Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Pavlo Boidachenko

Nr albumu: 1124969

Aplikacja uczenia maszynowego metodą SVM

Praca licencjacka
na kierunku informatyki

Praca wykonana pod kierunkiem
dr Grzegorz Surówka
Zakład Technologii Informatycznych

Kraków 2019

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....	.....
Kraków, dnia	Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

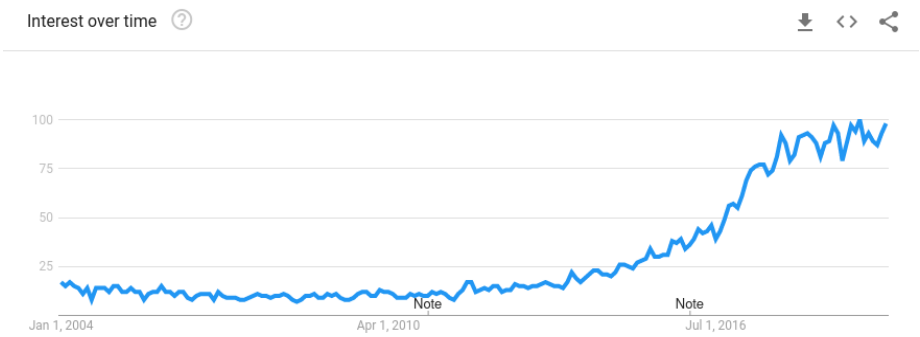
.....	.....
Kraków, dnia	Podpis kierującego pracą

Contents

1 Wstęp

1.1 Motywacja

W aktualne czasy temat Uczenia Maszynowego jest popularny1.1 jak nigdy do tego. Projekty z użyciem Uczenia Maszynowego pozwalają na tworzenie aplikacji które jeszcze 10 lat temu trudno było wyobrazić.



Rysunek 1.1: Machine Learning trends
Źródło: Google Trends

Rozpowszechnienie Uczenia Maszynowego również spowodowało i moje zainteresowanie tematem. Z tego powodu dla swojej pracy licencjackiej wybrałem temat: Aplikacja uczenia maszynowego metodą SVM. Po zakończeniu pracy spodziewam się podwyższyć swoją kompetencje w dziale Uczenia Maszynowego.

1.2 Cel

Celem mojej pracy licencjackiej jest stworzenie oprogramowania pozwalającego na generowanie modeli używając Maszyny wektorów wspierających *(ang. Support Vector Machine, SVM)* z graficznym interfejsem użytkownika. Program będą mogli użyć osoby potrzebujące szybko przetrenować kilka modeli, przetestować ich dla różnych parametrów, zwizualizować dane. Program ma na celu ułatwienie pracę z Maszyną wektorów wspierających poprzez graficzny interfejs użytkownika oparty na bibliotekę QT. Część funkcjonalna programu jest oparta o bibliotekę LIBSVM[1].

1.3 Zakres

Program powinien móc ustawiać parametry dla wybranej metody oraz jądra *(*ang. kernel), generować wykresy podawanych zbiorów danych, interpretować różne formaty zbiorów danych, wykonywać Sprawdzian krzyżowy (ang. Cross validation, CV), mieć metodę do optymalizacji parametrów, pokazywać wyniki trenowania oraz testowania modeli.

2 Metoda klasyfikacji SVM

W tym paragrafie w ogólnych zarysach jest opisana Maszyna Wektorów Wspierających oraz jej typy zaimplementowane w LIBSVM. Również będą krótkie opisy zaimplementowanych jąder.

2.1 Opis

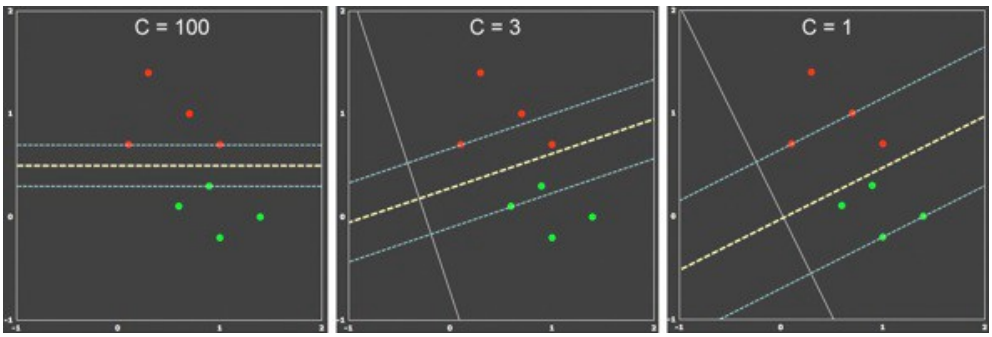
Swój program napisałem w oparciu o bibliotekę LIBSVM[1]. Maszyna Wektorów Wspierajacych(ang. Support Vector Machine. SVM) - klasyfikator, nauka którego ma na celu wyznaczenie hiperpłaszczyzny rozdzielającej dwie klasy z maksymalnym marginesem. Zaletą takiego klasyfikatora jest to że po uczeniu margines mówi jak dobrze są odseparowane klasy. LIBSVM implementuje pięć typów Maszyny Wektorów Wspierajacych C-SVC, ν-SVC, One class SVM, ϵ-SVR, ν-SVR.

2.2 C-SVC

C-Support Vector Classification - rodzaj klasyfikatora używający C jako parametr regularyzacji. Jeśli jest dany wektor $x_i \in R^n$, $i = 1, ..., l$ w dwóch klasach i wektor etykiet $y_i \in \{1, -1\}$ to C-SVC rozwiązuję tak sformułowany problem:

$$\begin{aligned} \min_{\boldsymbol{\omega}, b, \varepsilon} \quad & \tfrac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} + C\sum_{i=1}^{l}\varepsilon_i \\ \text{Z zastrzeżeniem że} \quad & y_i(\boldsymbol{\omega}^T\phi(x_i) + b) \geq 1 - \varepsilon_i \\ & \varepsilon_i \geq 0, i = 1, ..., l \end{aligned}$$

Parametr C służy do ustawienia marginesu: duży C $\rightarrow$ mały margines, mały C $\rightarrow$ duży margines.



Rysunek 2.1: Zależność marginesu od parametru C
Źródło: `https://medium.com/@pushkarmandot`

Dobry model dobrze separuję dane i razem z tym ma duży margines. Natomiast w w rzeczywistości jedno wyłącza drugie: duży margines włącza punkty z dwóch klas, a dobre separowanie może powodować przeuczanie(ang. Overfitting). Przeuczanie może skutkować tym że model jest dobry na danych treningowych ale jest zły na danych testowych.

2.3 ν-SVM

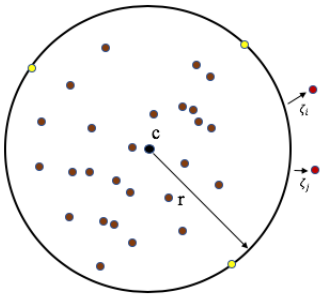
ν-Support Vector Classification - rodzaj klasyfikatora używający ν jako parametr regularyzacji. Jest bardzo podobny do C-SVM, z różnicą że $\nu \in [0, 1]$. Przyjemną właściwością ν jest to że on jest dolną granicą stosunku wektorów wspierających i górną granicą stosunku błędu uczenia.

Jeśli jest dany wektor $x_i \in R^n$, $i = 1, ..., l$ w dwóch klasach i wektor $y \in R^l$ taki że $y_i \in \{1, -1\}$ to pierwotny problem optymalizacji wygląda następująco:

$$\begin{aligned} \min_{\boldsymbol{\omega}, b, \varepsilon, \rho} \quad & \tfrac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} - \nu\rho + \tfrac{1}{l}\sum_{i=1}^{l}\varepsilon_i \\ \text{Z zastrzeżeniem że} \quad & y_i(\boldsymbol{\omega}^T\phi(x_i) + b) \geq \rho - \varepsilon_i \\ & \varepsilon_i \geq 0,\ i = 1, ..., l,\ \rho \geq 0 \end{aligned}$$

2.4 One-class SVM

One-class Support Vector Machine - rodzaj klasyfikatora uczenia nienadzorowanego, które zakłada brak etykiet w danych uczących. Ma na celu znalezienie niewiadomych wzorców/anomalii(klastrów) w danych wejściowych.



Rysunek 2.2: Hipersfera zawierająca punkty danych. Ma środek c i promień R. Punkty na krawędzi są wektorami wspierającymi.
Źródło: Wikipedia

Jeśli dany jest wektor $x_i \in R^n$, $i = 1, ..., l$ bez informacji o klasach, to pierwotny problem optymalizacji wygląda następująco:

$$\begin{aligned} \min_{\boldsymbol{\omega}, \varepsilon, \rho} \quad & \tfrac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} - \rho + \tfrac{1}{\nu l}\sum_{i=1}^{l}\varepsilon_i \\ \text{Z zastrzeżeniem że} \quad & \boldsymbol{\omega}^T\phi(x_i) \geq \rho - \varepsilon_i, \\ & \varepsilon \geq 0,\ i = 1, ..., l \end{aligned}$$

2.5 ϵ-SVR

Jeśli wektor etykiet $y_i \in R$ to jest używana metoda regresji. ϵ-Support Vector Classification - używa C i ϵ jako parametrów regularyzacji. Celem jest znalezienie takiej funkcji $f(x)$ że jej wartość odchyla się od y_n na wartość nie większą od ϵ dla każdego punktu z zbioru treningowego.

Jeśli jest dany zbiór danych treningowych $\{(x_1, x_1), ..., (x_l, z_l)\}$, gdzie $x - I \in R^n$ jest wektorem cech, a $z_i \in R^1$ jest wyjściem. Przy danych parametrach $C > 0$ i $\epsilon > 0$, standardowa forma SVR to:

$$\begin{aligned} \min_{\boldsymbol{\omega},b,\boldsymbol{\varepsilon},\boldsymbol{\varepsilon}^*} \quad & \tfrac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} + C\sum_{i=1}^{l}\varepsilon_i + C\sum_{i=1}^{l}\varepsilon_i^* \\ \text{Z zastrzeżeniem że} \quad & \boldsymbol{\omega}^T\phi(x_i) + b - z_i \leq \epsilon + \varepsilon_i, \\ & z_i - \boldsymbol{\omega}^T\phi(x_i) - b \leq \epsilon + \varepsilon_i^*, \\ & \varepsilon_i, \varepsilon_i^* \geq 0, i = 1, ..., l \end{aligned}$$

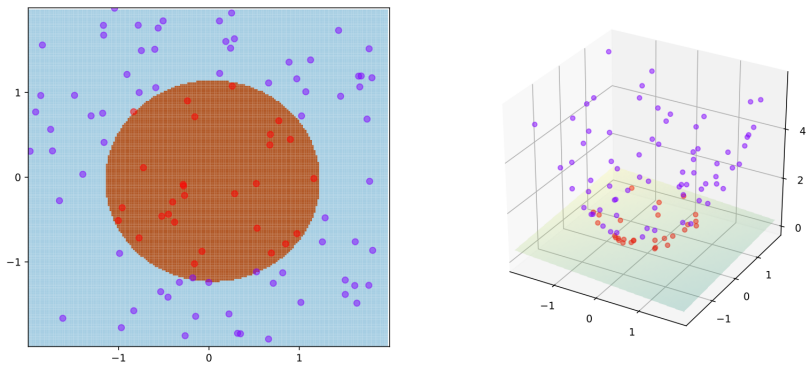
2.6 ν-SVR

ν-Support Vector Regression - podobnie do ν-SVC, używa parameter $\nu \in (0, 1]$ dla kontroli liczby wektorów wspierających. Również używa parametru ϵ. Z parametrami (C, ν) ν-SVR rozwiązuję:

$$\begin{aligned} \min_{\boldsymbol{\omega},b,\boldsymbol{\varepsilon},\boldsymbol{\varepsilon}^*,\epsilon} \quad & \tfrac{1}{2}\boldsymbol{\omega}^T\boldsymbol{\omega} + C(\nu\epsilon + \tfrac{1}{l}\sum_{i=1}^{l}(\varepsilon_i + \varepsilon_i^*)) \\ \text{Z zastrzeżeniem że} \quad & (\boldsymbol{\omega}^T\phi(x_i) + b) - z_i \leq \epsilon + \varepsilon_i, \\ & z_i - (\boldsymbol{\omega}^T\phi(x_i) + b) \leq \epsilon + \varepsilon_i^*, \\ & \varepsilon_i, \varepsilon_i^* \geq 0,\ i = 1, ..., l,\ \epsilon \geq 0 \end{aligned}$$

2.7 Jądra

W przypadku kiedy klasy nierozdzielne liniowo to jest używany trik z zastosowaniem funkcji jądrowych(ang. Kernel Trick). Funkcję jądrowe pozwalają odwzorować zbiór danych w przestrzeń z większą liczbą wymiarów, w której klasy danego zbioru będzie można rozdzielić hiperpłaszczyzną.



Rysunek 2.3: Przykład stosowania funkcji jądrowej dla nierozdzielnego liniowo zbioru danych
Źródło: Wikipedia

W bibliotece LIBSVM są zaimplementowane następne funkcje jądrowe:

$$\begin{aligned} \text{Liniowa} \quad & K(x, x') = x \cdot x' \\ \text{Wielomianowa} \quad & K(x, x') = (\gamma * x \cdot x' + coef0)^{degree}) \\ \text{RBF} \quad & K(x, x') = -\gamma * (\sqrt{x} + \sqrt{x'} - 2 * x \cdot x') \\ \text{Sigmoidalna} \quad & K(x, x') = tanh(\gamma * x \cdot x' + coef0) \end{aligned}$$

Autorzy biblioteki LIBSVM polecają [3] że pierwsza funkcja jądrowa którą warto spróbować to RBF. Jak pokazują [5] jeśli używać RBF z optymalizacją hiperparametrów to nie ma potrzeby rozważać liniową funkcję jądrową. W sigmoidalnej funkcji jądrowej macierz jądrowa może nie być pozytywnie określona i generalnie dokładność modelu nie jest lepsza od RBF [6].

3 Projekt aplikacji

W tym paragrafie są opisane użyte technologie przy napisaniu aplikacji i interfejs użytkownika.

3.1 Opis

3.2 Technologie

W roli jądra mojej aplikacji występuję biblioteka *LIBSVM* napisana w 2011 roku przez Chih-Chung Chang i Chih-Jen Lin w Państwowym Uniwersytecie Tajwańskim. Celem autorów było zrobienie narzędzia z prostym interfejsem umożliwiające używanie Maszyny Wektorów Wspierajacych użytkownikom które nie zajmują się zawodowo nauczaniem maszynowym. Domyślnie użycie biblioteki *LIBSVM* polega na skompilowaniu i używaniu dwóch programów: *svm-train* i *svm-predict*. *svm-train* służy do trenowania modelu, a *svm-predict* do testowania, chociaż również może być używany do klasyfikacji nowych niegrupowanych danych.

```
Usage: svm-train [options] training_set_file [model_file]
options:
-s svm_type : set type of SVM (default 0)
        0 -- C-SVC              (multi-class classification)
        1 -- nu-SVC             (multi-class classification)
        2 -- one-class SVM
        3 -- epsilon-SVR        (regression)
        4 -- nu-SVR             (regression)
-t kernel_type : set type of kernel function (default 2)
        0 -- linear: u'*v
        1 -- polynomial: (gamma*u'*v + coef0)^degree
        2 -- radial basis function: exp(-gamma*|u-v|^2)
        3 -- sigmoid: tanh(gamma*u'*v + coef0)
        4 -- precomputed kernel (kernel values in training_set_file)
-d degree : set degree in kernel function (default 3)
-g gamma : set gamma in kernel function (default 1/num_features)
-r coef0 : set coef0 in kernel function (default 0)
-c cost : set the parameter C of C-SVC, epsilon-SVR, and nu-SVR (default 1)
-n nu : set the parameter nu of nu-SVC, one-class SVM, and nu-SVR (default 0.5)
-p epsilon : set the epsilon in loss function of epsilon-SVR (default 0.1)
-m cachesize : set cache memory size in MB (default 100)
-e epsilon : set tolerance of termination criterion (default 0.001)
-h shrinking : whether to use the shrinking heuristics, 0 or 1 (default 1)
-b probability_estimates : whether to train a SVC or SVR model for probability estimates, 0 or 1 (default 0)
-wi weight : set the parameter C of class i to weight*C, for C-SVC (default 1)
-v n: n-fold cross validation mode
-q : quiet mode (no outputs)
```

Rysunek 3.1: Użycie programu svm-train

```
Usage: svm-predict [options] test_file model_file output_file
options:
-b probability_estimates: whether to predict probability estimates, 0 or 1 (default 0); for one-class SVM only 0 is su
pported
-q : quiet mode (no outputs)
```

Rysunek 3.2: Użycie programu svm-predict

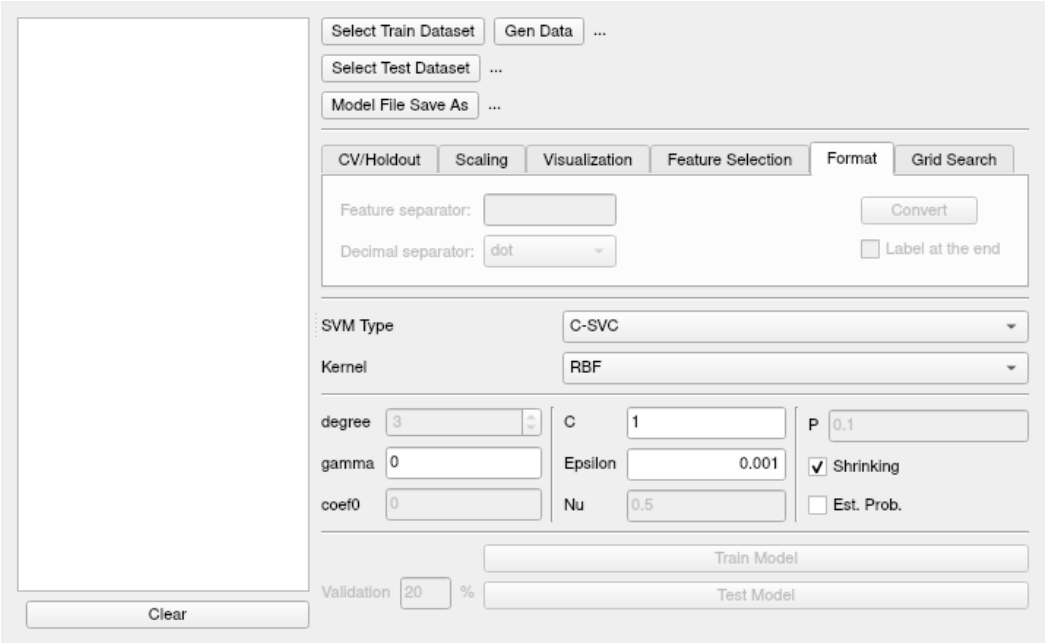
Aplikacja głównie jest napisana w języku *C++*. Wybrałem ten język programowania przez jego wydajność mimo to że jest on językiem wysokiego poziomu. Na wybór *C++* również wpłynęła moja sympatia do tego języka programowania, a napisanie dużego projektu w nim było dla mnie prawdziwym wyzwaniem, które pozwoliło pogłębić swoją wiedze w programowaniu i używaniu języka *C++*. Niektóre części aplikacji są napisane w języku Python, w szczególności te, które dotyczą rysowania wykresów, pracy z plikami i zmiennymi typu *string*. Standardowa biblioteka Python przedstawia sporo narzędzi dla pracy z plikami i zmiennymi typu *string*, co pozwala oszczędzać czas

programisty w porównaniu do C++, w którym napisanie podobnej funkcjonalności zajęłoby o wiele więcej czasu. Dla rysowania wykresów użyłem biblioteki *matplotlib* [4] która pozwala stosunkowo łatwo rysować jedno, dwu i trzy-wymiarowe wykresy, a w kombinacji z *numpy* i *scipy* generować wykresy gęstości.

Interfejs graficzny był pisany za pomocą biblioteki *Qt* [2] napisanej w C++. Twórcy biblioteki *Qt* stworzyli IDE dedykowane do pracy z biblioteką - *Qt Creator*. Jest ono wyposażone w szczegółową dokumentacje do biblioteki, wbudowany program *Qt Designer* pozwala projektować interfejs graficzny metodą przeciągnij i upuść(ang. drag and drop), automatyczne uzupełnienie kodu, prowadzi analizę semantyczną kodu, wskazuje na błędy do kompilacji.

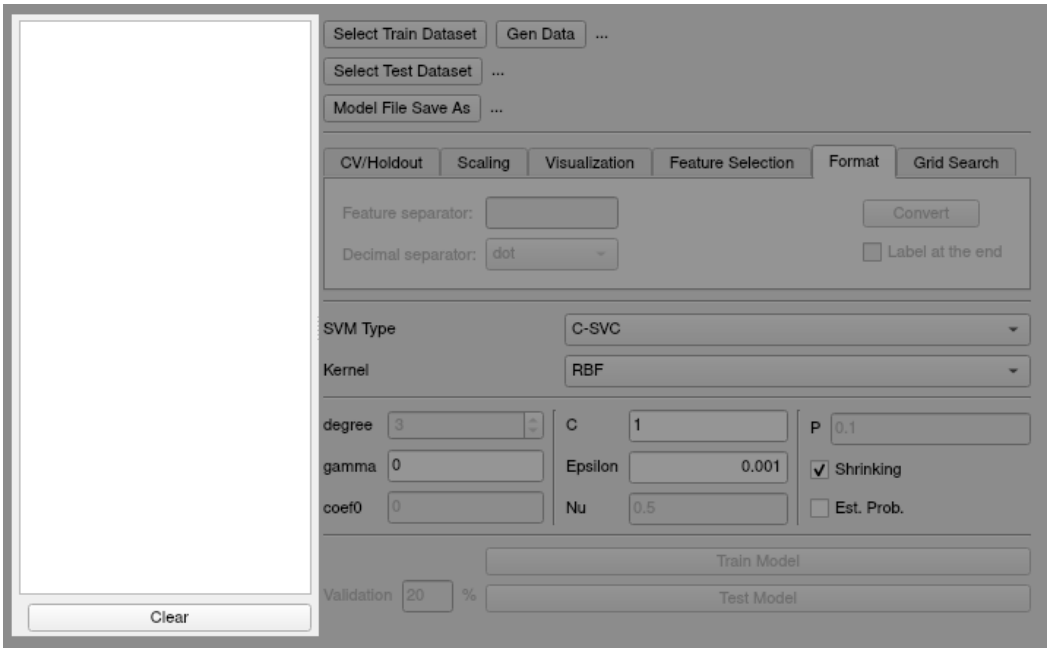
3.3 Interfejs użytkownika i realizacja funkcjonalności

Po uruchomieniu programu pierwsze co zobaczy użytkownik to okno główne. Po lewej stronie jest pole tekstowe w którym pojawiają się komunikaty o wynikach trenowania lub testowania modelu. W prawej górnej części okna są kontrolki do wybierania plików z danymi lub zapisywania pliku modelu. Żeby rozdzielić główną funkcjonalność od pobocznej i zrobić okno główne programu bardziej kompaktowym niektóre elementy są zrealizowane w postaci kart, natomiast funkcjonalność która dotyczy trenowania i testowania modelu jest dostępna bezpośrednio z okna głównego.



Rysunek 3.3: Okno główne programu

3.3.1 Pole tekstowe dla komunikatów.



Rysunek 3.4: Pole dla komunikatów

Pole tekstowe 3.4 w lewej części okna głównego programu przyznaczone dla wypisywania komunikatów z biblioteki *LIBSVM* lub używanych skryptów Python. W zasadzie opisywana kontrolka jest widżetem biblioteki *Qt TextEdit* w którym jest wyłączona możliwość edytowania zawartości. Dla tego żeby nie przerabiać kodu źródłowego biblioteki *LIBSVM* zamieniając każde wywołanie funkcji *printf* na odpowiednią funkcje, któraby dodawała tekst na *TextEdit*, szukałem sposobu na przekierowanie całego wyjścia aplikacji na widżet w interfejsie graficznym. Niestety biblioteka *Qt* nie przedstawia prostej możliwości na zrobienie tego, więc musiałem zraimplementować podobną funkcjonalność samodzielnie. Za tą funkcjonalność odpowiada klasa *OutputHandler*, która w swoim konstruktorze tworzy potok do którego przekierowuje *stdout*.

```
OutputHandler::OutputHandler()
{
    _saved_stdout = dup(STDOUT_FILENO);
    pipe(_stdout_pipe);
    dup2(_stdout_pipe[1], STDOUT_FILENO);
    close(_stdout_pipe[1]);
}
```

Listing 1: Konstruktor klasy *OutputHandler*

Za tym uruchamia wątek który ciągle sprawdza czy coś było zapisane do potoku i jeśli tak to wywołuje sygnał który dopisuje te dane do *TextEdit* na interfejsie graficznym.

```
void OutputHandler::handleOutput()
{
    int bytes_read;
    char buf[1024];
    while((bytes_read = (int)read(_stdout_pipe[0], buf, sizeof(buf))))
    {
        emit updateOutput(QString::fromLatin1(buf, bytes_read));
        if(_cmd_out)
        {
            write(_saved_stdout, buf, bytes_read);
        }
    }
}
```

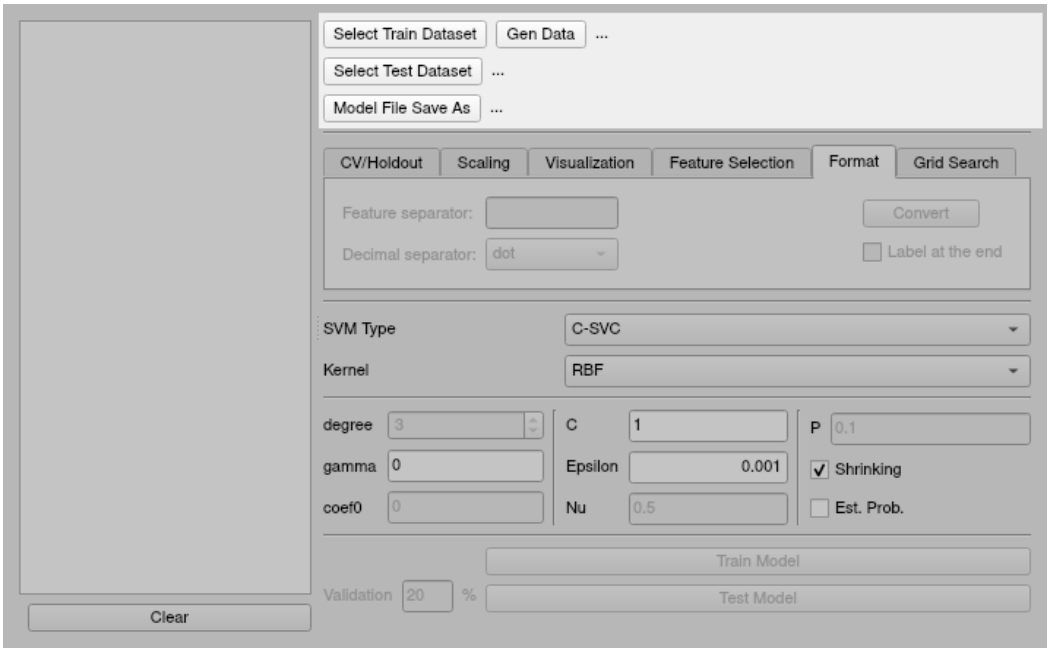
Listing 2: Funkcja uruchamiana w wątku

Takie podejście pozwala używać standardowe sposoby na wypisywanie komunikatów i nie myśleć o tym czy wypisywany komunikat zostanie wyświetlony na interfejsie użytkownika.

Pod omawianym polem tekstowym znajduje się przycisk *Clean*, który usuwa wszystkie wcześniej wypisane komunikaty z pola *TextEdit*.

3.3.2 Zarządzanie plikami z danymi

Pierwszym krokiem który będzie musiał zrobić użytkownik po uruchamianiu aplikacji - wybrać plik z danymi dla trenowania. Dla tego celu na ekranie głównym są odpowiednie kontrolki 3.5 które pozwalają wybrać plik z danymi do trenowania lub testowania i wybrać katalog w którym należy zapisać plik z wygenerowanym modelem. Domyślnie plik modelu ma nazwę [nazwa pliku z danymi do trenowania + .model]. Przycisk *Gen Data* pozwala wygenerować plik z danymi samodzielnie, jego funkcjonalność będzie omówiona później.



Rysunek 3.5: Zarządzanie plikami

Po wybraniu pliku z danymi program sprawdza czy dane są zapisane w formacie *LIBSVM*. W przypadku poprawności danych program parsuje plik wypisując liczbę rekordów, liczbę klas i liczbę cech jak dla danych trenujących tak i dla danych testowych.

Rysunek 3.6: Przykład wczytanych danych

Format danych *LIBSVM* przewiduję klasę na skrajnej lewej pozycji w linii, cechy są indeksowane, jeśli cecha równa się 0 to informacja o nich jest zbędna. Warto wspomnieć, że wybrana wersja biblioteki *LIBSVM* nie przewiduje użycia danych tekstowych:

```
1 1:3.5 2:2.25 3:1.17 # komentarz
-1 2:3  # pierwsza i trzecia cechy są 0
```

Poprawność danych wejściowych sprawdza skrypt w języku Python udostępniany w ramach biblioteki *LIBSVM*. Jeśli plik mieści niepoprawne dane to ścieżka do wybranego pliku jest farbowana na czerwono i jest wypisywany odpowiedni komunikat. W przypadku 3.7 w pliku z danymi linia 19 mieści cechę wartość której ma liczbę rzeczywistą z przecinkiem, a w formacie danych *LIBSVM* są dopuszczane tylko kropki.

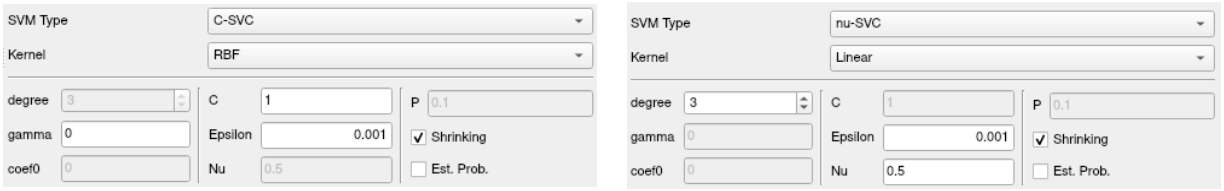
Rysunek 3.7: Przykład wczytanych niepoprawnych danych

Jeśli dane do testowania nie zgadzają się z danymi do trenowania(np. liczbą cech lub liczbą klas) to zbiór do testowania nie jest akceptowany:

Rysunek 3.8: Przykład niezgodności pomiędzy danymi do treningu a danymi do testowania

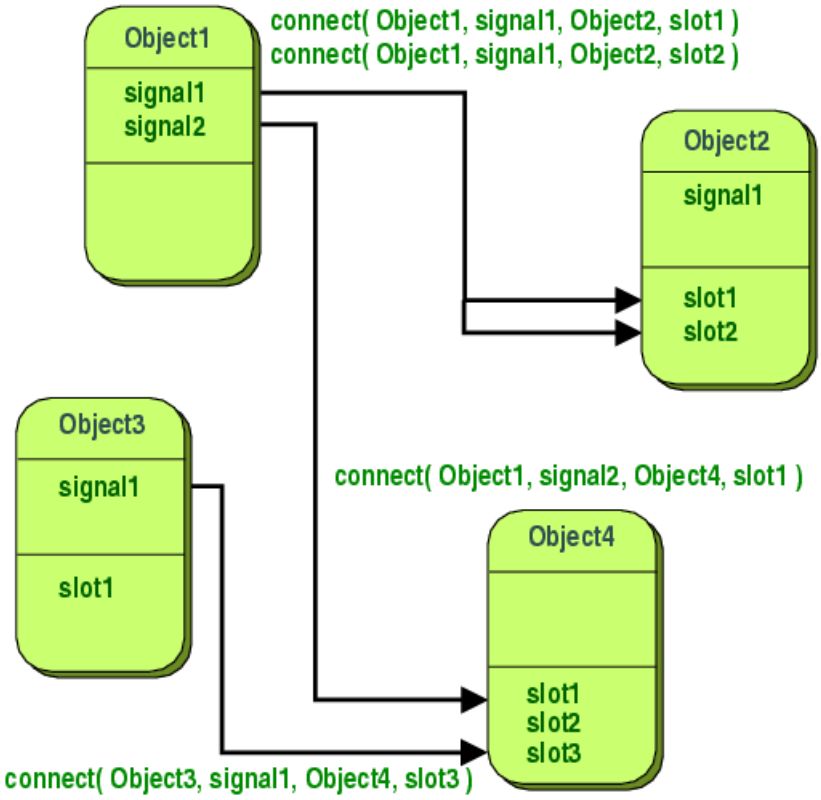
Dzięki bibliotece *Qt*, która w miarę możliwości używa natywnych do systemu operacyjnego kontrolek, wygląd selektora plików zależy od systemu operacyjnego. Domyślnie selektor plików otwiera katalog */home/[username]*, ale w przypadku kiedy pliki z danymi znajdują się głęboko w systemie plików to może być niewygodne, przez to aplikacja przechowuję ostatnio odwiedzony katalog w pliku */tmp/svmapp-lop* i zawsze otwiera w nim nowy selektor plików.

Niektóre akcje użytkownika mogą nie mieć sensu: zacząć proces trenowania bez wybrania pliku z danymi, zmieniać parametry modelu które nie dotyczą wybranego typu SVM lub wybranej funkcji jądrowej itp. Żeby zapobiec bezsensownym akcjom była zaimplementowana klasa *AvailabilityHandler*. Klasa odpowiada za deaktywacje kontrolek funkcjonalność których jest bezsensowna lub niebezpieczna 3.9



Rysunek 3.9: Przykład zmiany dostępności parametrów w zależności od wybranego typu SVM i funkcji jądra

Realizacją funkcjonalności dostępności parametrów jest ściśle związana z systemem sygnałów i gniazd w bibliotece *Qt*. Sygnały najczęściej są wywoływane przez kontrolki na interfejsie graficznym, a gniazdem jest funkcja do której są przekazywane odpowiednie parametry.



Rysunek 3.10: Wizualizacja systemu sygnałów i gniazd
Źródło: https://doc.qt.io/qt-5/signalsandslots.html

Przykład kodu w którym sygnały wywoływane w klasie *AvailabilityHandler* są łączone z gniazdami w kontrolkach z polami do wpisywania parametrów.

```
connect(availability_handler, &AvailabilityHandler::degreeEnabled, ui->degree_spinBox
    , &QSpinBox::setEnabled);
connect(availability_handler, &AvailabilityHandler::gammaEnabled, ui->gamma_lineEdit,
     &QLineEdit::setEnabled);
connect(availability_handler, &AvailabilityHandler::coef0Enabled, ui->coef0_lineEdit,
     &QLineEdit::setEnabled);
connect(availability_handler, &AvailabilityHandler::cEnabled, ui->C_lineEdit, &
    QLineEdit::setEnabled);
connect(availability_handler, &AvailabilityHandler::nuEnabled, ui->nu_lineEdit, &
    QLineEdit::setEnabled);
connect(availability_handler, &AvailabilityHandler::pEnabled, ui->P_lineEdit, &
    QLineEdit::setEnabled);
```

Żeby wyżej wymienione sygnały wywoływały się automatycznie sygnały kontrolek *Combo Box* zmieniających typ SVM lub funkcje jądrową są łączone z gniazdami w klasie *AvailabilityHandler*:

```
connect(ui->svmType_comboBox, SIGNAL(currentIndexChanged(int)), availability_handler
    , SLOT(filterSVMTypeParams(int)));
connect(ui->kernel_comboBox, SIGNAL(currentIndexChanged(int)), availability_handler,
    SLOT(filterKernelParams(int)));
```

Z sygnału *currentIndexChanged* jest przekazywany indeks wybranego elementu do gniazda *filterSVMTypeParams* w którym są wywoływany odpowiednie sygnały zarządzające dostępnością parametrów.

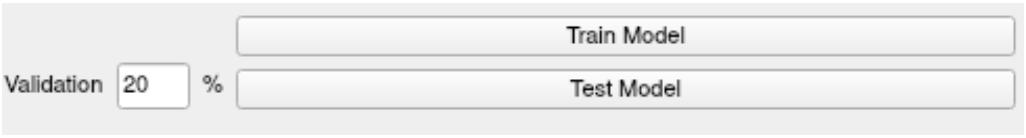
3.3.3 Parametry SVM i funkcji jądrowych

Wszystkie parametry oprócz *degree* są w postaci kontrolki *Line Edit* ze względu na to że są one liczbami rzeczywistymi, *degree* z kolei jest liczbą naturalną a więc używana jest kontrolka *Spin Box*. Po uruchamianiu trenowania program czyta pola z parametrami i sprawdza czy są poprawne. W pola typu *Line Edit* można wprowadzać litery(ponieważ jest dozwolony zapis liczb w notacji naukowej: *1e-3*). Jeśli wprowadzoną liczba nie jest w formacie naukowym lub nie jest liczbą rzeczywistą z separatorem kropką, to pole z błędnymi danymi jest obramiane na czerwono a proces treningu jest powstrzymywany.

Rysunek 3.11: Przykład niepoprawnych danych w polu parametru C

3.3.4 Przyciski trenowania i testowania

Przyciski trenowania i testowania 3.12 są umieszczone w dolnej części okna głównego programu.



Rysunek 3.12: Przyciski do trenowania i testowania modelu

Po wciśnięciu przycisku *Train Model* aplikacja, jak opisano wyżej, parsuje i sprawdza pola z parametrami i uruchamia proces trenowania dokładnie ten który oferuję program *svm-train* z biblioteki *LIBSVM*. Żeby ściślej powiązać aplikację z biblioteką *LIBSVM* przerobiłem biblioteczne plik źródłowe *svm-train.c* i *svm-predict.c* na klasę *SVMController*, pozwoliło to zapobiec używaniu polecenia *system()* które służy do uruchamiania programów zewnętrznych. Nie udało się jednak całkiem uniknąć tego - skrypty w języku Python używają *system()*.

Przykład wyjścia algorytmu trenującego LIBSVM C-SVC z funkcją jądrową RBF i parametrami $C = 1$, $gamma = 0$:

```
*
optimization finished, #iter = 318
nu = 0.469505
obj = -383.582676, rho = 0.903354
nSV = 408, nBSV = 401
Total nSV = 408
```

Gdzie *obj* oznacza wartość optymalną dla problemu podwójnego SVM, *rho* jest błędem systematycznym w funkcji decyzyjnej $sgn(w^T x - rho)$, *nSV* jest liczbą wektorów wspierających, *nBSV* - liczba ograniczonych wektorów wspierających. ν-SVC równoważną formą do C-SVC, nu - jest odpowiednikiem podanego parametru C tylko w ν-SVC, czyli jeśli użyć ν-SVC i podać jako parametr ν liczbę 0.469505 to wygenerowany model będzie miał bardzo podobne charakterystyki.

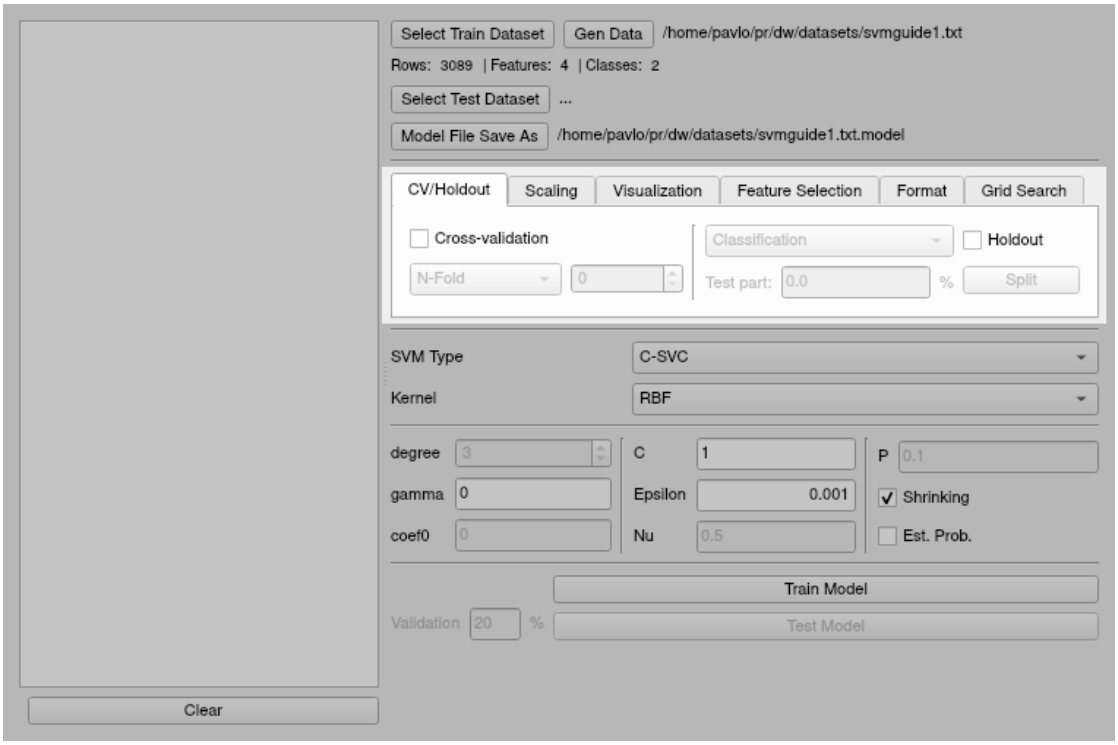
Za przyciskiem *Test Model* poza funkcjonalnością programu *svm-predict* stoi jeszcze jedna - walidacja. Klasa *SVMController* wczytuje plik z danymi do testowania i wypisuje procent dobrych zgadnięć wygenerowanego modelu, po tym w zależności od wartości w polu *Validation* program wyciąga odpowiedni procent danych ze zbioru do trenowania i przeprowadza testowanie na nim. Na ogół testowanie modelu na danych trenujących nie ma sensu ale taka operacja pozwala zidentyfikować czy dany model nie jest przeuczony(np. wynikiem walidacji jest 100%).

Przykład wyjścia algorytmu testującego:

```
Accuracy = 66.925% (2677/4000) (classification)
Validation Accuracy = 99.6769% (617/619) (classification)
```

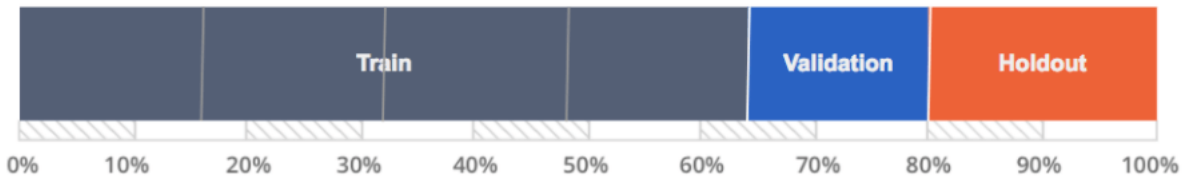
3.3.5 Walidacja krzyżowa i holdout

Walidacja krzyżowa(ang. Cross-Validation) i holdout były umieszczone na jednej karcie 3.13 dla zachowania ścisłości interfejsu graficznego, mimo to że ich funkcjonalność nie jest połączona.



Rysunek 3.13: Karta z kontrolkami do walidacji krzyżowej i holdout

Holdout polega na rozdzielaniu zbioru na dane trenujące i dane testujące. Najczęściej zbiór danych jest w jednym pliku, natomiast biblioteka *LIBSVM* jest zorganizowana tak, że potrzebuje plik z danymi trenującymi i plik z danymi testującymi. Za poprawne rozdzielenie danych odpowiadają dwa skrypty w języku Python: h4c.py i h4r.py. Pierwszy służy do klasyfikacji i stara się zachować stosunek klas w zbiorze testującym, drugi, z kolei, służy do regresji i po prostu wybiera podany procent rekordów tworząc plik z danymi do testowania. Na karcie jest umieszczona kontrolka *Combo Box* która pozwala wybrać pierwsze(*Classification*) lub drugie(*Regression*) podejścia.

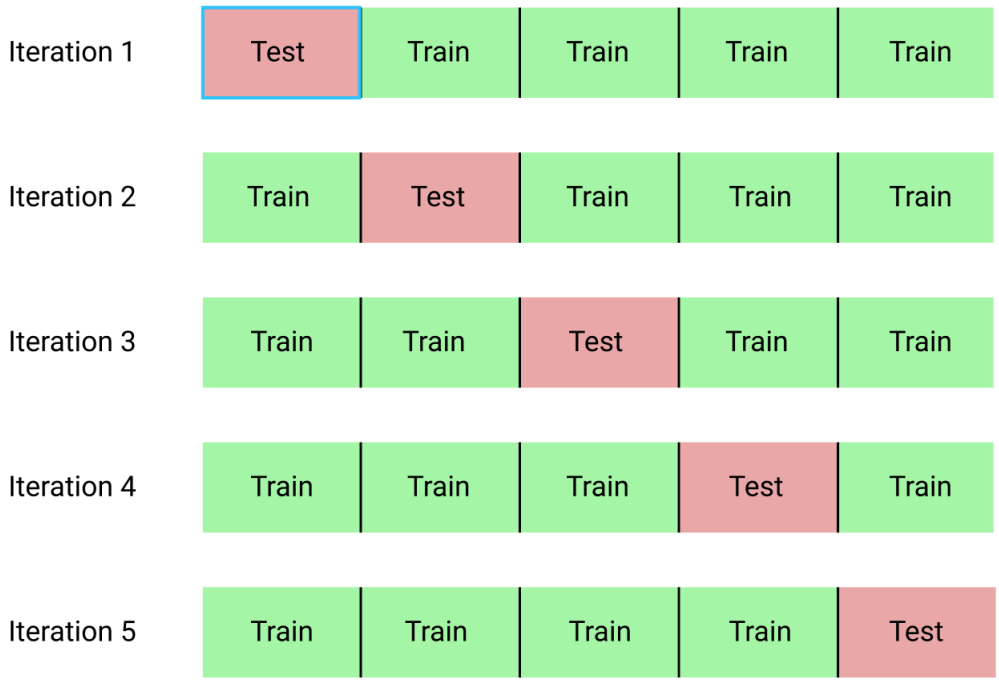


Rysunek 3.14: Przykład podziału zbioru danych. *Train* jest do trenowania, a *Holdout* i *Validation* służą do testowania modelu
Źródło: `https://www.datarobot.com/wiki/training-validation-holdout/`

Skrypt holdout dla regresji jest również używany w opisanej wyżej walidacji: on wybiera losowe rekordy z danych trenujących i tworzy z nich tymczasowy plik testujący na którym dalej działa algorytm testowania z biblioteki *LIBSVM*, przez to wynik walidacji może się różnić przy takich

samych parametrach modelu.

Walidacja krzyżowa jest bardzo ważnym narzędziem w nauczaniu maszynowym. Jeśli danych jest mało to rozdzielanie zbioru na dane trenujące i testujące może spowodować niedostateczne nauczanie się modelu, w takim przypadku używamy walidacji krzyżowej. N-krotna walidacja krzyżowa polega na rozdzielaniu zbioru na N równych części jedna z których służy to testowania a wszystkie inne do trenowania. Proces jest iteratywny, więc każda część w którejś iteracji będzie służyć do testowania.



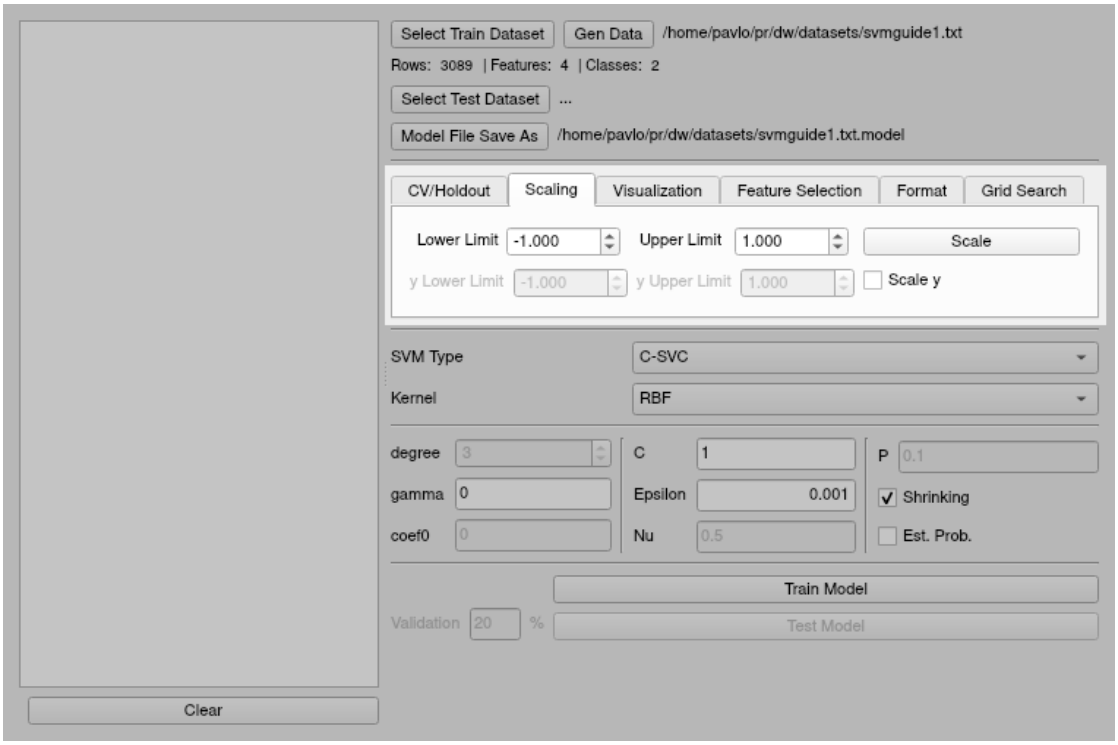
Rysunek 3.15: Ilustracja procesu 5-krotnej walidacji krzyżowej
Źródło: `https://towardsdatascience.com/cross-validation-explained-evaluating-estimator-performance-e51e5430ff85`

Również jest dostępna walidacja krzyżowa typu *Leave-one-out*, która jest rodzajem N-krotnej walidacji, tylko zbiór testujący zawsze jest jednoelementowy, czyli jest to równoważne podaniu jako N liczby $rozmiar\_zbioru - 1$.

Warto wspomnieć, że walidacja krzyżowa nie generuje modelu, a służy tylko do określania jak dobry jest model z podanymi parametrami. Kiedy użytkownik uzna dokładność modelu akceptującą on wyłączy walidacje krzyżową i wygeneruje model na całym zbiorze trenującym.

3.3.6 Skalowanie danych

Karta *Scaling* 3.16 służy do skalowania danych. Główną zaletą skalowania danych jest unikanie dominacji cech z większych zakresów liczbowych nad cechami z mniejszych zakresów liczbowych. Skalowanie również zmniejsza obciążenie numeryczne dla algorytmu, ponieważ funkcje jądrowe często zależą od iloczynu wektorowego lub skalarnego cech np. liniowa i wielomianowa funkcje jądrowe, dla których duże wartości w wektorach cech mogą powodować problemy numeryczne.



Rysunek 3.16: Karta z kontrolkami dla skalowania danych

Twórcy biblioteki *LIBSVM* proponują wszystkim początkującym zaczynać swoją pracę z generacją modelu właśnie od skalowania, ponieważ tak prosty krok może mieć bardzo wysoki wpływ na dokładność modelu.

Przykład wyjścia algorytmu trenującego na tym samym zbiorze danych z takimi samymi parametrami do i po skalowaniu danych:

```
...*..*
optimization finished, #iter = 5371
nu = 0.606150
obj = -1061.528918, rho = -0.495266
nSV = 3053, nBSV = 722
Total nSV = 3053
Accuracy = 66.925% (2677/4000) (classification)
Validation Accuracy = 99.6769% (617/619) (classification)
```

```
    /* moment w którym było przeprowadzone skalowanie danych */
```

```
*
optimization finished, #iter = 496
nu = 0.202599
obj = -507.307046, rho = 2.627039
nSV = 630, nBSV = 621
Total nSV = 630
Accuracy = 96.15% (3846/4000) (classification)
Validation Accuracy = 95.7997% (593/619) (classification)
```

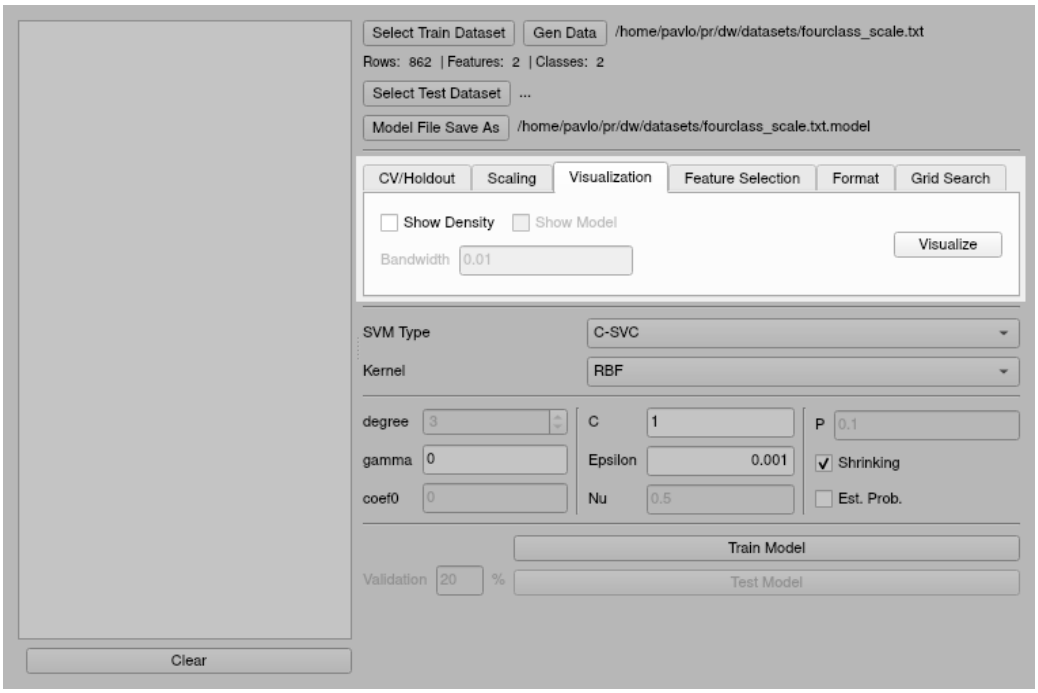
Na tym przykładzie widać, że dokładność modelu wzrosła na 30% przy dziesięciokrotnym zmniejszeniu liczby iteracji algorytmu.

Biblioteka *LIBSVM* zawiera w sobie program *svm-scale* który służy do poprawnego skalowania danych. W ramach projektu aplikacji kod tego programu został przerobiony i zawarty w klasie *svmscale*. Po wybraniu zbioru trenującego kontrolki skalowania danych robią się aktywne. Po skalowaniu aplikacja tworzy nowe pliki z danymi które kończą się na *.scale*. Warto zauważyć że zbiór trenujący i testujący muszą być skalowane razem, a więc przypadek w którym użytkownik wybiera zbiór trenujący, skaluję go, wybiera zbiór testujący i przeprowadza skalowanie ponownie może powodować nieprzewidywalne konsekwencje. Jeśli użytkownik planuje skalować dane i testować swój model to poprawnym podejściem jest wybranie zbiorów danych, a już za tym skalowanie danych.

Również jest dostępne skalowanie klas, które aktywuje się ptaszkiem w polu *Scale y*. Może to być przydatne w regresji, kiedy liczby w kolumnie Y(kolumna klas) nie są dyskretne. Za realizacją skalowania klas stoi wyżej wymieniony program *svm-scale*.

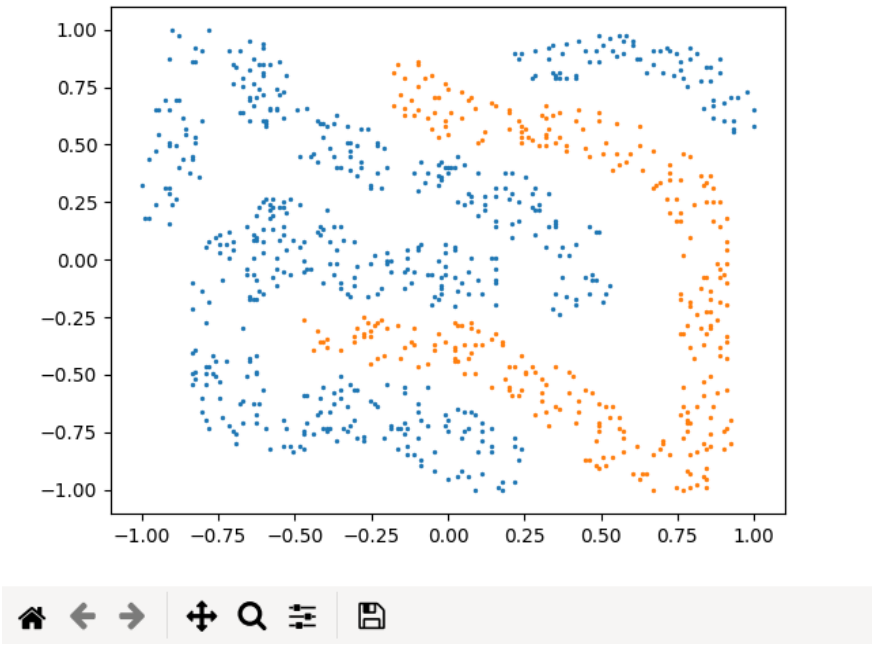
3.3.7 Wizualizacja danych

Karta *Visualization* 3.18 służy do wizualizacji jedno-, dwu- i trzy-wymiarowych danych.



Rysunek 3.17: Karta z kontrolkami dla wizualizacji danych

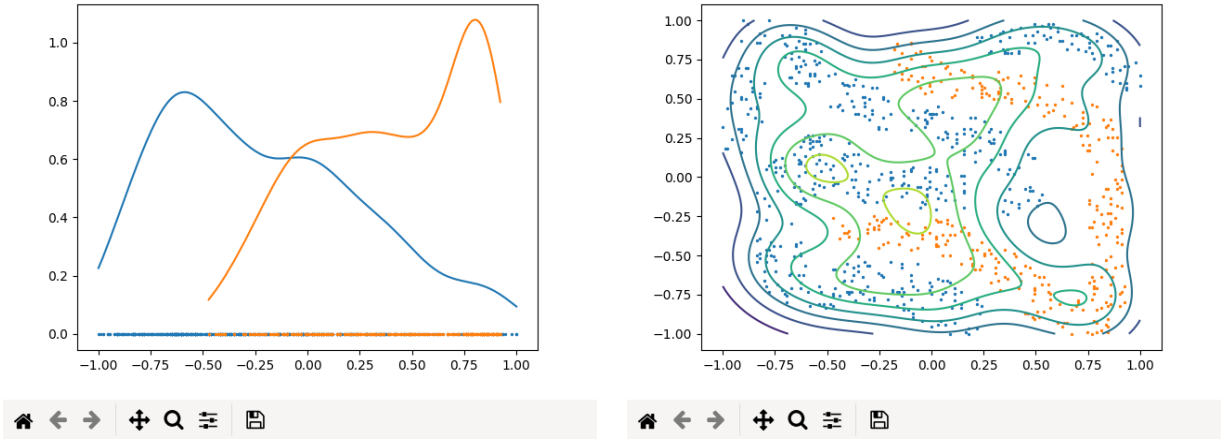
Kontrolki na karcie są dostępne tylko wtedy gdy jest wybrany zbiór trenujący z liczbą cech od 1 do 3.



Rysunek 3.18: Przykład wizualizacji dwuwymiarowych danych

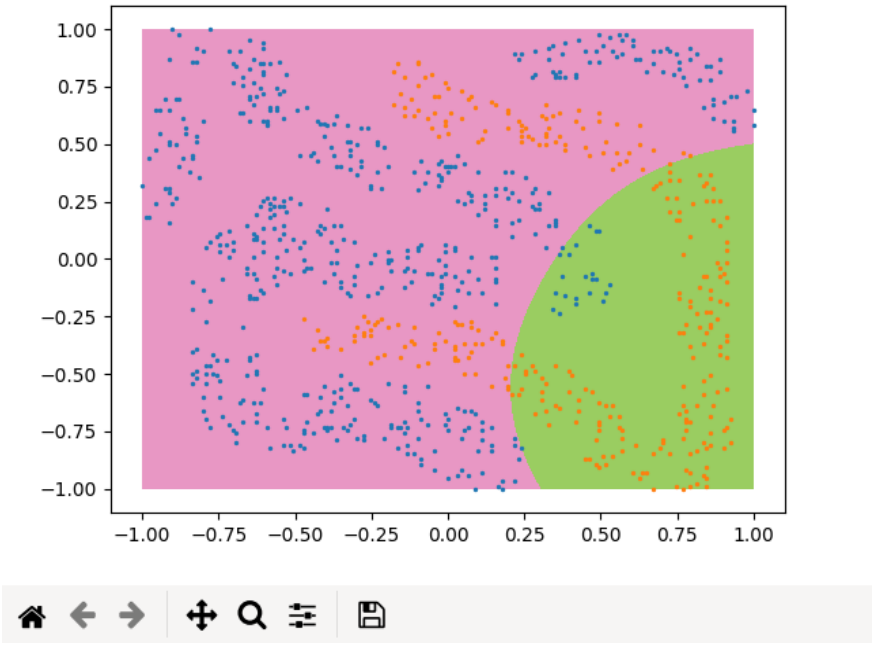
Dla jedno- i dwu-wymiarowych danych jest dostępne rysowanie gęstości danych. Algorytm rysują-

cy dzieli pole wykresu na kwadraty i zlicza liczbę punktów w każdym z nich. Na podstawie tego jest generowany konturowy wykres gęstości. Za liczbę kwadratów odpowiada wartość *Bandwidth*, która może być ustawiana przez użytkownika na interfejsie graficznym. Im mniejszy jest *Bandwidth* tym bardziej gładki wychodzi wykres, ale razem z tym zwiększa się obciążenie algorytmu rysującego.



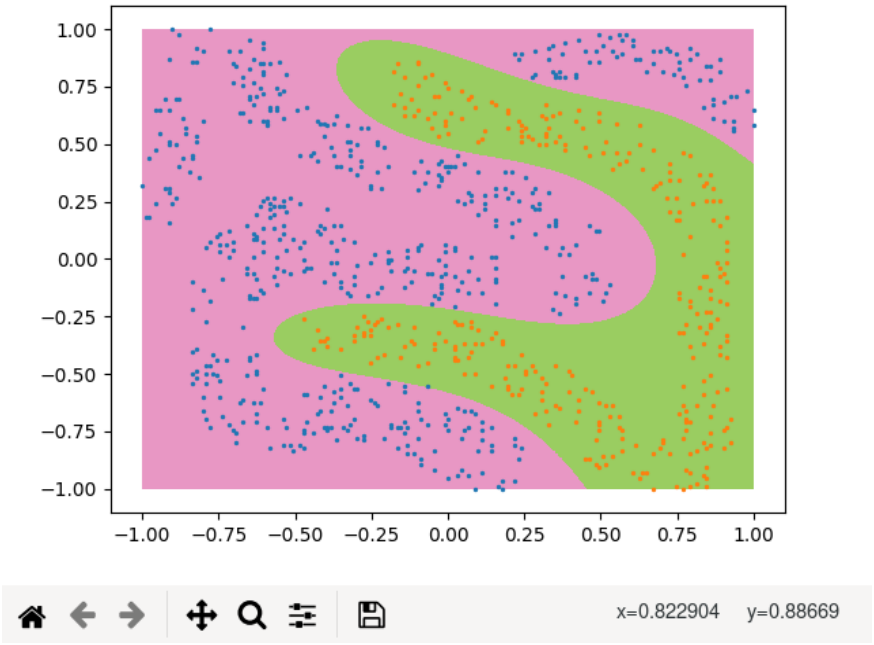
Rysunek 3.19: Przykład rysowania gęstości jedno- i dwu-wymiarowych danych

Dla dwuwymiarowych danych jest dostępne rysowanie modelu.



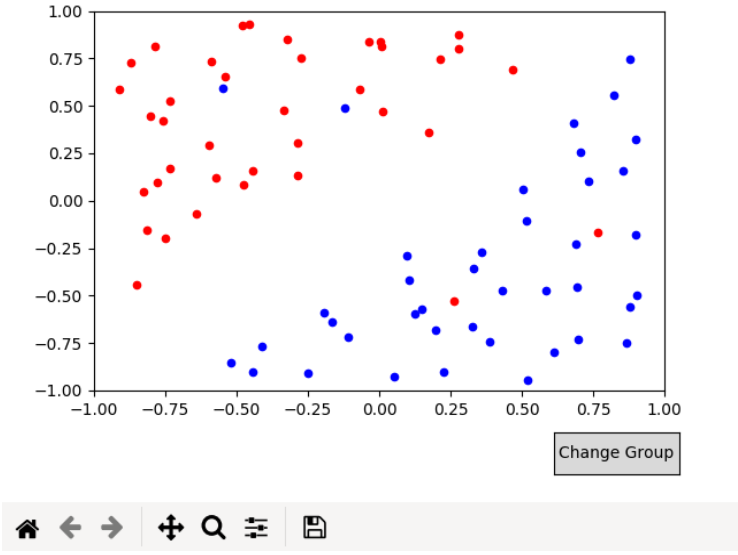
Rysunek 3.20: Przykład rysowania modelu

Dla wyżej narysowanego modelu 3.20 już z wykresu widać że parametry dobrane kiepsko. Dobierając odpowiednie parametry i skalując dane można osiągnąć lepsze wyniki:



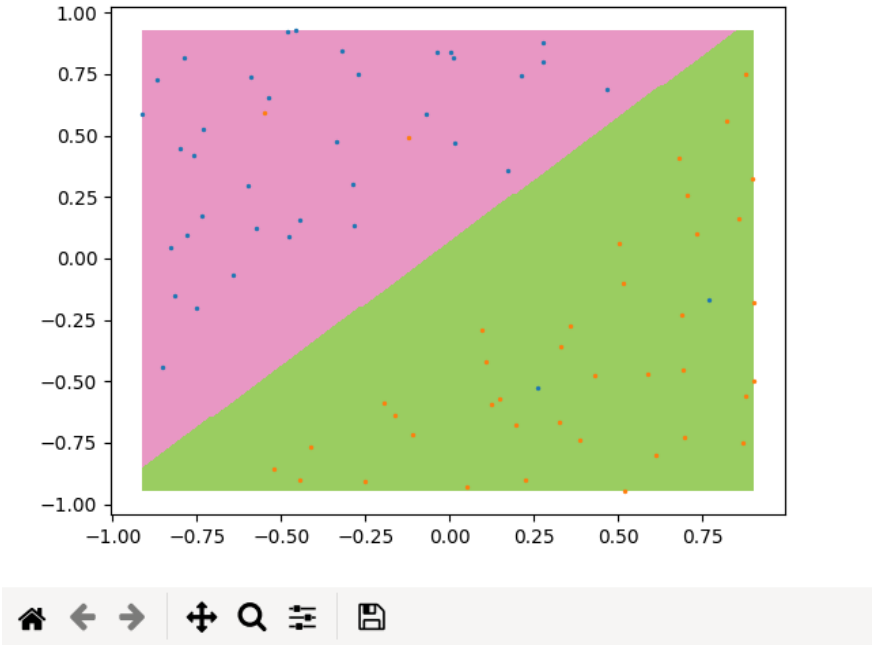
Rysunek 3.21: Przykład rysowania modelu

Przycisk *Gen Points* obok kontrolek do wybierania plików z danymi pozwala samodzielnie generować zbiór danych. Najpierw trzeba będzie wybrać nazwę pliku i katalog w którym należy go umieścić. Za tym pojawi się okno z pustym wykresem, klikając po nim użytkownik będzie generował punkty na tej płaszczyźnie, przycisk *Change Group* pozwala zmienić klasę aktualnie umieszczanych punktów.



Rysunek 3.22: Przykład ręcznego generowania danych

Wygenerowany model z liniową funkcją jądrową:

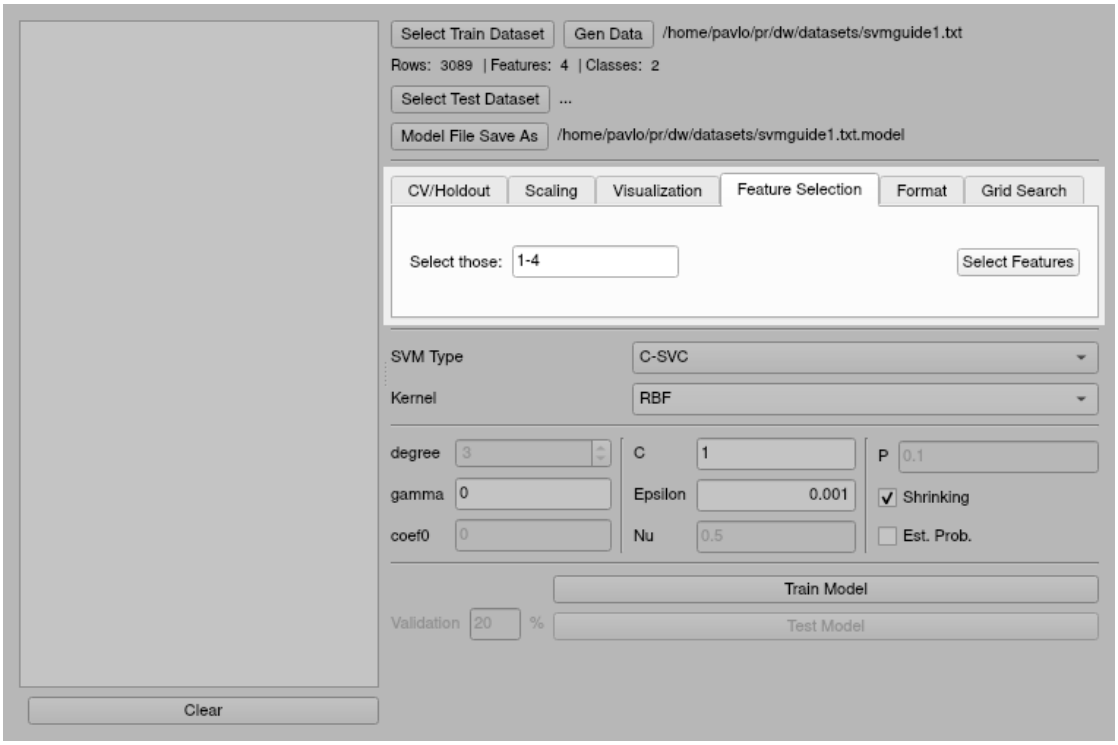


Rysunek 3.23: Model z ręcznie wygenerowanych danych

Dla rysowania wykresów był napisany skrypt w języku Python *plotter.py*. Wykresy gęstości pomaga rysować funkcja *gaussian_kde* z biblioteki *scipy*. Żeby narysować model skrypt generuje siatkę punktów na całym wykresie zapisuję ich do tymczasowego pliku i przekazuje go do programu *svm-predict* z biblioteki *LIBSVM*, który w wyniku swojego działania zwraca swoje zgadnięcia odnośnie podanych punktów, co pozwala kolorować tło wykresu. Skutkiem takiego podejścia jest 'kąciastość' linii rozdzielającej klasy. Wartość *Bandwidth* pozwala kontrolować liczbę punktów w generowanej siatce i robić bardziej gładką linie.

3.3.8 Wybór cech

Czasami jest potrzeba wybrać tylko niektóre cechy z zbioru danych. Dla takich potrzeb w aplikacji istnieje funkcjonalność wyboru cech na karcie *Feature Selection* 3.24.



Rysunek 3.24: Karta z kontrolkami do wyboru cech

Na karcie jest umieszczone pole typu *Text Edit*, w które użytkownik musi wprowadzić które cechy on chce wybrać. Domyślnie w polu jest wpisany przedział cech w wczytanym pliku. Użytkownik może zmienić ten przedział lub wskazać konkretne cechy do wyboru, załóżmy że wczytany zbiór danych ma 30 cech, wtedy:

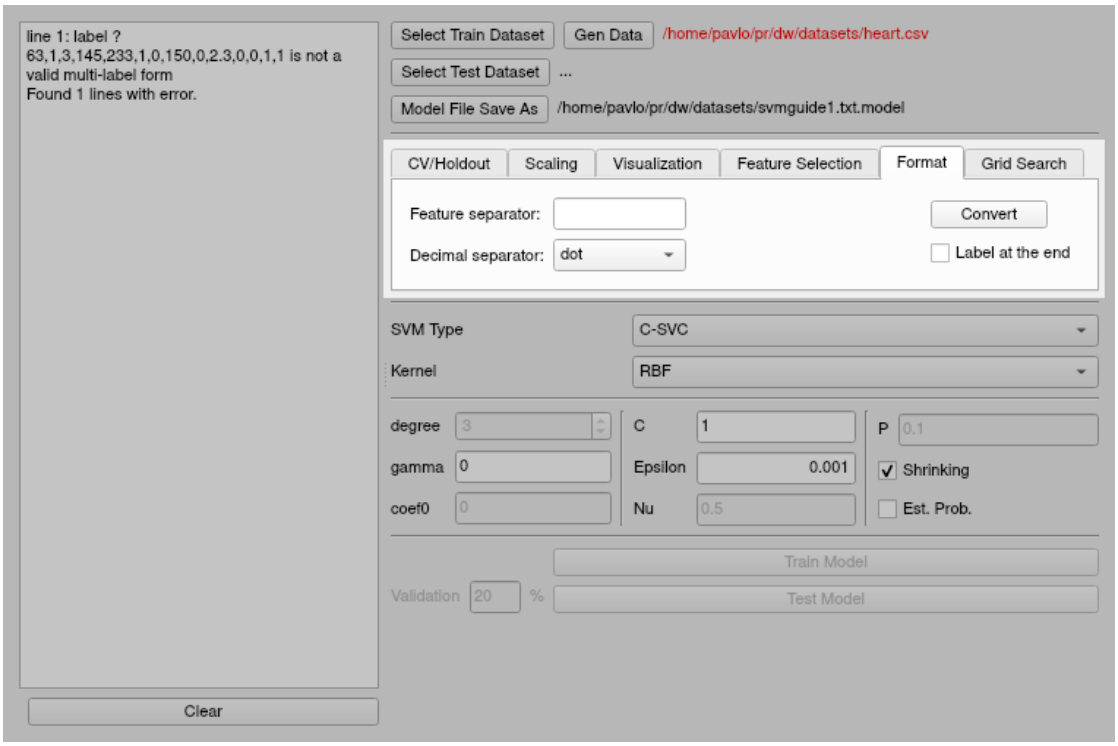
```
1-10                # wybór cech od 1 do 10
1-5,10-25           # wybór cech od 1 do 5 i od 10 do 25
1-5,7,10,15-20,25   # wybór cech od 1 do 5, 7 i 10, od 15 do 20 i 25
```

Przycisk *Select Features* uruchamia skrypt w języku Python *f_select.py* który z wybranych zbiorów danych wybiera cechy i zapisuję ich w formacie *LIBSVM* do plików z nazwą kończąca się na *.fseleced*. Analogicznie do skalowania jeśli użytkownik chce swoje dane testować to przy wyborze cech musi być wybrany odpowiedni plik z danymi testującymi.

3.3.9 Zmiana formatu danych na *LIBSVM*

W Internecie trudno znaleźć dane przygotowane w formacie *LIBSVM*, większość z nich jest w formacie CSV. Natomiast algorytm trenujący biblioteki *LIBSVM* nie przyjmuje danych w innych formatach. Żeby dać swobodę użytkownikom i pozbawić ich od potrzeby pisania własnych programów dla konwersji została dodana prosta funkcjonalność umieszczona na karcie *Format* 3.25 pozwalająca konwertować dane do formatu *LIBSVM*. Użytkownik musi podać separator pomiędzy

cechami i określić czy cecha jest na początku lub końcu rekordu. Niestety nie ma opcji żeby liczba określająca klasę znajdywała się pomiędzy liczbami oznaczające cechy. Również trzeba określić separator dziesiętny liczb rzeczywistych(kropka lub przecinek).



Rysunek 3.25: Karta z kontrolkami do formatowania danych

Po ustawieniu wszystkich parametrów formatowania i kliknięciu przycisku *Convert* aplikacja uruchamia skrypt w języku Python *convert2svm.py* który dzięki podanemu separatorowi analizuje podany plik u układa dane w formacie *LIBSVM*.

Na przykład to są 5 pierwszych linijek ze zbioru danych w formacie CSV:

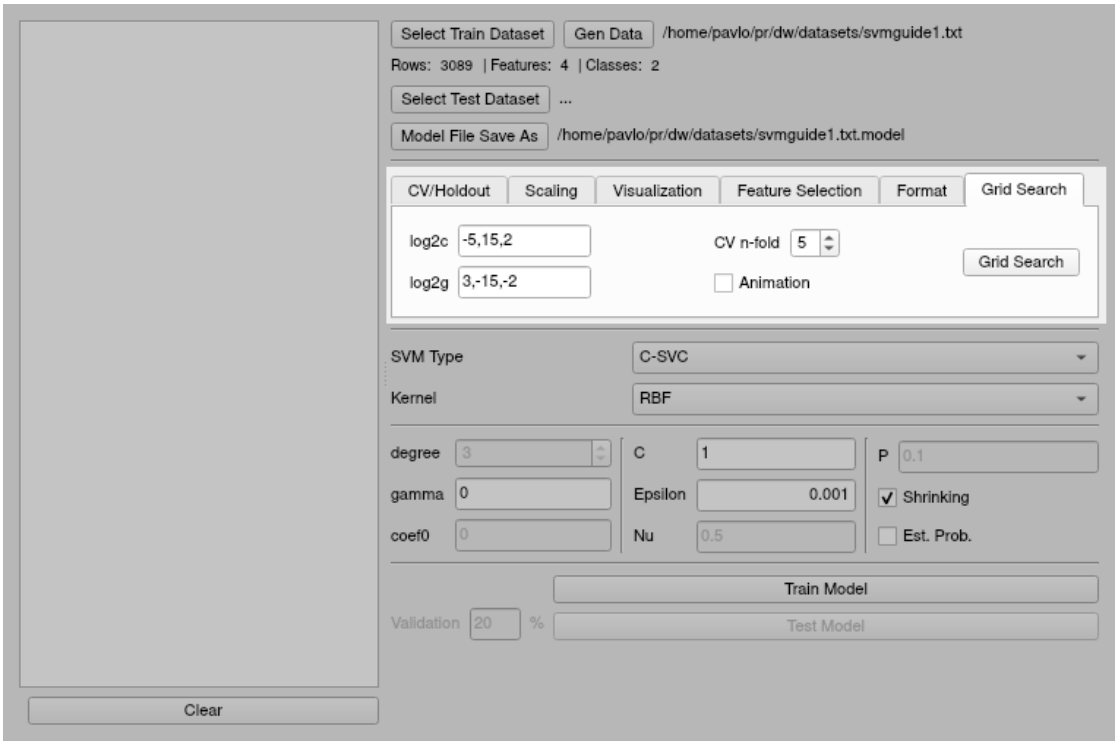
```
63,1,3,145,233,1,0,150,0,2.3,0,0,1,1
37,1,2,130,250,0,1,187,0,3.5,0,0,2,1
41,0,1,130,204,0,0,172,0,1.4,2,0,2,1
56,1,1,120,236,0,1,178,0,0.8,2,0,2,1
57,0,0,120,354,0,1,163,1,0.6,2,0,2,1
```

Te same dane po formatowaniu(ustawiono flagę że liczby określające klasy są na końcach linii):

```
1 1:63 2:1 3:3 4:145 5:233 6:1 7:0 8:150 9:0 10:2.3 11:0 12:0 13:1
1 1:37 2:1 3:2 4:130 5:250 6:0 7:1 8:187 9:0 10:3.5 11:0 12:0 13:2
1 1:41 2:0 3:1 4:130 5:204 6:0 7:0 8:172 9:0 10:1.4 11:2 12:0 13:2
1 1:56 2:1 3:1 4:120 5:236 6:0 7:1 8:178 9:0 10:0.8 11:2 12:0 13:2
1 1:57 2:0 3:0 4:120 5:354 6:0 7:1 8:163 9:1 10:0.6 11:2 12:0 13:2
```

3.3.10 Optymalizacja parametrów

Przeszukiwanie siatką(ang. grid search) jest algorytmem pozwalającym na dobór parametrów modelu. Biblioteka *LIBSVM* dostarcza skrypt w języku Python *grid-search* dla parametrów C i *gamma* który był zintegrowany z aplikacją. Probuje on różne kombinacje (C, γ)(działa w oparciu o funkcje jądrową RBF) i wypisuje tą dla której walidacja krzyżowa dała najlepszy wynik.



Rysunek 3.26: Karta z kontrolkami do przeszukiwania siatką

Na karcie *Grid Search* znajdują się dwa pola tekstowych pozwalających na ustawienie zakresów parametrów i kroków algorytmu. W pole należy wpisać trzy liczby: początek, koniec i krok. Zakres próbowanych parametrów więc będzie wyglądał tak: $2^{begin,...,begin+k*step,...end}$. Przeszukiwanie siatką pomaga wybrać potrzebne parametry nie zgadując i nie wprowadzając ręcznie każdej pary, wadą jednak jest że proces jest dość wolny.

Na karcie *Grid Search* znajduje się kontrolka typu *Check Box* pozwalająca włączyć animacje procesu przeszukiwania jeśli na używanym systemie jest skonfigurowana aplikacja *gnuplot*. Również można wybrać krotność walidacji krzyżowej używanej dla testowania każdej pary parametrów.

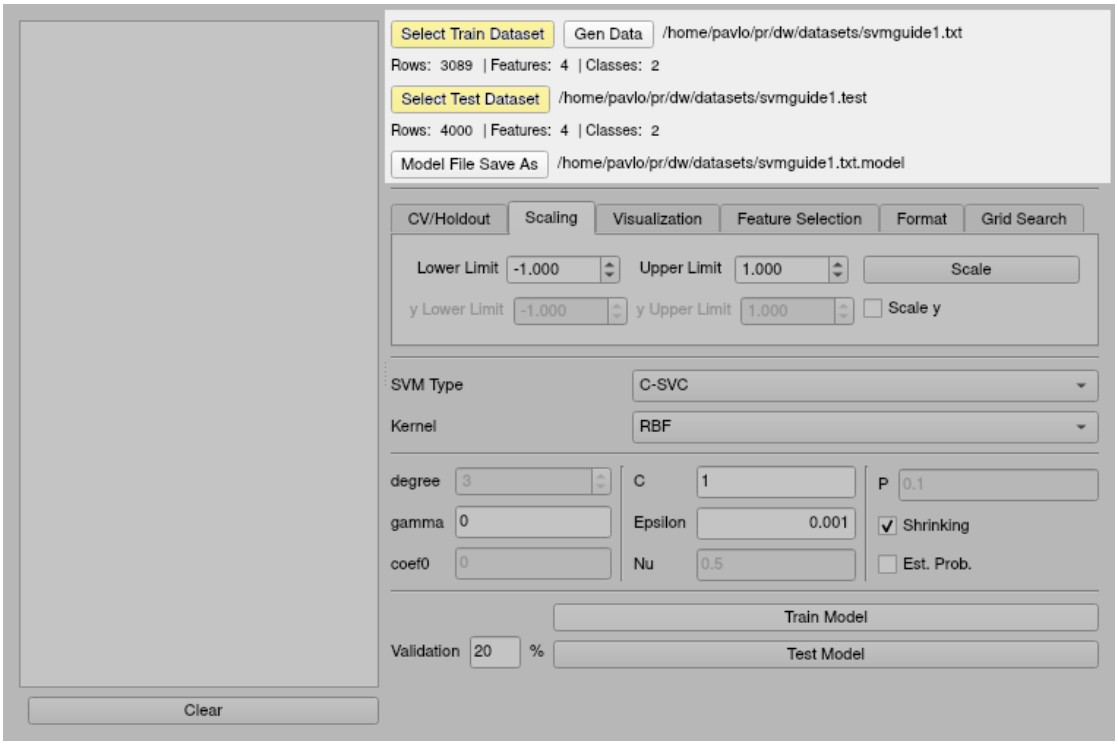
4 Przykłady działania aplikacji

W tym paragrafie będą przedstawione przykłady użycia aplikacji.

4.1 Przykład 1

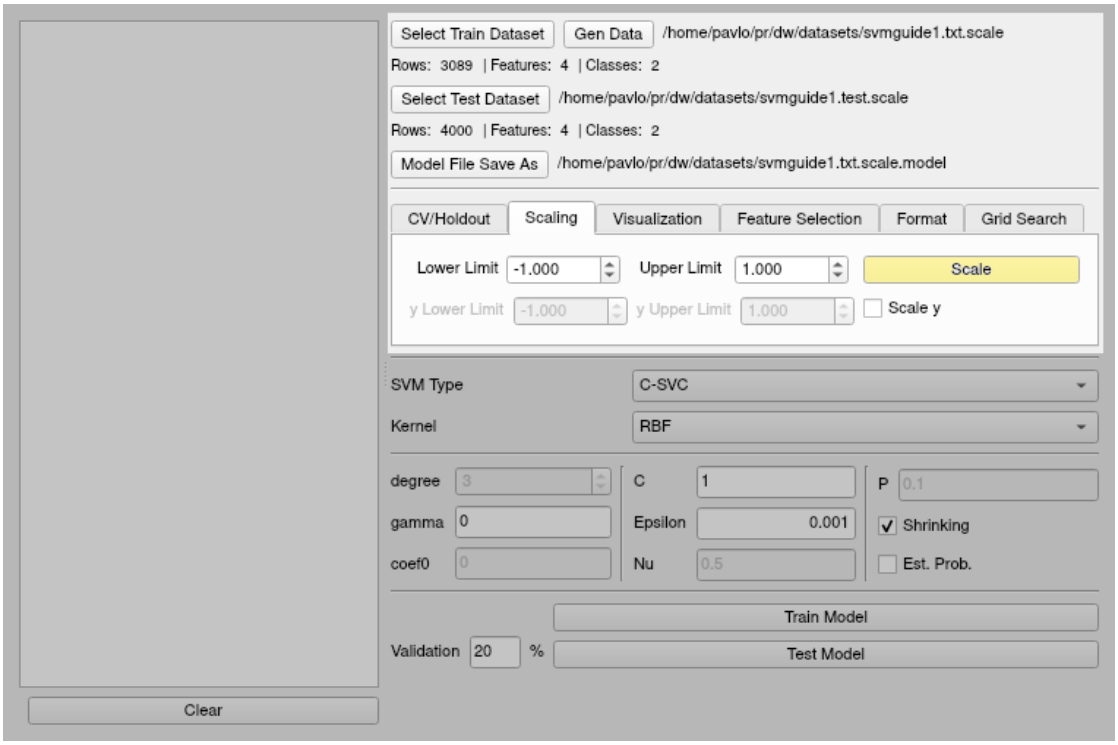
Przykład pokazujący pracę z plikiem z nieprzeskalowanymi danymi w formacie *LIBSVM*

Krok 1: Wczytać plik z danymi



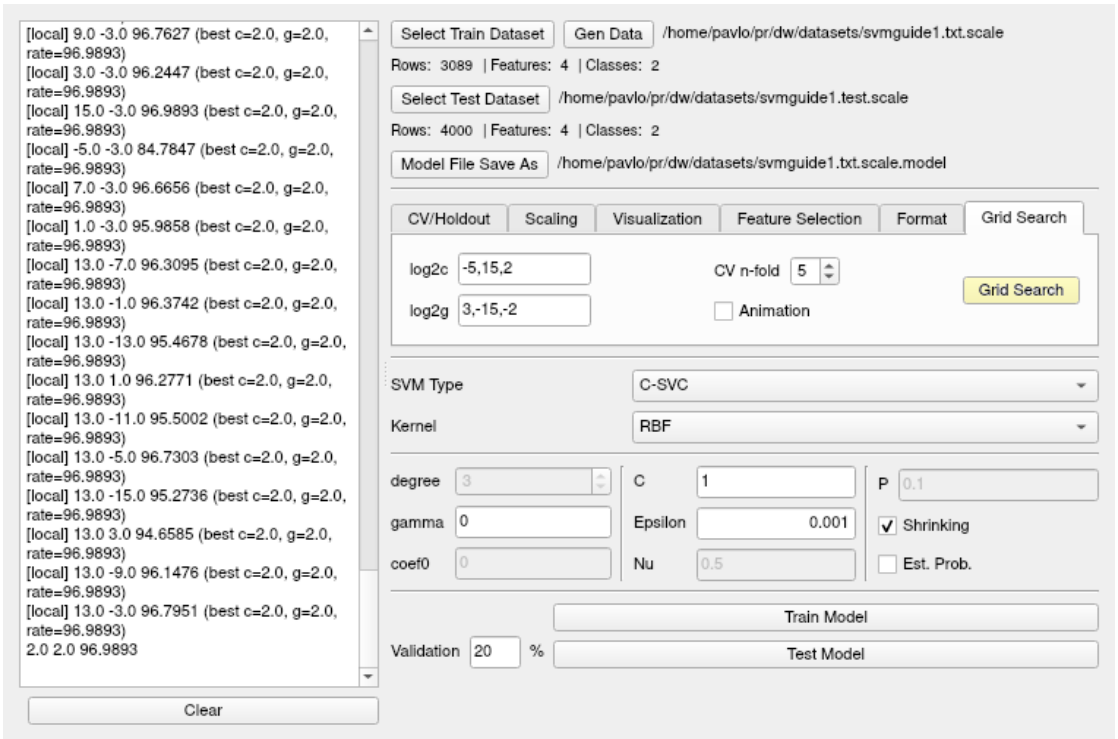
Pojawiły się ścieżki z plikami, a pod nimi informacje o wczytanych zbiorach danych, więc dane wczytane poprawnie.

Krok 2: Wyskalować dane do przedziału $[-1, 1]$



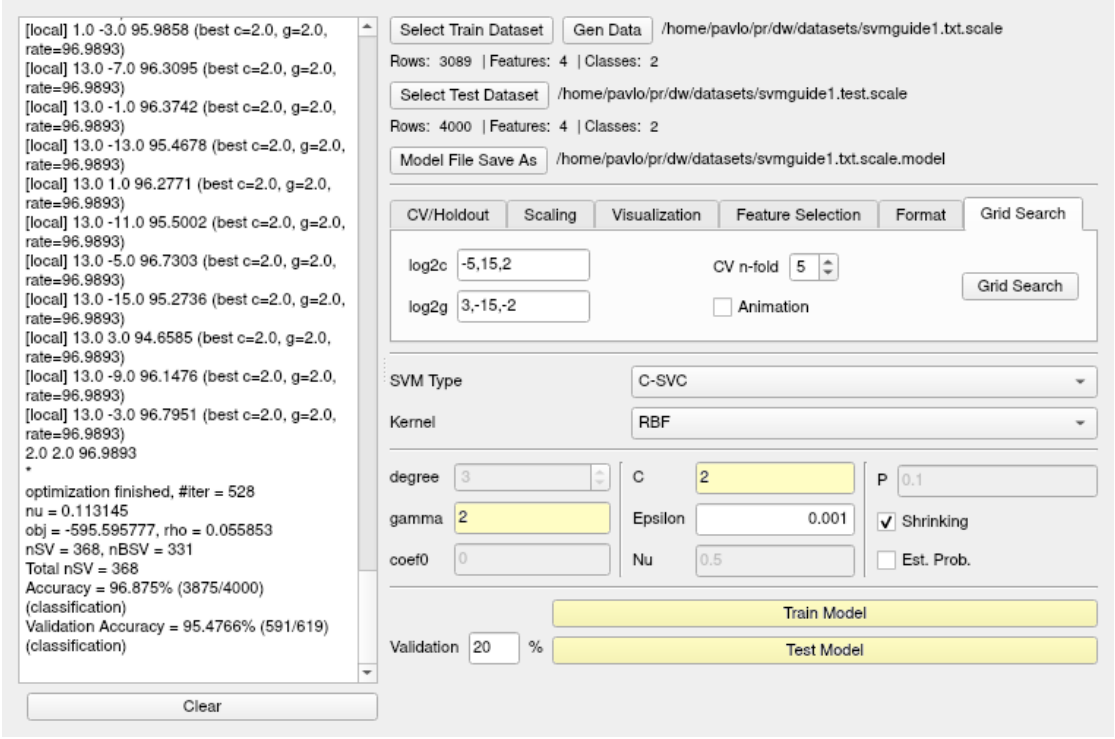
Do nazw plików z danymi dodano rozszerzenie *.scale*, więc dane są wyskalowane.

Krok 3: Przeprowadzić przeszukiwanie siatką, żeby dobrać pasujące parametry.



Algorytm przeszukiwania siatką zwrócił że przy parametrach $C = 2$, $gamma = 2$ precyzja modelu będzie około 96%.

Krok 4: Wytrenować i przetestować model dla parametrów zwróconych algorytmem optymalizującym.

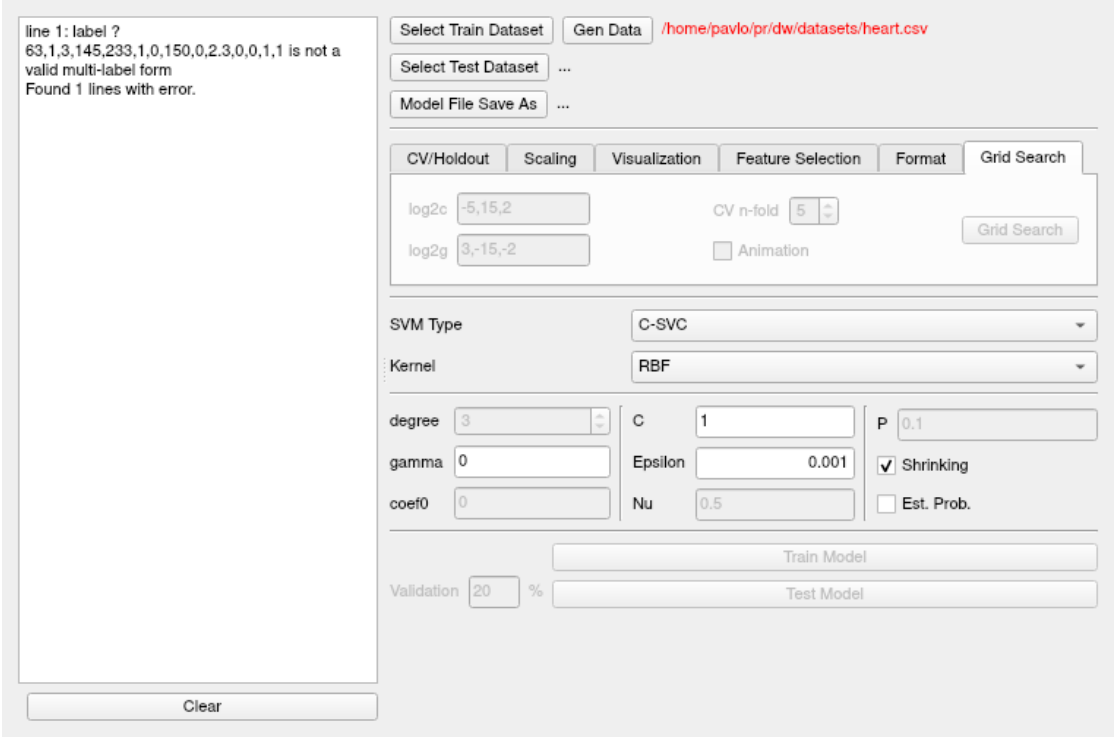


Wytrenowany model znajduje się w ścieżce podanej naprzeciwko przycisku *Model File Save as*.

4.2 Przykład 2

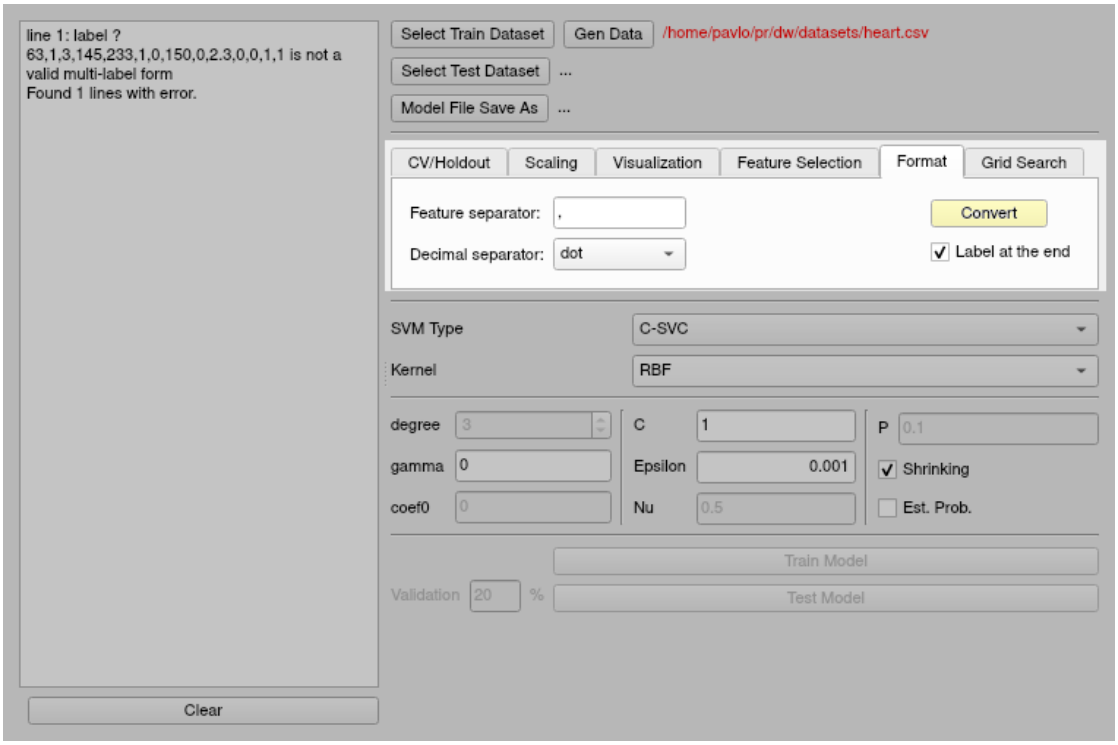
Przykład z danymi w formacie *CSV*.

Krok 1: Wczytać plik z danymi.



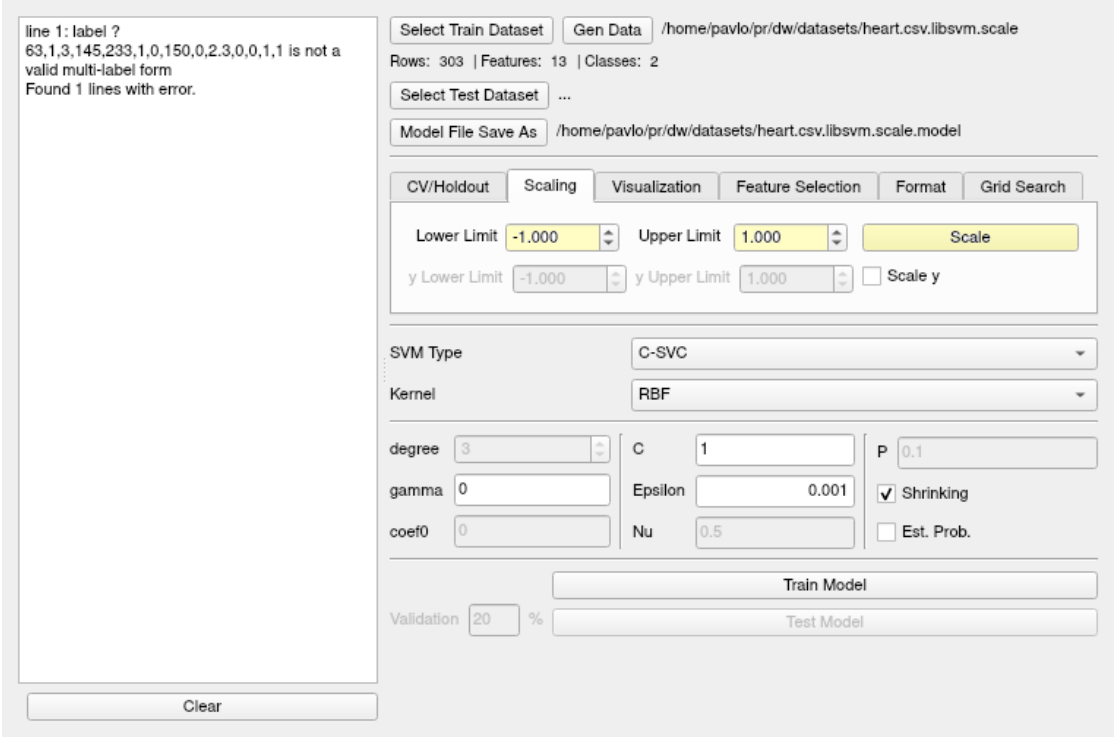
Ścieżka do wczytanego pliku jest czerwonego koloru, co sygnalizuje o błędzie, komunikat w polu tekstowym mówi że wczytane dane nie są w formacie *LIBSVM*.

Krok 2: Skonwertować dane do formatu *LIBSVM*.



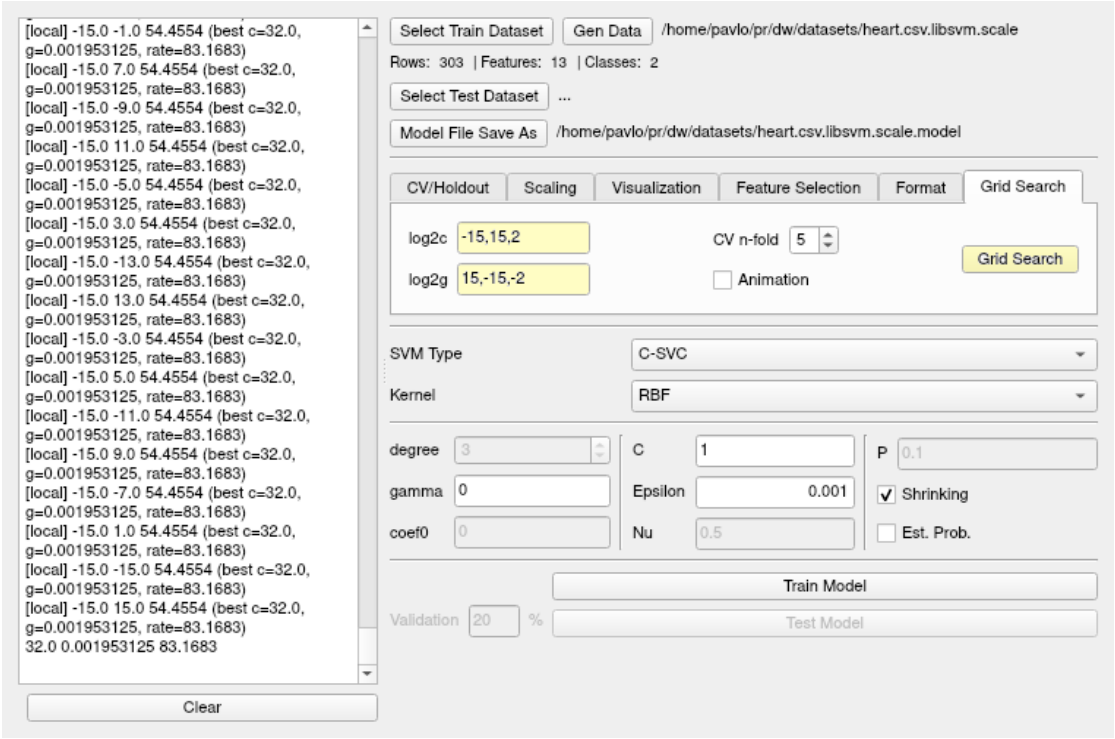
Należy wprowadzić separator danych i zaznaczyć ptaszkiem czy liczba wskazująca klasę jest na początku lub końcu linii. W danym przypadku separatorem jest przecinek, a kolumna klas jest na końcu. Po kliknięciu na przycisk *Convert* program automatycznie wczyta plik z skonwertowanymi danymi i wypisze informację o nich.

Krok 3: Wyskalować dane do przedziału $[-1, 1]$.



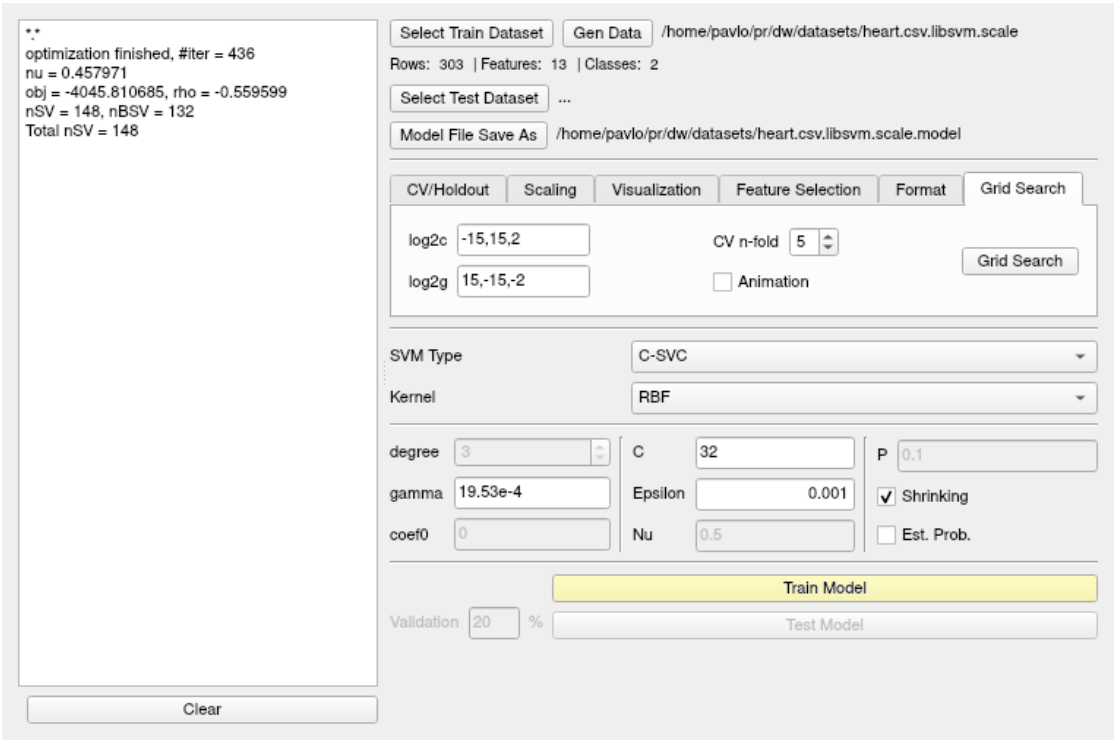
Skoro w danym zbiorze jest tylko 303 rekordy nie będziemy dzielić go na zbiór testujący i trenujący, a użyjemy walidacji krzyżowej. Do nazwy pliku z danymi było dodane *.scale*, więc dane są wyskalowane.

Krok 4: Uruchomić przeszukiwanie siatką, żeby dobrać odpowiednie parametry.



Algorytm zwrócił parametry $C = 32$, $gamma = 0.001953$ na których 5-krotna walidacja krzyżowa pokazuje precyzje modelu w 83%. Mając bardziej szczegółową wiedze o tym zbiorze danych np. co reprezentują cechy można było by spróbować wyrzucić te które nie dotyczą rozwiązywanego zadania i uzyskać większą dokładność modelu.

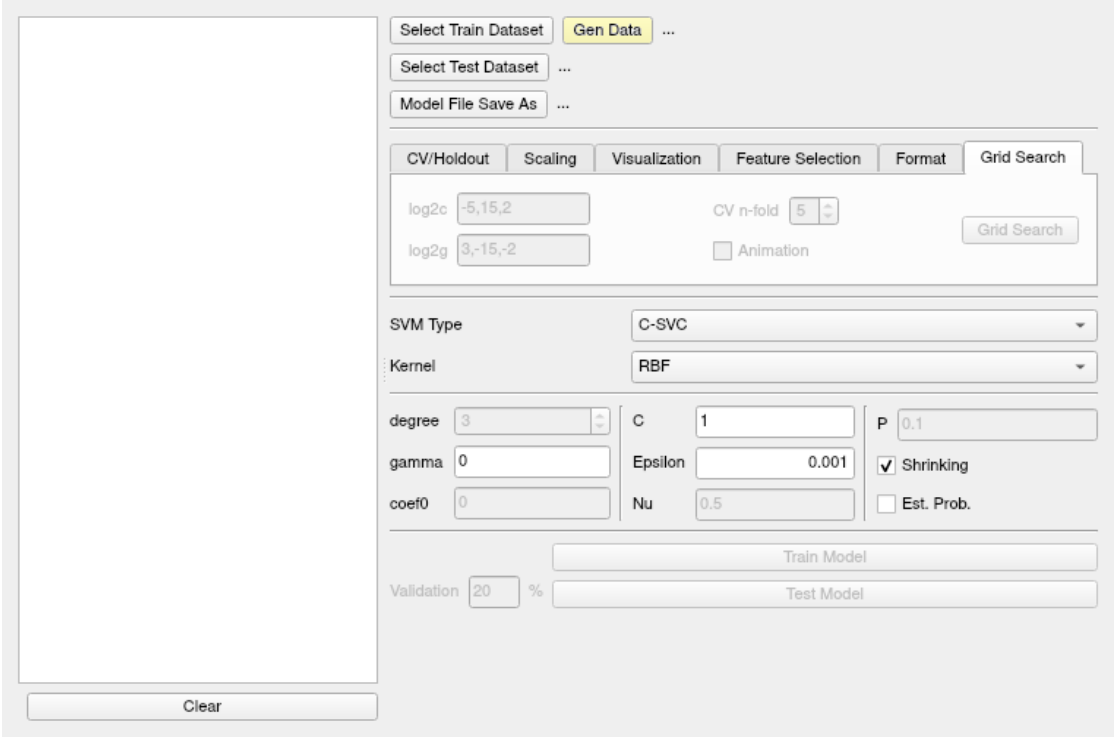
Krok 5: Wygenerować model.



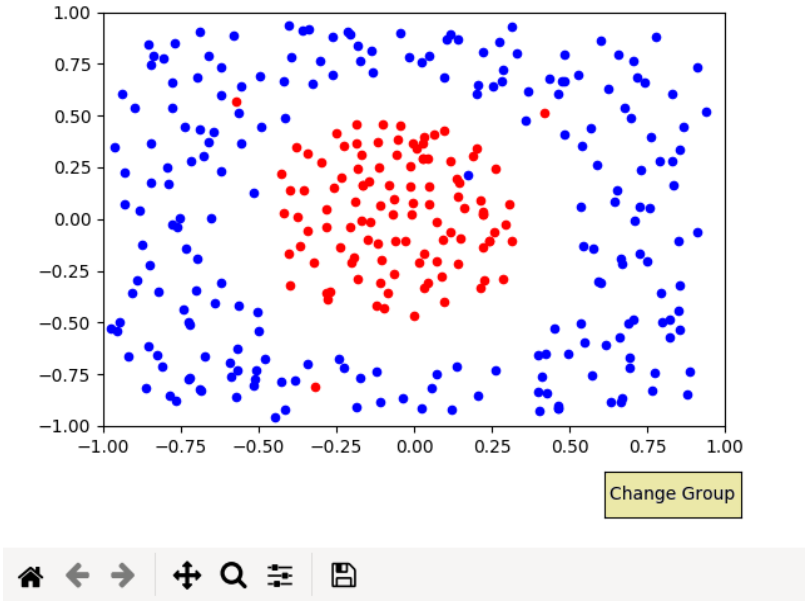
4.3 Przykład 3

Przykład reprezentujący pracę z ręcznie wygenerowanymi danymi.

Krok 1: Wcisnąć przycisk *Gen Data* i wybrać lokalizację i nazwę dla pliku z ręcznie wygenerowanymi danymi.

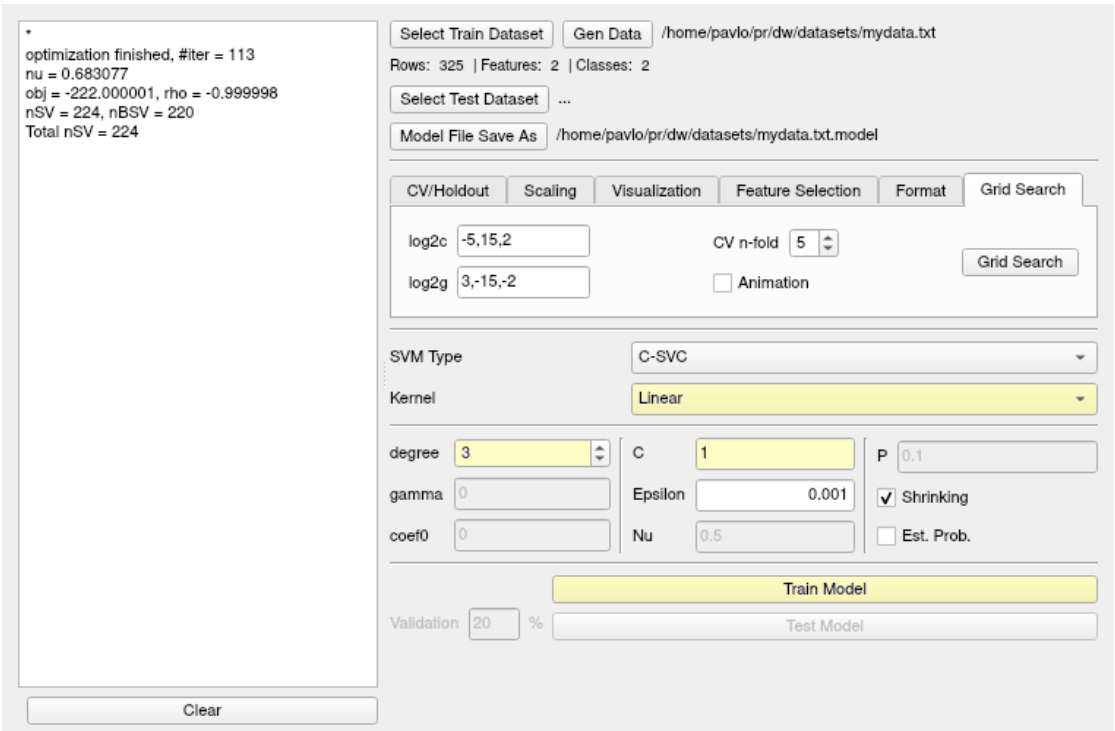


Krok 2: Klikając na płaszczyźnie generować punkty. Dla zmiany klasy należy kliknąć w przycisk *Change Group*.

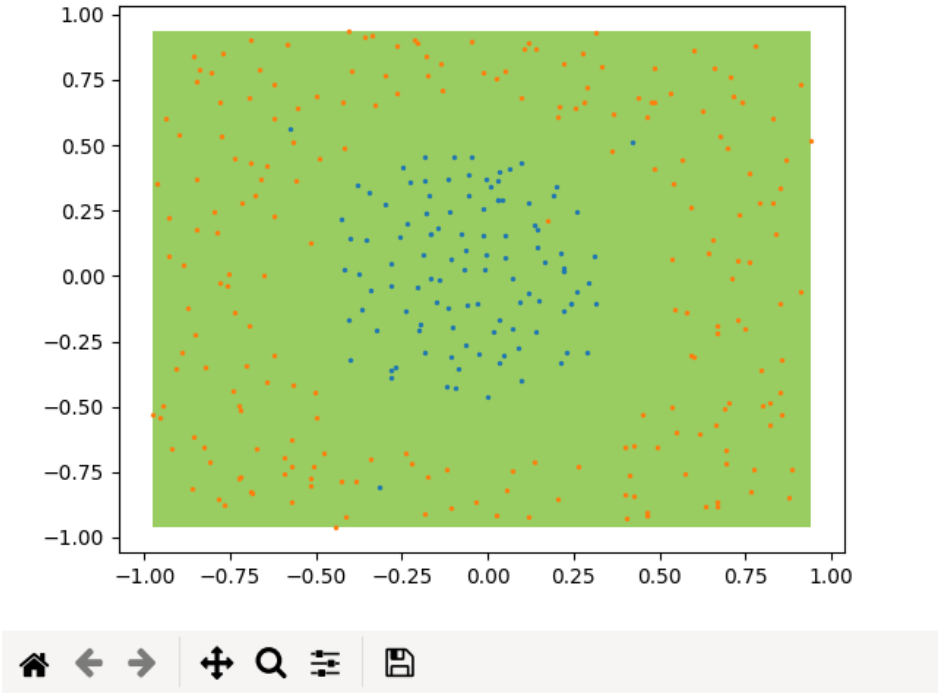


Staram się imitować klasyczny przykład na którym jest pokazywane działanie funkcji jądrowych. W kolejnym kroku nie ma potrzeby skalować dane skoro granice płaszczyzny na której generujemy dane są w przedziałach $x \in [-1, 1]$ i $y \in [-1, 1]$

Krok 3: W celu zadowolenia interesu użyjmy liniowej funkcji jądrowej z domyślnymi parametrami.

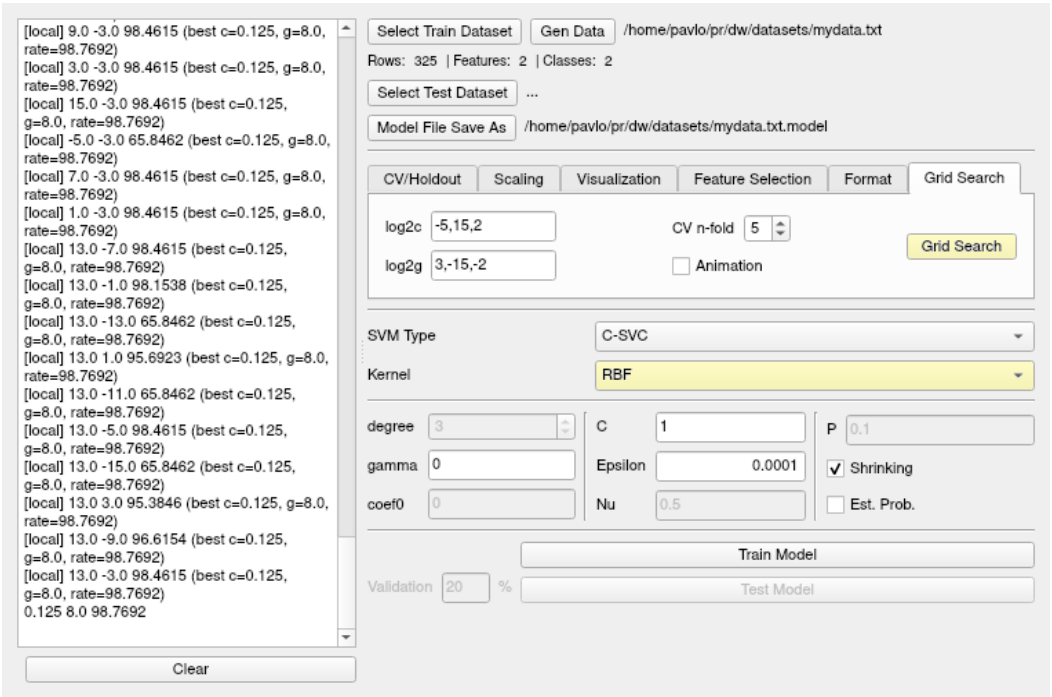


Krok 4: Po wygenerowania modelu można zwizualizować go.



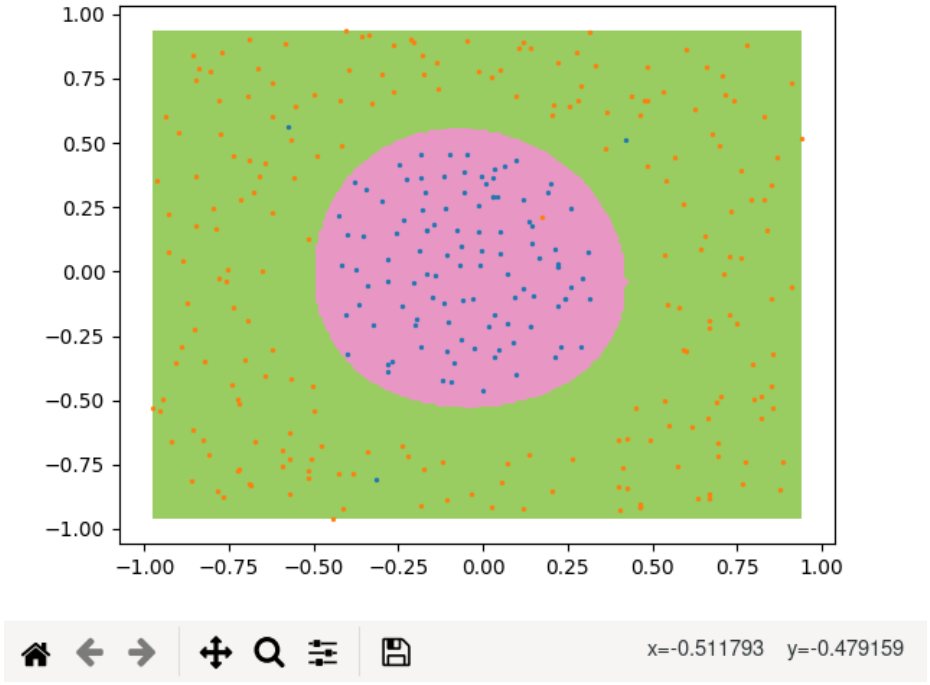
Algorytm *SVM* przydzielił wszystkie punkty do jednej klasy i według 5-krotnej walidacji krzyżowej uzyskał precyzje 65.84%. Jest to, oczywiście, kiepski model.

Krok 5: Wróćmy do funkcji jądrowej RBF i spróbujmy uruchomić przeszukiwanie siatką.



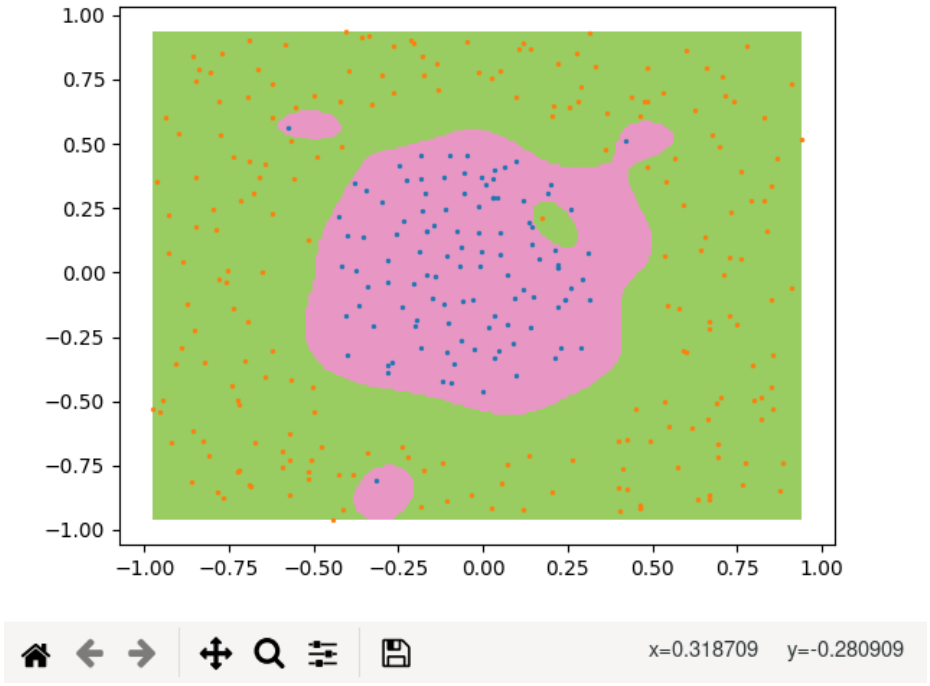
Algorytm proponuje użyć parametrów $C = 0.125$ i $gamma = 8$ dzięki którym potrafimy uzyskać znacznie lepszą precyzje: 98.76%!

Krok 6: Po wygenerowaniu modelu można go zwizualizować.



Z wykresu widać że ten model jest znacznie lepszy. Algorytm poprawnie zidentyfikował szumy(punkty znajdujące się pomiędzy punktami z innej klasy) i nie widać przeuczania.

Krok 7: W celach zadowolenia interesu sprobujmy sztucznie wywołać zjawisko przeuczania, np. podając jako parametry $C = 500$ i $gamma = 20$. Po wygenerowaniu modelu będzie on miał taki wygląd.



W tym przypadku model bierze pod uwagę punkty które były wprowadzone specjalnie w roli szumów, 5-krotna walidacja krzyżowa zwraca gorszy wynik: 95.69%

5 Podsumowanie

W tym paragrafie będą podsumowane dokonane prace i opisany dalszy możliwy rozwój aplikacji.

5.1 Odniesienie do celu pracy

W ramach pracy udało się stworzyć aplikację z interfejsem graficznym, która potrafi trenować i testować modeli, wizualizować dane i wytrenowany model, skalować i przekształcać dane. Program może być przydatny dla osób chcących szybko przetestować kilka modeli z różnymi parametrami. Przez możliwość wizualizacji modelu aplikacja może być szczególnie przydatna w celach dydaktycznych.

5.2 Rozwój aplikacji

Jeśli do aplikacji w przyszłości będzie dodawana nowa funkcjonalność to w każdym przypadku będzie konieczna zmiana schematu interfejsu graficznego. W trakcie pracy dodając nową funkcjonalność tworzyłem nową kartę na której umieszczałem odpowiednie kontrolki, kontynuacja tego może spowodować zaśmiecanie interfejsu przez co użytkownikom będzie trudno znaleźć kontrolki i meni których potrzebują. Obecnie aplikacja używa jednego okna za wyjątkiem okien systemowych dla wyboru plików i okien do wizualizacji danych, rozwiązaniem zaśmiecania interfejsu może być użycie wielu okien. Na przykład osobne okno do wizualizacji danych, okno dla pracy z plikami i danymi, okno do trenowania i testowania modelu itp. Innym podejściem które wydaje się sensownym - praca w jednym oknie, ale zmiana jego treści w zależności od akcji użytkownika. Wadą, oczywiście, będzie to że użytkownik zobaczy dostępne kontrolki dopiero po akcji(np kontrolki do wizualizacji nie są umieszczane na ekranie do wczytania plików z danymi), przez co trudno będzie planować swoją prace.

W przyszłości warto dodać wstępnie obliczone funkcje jądrowe(ang. *Precomputed kernel*). Biblioteka *LIBSVM* ma taką funkcjonalność, natomiast nie została ona zaimplementowana w obecnej wersji aplikacji. Polega ona na tym, że wartości funkcji jądrowej znajdują się w pliku trenującym.

W nauczaniu maszynowym dane często nie są zbalansowane czyli jedna klasa występuje częściej od innej. W przyszłości warto dodać funkcjonalność którą oferuje *LIBSVM*: ustawianie wag dla poszczególnych klas, co pozwala walczyć z niezbalansowanymi danymi.

Podział na pliki trenujące i testujące był zaimplementowany żeby odwzorować działanie biblioteki *LIBSVM*, natomiast w rzeczywistości użytkownik najczęściej ma jeden plik z danymi. Aplikacja może działać z jednym plikiem, używając walidacji krzyżowej lub wyboru danych do testowania z pliku trenującego(ang. *Holdout*), ale wydaje się sensowniejszym zrezygnować z założenia że użytkownik ma dwa pliki(testujący i trenujący) na korzyść założenia że użytkownik po prostu ma jakiś zbiór danych, nie koniecznie w formacie *LIBSVM*.

Istniejący konwerter danych do formatu *LIBSVM* jest bardzo naiwny, i zakłada że dane mają separator, a liczba oznaczająca klasę jest na początku lib końcu linii. W przyszłości warto zbadać najpopularniejsze formaty do zapisu danych i upewnić się że zaimplementowany konwerter sobie z nimi poradzi.

Na obecną chwile aplikacja nie umie pracować z danymi w tekstowej postaci, w przyszłości warto to dodać.

Warto przerobić system wątków i uruchamiania skryptów. Teraz jest tak, że skrypty w języku *Python* uruchamiają się na tym samym wątku na którym jest obliczany interfejs graficzny, czyli w

trakcie działania skryptu interfejs jest zamrażany. W przypadku skryptów które działają szybko(np. sprawdzenie poprawności danych) to nie ma znaczenia, użytkownik po prostu nie zauważy tego, natomiast skrypt przeszukiwania siatką może trwać dość długo(około 30-40 sekund). Tak długa nieaktywność interfejsu może naprowadzić użytkownika na myśl, że program się zawiesił i przestał działać. Rozwiązaniem może być uruchamianie skryptów w osobnych wątkach, a na interfejsie graficznym umieszczenie animacji oznaczającej że program coś liczy i nie trzeba go przerywać.

Można pomyśleć nad dodaniem innych bibliotek w oparciu o które może działać aplikacja, pozwoliło by to użytkownikom porównywanie wydajności różnych implementacji *SVM*.

Program ma dość sporo zależności dla uruchamianych skryptów. Należałoby zaimplementować system sprawdzający czy wszystkie zależności są zainstalowane i czy wszystkie skrypty są w katalogu roboczym.

References

1 Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm` (cited on pages 4, 5).

2 The Qt Company. *Qt library.* URL: `https://www.qt.io/` (cited on page 10).

3 Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. "A practical guide to support vector classification". In: (2003) (cited on page 8).

4 John D. Hunter. *Matplotlib: Python plotting.* URL: `https://matplotlib.org/` (cited on page 10).

5 S Sathiya Keerthi and Chih-Jen Lin. "Asymptotic behaviors of support vector machines with Gaussian kernel". In: *Neural computation* 15.7 (2003) (cited on page 8).

6 Hsuan-Tien Lin and Chih-Jen Lin. "A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods". In: *submitted to Neural Computation* 3 (2003) (cited on page 8).