Lab Sheet 3: Text generation with Logistic regression & RNNs.

Expected Lab Duration: ~2 hours

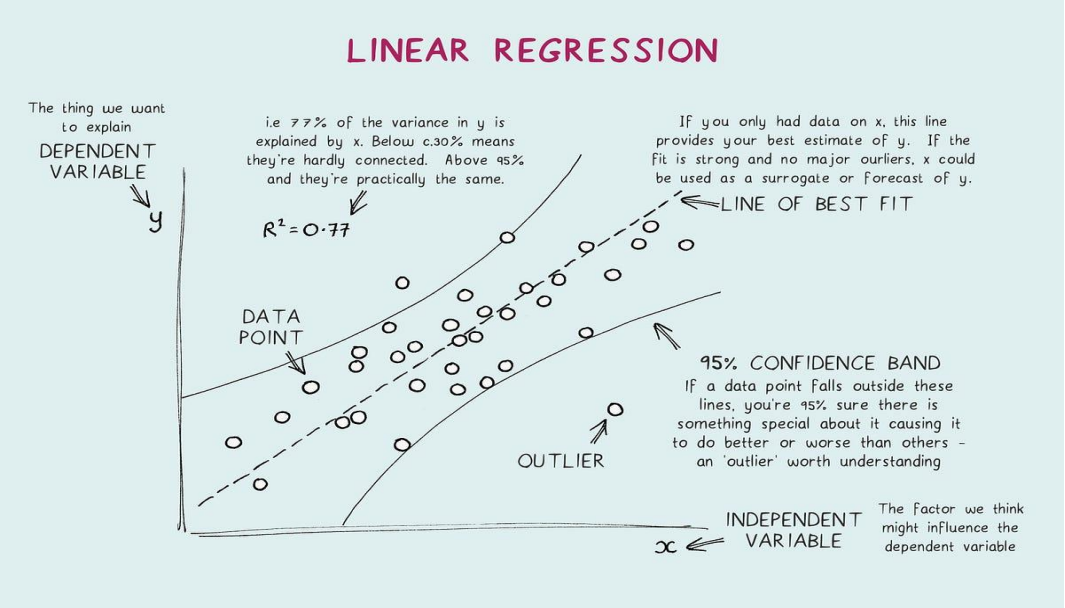
3.1 - Logistic regression, an introduction

Introduction to Logistic regression: "Read this paper if you want to learn logistic regression"

1. Linear Regression Recap:

Linear regression is a supervised learning algorithm used for predicting a continuous outcome (dependent variable) based on one or more predictor variables (features).

The linear regression model assumes a linear relationship between the input features and the output. Quick blog explaining linear regression.



2. Limitation for Classification:

While linear regression is suitable for regression problems, it is not directly applicable to classification problems where the goal is to predict a categorical outcome.

3. Binary Classification:

- Logistic regression is a type of regression analysis used for binary classification problems (two classes: 0 or 1).

- The logistic regression model predicts the probability that an instance belongs to a particular class.

4. Sigmoid (Logistic) Function:

- The key idea behind logistic regression is to transform the linear regression output using the sigmoid (logistic) function.
- The sigmoid function is defined as: $\sigma(z) = \frac{1}{1 + e^{-z}}$, where z is the linear combination of input features and weights. $z = b + w1x1 + w2x2 +$

5. Logistic Regression Model:

- The logistic regression model can be expressed as:
  $P(Y = 1|X) = \frac{1}{1 + e^{-(b+w1x1+w2x2+.)}}$
- $P(Y = 1|X)$ represents the probability that the output *Y* is 1 given the input features *X*.
- *b* is the bias term and, $w1, w2, w3, wn$ are the weights associated with the input features.

6. Decision Boundary:

- The logistic regression model predicts class 1 if $P(Y = 1|X) \geq 0.5$ and class 0 otherwise.

- The decision boundary is the line or hyperplane where $P(Y = 1|X) = 0.5$. It is determined by the weights and bias.

7. Training Logistic Regression:

- The model is trained by minimizing a cost function, often the cross-entropy loss, which measures the difference between predicted and actual probabilities.

- Optimization algorithms like gradient descent are used to find the optimal weights and bias.

8. Advantages:

- Logistic regression is computationally efficient, simple, and interpretable.

- It provides probabilities that can be used to make informed decisions.

9. Limitations:

- Assumes a linear relationship between features and the log-odds of the predicted probability.

- Sensitive to outliers.

10. Use Cases:

Commonly used in binary classification tasks such as spam detection, medical diagnosis, etc.

3.2 - Classifying text with Logistic regression

Problem Statement: IMDb Movie Review Sentiment Analysis

Background

The Internet Movie Database (IMDb) is a popular platform where users can rate and review movies. As a data scientist, your task is to build a sentiment analysis model that can predict whether a given movie review is positive or negative based on the text content.

Dataset

You have been provided with a dataset containing IMDb movie reviews. (Download it from here) Each review is labeled as either “positive” or “negative.” The dataset consists of the following features:

- **Text**: The actual movie review text.
- **Label**: The sentiment label (1 for positive, 0 for negative).

Objective

Your objective is to create a binary classification model using logistic regression that can accurately predict the sentiment of a movie review based on its text content.

Steps to Achieve the Objective

1. **Data Preprocessing**:
    - Clean and preprocess the text data by removing special characters, stopwords, and performing tokenization.
    - Convert the text into numerical representations (e.g., using TF-IDF or word embeddings).
2. **Feature Engineering**:

    - Extract relevant features from the text data (Bag of words, we will be using this!).
    - Create a feature matrix for training the model.
3. **Model Building**:
    - Split the dataset into training and validation sets.
    - Train a logistic regression model on the training data.
    - Tune hyperparameters (e.g., regularization strength) using cross-validation.
4. **Model Evaluation**:
    - Evaluate the model's performance on the validation set using metrics such as accuracy, precision, recall, and F1-score.
    - Analyze the confusion matrix to understand false positives and false negatives.
5. **Model Deployment**:
    - Once satisfied with the model's performance, deploy it for real-world predictions.
    - Provide an API or interface for users to input movie reviews and receive sentiment predictions.

Evaluation Metrics

Choose appropriate evaluation metrics based on the problem context. Commonly used metrics for binary classification include accuracy, precision, recall, F1-score, and area under the receiver operating characteristic curve (AUC-ROC).

Challenges

- **Imbalanced Classes**: The dataset may have an imbalance between positive and negative reviews.
- **Feature Extraction**: Choosing the right features from the text data is crucial.
- **Overfitting**: Regularization techniques should be applied to prevent overfitting.

3.4 - Exercise

Complete whatever section is marked with **#todo** with lines of code. Make sure to debug any errors in the code. The final code should be submitted in your notebook files.

Imports

```
import re
import numpy as np
from collections import Counter
```

Function to preprocess text data

```
def preprocess_text(text):
    # todo: Convert text to lowercase
    # todo: Remove special characters and numbers
    return text
```

Function to create a Bag of Words representation

```
def create_bow(corpus):
    word_counts = Counter()
    for doc in corpus:
        words = doc.split()
        word_counts.update(words)
    word_to_index = {word: idx for idx, (word, _) in
enumerate(word_counts.items())}
    return word_to_index
```

Function to convert text data into Bag of Words features

```
def text_to_bow(text, word_to_index):
    bow_vector = np.zeros(len(word_to_index))
    words = text.split()
    for word in words:
        if word in word_to_index:
            bow_vector[word_to_index[word]] += 1
    return bow_vector
```

Logistic Regression Model

```
class LogisticRegression:
    def __init__(self, learning_rate=0.01, epochs=1000):
        self.learning_rate = learning_rate
        self.epochs = epochs

    def sigmoid(self, z):
        #todo: return sigmoid of z

    def fit(self, X, y):
        m, n = X.shape
        self.weights = np.zeros(n)
        for epoch in range(self.epochs):
            z = np.dot(X, self.weights)
            predictions = self.sigmoid(z)
            gradient = np.dot(X.T, (predictions - y)) / m
            #todo: update self.weights corresponding to lr and gradient


    def predict(self, X):
        z = np.dot(X, self.weights)
        predictions = self.sigmoid(z)
        return np.round(predictions)
```

Load the dataset

```
import pandas as pd
df = pd.read_csv('reviews.csv')
```

Preprocess the text data

```
df['review'] = df['review'].apply(preprocess_text)
```

Map 'positive' to 1 and 'negative' to 0

```
df['sentiment'] = df['sentiment'].map({'positive': 1, 'negative': 0})
```

Split the data into training and validation sets (80-20 split)

```
train_size = int(0.8 * len(df))
train_data, val_data = df[:train_size], df[train_size:]
```

Create Bag of Words representation for training set

```
word_to_index = create_bow(train_data['review'])
X_train = np.array([text_to_bow(text, word_to_index) for text in
train_data['review']])
y_train = train_data['sentiment'].values
```

Create Bag of Words representation for validation set

```
#todo
```

Train the logistic regression model

```
lr_model = LogisticRegression(learning_rate=0.01, epochs=1000)
lr_model.fit(X_train, y_train)
```

Evaluate the model on the validation set

```
predictions = lr_model.predict(X_val)
```

Calculate accuracy

```
accuracy = np.mean(predictions == y_val)
print(f'Accuracy: {accuracy:.2%}')
```

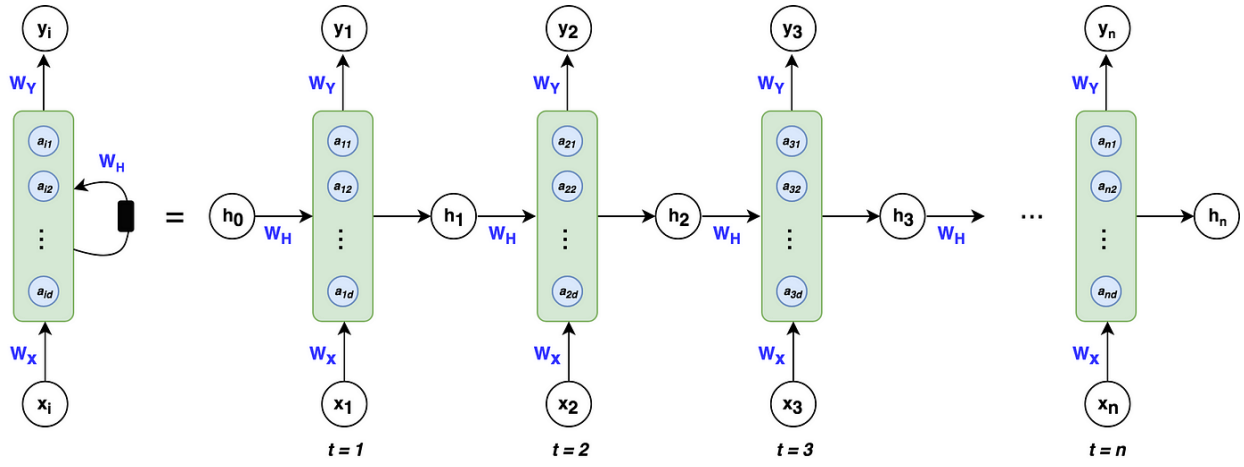
3.5 - Introduction to Recurrent Neural Networks (RNNs)

What are RNNs?

Recurrent Neural Networks (RNNs) are a type of neural network specifically designed for processing **sequential data**. Unlike traditional feedforward neural networks, which handle fixed-size inputs and produce fixed-size outputs, RNNs can handle **variable-length sequences** as both inputs and outputs. This makes them particularly useful for tasks involving text, time series data, and other sequential information.

How do RNNs work?

1. **The Basics:**
    - At the heart of an RNN is a hidden state vector (**h**), which evolves over time as the network processes each element in the sequence.
    - The key idea is that the hidden state at time step **t** depends not only on the current input (x_t) but also on the previous hidden state(h_{t-1}).
    - RNNs use the **same set of weights** for each time step, making them **recurrent**.



2. **Math Behind RNNs:**
    - Let's consider a simple "many-to-many" RNN with input vectors x_0**,**x_1 **, …,** x_n and output vectors $y_0, y_1, , y_n$.
    - The hidden state at time step **t** is calculated as follows:
        - $(h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h))$
            - $(W_{xh}) and (W_{hh})$ are weight matrices for input-to-hidden and hidden-to-hidden connections.
            - (b_h) is the bias term.
            - $(\tanh)$ is the hyperbolic tangent activation function.
    - The output at time step **t** is then computed using the hidden state:
        - $(y_t = W_{hy}h_t + b_y)$
            - (W_{hy}) is the weight matrix for hidden-to-output connections.
            - (b_y) is the output bias term.

3. **Applications of RNNs:**
    - **Machine Translation:** RNNs are used for translating text from one language to another (e.g., Google Translate).
    - **Sentiment Analysis:** RNNs can classify text sentiment (positive/negative) based on input sequences.
    - **Speech Recognition:** RNNs process audio sequences to recognize spoken words.
    - **Time Series Prediction:** RNNs can predict future values in time series data.

4. **Building an RNN from Scratch:**
    - You can create a basic RNN using Python and libraries like **numpy**.
    - Preprocess your data (tokenize, encode, etc.).
    - Initialize weight matrices and biases.
    - Implement the forward pass equations mentioned earlier.
    - Train your RNN using gradient descent and backpropagation.
    - Evaluate its performance and generate text!

Next Steps:

- **Learn More:**
    - Explore more advanced RNN architectures like **LSTM (Long Short-Term Memory)** and **GRU (Gated Recurrent Unit)**.
    - Dive into libraries like **TensorFlow**, **Keras**, or **PyTorch** for building powerful RNN models.

References:

1. An Introduction to Recurrent Neural Networks for Beginners[1]
2. DataCamp Tutorial on RNNs[2]
3. Coursera Course on CNNs and RNNs[3]
4. Machine Learning Mastery: Introduction to RNNs[4]

3.6 - Exercise: Generating text with RNNs!

Problem Statement: Shakespearean Text Generation with Recurrent Neural Networks (RNNs)

Background:

William Shakespeare, the renowned playwright and poet, left behind a rich legacy of literary works. His plays, sonnets, and poems are celebrated for their eloquence, wit, and timeless themes. As an aspiring data scientist, you've been tasked with creating an AI model that can generate text in the style of Shakespeare.

Task:

Your goal is to train a Recurrent Neural Network (RNN) on the popular Shakespeare dataset, which contains a collection of Shakespearean texts. The trained model should be able to generate new text passages that mimic the Bard's distinctive language and poetic flair.

Dataset:

The Shakespeare dataset consists of various plays, sonnets, and other writings attributed to William Shakespeare. Each line of text is a sequence of words, punctuations, and special characters. You'll preprocess this dataset to create input sequences for training the RNN.

Model Architecture:

Design and implement an RNN architecture for text generation. Do not consider using Long Short-Term Memory (LSTM) cells or Gated Recurrent Units (GRUs) to capture long-range dependencies in the text, for simplicity's sake. Experiment with different hyperparameters, such as the number of layers, hidden units, and sequence length.

Training Objective:

Train the RNN to predict the next word in a sequence given the preceding context. Use a cross-entropy loss function to optimize the model's predictions. Monitor the training process to prevent overfitting and ensure convergence.

Again, complete whatever section is marked with **#todo** with lines of code. Make sure to debug any errors in the code. The final code should be submitted in your notebook files.

```
import torch
import torch.nn as nn
import torch.optim as optim
import string
import random
```

Load and preprocess the Shakespeare dataset

```
with open('shakespeare.txt', 'r', encoding='utf-8') as file:
    data = file.read()
```

Create a mapping of unique characters to integers

```
chars = sorted(list(set(data)))
char_to_index = {char: i for i, char in enumerate(chars)}
index_to_char = #todo
```

Convert the text into integer sequences

```
int_data = [char_to_index[char] for char in data]
```

Define the RNN model

```
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN, self).__init__()
        self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x, hidden):
        x = self.embedding(x)
        output, hidden = self.rnn(x, hidden)
        output = self.fc(output)
        return output, hidden
```

Hyperparameters

```
input_size = len(chars)
hidden_size = 128
output_size = len(chars)
seq_length = 100  # Adjust as needed
learning_rate = 0.01
```

Initialize the model, loss function, and optimizer

```
rnn_model = #todo
criterion = #todo
optimizer = #todo
```

Training the RNN

```
num_epochs = 50  # Adjust as needed

for epoch in range(num_epochs):
    hidden_state = None
    for i in range(0, len(int_data) - seq_length, seq_length):
        inputs = torch.tensor(int_data[i:i+seq_length])
        targets = torch.tensor(int_data[i+1:i+seq_length+1])  # Predict the
next character

        optimizer.zero_grad()
        outputs, hidden_state = rnn_model(inputs, hidden_state)
        loss = criterion(outputs.view(-1, output_size), targets.view(-1))
        loss.backward()
        optimizer.step()

    if (epoch + 1) % 10 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')
```

Text generation using the trained RNN

```
def generate_text(model, seed, length=500):
    model.eval()
    generated_text = seed

    for _ in range(length):
        seed_encoded = torch.tensor([char_to_index[char] for char in seed])
        output, _ = model(seed_encoded, None)
        probabilities = torch.softmax(output[-1], dim=0).detach().numpy()
        predicted_index = random.choices(range(len(chars)),
weights=probabilities)[0]
        #todo: update seed and generated_text based on probabilities

    return generated_text
```

Generate and print text

```
seed_text = "shall i compare thee to a summer's day?\n"
generated_text = generate_text(rnn_model, seed_text)
print("\nGenerated Text:")
print(generated_text)
```