Politechnika Rzeszowska
Wydział Elektrotechniki i Informatyki

Sztuczna Inteligencja

Klasyfikator win w Burn

Projekt

28.01.2025 r.

Kamil Pomykała, 174725

Spis treści

1. Opis problemu

Klasyfikacja win, polegająca na przypisaniu konkretnego trunku do właściwej odmiany winogron na podstawie jego parametrów chemicznych, odgrywa ważną rolę w analizie jakości oraz unikalnych cech wina. Zróżnicowanie cech win pod względem składu chemicznego sprawia, że klasyfikacja win jest trudnym zadaniem.

Dzięki nowoczesnym rozwiązaniom, takim jak sztuczna inteligencja i algorytmy uczenia głębokiego, możliwe jest zautomatyzowanie procesu klasyfikacji win. Problem ten można sprowadzić do zadania wieloklasowej klasyfikacji, w którym model, analizując cechy takie jak zawartość alkoholu, poziom kwasowości czy intensywność barwy, jest w stanie rozpoznać odmianę winogron wykorzystaną do produkcji danego wina.

1.1. Istota problemu

Klasyfikacja win na podstawie ich składu chemicznego ma znaczenie zarówno dla producentów, jak i konsumentów. Dla producentów stanowi narzędzie umożliwiające kontrolę jakości, identyfikację charakterystycznych cech produktu oraz optymalizację procesów wytwarzania. Z kolei dla konsumentów wiedza o właściwościach wina pozwala na lepsze dopasowanie do indywidualnych preferencji smakowych.

Automatyzacja tego procesu za pomocą modeli sztucznej inteligencji nie tylko przyspiesza i ułatwia analizę, ale również otwiera nowe możliwości w zakresie badań nad różnorodnością win oraz ich właściwościami.

1.2. Technologiczne aspekty rozwiązania

Zastosowanie sztucznej inteligencji, w szczególności algorytmów uczenia maszynowego, stanowi kluczowy element w procesie automatycznej klasyfikacji win. Nowoczesne techniki, takie jak sieci neuronowe czy głębokie uczenie (Deep Learning), umożliwiają modelom analizowanie złożonych wzorców w danych chemicznych, które są trudne do wychwycenia przez tradycyjne metody.

Do implementacji tych sieci najczęściej wykorzystuje się popularne biblioteki, takie jak `PyTorch` i `TensorFlow`, które oferują szeroki zestaw narzędzi do budowy, trenowania i optymalizacji sieci neuronowych. Dodatkowo, `Burn`, biblioteka stworzona w języku `Rust`, staje się coraz bardziej popularną alternatywą do bibliotek w językach wysokopoziomowych, dzięki swojej wydajności i możliwościach języka niskopoziomowego.

1.3. Cel projektu

Celem projektu jest opracowanie modelu sztucznej inteligencji, który będzie w stanie automatycznie klasyfikować wina na podstawie ich cech chemicznych. Projekt zakłada stworzenie systemu wykorzystującego algorytmy głębokiego uczenia, który na podstawie danych, precyzyjnie przypisze wino do jednej z określonych odmian. Dążymy do opracowania modelu o wysokiej dokładności, który może zostać wykorzystany zarówno w badaniach naukowych, jak i w przemyśle winiarskim do automatycznej klasyfikacji produktów.

2. Opis części praktycznej

2.1. Uczenie modelu

Uczenie modelu w kontekście Deep Learningu polega na wykorzystaniu takich algorytmów sztucznej inteligencji jak gradientowe wstecznej propagacji (backpropagation), konwolucyjne sieci neuronowe (CNN), rekurencyjne sieci neuronowe (RNN), sieci neuronowe o długiej pamięci krótkotrwałej (LSTM) oraz transformatory, do rozpoznawania wzorców i zależności w danych.

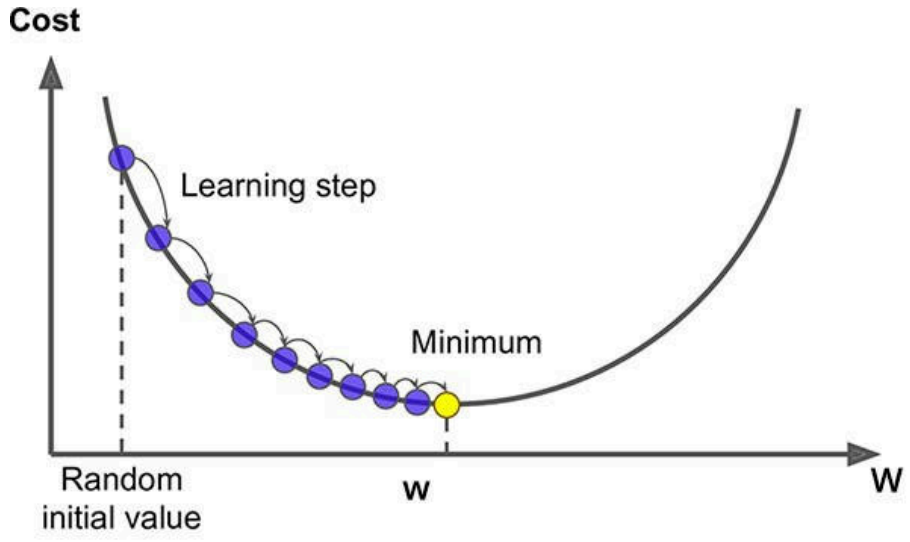
Deep Learning charakteryzuje się zdolnością do pracy z dużymi i złożonymi zestawami danych, wysoką skalowalnością oraz możliwością uczenia modeli o bardzo wielu parametrach, co pozwala na rozwiązywanie problemów wymagających znacznych mocy obliczeniowych i głębokiego modelowania nieliniowych zależności. Co istotne, techniki *Deep Learningu* sprawdzają się również w przypadku prostszych i dobrze przygotowanych zbiorów danych, gdzie dzięki swojej elastyczności potrafią efektywnie modelować struktury i wzorce w danych.

Uczenie modelu odbywa się w procesie zwanym treningiem, który polega na minimalizowaniu funkcji błędu. Funkcja ta mierzy, jak bardzo przewidywania modelu różnią się od rzeczywistych etykiet w danych. Podczas treningu sieć neuronowa iteracyjnie dostosowuje swoje parametry (*wagi i biasy*), aby jak najlepiej odwzorować prawdziwe dane wyjściowe.

2.1.1. Algorytmy optymalizacji

Proces optymalizacji w sieciach neuronowych opiera się najczęściej na zastosowaniu algorytmu **spadku gradientu** (*Gradient Descent*), który umożliwia modyfikację wag w celu minimalizacji błędu predykcji. Istnieje wiele wariantów tego algorytmu, takich jak *Stochastic Gradient Descent (SGD)*, *Mini-batch Gradient Descent*, a także bardziej zaawansowane metody, jak *Adam (Adaptive Moment Estimation)*, *RMSprop* czy *Adagrad*.

Każdy z tych wariantów oferuje różne podejścia do aktualizacji wag, co pozwala na przyspieszenie procesu uczenia oraz poprawę stabilności w zależności od charakterystyki danych i struktury modelu.



Rysunek 1: Ilustracja działania algorytmu spadku gradientu [1]

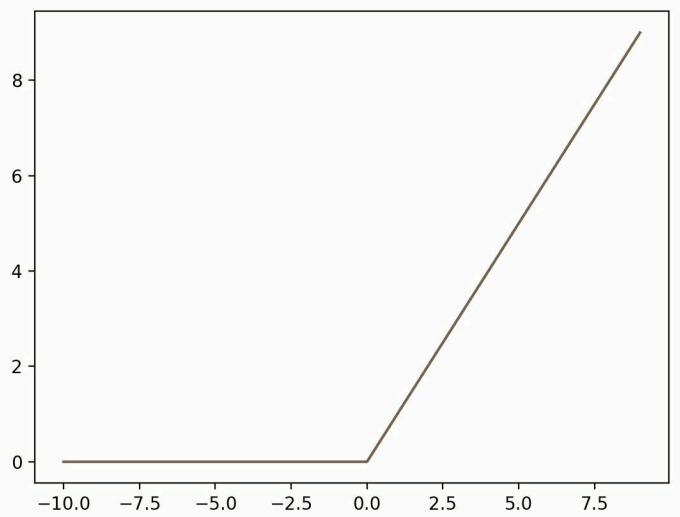
2.1.2. Funkcje aktywacji

Funkcje aktywacji odgrywają kluczową rolę w działaniu sieci neuronowych, wprowadzając nieliniowość do modelu i umożliwiając modelowanie bardziej złożonych zależności. Bez funkcji aktywacji sieci byłyby jedynie liniowymi modelami, co znacznie ograniczałoby ich zdolność do reprezentowania skomplikowanych wzorców.

Jedną z najczęściej stosowanych funkcji aktywacji jest **ReLU** (*Rectified Linear Unit*), która zwraca wartość wejściową, jeśli jest dodatnia, a zero w przeciwnym wypadku. Funkcja ta jest szybka w obliczeniach i skutecznie radzi sobie z problemem zanikania gradientów, co czyni ją podstawowym wyborem w wielu współczesnych modelach głębokiego uczenia.

$$f(x) = \max(0, x)$$

Równanie 1: Wzór funkcji ReLU

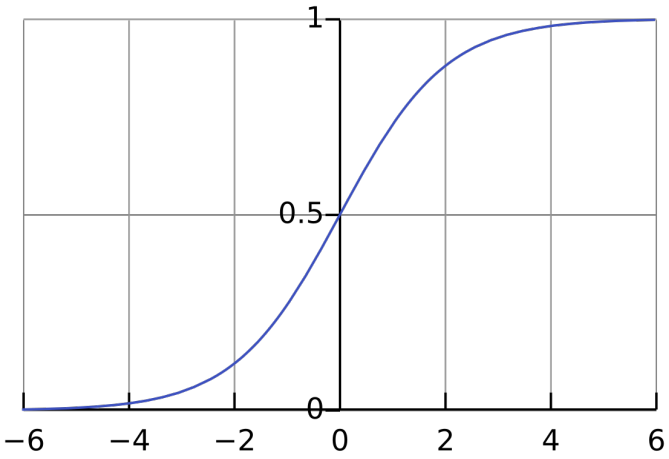


Rysunek 2: Wykres liniowy ReLU [2]

Inną popularną funkcją jest **sigmoid**, która przekształca wartości wejściowe na zakres od 0 do 1, dzięki czemu świetnie nadaje się do problemów binarnej klasyfikacji. Jednak jej zastosowanie w głębszych sieciach bywa ograniczone z powodu problemu zanikania gradientów, który może utrudniać efektywne uczenie.

$$f(x) = \frac{1}{1+e^{-x}}$$

Równanie 2: Wzór funkcji sigmoidalnej [3]

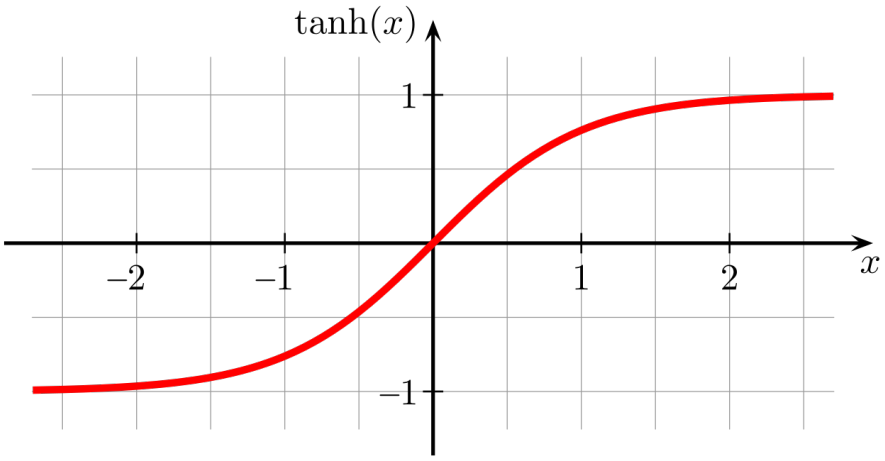


Rysunek 3: Wykres funkcji sigmoidalnej [3]

Podobnie funkcja **tanh** (*hiperboliczna tangens*) odwzorowuje wartości na zakres od −1 do 1, co sprawia, że jest bardziej symetryczna względem zera w porównaniu do sigmoid. To może prowadzić do szybszej zbieżności w niektórych przypadkach, ale również cierpi z powodu problemu zanikania gradientów w głębokich sieciach.

$$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Równanie 3: Wzór funkcji tanh



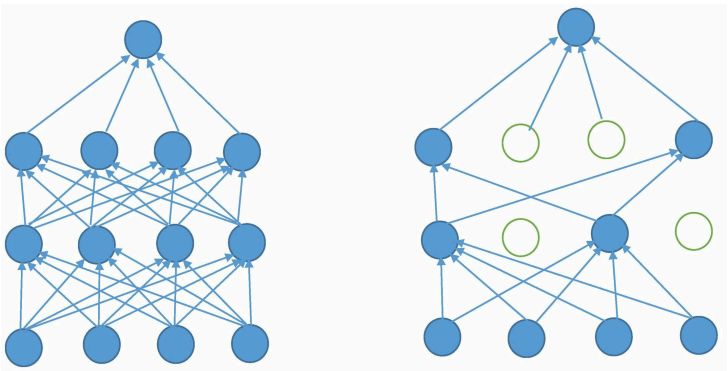
Rysunek 4: Wykres funkcji tanh [4]

Warto również wspomnieć o nowoczesnych wariantach funkcji aktywacji, takich jak **Leaky ReLU**, która jest modyfikacją ReLU i pozwala na przekazywanie niewielkiej wartości (np. 0.01x) dla ujemnych wejść, co zmniejsza problem martwych neuronów. Funkcja **ELU** (*Exponential Linear Unit*) dodatkowo łagodzi problem zerowych wartości w ujemnym zakresie, co może przyspieszać uczenie modeli.

2.1.3. Regularyzacja i unikanie przeuczenia

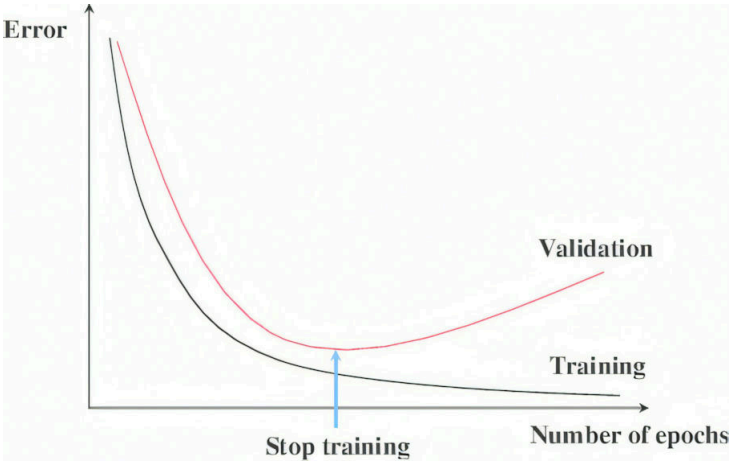
Podczas treningu model głębokiego uczenia stara się nauczyć ogólnych wzorców, które będą skuteczne nie tylko dla danych treningowych, ale również dla nowych, niewidzianych wcześniej przykładów. Proces ten, zwany generalizacją, jest kluczowy dla efektywności modelu w rzeczywistych zastosowaniach. Jednak modele głębokiego uczenia, ze względu na swoją wysoką złożoność, są narażone na ryzyko przeuczenia (*overfittingu*), gdzie model zapamiętuje szczegóły danych treningowych kosztem ogólnych wzorców.

Aby uniknąć przeuczenia, stosuje się techniki regularyzacji, które mają na celu kontrolowanie złożoności modelu i zapobieganie nadmiernemu dopasowaniu do danych treningowych. Jedną z najpopularniejszych metod regularyzacji jest ***dropout***, która polega na losowym wyłączaniu neuronów w trakcie treningu, co pozwala na zwiększenie odporności modelu na przeuczenie.



Rysunek 5: Schemat działania dropout. Sieć neuronowa bez (lewa strona) i z dropoutem (prawa strona) [5]

Innym popularnym podejściem jest ***early stopping***, które polega na monitorowaniu skuteczności modelu na zbiorze walidacyjnym i zatrzymywaniu treningu, gdy skuteczność przestaje rosnąć. Dzięki temu można uniknąć nadmiernego dopasowania modelu do danych treningowych i poprawić jego zdolność do generalizacji.



Rysunek 6: Schemat działania early stopping [6]

2.2. Nadzorowane uczenie

Uczenie nadzorowane to metoda uczenia maszynowego, w której model jest trenowany na danych wejściowych, które są już przypisane do odpowiednich klas (czyli wyników). W przypadku klasyfikacji win oznacza to, że dla każdego przykładu (wina) znamy już jego kategorię, czyli odmianę winogron, z której pochodzi. Model wykorzystuje te informacje, aby nauczyć się, jak przypisać nowe, nieznane dane do właściwej klasy.

W procesie uczenia modelu, sieć neuronowa analizuje cechy chemiczne wina (takie jak zawartość alkoholu, kwasowość, intensywność barwy itp.), a następnie na podstawie tych danych podejmuje decyzję, do której z trzech odmian winogron należy dane wino. Model jest trenowany na wcześniej oznaczonych danych (winach o znanych odmianach), a celem jest minimalizacja błędu klasyfikacji, aby przewidywania modelu były jak najdokładniejsze.

2.3. Ocena skuteczności modelu

Testowanie modelu polega na sprawdzeniu jego zdolności do poprawnego przypisywania nowych, niewidzianych wcześniej danych do odpowiednich klas. W tym przypadku model, który został wytrenowany na danych zawierających cechy chemiczne win, jest testowany na osobnym zbiorze testowym, który nie był używany w trakcie treningu. Celem jest ocena, jak dobrze model generalizuje swoje umiejętności na nowych danych, a nie tylko na tych, które były obecne w zestawie treningowym.

Podczas testowania modelu ocenia się jego skuteczność przy pomocy odpowiednich metryk, takich jak **dokładność** (*accuracy*), która wskazuje, jaka część wszystkich przewidywań była poprawna oraz **strata** (*loss*), która określa, jak dobrze model radzi sobie z minimalizacją błędów w klasyfikacji.

2.3.1. Dokładność

Dokładność modelu to procent poprawnych przewidywań w stosunku do wszystkich przewidywań. Im wyższa wartość dokładności, tym lepsza jest skuteczność modelu. Na przykład, jeśli model przewidział poprawnie 90 z 100 próbek, to jego dokładność wynosi 90%. Dokładność jest jedną z najczęściej stosowanych metryk w problemach klasyfikacji, ponieważ jest intuicyjna i łatwa do interpretacji.

$$\text{Accuracy} = \frac{\text{Liczba poprawnych przewidywań}}{\text{Liczba wszystkich próbek}}$$

2.3.2. Strata

Strata (*loss*) to wartość funkcji błędu, która mierzy, jak bardzo przewidywania modelu różnią się od rzeczywistych etykiet w danych.

Dla problemu klasyfikacji, jedną z najczęściej stosowanych funkcji *loss* jest ***cross-entropy loss***, która mierzy różnicę między przewidywaną a rzeczywistą rozkładem prawdopodobieństw dla poszczególnych klas. Im mniejsza wartość *loss*, tym lepsza jest skuteczność modelu, ponieważ oznacza to, że przewidywania modelu są bardziej zbliżone do rzeczywistych etykiet.

3. Analiza danych wejściowych

Analiza danych wejściowych to kluczowy etap w każdym projekcie uczenia maszynowego, który pozwala na zrozumienie struktury danych, ich właściwości oraz przygotowanie ich do dalszego przetwarzania i trenowania modelu. W tym przypadku celem analizy jest dokładne zrozumienie cech chemicznych win, które stanowią podstawę do klasyfikacji różnych odmian winogron.

3.1. Opis zbioru danych

Zbiór danych, który jest wykorzystywany w tym projekcie, pochodzi z analizy chemicznej win pochodzących z tego samego regionu we Włoszech, ale z trzech różnych odmian winogron. Zawiera on 13 cech chemicznych, które zostały zmierzone dla każdego z win, takich jak zawartość alkoholu, kwasowość, intensywność barwy oraz inne właściwości, które wpływają na charakterystykę wina. W zbiorze danych znajdują się również etykiety, które informują o rodzaju odmiany winogron, z której pochodzi dane wino.

Dokładnym źródłem danych jest *„PARVUS - An Extendible Package for Data Exploration, Classification and Correlation”*. Instytut Analiz Farmaceutycznych i Żywnościowych oraz Technologii, Genua, Włochy.

3.2. Struktura zbioru danych

Każdy wiersz w zbiorze danych reprezentuje jedno wino, a 13 kolumn odpowiada różnym cechom chemicznym, które zostały zmierzone w próbce. Wartości w tych kolumnach są liczbowe, co oznacza, że dane są już w formie numerycznej, gotowe do dalszego przetwarzania. Również warto wspomnieć, że nie ma brakujących danych w zbiorze, co ułatwia analizę.

Zbiór danych podzielony jest na 178 próbek, a liczby próbek dla poszczególnych odmian winogron to:

- Klasa 1: 59 próbek - (33.1%)
- Klasa 2: 71 próbek - (39.9%)
- Klasa 3: 48 próbek - (27.0%)

Atrybuty chemiczne, które zostały zmierzone dla każdego wina, to:

1. **Alkohol** – Zawartość alkoholu w winie, wyrażona jako procent objętości alkoholu w stosunku do całkowitej objętości płynu. Jest to jeden z kluczowych wskaźników wpływających na smak i charakterystykę wina.
2. **Kwas jabłkowy** – Zawartość kwasu jabłkowego w winie. Jest to naturalny kwas organiczny, który wpływa na smak wina, nadając mu kwaskowatość. Wina o wyższej zawartości tego kwasu mogą być bardziej kwasowe i orzeźwiające.
3. **Popiół** – Zawartość popiołu w winie, który jest pozostałością po spaleniu organicznych składników w winie. Zawartość popiołu może mieć wpływ na mineralność wina.
4. **Zasadowość popiołu** – Określa poziom zasadowości popiołu w winie. Zasadowość wpływa na pH wina i może oddziaływać na smak, zwłaszcza w kontekście równowagi kwasowości i słodyczy.
5. **Magnez** – Zawartość magnezu w winie. Magnez jest jednym z minerałów, który wpływa na smak wina, a jego obecność może wpływać na ogólną jakość i strukturę wina.
6. **Całkowita zawartość fenoli** – Fenole są naturalnymi związkami organicznymi występującymi w winie, które odpowiadają za smak, zapach oraz właściwości zdrowotne wina. Ich zawartość może wpływać na goryczkę oraz astringencję wina.
7. **Flawanoidy** – Jest to grupa fenoli, która wpływa na smak, zapach i kolor wina. Flawanoidy są również antyoksydantami, co ma znaczenie dla trwałości wina.
8. **Fenole nieflawanoidowe** – Inna grupa fenoli, która nie należy do flawanoidów, ale również wpływa na smak i zapach wina, nadając mu specyficzne cechy organoleptyczne.
9. **Proantocyjaniny** – Związki odpowiedzialne za kolor wina, zwłaszcza czerwonych win. Są to silne antyoksydanty, które również wpływają na strukturę smaku i astringencję.
10. **Intensywność koloru** – Określa głębokość koloru wina, co jest ważnym atrybutem w przypadku win czerwonych i białych. Intensywność koloru może wskazywać na dojrzałość wina oraz jego skład chemiczny.
11. **Odcień** – Dotyczy barwy wina, który może się różnić w zależności od odmiany winogron, procesu fermentacji i starzenia wina. Odcień jest kluczowym elementem oceny jakości wina.
12. **OD280/OD315 rozcieńczonych win** – Stosunek absorbancji przy długości fali 280 nm do 315 nm, który jest wykorzystywany do oceny jakości i zawartości substancji organicznych w winie, takich jak fenole.

13. **Prolina** – Aminokwas występujący w winie, który wpływa na strukturę wina oraz jego stabilność. Zawartość proliny może mieć wpływ na właściwości smakowe i chemiczne wina.

4. Przygotowanie danych do nauki

Odpowiednie przygotowanie danych ma na celu zapewnienie, że model będzie w stanie uczyć się skutecznie i osiągać jak najlepsze wyniki. W tym etapie dokonuje się kilku istotnych operacji, takich jak czyszczenie danych, normalizacja, dodanie brakujących wartości, podział na zbiory treningowe i testowe, a także inne techniki, które pomagają w dostosowaniu danych do wymagań modelu.

Poniżej przedstawię kroki, które ja podjąłem w celu przygotowania danych do trenowania modelu klasyfikacji win.

4.1. Czyszczenie danych

Pierwszym krokiem w przygotowaniu danych jest ich czyszczenie, czyli usunięcie zbędnych informacji, brakujących wartości czy duplikatów. W przypadku zbioru danych, który został wykorzystany w tym projekcie, nie było potrzeby usuwania żadnych próbek. Każdy atrybut może potencjalnie wpłynąć na klasyfikację wina, dlatego nie ma potrzeby eliminacji.

Jedynym ważnym aspektem była zmiana etykiet klas, aby numeracja zaczynała się od 0, co jest wymagane przez bibliotekę `Burn`. Wcześniej etykiety klas były oznaczone jako 1, 2 i 3, co zostało zmienione na 0, 1 i 2.

Bez tej zmiany mój model miał problem z uczeniem się. Uzyskałem zaledwie 70% dokładności. Po zmianie etykiet klas na 0, 1 i 2 model od razu zaczął osiągać wyniki na poziomie 95% dokładności.

4.2. Normalizacja danych

Normalizacja danych jest ważnym krokiem w przypadku modeli, które wykorzystują algorytmy uczenia maszynowego, ponieważ pozwala na ujednolicenie skali wartości atrybutów. W przypadku zbioru danych z cechami chemicznymi win, wartości poszczególnych atrybutów różnią się znacząco, co może wpłynąć na proces uczenia modelu.

Np. zawartość alkoholu w winie mieści się w zakresie od 11.0% do 14.8%, podczas gdy zawartość kwasu jabłkowego waha się od 1.74 do 4.23. Aby uniknąć problemów związanych z różnicą w skali wartości, zastosowałem normalizację.

W procesie normalizacji dane są przeskalowane w taki sposób, aby mieściły się w określonym zakresie. W moim przypadku od -1 do 1.

$$x_{\text{norm}} = \frac{2*(x - x_{\text{min}})}{x_{\text{max}} - x_{\text{min}}} - 1$$

Wzór 4: Normalizacji danych do zakresu [-1, 1]

```rust
for (i, row) in x.iter().enumerate() {
  for (j, &value) in row.iter().enumerate() {
    if max[j] - min[j] == 0.0 {
      x_norm[i][j] = 0.0; // Avoid division by zero
    } else {
      // Normalize to range [-1, 1]
      x_norm[i][j] = -1.0 + 2.0 * (value - min[j]) / (max[j] - min[j]);
    }
  }
}
```

Program 1: Normalizacja danych

4.3. Podział na zbiory treningowe i testowe

Podział danych na zbiory treningowe i testowe jest kluczowym elementem w procesie uczenia maszynowego, ponieważ pozwala na ocenę skuteczności modelu na nowych, niewidzianych wcześniej danych. W moim przypadku zdecydowałem się na podział danych w stosunku 80% do 20%, gdzie 80% danych zostało wykorzystane do treningu, a 20% do testowania. W późniejszych etapach zmieniałem ten podział w celach eksperymentów (Sekcja 6).

Podział danych na zbiory treningowe i testowe pozwala na ocenę skuteczności modelu na nowych, niewidzianych wcześniej danych. W ten sposób można sprawdzić, jak dobrze model generalizuje swoje umiejętności na nowych danych, a nie tylko na tych, które były obecne w zestawie treningowym.

W tym celu napisałem funkcję `split_data`, która dokonuje losowego podziału danych na zbiory treningowe i testowe. Każda klasa jest losowana osobna, aby zachować równowagę między klasami w obu zbiorach.

```rust
fn split_data(
  x: Vec<Vec<f32>>,
  y_t: Vec<i32>,
) -> (Vec<Vec<f32>>, Vec<i32>, Vec<Vec<f32>>, Vec<i32>) {
  let class_counts = get_class_count(&y_t);
  let mut x_train = Vec::new();
  let mut y_t_train = Vec::new();
  let mut x_test = Vec::new();
  let mut y_t_test = Vec::new();

  let wines = x.iter().zip(y_t.iter()).collect::<Vec<_>>();

  for (class, count) in class_counts.iter() {
    let num_train = (count * 85) / 100;

    let mut class_wines = wines.iter().filter(|(_, &y)| y == *class).collect::<Vec<_>>();
    class_wines.shuffle(&mut thread_rng());

    let (test, train) = class_wines.split_at(num_train as usize);

    for (x, y) in train.into_iter() {
      x_train.push(x.clone().clone());
```

```
23          y_t_train.push(y.clone().clone());
24      }
25
26      for (x, y) in test.iter() {
27          x_test.push(x.clone().clone());
28          y_t_test.push(y.clone().clone());
29      }
30  }
31
32  (x_train, y_t_train, x_test, y_t_test)
33 }
```

Program 2: Podział danych na zbiory treningowe i testowe

4.4. Zapis danych do plików

Tak przygotowane dane zostały zapisane do plików w formacie PKL. Do zserializowania danych wykorzystałem bibliotekę `serde-pickle` z pakietu `serde`, która pozwala na serializację i deserializację danych w formacie PKL. Dane zostały zapisane w postaci czterech plików: `x_train.pkl`, `y_t_train.pkl`, `x_test.pkl` i `y_t_test.pkl`.

```rust
/// Save `x` and `y_t` to disk in pickle format.
fn dump_to_pkl(x: Vec<Vec<f32>>, y_t: Vec<i32>, prefix: &str) {
    // Create BTreeMaps to serialize
    let mut x_map = BTreeMap::new();
    let mut y_map = BTreeMap::new();
    x_map.insert(String::from("x"), x);
    y_map.insert(String::from("y_t"), y_t);

    // Serialize to pickle format
    let x_serialized = serde_pickle::to_vec(&x_map, Default::default()).unwrap();
    let y_serialized = serde_pickle::to_vec(&y_map, Default::default()).unwrap();

    // Save to disk
    std::fs::write(format!("./data/x_{}.pkl", prefix), &x_serialized).unwrap();
    std::fs::write(format!("./data/y_t_{}.pkl", prefix), &y_serialized).unwrap();
}
```

Program 3: Funkcja zapisująca dane do plików PKL

4.5. Podsumowanie

Przygotowanie danych do trenowania modelu klasyfikacji win wymagało kilku istotnych kroków, takich jak czyszczenie, normalizacja, podział na zbiory treningowe i testowe oraz zapis do plików. Dzięki tym operacjom dane są gotowe do dalszego przetwarzania i trenowania modelu.

y_t	alcohol	malic acid	ash	alcalinity of ash
0	0.67894757	-0.6205534	0.0053474903	-0.41237122
0	-0.04210496	-0.66007906	0.24064171	-0.25773203
0	0.72105277	-0.5335969	0.4545455	-0.030927837
0	0.3052634	-0.58102775	0.3796792	-0.13402063
0	0.62631583	-0.7075099	0.026737928	-0.36082482
0	0.063158035	-0.6086957	-0.2727273	-0.8144331

Rysunek 7: Tabela z częścią danych przygotowanych do trenowania modelu

5. Trenowanie modelu

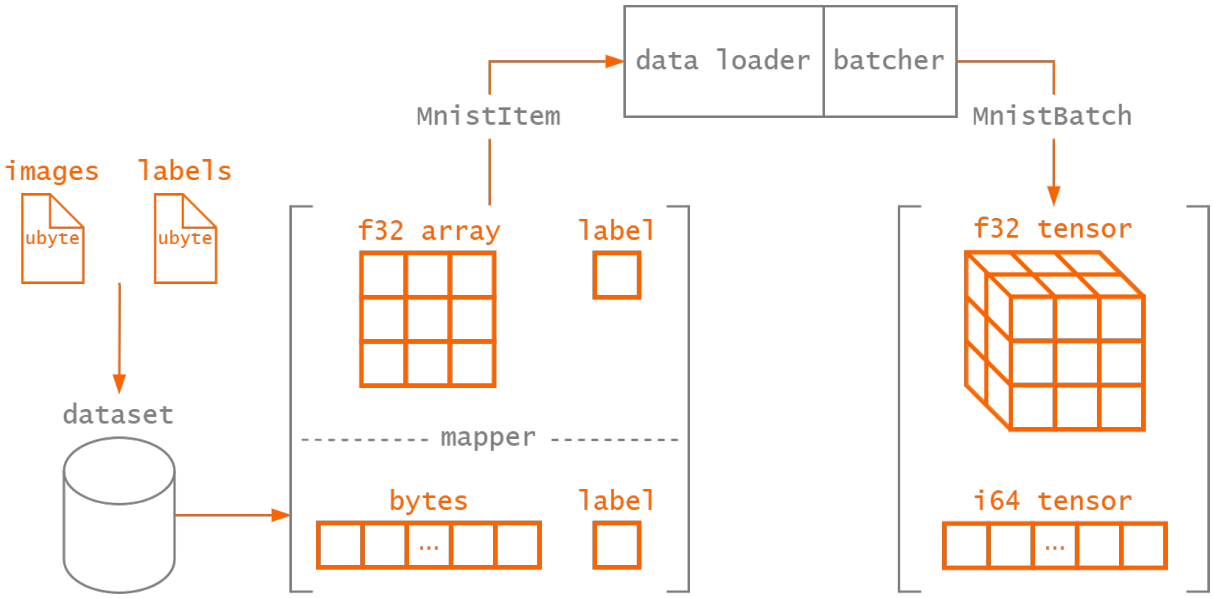
Po przygotowaniu danych przyszedł czas na trenowanie modelu. W tym projekcie zdecydowałem się na wykorzystanie biblioteki `Burn` do implementacji modelu, jak i jego trenowania.

Przed samym przystąpieniem do trenowania modelu, potrzebne jest zdefiniowanie architektury sieci neuronowej, przygotowanie danych treningowych i testowych oraz wybór odpowiednich hiperparametrów, takich jak współczynnik uczenia, liczba epok czy rozmiar wsadu (*batch size*).

5.1. Przygotowanie danych

Trenowanie modelu w bibliotece `Burn` wymaga tzw. *Batcherów* (plik `batcher.rs` w części skryptowej), które są odpowiedzialne za dostarczanie danych do modelu w postaci wsadów (*batches*). *Batcher* przyjmuje strukturę *Dataset* (plik `dataset.rs`, w części skryptowej) jako dane wejściowe i dzieli je na wsady o określonym rozmiarze.

W moim przypadku, struktura `WineDataset` przechowuje w pamięci systemowej dane treningowe i testowe, które zostały wcześniej przygotowane i zapisane do plików PKL. Następnie, `WineBatcher` dzieli te dane na wsady `WineBatch`, które są przekazywane do modelu podczas treningu.



Rysunek 8: Schemat przepływu danych w procesie trenowania modelu [7]

Ważnym elementem w trenowaniu modelu jest implementacja tensorów, które są podstawowymi strukturami danych w bibliotece `Burn`. Tensor reprezentuje wielowymiarowy wektor lub macierz, który jest przechowywany w pamięci urządzenia. W przypadku mojego programu tensory tworzone są w funkcji `batch` w strukturze `WineBatcher`. Przechowują one dane wejściowe i wyjściowe dla modelu, które są przekazywane do sieci neuronowej.

Pierwsze tensor jest tworzony dla każdego wina z danych treningowych, a drugi tensor przechowuje etykietę klasy dla tego wina. Następnie, te tensory są łączone w jeden tensor dla danych wejściowych i wyjściowych, który jest przekazywany bezpośrednio do modelu.

```rust
impl<B: Backend> Batcher<WineItem, WineBatch<B>> for WineBatcher<B> {
  fn batch(&self, data: Vec<WineItem>) -> WineBatch<B> {
    let x = data
      .iter()
      .map(|wine| {
        Tensor::<B, 2>::from_data(
          [[
            wine.alcohol,
            wine.malic_acid,
            wine.ash,
            wine.alcalinity_of_ash,
            wine.magnesium,
            wine.total_phenols,
            wine.flavanoids,
            wine.nonflavanoid_phenols,
            wine.proanthocyanins,
            wine.color_intensity,
            wine.hue,
            wine.od280_od315_of_diluted_wines,
            wine.proline,
          ]],
```

```
            &self.device,
          )
        })
        .collect::<Vec<_>>();

    let y = data
      .iter()
      .map(|wine| Tensor::<B, 1, Int>::from_data([wine.class], &self.device))
      .collect::<Vec<_>>();

    let x = Tensor::<B, 2>::cat(x, 0).to_device(&self.device);
    let y = Tensor::<B, 1, Int>::cat(y, 0).to_device(&self.device);

    WineBatch { x, y }
  }
}
```

Program 4: Implementacji funkcji `batch`, której wymaga struktura `WineBatcher`

5.2. Definicja modelu

Model sieci neuronowej jest zdefiniowany w strukturze `WineModel` (plik `model.rs` w części skryptowej). Składa się z trzech warstw sieci neuronowej: jednej warstwy wejściowej, jednej warstwy ukrytej i jednej warstwy wyjściowej.

Warstwa wejściowa ma 13 neurony, które odpowiadają 13 cechom chemicznym wina. Warstwa ukryta ma liczbę neuronów, która jest określona przez hiperparametry `K1` i `K2`. Na początku eksperymentów przyjąłem wartości `K1 = 64` i `K2 = 32`. Warstwa wyjściowa ma 3 neurony, które odpowiadają trzem klasom winogron.

Model wykorzystuje funkcję aktywacji **ReLU** (*Rectified Linear Unit*) w warstwie ukrytej i wejściowej oraz funkcję regulacji **Dropout** w celu zapobiegania przeuczeniu. Funkcja **Dropout** losowo wyłącza część neuronów w warstwie ukrytej podczas treningu, co pomaga w regularyzacji modelu i poprawia jego zdolność do generalizacji.

```rust
impl<B: Backend> WineModel<B> {
  pub fn forward(&self, x: Tensor<B, 2>) -> Tensor<B, 2> {
    let x = self.input_layer.forward(x);
    let x = self.activation.forward(x);
    let x = self.dropout.forward(x);

    let x = self.hidden_layer.forward(x);
    let x = self.activation.forward(x);
    let x = self.dropout.forward(x);

    self.output_layer.forward(x)
  }
}
```

Program 5: Implementacji funkcji `forward`, która przekazuje dane przez warstwy sieci neuronowej

5.3. Trenowanie modelu

Trenowanie modelu polega na iteracyjnym dostosowywaniu wag i biasów sieci neuronowej w celu minimalizacji funkcji błędu. W tym projekcie wykorzystałem algorytm optymalizacji **Adam**, który jest jednym z najczęściej stosowanych algorytmów optymalizacji w uczeniu maszynowym. **Adam** łączy zalety algorytmów **RMSprop** i **Adagrad**, co pozwala na skuteczne aktualizowanie wag sieci neuronowej.

W trakcie trenowania modelu obliczana jest wartość funkcji straty (*loss*), która mierzy, jak bardzo przewidywania modelu różnią się od rzeczywistych etykiet w danych. Następnie, wartość straty jest wykorzystywana do aktualizacji wag sieci neuronowej w procesie optymalizacji.

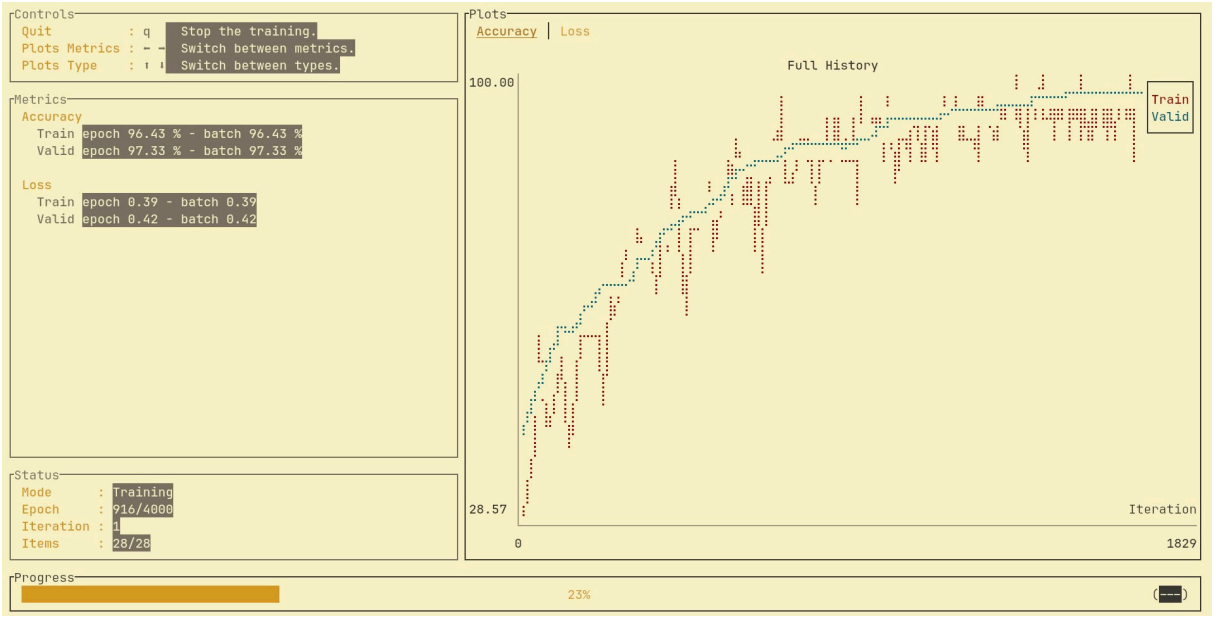
`Burn` dostarcza wiele wbudowanych metryk, które można wykorzystać do oceny skuteczności modelu podczas treningu. W moim przypadku, wykorzystałem metryki **dokładność** (*accuracy*) i **strata** (*loss*), które są kluczowe w problemach klasyfikacji.

`Burn` również dostarcza bardzo przejrzysty interfejs pokazujący przebieg treningu, w tym wartości metryk, czas treningu, aktualizacje wag oraz inne informacje, które są przydatne podczas analizy wyników.

```rust
pub fn train<B: AutodiffBackend>(artifact_dir: &str, config: TrainingConfig, device: B::Device) {
    create_artifact_dir(artifact_dir);
    config
        .save(format!("{artifact_dir}/config.json"))
        .expect("Config should be saved successfully");

    B::seed(config.seed);

    let train_batch = WineBatcher::<B>::new(device.clone());
    let test_batch = WineBatcher::<B::InnerBackend>::new(device.clone());

    let dataloader_train = DataLoaderBuilder::new(train_batch)
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(WineDataset::train());

    let dataloader_test = DataLoaderBuilder::new(test_batch)
        .batch_size(config.batch_size)
        .shuffle(config.seed)
        .num_workers(config.num_workers)
        .build(WineDataset::test());

    let learner = LearnerBuilder::new(artifact_dir)
        .metric_train_numeric(AccuracyMetric::new())
        .metric_valid_numeric(AccuracyMetric::new())
        .metric_train_numeric(LossMetric::new())
        .metric_valid_numeric(LossMetric::new())
        .with_file_checkpointer(CompactRecorder::new())
        .devices(vec![device.clone()])
```

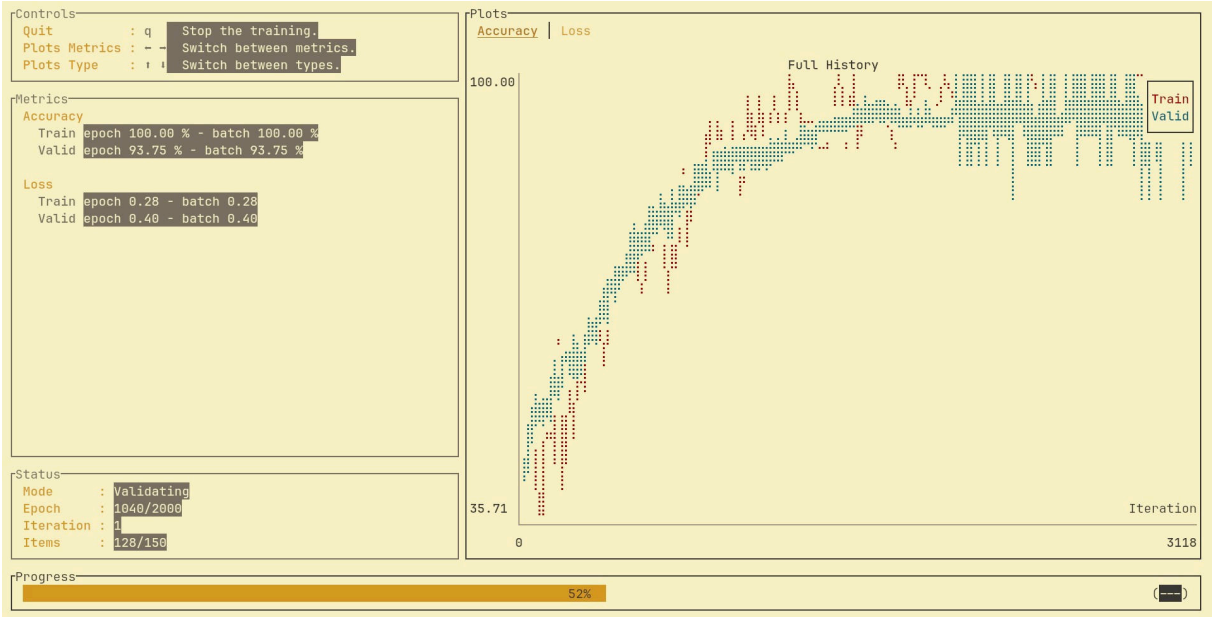
```
31      .num_epochs(config.num_epochs)
32      .summary()
33      .build(
34          config.model.init::<B>(&device),
35          config.optimizer.init(),
36          config.learning_rate,
37      );
38
39  let model_trained = learner.fit(dataloader_train, dataloader_test);
40
41  model_trained
42      .save_file(format!("{artifact_dir}/model"), &CompactRecorder::new())
43      .expect("Trained model should be saved successfully");
44  }
```

Program 6: Funkcji `train`, która trenuje model sieci neuronowej

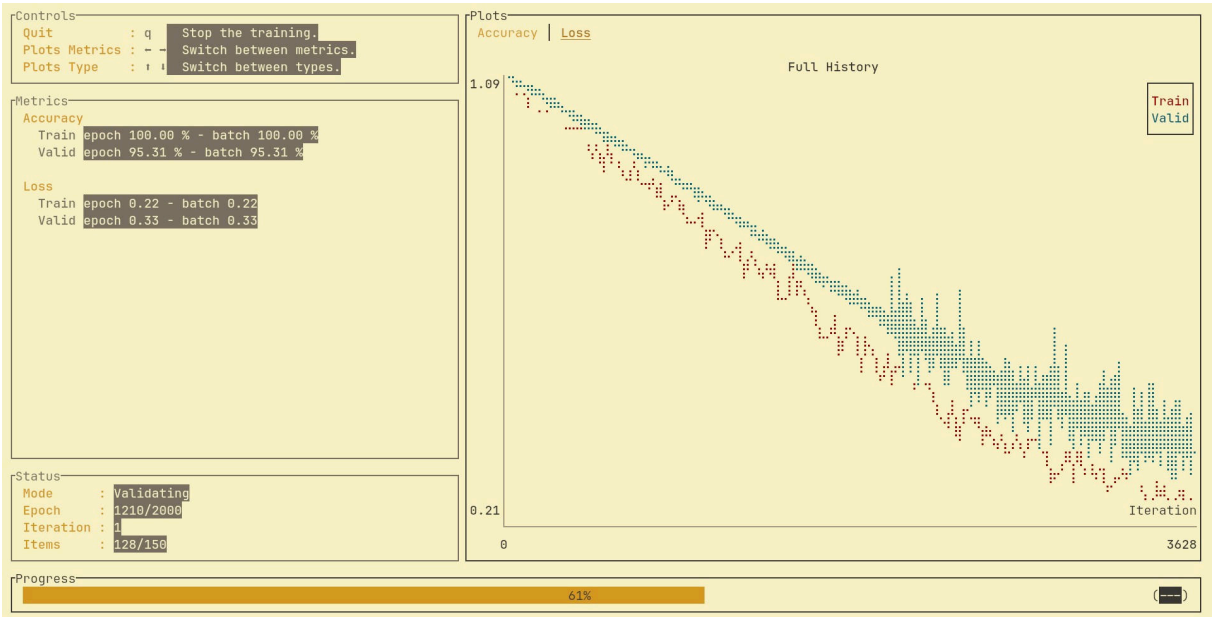


Rysunek 9: Interfejs `Burn` podczas treningu modelu

Po zakończeniu trenowania modelu można ocenić jego skuteczność na zbiorze testowym. Początkowo wykorzystałem hiperparametry `k1 = 64`, `k2 = 32`, `num_epochs = 2000`, `batch_size = 128` i `learning_rate = 1.0e-3`. Te parametry pozwoliły mi uzyskać dokładność na poziomie około 100% na zbiorze treningowym, a 94% na zbiorze testowym. Wyniki te były obiecujące, ale postanowiłem przeprowadzić serię eksperymentów, aby znaleźć optymalne wartości hiperparametrów.



Rysunek 10: Wykres dokładności modelu dla hiperparametrów `k1 = 64`, `k2 = 32`, `num_epochs = 2000`, `batch_size = 128` *i* `learning_rate = 1e-3`



Rysunek 11: Wykres straty modelu dla hiperparametrów `k1 = 64`, `k2 = 32`, `num_epochs = 2000`, `batch_size = 128` *i* `learning_rate = 1e-3`

Split	Metric	Min.	Epoch	Max.	Epoch
Train	Loss	0.063	1950	1.112	2
Train	Accuracy	21.429	62	100.000	2000
Valid	Loss	0.175	1958	1.095	1
Valid	Accuracy	41.333	1	94.000	1001

Rysunek 12: Tabela wyników modelu dla hiperparametrów `k1 = 64`, `k2 = 32`, `num_epochs = 2000`, `batch_size = 128` *i* `learning_rate = 1e-3`

6. Eksperymenty

W celu poprawy skuteczności modelu przeprowadziłem serię eksperymentów, w których zmieniałem hiperparametry modelu i nie tylko. Zmiany te miały na celu znalezienie optymalnych wartości, które pozwolą na uzyskanie jak najlepszych wyników.

6.1. Generowanie danych

Pierwszym eksperymentem było zbadanie kilku różnych prób danych treningowych i testowych, aby sprawdzić, jak zmiana podziału danych wpłynie na skuteczność modelu. W trakcie eksperymentów zmieniałem stosunek danych treningowych do testowych, aby sprawdzić, czy zmiana proporcji ma wpływ na wyniki modelu.

Ostatecznie, najlepszym stosunkiem okazał się podział 85% danych treningowych do 15% danych testowych. Ten podział pozwolił mi uzyskać najlepsze wyniki.

Zauważyłem też, że wyniki potrafiły się różnić bez zmiany stosunku podziału, a po samym wygenerowaniu ponownie danych. Wynika to z losowości podziału danych na zbiory treningowe i testowe, co może wpływać na wyniki modelu.

6.2. Zmiana hiperparametrów

Pierwszym krokiem w eksperymentach było zmienienie hiperparametrów modelu, takich jak `num_epochs`, `batch_size`, `learning_rate` i ilość neuronów w warstwie ukrytej (`K1` i `K2`). W trakcie eksperymentów testowałem różne wartości tych parametrów, aby znaleźć optymalne wartości, które pozwolą na uzyskanie jak najlepszych wyników.

6.2.1. Zmiana liczby neuronów w warstwie ukrytej

Pierwszym eksperymentem było zbadanie, jak zmiana liczby neuronów w warstwie ukrytej wpłynie na skuteczność modelu. Poniższa tabela przedstawia wyniki modelu dla różnych wartości parametrów `K1` i `K2`. Ważne jest, żeby wartości `K2` były zawsze mniejsze lub równe `K1`, aby zachować redukcję złożoności modelu.

K1	*K2*	*Dokładność treningowa*	*Dokładność testowa*	*Strata treningowa*	*Strata testowa*
32	16	100%	96.667%	0.018	0.106
64	32	**100%**	**97.333%**	$\mathbf{2.150 \cdot 10^{-3}}$	**0.101**
64	48	100%	96.667%	$6.878 \cdot 10^{-4}$	0.103
96	48	100%	96.667%	$6.416 \cdot 10^{-4}$	0.115
128	64	100%	96%	$2.755 \cdot 10^{-4}$	0.134

Tabela 1: Tabela wyników modelu dla różnych wartości parametrów `K1` i `K2`

Wyniki eksperymentów pokazują, że zmiana liczby neuronów w warstwie ukrytej nie miała znaczącego wpływu na skuteczność modelu. Ostatecznie, najlepszymi wartościami hiperparametrów okazały się `K1 = 64` i `K2 = 32`, które pozwoliły mi uzyskać dokładność na poziomie 100% na zbiorze treningowym, a 97.333% na zbiorze testowym. Strata modelu była również na niskim poziomie - około $2.150 \cdot 10^{-3}$ dla zbioru treningowego i 0.101 dla zbioru testowego. Co ciekawa strata dla była trochę większa niż dla innych wartości, ale dokładność była najwyższa.

6.2.2. Zmiana liczby epok

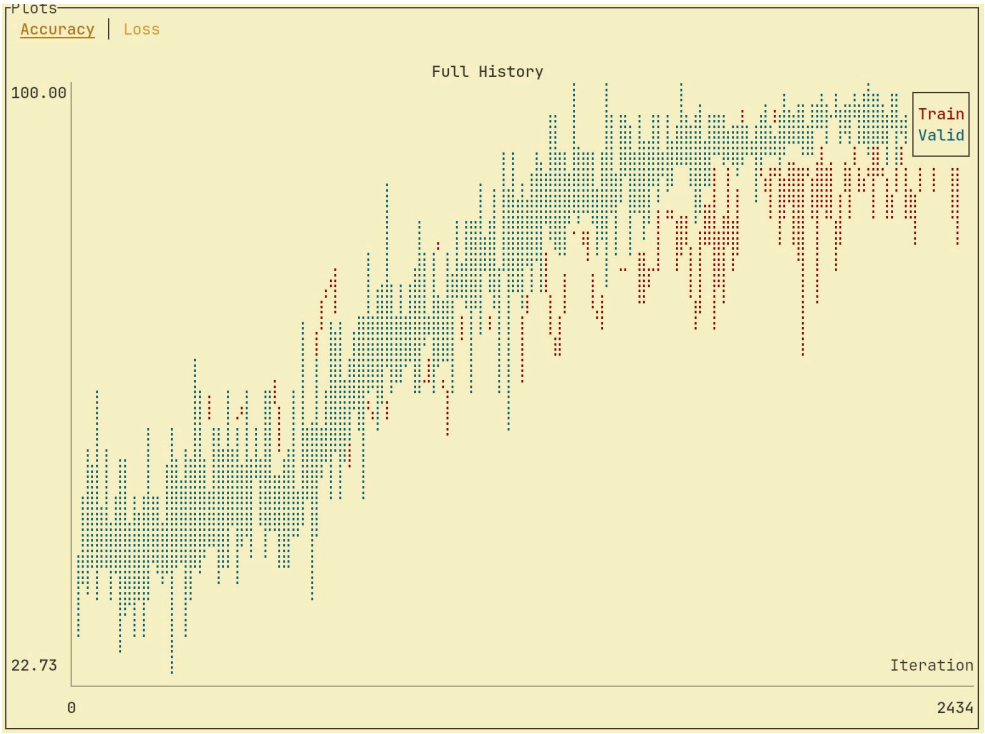
Kolejnym eksperymentem było zbadanie, jak zmiana liczby epok wpłynie na skuteczność modelu. W trakcie eksperymentów zmieniałem wartość parametru `num_epochs`, aby sprawdzić, jak długi czas trenowania ma wpływ na wyniki modelu.

Ostatecznie, najlepszą wartością dla dotychczasowych eksperymentów okazała się liczba epok równa 2000. Ta wartość najlepiej nauczała model. Wartości ≥ 3000 powodowały przeuczenie modelu, co objawiało się wzrostem straty, a wartości ≤ 1500 powodowały niedouczenie modelu, co objawiało się niższą dokładnością.

6.2.3. Zmiana rozmiaru wsadu

Kolejnym eksperymentem było zbadanie, jak zmiana rozmiaru wsadu wpłynie na skuteczność modelu. W trakcie eksperymentów zmieniałem wartość parametru `batch_size`, aby sprawdzić, jak liczba próbek przekazywanych do modelu jednocześnie ma wpływ na wyniki modelu.

Za mała ilość próbek w wsadzie prowadzić do dużych wahań w procesie uczenia. Biblioteka w takich przypadkach nie wyświetla tabeli z wynikami.



Rysunek 13: Wykres uczenia modelu dla małego rozmiaru wsadu

Z kolei zbyt duża ilość próbek prowadzić do wydłużonego procesu uczenia. W takich przypadkach model uczy się wolniej i osiąga te same wyniki od pewnego momentu.

Poniższa tabela przedstawia wyniki modelu dla różnych wartości parametru `batch_size`.

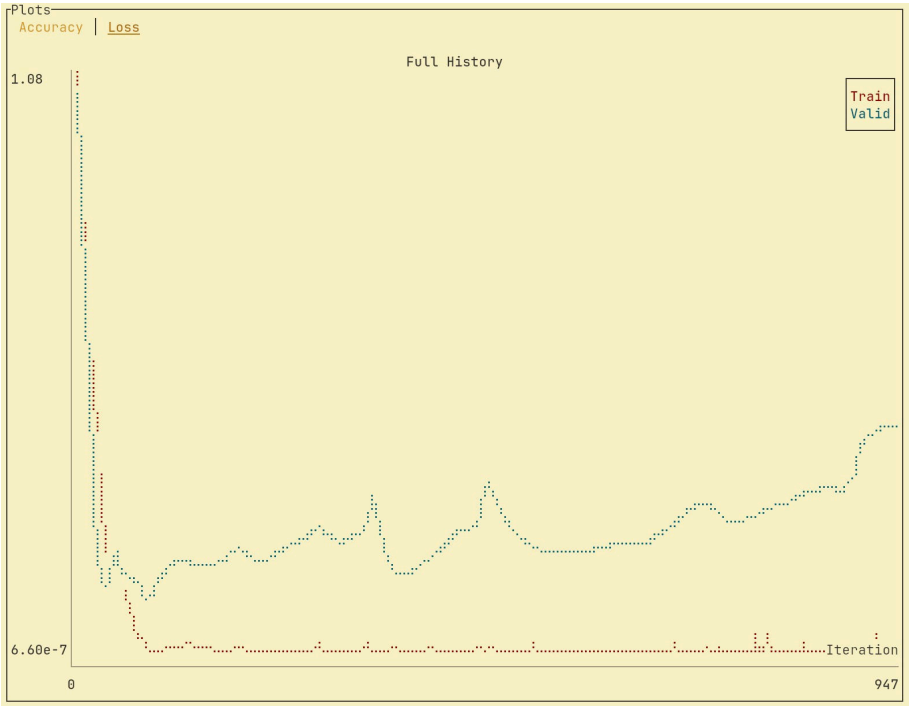
Rozmiar wsadu	*Dokładność treningowa*	*Dokładność testowa*	*Strata treningowa*	*Strata testowa*
64	-	-	-	-
128	**100%**	**97.333%**	$\mathbf{2.150 \cdot 10^{-3}}$	**0.061**
192	100%	97.333%	$2.150 \cdot 10^{-3}$	0.101
256	100%	97.333%	$2.150 \cdot 10^{-3}$	0.101
512	100%	97.333%	$2.150 \cdot 10^{-3}$	0.101
1024	100%	97.333%	$2.150 \cdot 10^{-3}$	0.101

Jak widać, zmiana rozmiaru wsadu nie ma praktycznie wpływu na skuteczność modelu. Ostatecznie, najlepszą wartością hiperparametru okazał się `batch_size = 128`, ale na wykresie uczenia widać było nadal wahania. Dlatego zdecydowałem się na wartość `batch_size = 256`, która pozwoliła mi uzyskać stabilne wyniki.

6.2.4. Zmiana współczynnika uczenia

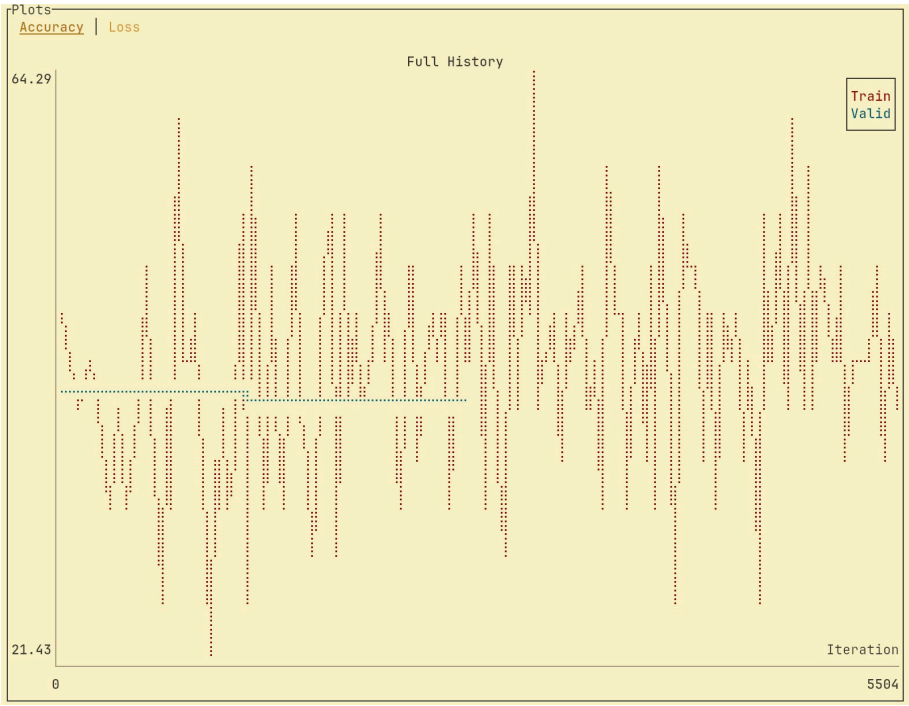
Ostatnim eksperymentem było zbadanie, jak zmiana współczynnika uczenia wpłynie na skuteczność modelu. W trakcie eksperymentów zmieniałem wartość parametru `learning_rate`, aby sprawdzić, która wartość pozwoli na uzyskanie najlepszych i stabilnych wyników.

Dla wysokiego współczynnika uczenia model uczył się szybko, ale miał problem z osiągnięciem stabilnych wyników. Stara była bardzo niska dla zbioru treningowego, ale rosnąca dla zbioru testowego.



Rysunek 14: Wykres uczenia dla `learning_rate = 1.0e-2`

Dla niskiego współczynnika uczenia model uczył się wolno i nie osiągał dobrych wyników. Nawet po wielu epokach model nie osiągał wysokiej dokładności.



Rysunek 15: Wykres uczenia dla `learning_rate = 1.0e-6`

Poniższa tabela przedstawia wyniki modelu dla różnych wartości parametru `learning_rate`.

Współczynnik uczenia	*Dokładność treningowa*	*Dokładność testowa*	*Strata treningowa*	*Strata testowa*
$1.0 \cdot 10^{-2}$	100%	97.333%	$2.150 \cdot 10^{-3}$	0.101
$5.0 \cdot 10^{-3}$	**100%**	**98.667%**	$\mathbf{0.000 \cdot 10^{-0}}$ ***(Graniczna wartość)***	**0.091**
$1.0 \cdot 10^{-3}$	100%	97.333%	$1.022 \cdot 10^{-7}$	0.103
$5.0 \cdot 10^{-4}$	100%	97.333%	$3.572 \cdot 10^{-6}$	0.105
$1.0 \cdot 10^{-4}$	100%	97.333%	$2.150 \cdot 10^{-3}$	0.101
$5.0 \cdot 10^{-5}$	100%	96.667%	0.025	0.109
$1.0 \cdot 10^{-5}$	96.429%	89.333%	0.730	0.797
$1.0 \cdot 10^{-6}$	71.429%	40.677%	1.017	1.061

Widać jak bardzo zmiana współczynnika uczenia wpływa na skuteczność modelu. W najlepszych przypadkach model osiągał podobną dokładność. Różnica była w stracie, która była najniższa dla wartości `learning_rate = 5.0e-3`, ale nadal wartość ta nie pozwalała na osiągnięcie stabilnych wyników. Ostatecznie, najlepszymi i stabilnymi wartościami hiperparametrów okazały się `K1 = 64`, `K2 = 32`, `num_epochs = 2000`, `batch_size = 256` i `learning_rate = 5.0e-4`.

6.3. Zmiana architektury modelu

Kolejnym krokiem w eksperymentach było zmienienie architektury modelu, tak aby sprawdzić, czy zmiana liczby neuronów w warstwie ukrytej wpłynie na skuteczność modelu.

6.3.1. Dodanie drugiej warstwy ukrytej

Pierwszym eksperymentem było dodanie drugiej warstwy ukrytej do modelu. W trakcie eksperymentów testowałem różne kombinacje wartości parametrów `K1`, `K2` i `K3`, aby sprawdzić, jak zmiana liczby neuronów w warstwach ukrytych wpłynie na skuteczność modelu.

Poniższa tabela przedstawia wyniki modelu dla różnych wartości parametrów `K1`, `K2` i `K3`.

K1	*K2*	*K3*	*Dokładność treningowa*	*Dokładność testowa*	*Strata treningowa*	*Strata testowa*
48	32	16	100%	96.667%	$3.444 \cdot 10^{-4}$	0.116
64	48	32	**100%**	**98%**	$\mathbf{5.458 \cdot 10^{-6}}$	**0.089**
128	64	32	100%	96%	$8.391 \cdot 10^{-6}$	0.131
256	128	64	100%	96%	$6.982 \cdot 10^{-7}$	0.135

Tabela 4: Tabela wyników modelu dla różnych wartości parametrów `K1`, `K2` i `K3`

Wyniki eksperymentów pokazują, że dodanie drugiej warstwy ukrytej miało niewielki wpływ na skuteczność modelu. Ostatecznie, najlepszymi wartościami hiperparametrów okazały się `K1 = 64`, `K2 = 48` i `K3 = 32`, które pozwoliły mi uzyskać dokładność na poziomie 100% na zbiorze treningowym, a 98% na zbiorze testowym. Strata modelu była również na niskim poziomie - około $5.458 \cdot 10^{-6}$ dla zbioru treningowego i 0.089 dla zbioru testowego.

6.3.2. Usunięcie warstwy ukrytej

Kolejnym eksperymentem było usunięcie warstwy ukrytej z modelu. W trakcie eksperymentów testowałem różne wartości parametrów `K1`, aby sprawdzić, jak zmiana liczby neuronów w warstwie wejściowej wpłynie na skuteczność modelu.

Poniższa tabela przedstawia wyniki modelu dla różnych wartości parametru `K1`.

K1	*Dokładność treningowa*	*Dokładność testowa*	*Strata treningowa*	*Strata testowa*
16	100%	96%	0.018	0.148
32	100%	96%	$4.869 \cdot 10^{-3}$	0.116
64	**100%**	**96%**	$\mathbf{1.625 \cdot 10^{-3}}$	**0.118**
128	100%	96%	$6.623 \cdot 10^{-4}$	0.129
256	100%	96%	$3.070 \cdot 10^{-4}$	0.130
512	100%	96%	$1.593 \cdot 10^{-4}$	0.119

Tabela 5: Tabela wyników modelu dla różnych wartości parametru `K1`

Jak widać, usunięcie warstwy ukrytej miało niewielki, ale negatywny wpływ na skuteczność modelu. Ostatecznie, najlepszą wartością hiperparametru okazał się `K1 = 64`, który pozwolił mi uzyskać dokładność na poziomie 100% na zbiorze treningowym, a 96% na zbiorze testowym. Strata modelu była również na niskim poziomie - około $1.625 \cdot 10^{-3}$ dla zbioru treningowego i 0.118 dla zbioru testowego.

6.4. Zmiana funkcji aktywacji

Osatnim eksperymentem było zbadanie, jak zmiana funkcji aktywacji wpłynie na skuteczność modelu. W trakcie eksperymentów testowałem różne funkcje aktywacji, takie jak **ReLU**, **LeakyReLU**, **Sigmoid** i **Tanh**.

Poniższa tabela przedstawia wyniki modelu dla różnych funkcji aktywacji.

Funkcja aktywacji	*Dokładność treningowa*	*Dokładność testowa*	*Strata treningowa*	*Strata testowa*
ReLU	100%	97.333%	$6.054 \cdot 10^{-5}$	0.105
LeakyReLU	100%	97.333%	$6.413 \cdot 10^{-5}$	0.105
Sigmoid	100%	94.667%	0.038	0.184
Tanh	100%	97.333%	$5.535 \cdot 10^{-4}$	0.111

Tabela 6: Tabela wyników modelu dla różnych funkcji aktywacji

Wyniki eksperymentów pokazują, że funkcja aktywacji **ReLU** i **LeakyReLU** osiągnęły najlepsze wyniki. **Sigmoid** miała najgorsze wyniki, co sugeruje, że ta funkcja aktywacji nie nadaje się do tego problemu. **Tanh** osiągnęła wyniki podobne do **ReLU** i **LeakyReLU**, ale była nieco gorsza.

7. Podsumowanie

W niniejszej pracy omówiono proces tworzenia i testowania modelu klasyfikacyjnego przy użyciu danych chemicznych dotyczących różnych rodzajów winogron. Dzięki zastosowaniu nowoczesnych technik sztucznej inteligencji, takich jak głębokie uczenie, możliwe było zbudowanie modelu, który skutecznie klasyfikuje wina na podstawie ich atrybutów chemicznych. W ramach przygotowania danych do nauki omówiono m.in. techniki normalizacji, które miały na celu zapewnienie, że cechy wejściowe mają jednolitą skalę i nie wpływają negatywnie na proces uczenia modelu. Testowanie modelu uwzględniało m.in. miarę loss oraz dokładność, które pozwoliły na ocenę skuteczności modelu.

W pracy uwzględniono także kluczowe aspekty technologiczne związane z używaną biblioteką - `Burn`. Przygotowanie danych i ich odpowiednia obróbka są niezbędne, by model mógł działać skutecznie, a wyniki testów wskazują na pozytywne efekty zastosowanych rozwiązań w kontekście klasyfikacji win. Dzięki tym technologiom możliwe jest uzyskanie dokładnych prognoz dotyczących typu wina, co może mieć zastosowanie w różnych dziedzinach, od przemysłu winiarskiego po zastosowania badawcze w naukach chemicznych.

7.1. Wnioski

1. **Efektywność sztucznej inteligencji w klasyfikacji danych chemicznych** – Zastosowanie głębokiego uczenia do klasyfikacji win na podstawie ich atrybutów chemicznych okazało się skuteczne. Model uzyskał wysoką dokładność, co sugeruje, że takie podejście może być użyteczne w analizach przemysłowych oraz badaniach naukowych.
2. **Znaczenie przygotowania danych** – Dobrze przeprowadzone przygotowanie danych, w tym normalizacja i odpowiedni podział na zbiory treningowe oraz testowe, miało kluczowy wpływ na wydajność modelu. W szczególności normalizacja danych pozwoliła na poprawienie stabilności modelu i zapewnienie, że wszystkie cechy miały podobną wagę.
3. **Potencjał do zastosowań praktycznych** – Modele klasyfikacyjne oparte na głębokim uczeniu mogą mieć szerokie zastosowanie w przemyśle winiarskim, gdzie dokładne określenie rodzaju wina na podstawie jego właściwości chemicznych może pomóc w procesach produkcji, kontroli jakości i marketingu.

Bibliografia

[1] David Andrés, „Gradient Descent”. [Online]. Dostępne na: https://mlpills.dev/machine-learning/gradient-descent/%22

[2] Jason Brownlee, „A Gentle Introduction to the Rectified Linear Unit (ReLU)”. [Online]. Dostępne na: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep--learning-neural-networks/

[3] Wikipedia, „Sigmoid function”. [Online]. Dostępne na: https://en.wikipedia.org/wiki/Sigmoid_function

[4] Wikipedia, „Hyperbolic tangent function”. [Online]. Dostępne na: https://commons.wikimedia.org/wiki/File:Hyperbolic_Tangent.svg

[5] Imrus Salehin i Dae-Ki Kang, „A Review on Dropout Regularization Approaches for Deep Neural Networks within the Scholarly Domain”. [Online]. Dostępne na: https://www.mdpi.com/2079--9292/12/14/3106

[6] Konstantin Tretyakov, „The Mystery of Early Stopping”. [Online]. Dostępne na: https://fouryears.eu/2017/12/06/the-mystery-of-early-stopping/

[7] Guillaume Lagrange, „Building Blocks #1: Dataset & Data Loading”. [Online]. Dostępne na: https://burn.dev/blog/building-blocks-dataset/

8. Skrypt

Rust

```rust
use cli_table::*;
use si_project::dataset::WineItem;
use std::collections::BTreeMap;
use std::default::Default;
use std::fs::read_to_string;

fn main() {
  let data = load_data_file();
  let wines = parse_data(data);

  // 1. Extract `x` and `y_t`
  let (x, y_t) = extract_x_and_y_t(&wines);

  // 2. Normalize `x` → `x_norm`
  let x_norm = normalize_x(&x);

  // Sorting by `y_t` is unnecessary for this dataset. Data is already sorted by class.

  // 3. Split data into training and testing sets
  let (x_train, y_t_train, x_test, y_t_test) = split_data(x_norm, y_t);

  // 3. Display data in a table
  println!("Data test:");
  table_data(&x_train, &y_t_train);
  println!("\n\n\nData train:");
  table_data(&x_test, &y_t_test);

  // 4. Save everything to an PKL file
  dump_to_pkl(x_train, y_t_train, "train");
  dump_to_pkl(x_test, y_t_test, "test");

  println!("Data processing and saving completed.");
}

/// Load data file into a `String`.
fn load_data_file() -> String {
  read_to_string("./data/wine.data").unwrap()
}

/// Parse raw data into a Vector of `Wine` structs.
fn parse_data(data: String) -> Vec<WineItem> {
  data.lines()
    .map(|line| {
      let mut wine_data = line.split(',').map(|s| s.parse::<f32>().unwrap());
      WineItem {
        class: wine_data.next().unwrap() as i32,
        alcohol: wine_data.next().unwrap(),
        malic_acid: wine_data.next().unwrap(),
        ash: wine_data.next().unwrap(),
        alcalinity_of_ash: wine_data.next().unwrap(),
        magnesium: wine_data.next().unwrap(),
        total_phenols: wine_data.next().unwrap(),
        flavanoids: wine_data.next().unwrap(),
        nonflavanoid_phenols: wine_data.next().unwrap(),
        proanthocyanins: wine_data.next().unwrap(),
        color_intensity: wine_data.next().unwrap(),
        hue: wine_data.next().unwrap(),
        od280_od315_of_diluted_wines: wine_data.next().unwrap(),
        proline: wine_data.next().unwrap(),
      }
```

```
    })
    .collect()
}

/// Extract `x` (Wine attributes 1-13) and `y_t` (class attribute).
fn extract_x_and_y_t(wines: &[WineItem]) -> (Vec<Vec<f32>>, Vec<i32>) {
  let num_records = wines.len();
  let mut x = vec![vec![0.0; 13]; num_records];
  let mut y_t = vec![0; num_records];

  for (i, wine) in wines.iter().enumerate() {
    x[i][0] = wine.alcohol;
    x[i][1] = wine.malic_acid;
    x[i][2] = wine.ash;
    x[i][3] = wine.alcalinity_of_ash;
    x[i][4] = wine.magnesium;
    x[i][5] = wine.total_phenols;
    x[i][6] = wine.flavanoids;
    x[i][7] = wine.nonflavanoid_phenols;
    x[i][8] = wine.proanthocyanins;
    x[i][9] = wine.color_intensity;
    x[i][10] = wine.hue;
    x[i][11] = wine.od280_od315_of_diluted_wines;
    x[i][12] = wine.proline;

    y_t[i] = wine.class - 1;
  }

  (x, y_t)
}

/// Normalize `x` (array of wine attributes 1-13).
fn normalize_x(x: &Vec<Vec<f32>>) -> Vec<Vec<f32>> {
  let num_columns = x[0].len();

  let mut min = vec![f32::MAX; num_columns];
  let mut max = vec![f32::MIN; num_columns];

  for row in x.iter() {
    for (j, &value) in row.iter().enumerate() {
      if value < min[j] {
        min[j] = value;
      }
      if value > max[j] {
        max[j] = value;
      }
    }
  }

  let mut x_norm = x.clone();
  for (i, row) in x.iter().enumerate() {
    for (j, &value) in row.iter().enumerate() {
      if max[j] - min[j] == 0.0 {
        x_norm[i][j] = 0.0; // Avoid division by zero
      } else {
        // Normalize to range [-1, 1]
        x_norm[i][j] = -1.0 + 2.0 * (value - min[j]) / (max[j] - min[j]);
      }
    }
  }

  x_norm
```

```rust
}

/// Display data in a table.
fn table_data(x: &Vec<Vec<f32>>, y_t: &Vec<i32>) {
  let mut table = Vec::new();

  // Add data to table
  for (i, row) in x.iter().enumerate() {
    let mut row_data = Vec::new();
    row_data.push(y_t[i].to_string());
    for &value in row.iter() {
      row_data.push(value.to_string());
    }
    table.push(row_data);
  }

  // Add header to table
  let table = table.table().title(vec![
    "y_t".cell().bold(true),
    "alcohol".cell().bold(true),
    "malic acid".cell().bold(true),
    "ash".cell().bold(true),
    "alcalinity of ash".cell().bold(true),
    "magnesium".cell().bold(true),
    "total phenols".cell().bold(true),
    "flavanoids".cell().bold(true),
    "nonflavanoid phenols".cell().bold(true),
    "proanthocyanins".cell().bold(true),
    "color intensity".cell().bold(true),
    "hue".cell().bold(true),
    "od280 od315 of diluted wines".cell().bold(true),
    "proline".cell().bold(true),
  ]);

  print_stdout(table).unwrap();
}

/// Save `x` and `y_t` to disk in pickle format.
fn dump_to_pkl(x: Vec<Vec<f32>>, y_t: Vec<i32>, prefix: &str) {
  // Create BTreeMaps to serialize
  let mut x_map = BTreeMap::new();
  let mut y_map = BTreeMap::new();
  x_map.insert(String::from("x"), x);
  y_map.insert(String::from("y_t"), y_t);

  // Serialize to pickle format
  let x_serialized = serde_pickle::to_vec(&x_map, Default::default()).unwrap();
  let y_serialized = serde_pickle::to_vec(&y_map, Default::default()).unwrap();

  // Save to disk
  std::fs::write(format!("./data/x_{}.pkl", prefix), &x_serialized).unwrap();
  std::fs::write(format!("./data/y_t_{}.pkl", prefix), &y_serialized).unwrap();
}

/// Split data into training and testing sets.
use rand::seq::SliceRandom;
use rand::thread_rng;

fn split_data(
  x: Vec<Vec<f32>>,
  y_t: Vec<i32>,
) -> (Vec<Vec<f32>>, Vec<i32>, Vec<Vec<f32>>, Vec<i32>) {
```

```rust
  let class_counts = get_class_count(&y_t);
  let mut x_train = Vec::new();
  let mut y_t_train = Vec::new();
  let mut x_test = Vec::new();
  let mut y_t_test = Vec::new();

  let wines = x.iter().zip(y_t.iter()).collect::<Vec<_>>();

  for (class, count) in class_counts.iter() {
    let num_train = (count * 80) / 100;

    let mut class_wines = wines
      .iter()
      .filter(|(_, &y)| y == *class)
      .collect::<Vec<_>>();
    class_wines.shuffle(&mut thread_rng());

    let (test, train) = class_wines.split_at(num_train as usize);

    for (x, y) in train.into_iter() {
      x_train.push((*x).clone());
      y_t_train.push((*y).clone());
    }

    for (x, y) in test.iter() {
      x_test.push((*x).clone());
      y_t_test.push((*y).clone());
    }
  }

  (x_train, y_t_train, x_test, y_t_test)
}

fn get_class_count(y_t: &Vec<i32>) -> BTreeMap<i32, i32> {
  let mut class_counts = BTreeMap::new();
  for &class in y_t.iter() {
    *class_counts.entry(class).or_insert(0) += 1;
  }

  class_counts
}
```

Program 7: `prepare_data.rs`

```rust
use burn::data::dataset::{Dataset, InMemDataset};
use serde::{Deserialize, Serialize};
use std::collections::BTreeMap;
use std::fs::File;
use std::io;

#[derive(Debug, PartialEq, Clone, Serialize, Deserialize)]
pub struct WineItem {
  #[serde(rename = "class")]
  pub class: i32,

  #[serde(rename = "Alcohol")]
  pub alcohol: f32,

  #[serde(rename = "Malic acid")]
  pub malic_acid: f32,

  #[serde(rename = "Ash")]
  pub ash: f32,
```

```
20
21      #[serde(rename = "Alcalinity of ash")]
22      pub alcalinity_of_ash: f32,
23
24      #[serde(rename = "Magnesium")]
25      pub magnesium: f32,
26
27      #[serde(rename = "Total phenols")]
28      pub total_phenols: f32,
29
30      #[serde(rename = "Flavanoids")]
31      pub flavanoids: f32,
32
33      #[serde(rename = "Nonflavanoid phenols")]
34      pub nonflavanoid_phenols: f32,
35
36      #[serde(rename = "Proanthocyanins")]
37      pub proanthocyanins: f32,
38
39      #[serde(rename = "Color intensity")]
40      pub color_intensity: f32,
41
42      #[serde(rename = "Hue")]
43      pub hue: f32,
44
45      #[serde(rename = "OD280/OD315 of diluted wines")]
46      pub od280_od315_of_diluted_wines: f32,
47
48      #[serde(rename = "Proline")]
49      pub proline: f32,
50  }
51
52  pub struct WineDataset {
53      pub(crate) dataset: InMemDataset<WineItem>,
54  }
55
56  impl WineDataset {
57      pub fn subset(&self, indices: &[usize]) -> Self {
58          let data: Vec<WineItem> = indices
59              .iter()
60              .map(|&i| self.dataset.get(i).unwrap().clone())
61              .collect();
62          Self {
63              dataset: InMemDataset::new(data),
64          }
65      }
66  }
67
68  impl WineDataset {
69      pub fn from_pkl(split: &str) -> Result<Self, io::Error> {
70          let x_file = File::open(format!("./data/x_{}.pkl", split))?;
71          let y_file = File::open(format!("./data/y_t_{}.pkl", split))?;
72
73          let x_map: BTreeMap<String, Vec<Vec<f32>>> =
74              serde_pickle::from_reader(x_file, Default::default())
75                  .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?;
76          let y_map: BTreeMap<String, Vec<i32>> =
77              serde_pickle::from_reader(y_file, Default::default())
78                  .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?;
79
80          let mut data = Vec::new();
81
```

```
      for (key, x_values) in x_map.get("x").unwrap().iter().enumerate() {
        if let Some(y_values) = y_map.get("y_t").unwrap().get(key) {
          let wine = WineItem {
            class: *y_values,
            alcohol: x_values[0],
            malic_acid: x_values[1],
            ash: x_values[2],
            alcalinity_of_ash: x_values[3],
            magnesium: x_values[4],
            total_phenols: x_values[5],
            flavanoids: x_values[6],
            nonflavanoid_phenols: x_values[7],
            proanthocyanins: x_values[8],
            color_intensity: x_values[9],
            hue: x_values[10],
            od280_od315_of_diluted_wines: x_values[11],
            proline: x_values[12],
          };
          data.push(wine);
        }
      }

      let dataset = InMemDataset::new(data);

      Ok(Self { dataset })
    }

    pub fn train() -> Self {
      Self::from_pkl("train").unwrap()
    }

    pub fn test() -> Self {
      Self::from_pkl("test").unwrap()
    }
  }

  impl Dataset<WineItem> for WineDataset {
    fn get(&self, index: usize) -> Option<WineItem> {
      self.dataset.get(index)
    }

    fn len(&self) -> usize {
      self.dataset.len()
    }
  }
```

Program 8: `dataset.rs`

```rust
use crate::dataset::WineItem;
use burn::data::dataloader::batcher::Batcher;
use burn::prelude::{Backend, Int, Tensor};

#[derive(Clone)]
pub struct WineBatcher<B: Backend> {
  device: B::Device,
}

impl<B: Backend> WineBatcher<B> {
  pub fn new(device: B::Device) -> Self {
    Self { device }
  }
}

```

```rust
#[derive(Clone, Debug)]
pub struct WineBatch<B: Backend> {
  pub x: Tensor<B, 2>,
  pub y: Tensor<B, 1, Int>,
}

impl<B: Backend> Batcher<WineItem, WineBatch<B>> for WineBatcher<B> {
  fn batch(&self, data: Vec<WineItem>) -> WineBatch<B> {
    let x = data
      .iter()
      .map(|wine| {
        Tensor::<B, 2>::from_data(
          [[
            wine.alcohol,
            wine.malic_acid,
            wine.ash,
            wine.alcalinity_of_ash,
            wine.magnesium,
            wine.total_phenols,
            wine.flavanoids,
            wine.nonflavanoid_phenols,
            wine.proanthocyanins,
            wine.color_intensity,
            wine.hue,
            wine.od280_od315_of_diluted_wines,
            wine.proline,
          ]],
          &self.device,
        )
      })
      .collect::<Vec<_>>();

    let y = data
      .iter()
      .map(|wine| Tensor::<B, 1, Int>::from_data([wine.class], &self.device))
      .collect::<Vec<_>>();

    let x = Tensor::<B, 2>::cat(x, 0).to_device(&self.device);
    let y = Tensor::<B, 1, Int>::cat(y, 0).to_device(&self.device);

    WineBatch { x, y }
  }
}
```

Program 9: `batcher.rs`

```rust
use crate::batcher::WineBatch;
use burn::config::Config;
use burn::module::Module;
use burn::nn::loss::CrossEntropyLossConfig;
use burn::nn::{Dropout, DropoutConfig, Linear, LinearConfig, Relu};
use burn::prelude::{Backend, Tensor};
use burn::tensor::backend::AutodiffBackend;
use burn::tensor::Int;
use burn::train::{ClassificationOutput, TrainOutput, TrainStep, ValidStep};

#[derive(Module, Debug)]
pub struct WineModel<B: Backend> {
  input_layer: Linear<B>,
  hidden_layer: Linear<B>,
  output_layer: Linear<B>,
  activation: Relu,
  dropout: Dropout,
```

```rust
}

impl<B: Backend> WineModel<B> {
  pub fn forward(&self, x: Tensor<B, 2>) -> Tensor<B, 2> {
    let x = self.input_layer.forward(x);
    let x = self.activation.forward(x);
    let x = self.dropout.forward(x);

    let x = self.hidden_layer.forward(x);
    let x = self.activation.forward(x);
    let x = self.dropout.forward(x);

    self.output_layer.forward(x)
  }
}

impl<B: Backend> WineModel<B> {
  pub fn forward_classification(
    &self,
    x: Tensor<B, 2>,
    y: Tensor<B, 1, Int>,
  ) -> ClassificationOutput<B> {
    let output = self.forward(x);
    let loss = CrossEntropyLossConfig::new()
      .init(&output.device())
      .forward(output.clone(), y.clone());

    ClassificationOutput::new(loss, output, y)
  }
}

impl<B: AutodiffBackend> TrainStep<WineBatch<B>, ClassificationOutput<B>> for WineModel<B> {
  fn step(&self, batch: WineBatch<B>) -> TrainOutput<ClassificationOutput<B>> {
    let item = self.forward_classification(batch.x, batch.y);

    TrainOutput::new(self, item.loss.backward(), item)
  }
}

impl<B: Backend> ValidStep<WineBatch<B>, ClassificationOutput<B>> for WineModel<B> {
  fn step(&self, batch: WineBatch<B>) -> ClassificationOutput<B> {
    self.forward_classification(batch.x, batch.y)
  }
}

#[derive(Config, Debug)]
pub struct WineModelConfig {
  num_classes: usize,
  k1: usize,
  k2: usize,
  #[config(default = "0.5")]
  dropout: f64,
}

impl WineModelConfig {
  pub fn init<B: Backend>(&self, device: &B::Device) -> WineModel<B> {
    WineModel {
      input_layer: LinearConfig::new(13, self.k1).init(device),
      hidden_layer: LinearConfig::new(self.k1, self.k2).init(device),
      output_layer: LinearConfig::new(self.k2, self.num_classes).init(device),
      activation: Relu::new(),
      dropout: DropoutConfig::new(self.dropout).init(),
```

```
    }
  }
}
```

Program 10: `model.rs`

```rust
use crate::batcher::WineBatcher;
use crate::dataset::WineDataset;
use crate::model::WineModelConfig;
use burn::config::Config;
use burn::data::dataloader::DataLoaderBuilder;
use burn::module::Module;
use burn::optim::AdamConfig;
use burn::record::CompactRecorder;
use burn::tensor::backend::AutodiffBackend;
use burn::train::metric::{AccuracyMetric, LossMetric};
use burn::train::LearnerBuilder;

#[derive(Config)]
pub struct TrainingConfig {
  pub model: WineModelConfig,
  pub optimizer: AdamConfig,
  #[config(default = 2000)]
  pub num_epochs: usize,
  #[config(default = 256)]
  pub batch_size: usize,
  #[config(default = 1)]
  pub num_workers: usize,
  #[config(default = 42)]
  pub seed: u64,
  #[config(default = 5e-3)]
  pub learning_rate: f64,
}

pub fn train<B: AutodiffBackend>(artifact_dir: &str, config: TrainingConfig, device: B::Device) {
  create_artifact_dir(artifact_dir);
  config
    .save(format!("{artifact_dir}/config.json"))
    .expect("Config should be saved successfully");

  B::seed(config.seed);

  let train_batch = WineBatcher::<B>::new(device.clone());
  let test_batch = WineBatcher::<B::InnerBackend>::new(device.clone());

  let dataloader_train = DataLoaderBuilder::new(train_batch)
    .batch_size(config.batch_size)
    .shuffle(config.seed)
    .num_workers(config.num_workers)
    .build(WineDataset::train());

  let dataloader_test = DataLoaderBuilder::new(test_batch)
    .batch_size(config.batch_size)
    .shuffle(config.seed)
    .num_workers(config.num_workers)
    .build(WineDataset::test());

  let learner = LearnerBuilder::new(artifact_dir)
    .metric_train_numeric(AccuracyMetric::new())
    .metric_valid_numeric(AccuracyMetric::new())
    .metric_train_numeric(LossMetric::new())
    .metric_valid_numeric(LossMetric::new())
    .with_file_checkpointer(CompactRecorder::new())
```

```
    .devices(vec![device.clone()])
    .num_epochs(config.num_epochs)
    .summary()
    .build(
      config.model.init::<B>(&device),
      config.optimizer.init(),
      config.learning_rate,
    );

  let model_trained = learner.fit(dataloader_train, dataloader_test);

  model_trained
    .save_file(format!("{artifact_dir}/model"), &CompactRecorder::new())
    .expect("Trained model should be saved successfully");
}

fn create_artifact_dir(artifact_dir: &str) {
  // Remove existing artifacts before to get an accurate learner summary
  std::fs::remove_dir_all(artifact_dir).ok();
  std::fs::create_dir_all(artifact_dir).ok();
}
```

Program 11: `training.rs`

```
use burn::backend::cuda_jit::CudaDevice;
use burn::backend::{CudaJit};
use burn::data::dataset::Dataset;
use burn::{
  backend::{Autodiff, Wgpu},
  optim::AdamConfig,
};
use si_project::dataset::WineDataset;
use si_project::model::WineModelConfig;
use si_project::training::{train, TrainingConfig};

fn main() {
  // type MyBackend = Wgpu<f32, i32>;
  type MyBackend = CudaJit<f32, i32>;
  type MyAutodiffBackend = Autodiff<MyBackend>;

  // let device = burn::backend::wgpu::WgpuDevice::default();
  let device = CudaDevice::default();

  let artifact_dir = "./artifacts";
  train::<MyAutodiffBackend>(
    artifact_dir,
    TrainingConfig::new(WineModelConfig::new(3, 64, 32), AdamConfig::new()),
    device.clone(),
  );
}
```

Program 12: `main.rs`