. Pairwise Comparison:
   For each race ID, the rows corresponding to that race are retrieved. For every pair of horses in the race, the differences between their feature values are calculated. The common information such as race ID and positions of the two horses are stored in the results dictionary.

3. Data Transformation:
   A new DataFrame is created using the collected results. A function is defined to determine the winning horse between a pair based on their positions. This function assigns a value of 1 if horse 1 wins and 0 if horse 2 wins. The DataFrame is then prepared by dropping unnecessary columns.

4. Dataset Preparation:
   A custom dataset class called Pairwise_Dataset is defined, inheriting from torch.utils.data.Dataset. This class implements the necessary methods (len and getitem) to retrieve features and targets for each sample. The DataFrame is split into train, validation, and test sets using the train_test_split function. The Pairwise_Dataset class is used to create the train, validation, and test datasets. Data loaders are created using torch.utils.data.DataLoader to efficiently load the datasets during training.

5. Model Definition:

The Pairwise model is defined by inheriting from nn.Module. The model consists of linear layers, activation functions (such as ReLU), and dropout layers to prevent overfitting. Hyperparameters for the model, including embedding size, hidden layer size, and dropout probability, are set. An instance of the Pairwise model is created.

6. Model Training:
   The number of epochs and learning rate are set for training. The loss function, such as negative log-likelihood loss (nn.NLLLoss), and optimizer, such as Adam (optim.Adam), are defined. The training loop is performed, iterating over the train data loader, calculating predictions, loss, and gradients. The model parameters are updated using the optimizer based on the computed gradients. The average loss for each epoch is recorded.

7. Model Evaluation:
   The validation loop is performed to evaluate the model on the validation set and calculate the validation loss. The model's performance is further assessed by performing the testing loop, evaluating the model on the test set and calculating the test loss. The average test loss is computed.

Evaluation

nDCG

The evaluation metric used is Normalized Discounted Cumulative Gain (nDCG). The code calculates nDCG scores for each model based on their predictions compared to the actual results of horse races. The method can be summarised as follows:

The code defines the file names for the predictions made by the pointwise and pairwise models.

1. Two functions are defined: dcg and nDCG. These functions are used to calculate the Discounted Cumulative Gain (DCG) and the normalized DCG (nDCG), respectively.

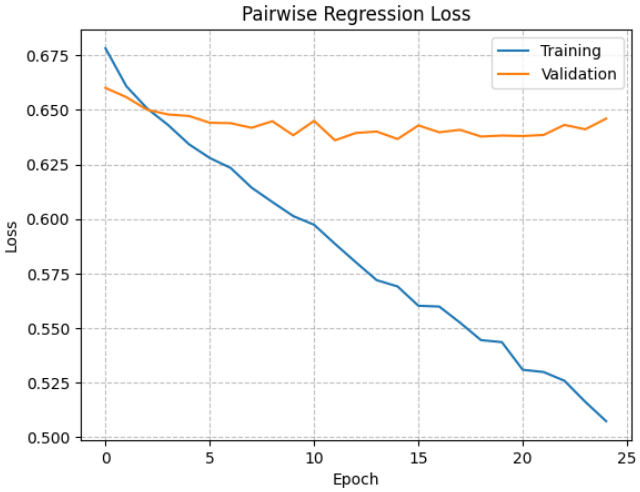
2. The calculateNDCG function takes a DataFrame (df), a flag indicating whether the cutoff point p is set to the default value or not (is_P_Default), and the cutoff point value (p). It iterates over the rows of the DataFrame to extract the predictions and actual results for each race.
3. The predictions and actual results are processed to remove unnecessary characters and split into lists.
4. The function calculates the nDCG score for each race by calling the nDCG function with the predicted and actual positions of the horses.
5. The nDCG scores for all races are stored in a list (ndcgs), which is returned by the calculateNDCG function.
6. The code reads the prediction data from the pointwise and pairwise files into separate DataFrames (df_PointWise and df_PairWise).
7. The first column of each DataFrame (index 0) is dropped since it contains unnecessary data.
8. The next step is to calculate the nDCG scores for the pointwise and pairwise models using the calculateNDCG function. The default value of p is used for this calculation.
9. The mean nDCG scores for each model are calculated using the statistics.mean function and printed to the console.
10. The code then sets the is_P_Default flag to False and assigns a value to p to evaluate the models by considering only the top p horses in each race.
11. The nDCG scores are calculated again with the updated parameters, and the mean scores are printed to the console.

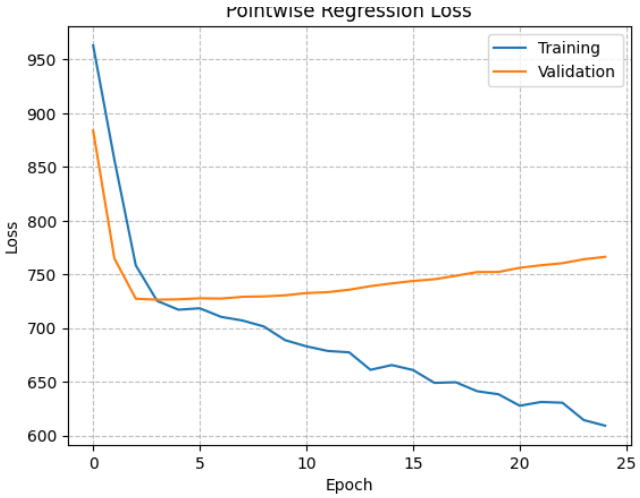
Overall, our code provides a way to evaluate the performance of the pointwise and pairwise models in predicting horse race outcomes using the nDCG metric. The evaluation is done using the provided prediction data, and the results are presented as mean nDCG scores for different scenarios.

Results

Behavior of training

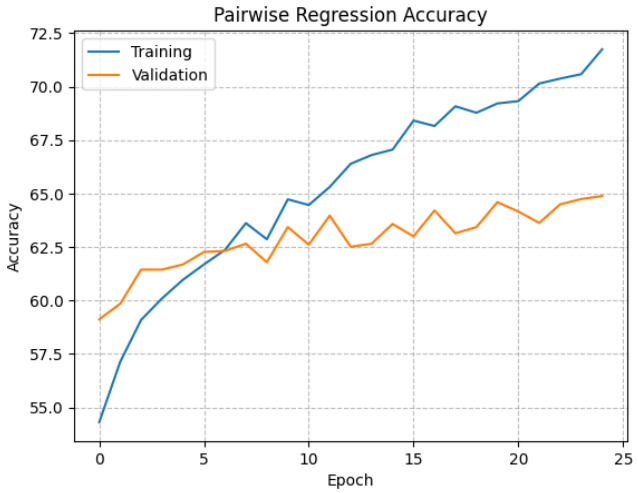
These graphical representations help visualize the behavior of training and validation losses and provide insights into how the model is learning and generalizing.





We can see here in the Pairwise regression and pointwise regression graphs are also overfitting after about 2 epochs. In the case of overfitting, the training loss continues to decrease, even reaching lower values, while the validation loss starts to increase or remains stagnant. The crossing point indicates the onset of overfitting, where the model starts to memorize the training data and fails to generalize well to new data.

The following graph will show the trend of pairwise regression accuracy over the epochs for both the training and validation datasets. It allows us to observe how the accuracy of the model changes during training and how it performs on unseen validation data.



In this scenario, the training and validation accuracy intersect, indicating a good balance between underfitting and overfitting. The accuracy continues to increase in a similar pattern as the model learns and improves.

NDCG scores

NDCG score calculated where all horses in the race are taken into account using predictions made by pointwise model: ~0.611

using predictions made by pairwise model: ~0.776

NDCG score calculated, where 3 horses in the race are taken into account using predictions made by pointwise model:
~0.308 using predictions made by pairwise model:
~0.593

Our results were conclusive. As can be seen in diagrams, our models had a maximum predicted winner rate of 77.6% when using a pairwise model with the entire available set of horses.

Conclusion

After analysing our results, it is clear that we can draw good results from our approach in our TF IDF pairwise approach. Our mistakes were likely caused by lack of data.

But again, despite our smaller data pool, our results were strong and suggest that our methods are working. Instead of just 3 month's worth of data, we can expand our web scraping to continue for a full year. Additionally, with our limited data, we were unable to effectively use BERT. Increasing our data size in future iterations could also allow us to use this and other embedding methods to improve our analysis on the horse racing predictions. Using the results we've collected and the lessons we've learned, we're planning on expanding our project and collecting more data to complete our goal of beating the bookies.

Bibliography

"Horse Racing Cards, Results & Betting | Racing Post." *Racingpost.com*, 2019, www.racingpost.com/.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova. *BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding*. May 2015, https://doi.org/10.48550/arXiv.1810.04805.

Silverman, Noah. "A HIERARCHICAL BAYESIAN ANALYSIS of HORSE RACING." *The Journal of Prediction Markets*, vol. 6, no. 3, 22 Jan. 2013, pp. 1–13, https://doi.org/10.5750/jpm.v6i3.590.

Wang, Yining, et al. *A Theoretical Analysis of NDCG Type Ranking Measures*. 24 Apr. 2013, https://doi.org/10.48550/arxiv.1304.6480. Accessed 15 June 2023.