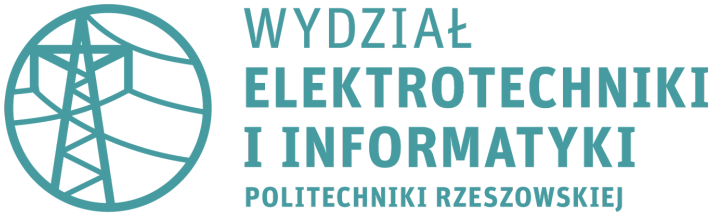
Politechnika Rzeszowska
Wydział Elektrotechniki i Informatyki



Sztuczna Inteligencja

Klasyfikator win w Burn

Projekt

24.01.2025 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 względu 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 zaawansowanych algorytmów sztucznej inteligencji, szczególnie sieci neuronowych, do rozpoznawania wzorców i zależności w danych. *Deep Learning*, będący poddziedziną uczenia maszynowego, różni się od tradycyjnych metod uczenia maszynowego tym, że automatycznie wykonuje proces ekstrakcji cech z surowych danych, bez potrzeby ręcznego selekcjonowania istotnych atrybutów. Proces ten odbywa się dzięki warstwom sieci neuronowych, które są odpowiedzialne za stopniowe przekształcanie danych wejściowych na reprezentacje coraz bardziej złożone.

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. Używa się tutaj algorytmu optymalizacji, najczęściej **spadku gradientu** (*Gradient Descent*), który pozwala na modyfikację wag w taki sposób, by minimalizować błąd predykcji.

W procesie tym kluczową rolę odgrywa funkcja aktywacji, która wprowadza nieliniowość do modelu i pozwala na modelowanie bardziej złożonych zależności. Przykładowe funkcje aktywacji to **ReLU** (*Rectified Linear Unit*), **sigmoida** czy **tanh**, które pomagają w decyzji, czy dane wejściowe mają być uwzględnione w obliczeniach w danej warstwie.

W trakcie treningu model uczy się generalizować, czyli wykrywać ogólne wzorce, które będą skuteczne nie tylko dla danych treningowych, ale także dla nowych, niewidzianych wcześniej przykładów. Aby uniknąć przeuczenia (*overfitting*), stosuje się techniki, takie jak regularizacja, ***dropout*** czy ***early stopping***, które pomagają utrzymać model w równowadze pomiędzy dokładnością a zdolnością do uogólniania.

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 najbardziej dokładne.

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. To źródło dostarcza narzędzie o nazwie PARVUS, które jest pakietem rozszerzalnym do eksploracji danych, klasyfikacji i analizy korelacji. Zostało opracowane przez zespół badaczy w Instytucie, który specjalizuje się w analizach farmaceutycznych i żywnościowych. Pakiet ten umożliwia dokładną analizę chemiczną win i może być używany do różnych celów badawczych, w tym klasyfikacji różnych rodzajów win.

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.

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 1: 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, nie widzianych 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 na 70% do 30% w celach eksperymentów.

Podział danych na zbiory treningowe i testowe pozwala na ocenę skuteczności modelu na nowych, nie widzianych 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. Po podziale dane są dodatkowo sortowane według klasy, aby zapewnić równomierne rozłożenie próbek w zbiorach treningowym i testowym.

```rust
fn split_data(
    x: Vec<Vec<f32>>,
    y_t: Vec<i32>
) -> (Vec<Vec<f32>>, Vec<i32>, Vec<Vec<f32>>, Vec<i32>) {
    let mut rng = thread_rng();
    let mut combined: Vec<(Vec<f32>, i32)> = x.into_iter().zip(y_t.into_iter()).collect();
    combined.shuffle(&mut rng);

    let num_records = combined.len();
    let num_train = (num_records as f32 * 0.8) as usize;

    // Split data into training and testing sets
    let (train_data, test_data): (Vec<_>, Vec<_>) = combined
      .into_iter()
      .enumerate()
      .partition(|&(i, _)| i >= num_train);

    // Sort by y_t class and unzip to separate x and y_t
    let mut train_sorted: Vec<_> = train_data.into_iter().map(|(_, data)| data).collect();
    train_sorted.sort_by_key(|&(_, y)| y);
    let (x_train, y_t_train): (Vec<_>, Vec<_>) = train_sorted.into_iter().unzip();

    let mut test_sorted: Vec<_> = test_data.into_iter().map(|(_, data)| data).collect();
    test_sorted.sort_by_key(|&(_, y)| y);
    let (x_test, y_t_test): (Vec<_>, Vec<_>) = test_sorted.into_iter().unzip();

    (x_train, y_t_train, x_test, y_t_test)
}
```

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	magnesium	total_phenols	flavanoids	nonflavanoid_phenols	proanthocyanins	color_intensity	hue	od280_od315_of_diluted_wines	proline
1	0.7973686	0.17588931	0.4919786	0.2783505	0.6086956	0.6965517	0.5978465	0.20754716	0.533123	0.37286687	0.49593496	0.8937729	0.3580599
1	0.8394738	0.18972331	0.50267375	0.2938144	0.5217391	0.76551723	0.5611815	0.245283	0.51104105	0.4351536	0.37398374	0.7472527	0.4935806
1	0.7368421	0.17984188	0.66310155	0.3402062	0.26086956	0.50689656	0.5598718	0.16981132	0.59305996	0.36860964	0.6178862	0.7692307	0.7039943
1	0.67105263	0.18181816	0.5347593	0.43814433	0.39130434	0.6482758	0.6012659	0.16981132	0.48580447	0.47952217	0.49593496	0.5897436	0.882311
1	0.8394738	0.64229244	0.61497325	0.1340206	0.6304348	0.6965517	0.5696203	0.13207547	0.52681386	0.32593855	0.3333333	0.82783886	0.34379458
1	0.6842106	0.21146242	0.71657753	0.3402062	0.45652175	0.64482756	0.5421941	0.32075474	0.3312303	0.5136519	0.6504065	0.5897436	0.7360913
1	0.7447368	0.12055335	0.486631	0.2783505	0.3043478	0.6896551	0.5928271	0.16981132	0.4542587	0.58682586	0.4308943	0.83516484	0.5470756
1	0.88421065	0.22332014	0.5828877	0.28618556	0.2826087	0.5241379	0.45991564	0.32075474	0.49526817	0.3387372	0.43902436	0.8461538	0.72182596
1	0.883579	0.56326186	0.4919786	0.2783505	0.3478261	0.7827586	0.5978465	0.26415095	0.56151426	0.3888737	0.4552845	0.7948718	0.5613409
1	0.531579	0.19565216	0.36363634	0.092783466	0.23913044	0.59999996	0.6181435	0.07547171	0.7886436	0.58511944	0.5203252	0.6007326	0.6219686
1	0.5921052	0.1778656	0.79144377	0.2525773	0.4347826	0.55862063	0.49367894	0.3962264	0.29968458	0.28327644	0.49593496	0.5531135	0.4293866
1	0.7657894	0.19565216	0.486631	0.35051563	0.41304347	0.6551724	0.6751055	0.35849053	0.52681386	0.6581786	0.5203252	0.67032963	0.70042795
1	0.5947369	0.24308299	0.7058824	0.3195876	0.3478261	0.6965517	0.6097847	0.33962265	0.3943218	0.46273038	0.47967482	0.57509154	0.7075686
1	0.6657896	0.1916996	0.58082135	0.2886598	0.51086956	0.7482759	0.6223629	0.3962264	0.60883284	0.41382256	0.3821138	0.77289385	0.36875892
1	0.6657896	0.19565216	0.5882353	0.5103093	0.5	0.6827586	0.51476794	0.13207547	0.6435332	0.4248014	0.48658486	0.64468867	0.6005706
1	0.7105262	0.15019761	0.71657753	0.61340207	0.33695653	0.6965517	0.6139241	0.30188677	0.6214512	0.3771331	0.5772358	0.52747256	0.71825963
1	0.84210515	0.1916996	0.57219255	0.25773197	0.6195652	0.6275862	0.5738397	0.28301886	0.59305996	0.37201363	0.4552845	0.970696	0.5613409
1	0.6868422	0.4604031	0.6417112	0.23711337	0.5	0.59310347	0.5675186	0.07547171	0.3943218	0.32593855	0.3982439	0.7655677	0.40442225

Rysunek 1: 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, podtrzebne 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 strukture *Dataset* (plik `dataset.rs`, w części skryptowej) jako dane wejściowe i dzieli je na wsady o określonym rozmiarze.

W moim przypadku, struktura `WineDataset` przechowuje 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.

5.1.1. Implementacja tensorów

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 przpadku 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`

6. Skrypt

```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`
```

```rust
    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;
  }

  (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
}

/// 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());
    }
```

```rust
    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>) {
  let mut rng = thread_rng();
  let mut combined: Vec<(Vec<f32>, i32)> = x.into_iter().zip(y_t.into_iter()).collect();
  combined.shuffle(&mut rng);

  let num_records = combined.len();
  let num_train = (num_records as f32 * 0.7) as usize;

  // Split data into training and testing sets
  let (train_data, test_data): (Vec<_>, Vec<_>) = combined
    .into_iter()
    .enumerate()
    .partition(|&(i, _)| i >= num_train);

```

```rust
    // Sort by y_t class
    let mut train_sorted: Vec<_> = train_data.into_iter().map(|(_, data)| data).collect();
    train_sorted.sort_by_key(|&(_, y)| y);
    let (x_train, y_t_train): (Vec<_>, Vec<_>) = train_sorted.into_iter().unzip();

    let mut test_sorted: Vec<_> = test_data.into_iter().map(|(_, data)| data).collect();
    test_sorted.sort_by_key(|&(_, y)| y);
    let (x_test, y_t_test): (Vec<_>, Vec<_>) = test_sorted.into_iter().unzip();

    (x_train, y_t_train, x_test, y_t_test)
}
```

Program 5: `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,

    #[serde(rename = "Alcalinity of ash")]
    pub alcalinity_of_ash: f32,

    #[serde(rename = "Magnesium")]
    pub magnesium: f32,

    #[serde(rename = "Total phenols")]
    pub total_phenols: f32,

    #[serde(rename = "Flavanoids")]
    pub flavanoids: f32,

    #[serde(rename = "Nonflavanoid phenols")]
    pub nonflavanoid_phenols: f32,

    #[serde(rename = "Proanthocyanins")]
    pub proanthocyanins: f32,

    #[serde(rename = "Color intensity")]
    pub color_intensity: f32,

    #[serde(rename = "Hue")]
    pub hue: f32,

    #[serde(rename = "OD280/OD315 of diluted wines")]
    pub od280_od315_of_diluted_wines: f32,

    #[serde(rename = "Proline")]
    pub proline: f32,
```

```rust
}

pub struct WineDataset {
  pub(crate) dataset: InMemDataset<WineItem>,
}

impl WineDataset {
  pub fn subset(&self, indices: &[usize]) -> Self {
    let data: Vec<WineItem> = indices
      .iter()
      .map(|&i| self.dataset.get(i).unwrap().clone())
      .collect();
    Self {
      dataset: InMemDataset::new(data),
    }
  }
}

impl WineDataset {
  pub fn from_pkl(split: &str) -> Result<Self, io::Error> {
    let x_file = File::open(format!("./data/x_{}.pkl", split))?;
    let y_file = File::open(format!("./data/y_t_{}.pkl", split))?;

    let x_map: BTreeMap<String, Vec<Vec<f32>>> =
      serde_pickle::from_reader(x_file, Default::default())
        .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?;
    let y_map: BTreeMap<String, Vec<i32>> =
      serde_pickle::from_reader(y_file, Default::default())
        .map_err(|e| io::Error::new(io::ErrorKind::InvalidData, e))?;

    let mut data = Vec::new();

    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()
  }
```

```
112
113   pub fn test() -> Self {
114     Self::from_pkl("test").unwrap()
115   }
116 }
117
118 impl Dataset<WineItem> for WineDataset {
119   fn get(&self, index: usize) -> Option<WineItem> {
120     self.dataset.get(index)
121   }
122
123   fn len(&self) -> usize {
124     self.dataset.len()
125   }
126 }
```

Program 6: `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 }
  }
}

#[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,
        )
      })
```

```
46        .collect::<Vec<_>>();
47
48      let y = data
49        .iter()
50        .map(|wine| Tensor::<B, 1, Int>::from_data([wine.class], &self.device))
51        .collect::<Vec<_>>();
52
53      let x = Tensor::<B, 2>::cat(x, 0).to_device(&self.device);
54      let y = Tensor::<B, 1, Int>::cat(y, 0).to_device(&self.device);
55
56      WineBatch { x, y }
57    }
58  }
```

Program 7: `batcher.rs`