Linear regression

Piotr Krzywicki

Wprowadzenie

Zadanie klasyfikacji czy regresji polegają na dopasowaniu do otykietowanego zbioru danych X, y funkcji, dla której z dostatecznie dużą dokładnością zachodzi $f_\theta(x) \approx y$ dla losowych x, y z tego zbioru.

Wprowadzenie

Żeby tego dokonać, uznajemy, że naszą funkcję f_θ daje się sparametryzować wagami θ.

Wprowadzenie

Przykładowy schemat wygląda następująco:

0. Do problemu dobierz architekturę modelu, parametryzowaną wagami θ, funkcję błędu $\mathcal{L}(x, y, \theta)$, której niska oczekiwana wartość dla danych wag gwarantuje dobrą jakość modelu
1. Niech X to dane wejściowe, y to etykiety
2. Zainicjalizuj parametry modelu θ
3. Powtarzaj dopóki $\mathcal{L}(\theta)$ nie będzie wystarczająco niska
    - Minimalizuj $\mathcal{L}(\theta)$ zmieniając wagi θ
        - zazwyczaj powyższa minimalizacja oparta jest na gradiencie
    - Innymi słowy zazwyczaj $\theta = \theta - \alpha * \nabla_\theta \mathcal{L}(\theta)$

Wprowadzenie

Algorytm regresji liniowej zakłada:

$$f_\theta(x) = \theta^T x$$
$$\mathcal{L}(x, y, \theta) = (f_\theta(x) - y)^2$$

Algorytm regresji logistcznej zakłada:

$$f_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$
$$\mathcal{L}(x, y, \theta) = H(f_\theta(x), y)$$

Algorytm regresji logistcznej z wieloma klasami (softmax regression/multinomal regression) zakłada:

$$f_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$$
$$\mathcal{L}(x, y, \theta) = H(f_\theta(x), y)$$

Wprowadzenie

gdzie:

$$H(p,q) = -\sum_{i=1}^{n} p(x_i) \log q(x_i) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) - \sum_{i=1}^{n} p(x_i) \log \frac{q(x_i)}{p(x_i)}$$

Wprowadzenie

Gradienty funkcji błędu w scenariuszu mamy już policzone, można się nimi nie przejmować. W dalszej części kursu i tak zajmie się nimi framework pytorch.

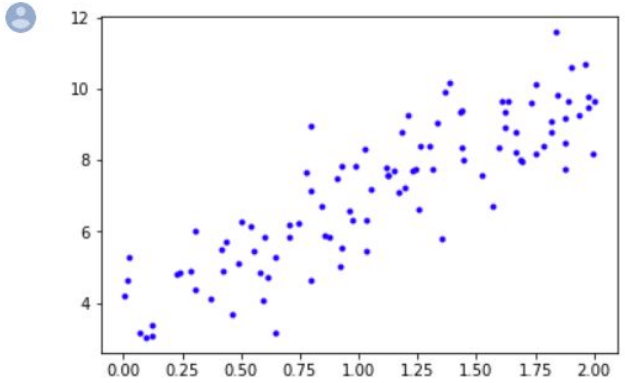
Regresja liniowa

Rozpoczynamy od preparowanego zbioru danych, który da się przybliżyć dobrze funkcją liniową.

```
[ ]  X = 2 * np.random.rand(100,1)
     y = 4 + 3*X + np.random.randn(100,1)
```

Zwizualizujmy nasz zbiór danych:

```
     plt.plot(X, y, "b.")
     plt.show()
     y = y[:, 0]
```



Regresja liniowa

Zaimplementujmy wyznaczania współczynników regresji liniowej za pomocą algorytmu spadku gradientu, minimazując błąd średniokwadratowy pomiędzy przewidywaniami naszego modelu, a wartościami prawdziwymi.

```python
def rmse(X, y, theta):
    return np.sqrt(np.mean((X @ theta - y) ** 2))

def linear_regression_gd(X, y, learning_rate=0.01, num_iterations=1000, logg
    """
    Performs gradient descent based linear regression on the input data.

    Parameters:
    X (ndarray): A NumPy array of shape (n_samples, n_features) representing
    y (ndarray): A NumPy array of shape (n_samples,) representing the target
    learning_rate (float): The learning rate for gradient descent. Default i
    num_iterations (int): The number of iterations for gradient descent. Def

    Returns:
    theta (ndarray): A NumPy array of shape (n_features+1,) representing the
    losses (list): values of loss function being optimized after each optimi
    """
```

Regresja liniowa

```python
# Add bias term to X
X = np.insert(X, 0, 1, axis=1)

# Initialize parameters to zeros
theta = np.zeros(X.shape[1])
losses = []

def prediction(X, theta):
  # Compute predictions
  y_pred = ...
  """
  TODO:
  your code goes here
  """
  return y_pred
```

```python
def gradient(X, theta):
  y_pred = prediction(X, theta)
  # Compute errors
  errors = y_pred - y
  # Compute gradients
  gradients = X.T @ errors / len(X)
  return gradients

# Perform gradient descent
for i in range(num_iterations):
    # Update parameters
    theta =
    """
    TODO:
    your code goes here
    """

    cost = rmse(X, y, theta)
    losses.append(cost)

    if i % logging_period == 0:
      print(f"RMSE after iteration {i}: {cost}")

return theta, losses
```

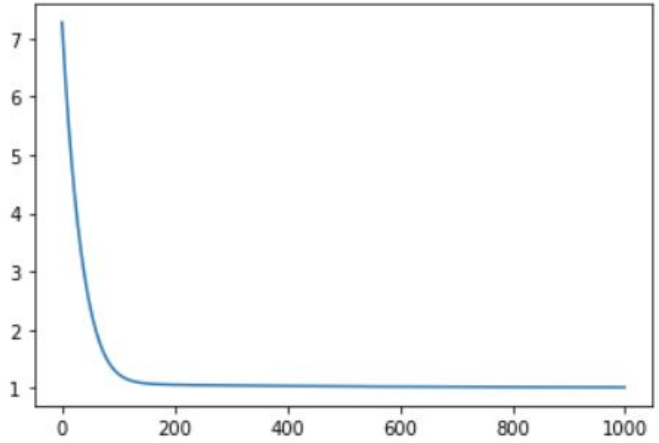
Regresja liniowa

Uruchommy nasz algorytm spadku gradientowego rozwiązujący zadanie regresji liniowej:

```
theta, losses = linear_regression_gd(X, y,
"""
TODO:
your code goes here
"""
)
plt.plot(losses)
plt.show()
```

Regresja liniowa

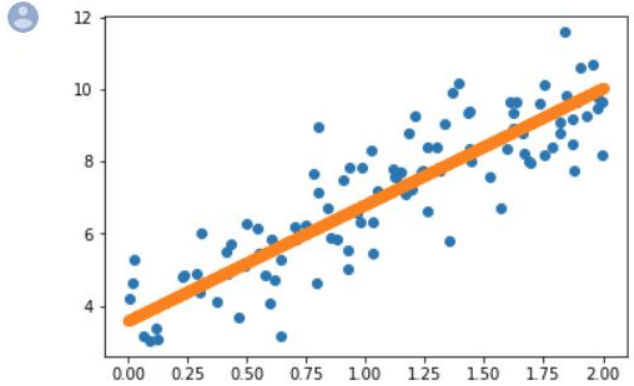
```
RMSE after iteration 0: 7.271707675111863
RMSE after iteration 100: 1.2429966410558813
RMSE after iteration 200: 1.055940964783732
RMSE after iteration 300: 1.0434430783712554
RMSE after iteration 400: 1.0349652783447179
RMSE after iteration 500: 1.0284762432539543
RMSE after iteration 600: 1.023510982732734
RMSE after iteration 700: 1.01971651923992
RMSE after iteration 800: 1.0168196729980163
RMSE after iteration 900: 1.0146098051666386
```



Regresja liniowa

Zwizualizujmy znalezioną prostą:

```
xs = np.linspace(0, 2, 1000)
plt.scatter(X[:, 0], y)
plt.scatter(xs,
"""
TODO:
your code goes here
"""
)
plt.show()
```



Regresja liniowa/wielomianowa – feature engineering

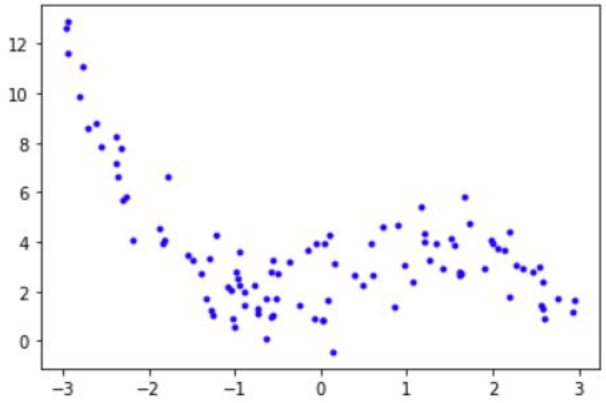
Dzięki technice inżynierii cech (feature engineering), polegającej na przekształceniu danych wejściowych w sposób dobrze dobrany do konkretnego problemu regresji, jesteśmy w stanie z dodatkową wiedzą ekspercką rozwiązać problemy regresji nieliniowej.

Zobaczmy to na przykładzie syntetycznego zbioru danych, który dobrze daje się przybliżyć wielomianem 3-go stopnia.

Regresja liniowa/wielomianowa – feature engineering

Na początku, wygenerujmy i zwizualizujmy nasz zbiór danych:

```
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X**2 - X**3/3 + X + 2 + np.random.randn(m, 1)

plt.plot(X,y, "b.")
plt.show()

y = y[:, 0]
```



Regresja liniowa/wielomianowa – feature engineering

Teraz, przekształćmy przestrzeń cech naszego problemu, żeby znalezienie dobrego modelu liniowego w tej przekształconej przestrzeni było już prostym zadaniem:

```
[ ]  X = np.concatenate([
     """
     TODO:
     your code goes here
     """
     ], axis=1)
```

Regresja liniowa/wielomianowa – feature engineering

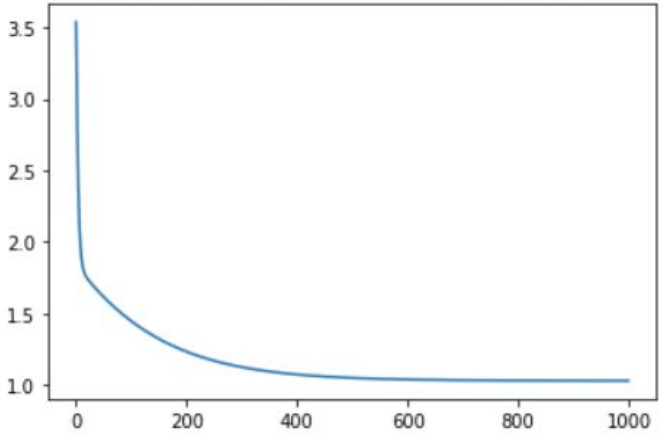
Zapuśćmy dokładnie ten sam co wcześniej algorytm rozwiązujący problem regresji liniowej, ale na przekształconej przez nas przestrzeni cech:

+ Code + Text

```
[ ] theta, losses = linear_regression_gd(X, y,
    """
    TODO:
    your code goes here
    """
    )
    plt.plot(losses)
    plt.show()
```

Regresja liniowa/wielomianowa – feature engineering

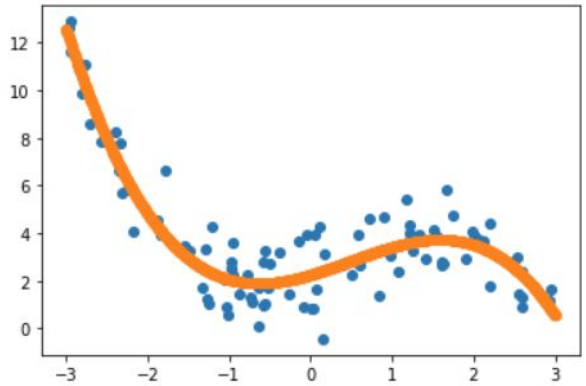
```
RMSE after iteration 0: 3.5381759740107133
RMSE after iteration 100: 1.444591289290609
RMSE after iteration 200: 1.2306193182709848
RMSE after iteration 300: 1.1226787737634336
RMSE after iteration 400: 1.07101825432535
RMSE after iteration 500: 1.0470468846612841
RMSE after iteration 600: 1.0360915275188092
RMSE after iteration 700: 1.0311168465641927
RMSE after iteration 800: 1.0288629716286257
RMSE after iteration 900: 1.0278422551832058
```



Regresja liniowa/wielomianowa – feature engineering

Zwizualizujmy dopasowanie naszego modelu:

```
[ ]  xs = np.linspace(-3, 3, 1000)
     plt.scatter(X[:, 0], y)
     plt.scatter(xs,
     """
     TODO:
     your code goes here
     """
     )
     plt.show()
```



Regresja liniowa – realny przykład

Spróbujmy zastosować nasz algorytm do rozwiązania rzeczywistego problemu. Spróbujemy na podstawie danych dotyczących zamówienia w danej restauracji za pomocą modelu liniowego przewidzieć napiwek.

```
tips_df = pd.read_csv('https://raw.githubusercontent.com/marcin119a/PODSTAWY
tips_df.head()
```

	index_of_row	total_bill	tip	sex	smoker	day	time	size
0	0	16.99	1.01	Female	No	Sun	Dinner	2
1	1	10.34	1.66	Male	No	Sun	Dinner	3
2	2	21.01	3.50	Male	No	Sun	Dinner	3
3	3	23.68	3.31	Male	No	Sun	Dinner	2
4	4	24.59	3.61	Female	No	Sun	Dinner	4

Regresja liniowa – realny przykład

Obróbmy najpierw nasz zbiór danych:

- przeróbmy zmienne kategoryczne na zmienne numeryczne za pomocą techniki one-hot encoding
- znormalizujmy nasz zbiór danych, żeby każda kolumna miała średnią wartość 0 i odchylenie standardowe 1
- Podzielmy nasz zbiór danych na:
    - zbiór treningowy: na którym będziemy uczyć nasz model
    - zbiór testowy: na którym będziemy sprawdzać jakość naszego modelu

Regresja liniowa – realny przykład

```
# Create dummy variables for the 'day' and 'time' columns, as they are categ
sex_dummies = pd.get_dummies(tips_df['sex'], prefix='sex')
smoker_dummies = pd.get_dummies(tips_df['smoker'], prefix='smoker')
day_dummies = pd.get_dummies(tips_df['day'], prefix='day')
time_dummies = pd.get_dummies(tips_df['time'], prefix='time')
tips_df = pd.concat([tips_df, day_dummies, time_dummies, sex_dummies, smoker
tips_df.drop(['day', 'time', 'sex', 'smoker', 'index_of_row', 'sex_Female',

# Split the dataset into training and testing sets
X = tips_df.drop('tip', axis=1)
y = tips_df['tip']
```

Regresja liniowa – realny przykład

```
# Split the data into training and testing sets
train_size = int(0.8 * len(X))
X_train, y_train = ...
X_test, y_test = ...
"""
^TODO:
your code goes there
"""
```

Regresja liniowa – realny przykład

```
# Scale the features manually
X_train = ...
X_test = ...
"""
^TODO:
your code goes there
"""

# Convert it to numpy
X_test = X_test.to_numpy()
X_train = X_train.to_numpy()
y_test = y_test.to_numpy()
y_train = y_train.to_numpy()
```

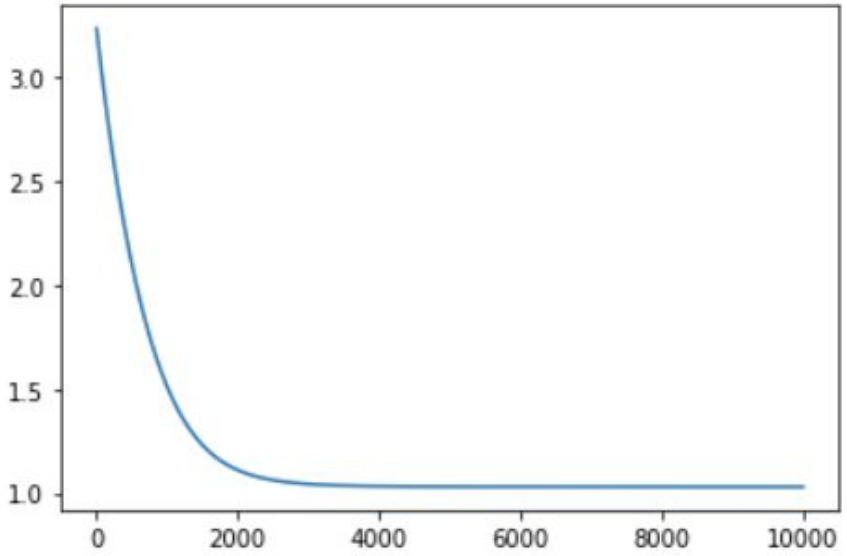
Regresja liniowa – realny przykład

Zapuśćmy algorytm rozwiązywania regresji liniowej na problemie przewidywania napiwków:

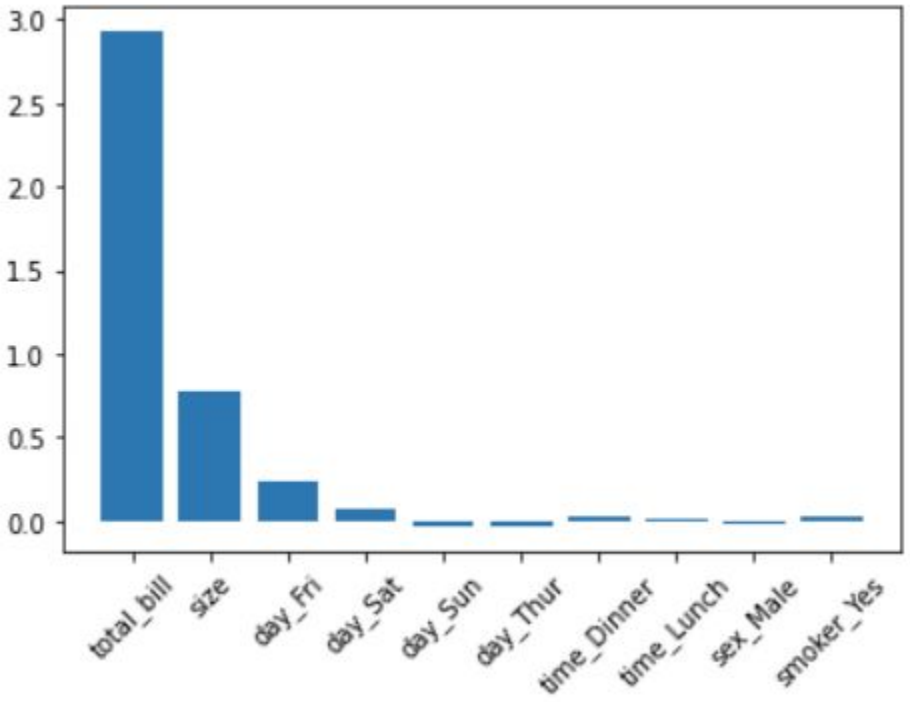
```
theta, losses = linear_regression_gd(X_train, y_train
"""
TODO:
your code goes here
"""
)
plt.plot(losses)
plt.show()
```

Regresja liniowa – realny przykład

```
RMSE after iteration 9500: 1.0289273476354668
RMSE after iteration 9600: 1.0289250627262045
RMSE after iteration 9700: 1.028922924851874
RMSE after iteration 9800: 1.028920923924141
RMSE after iteration 9900: 1.02891905056545
```



Regresja liniowa – realny przykład



Regresja liniowa – realny przykład

Możemy łatwo zinterpretować współczynniki naszej regresji liniowej jako wpływ każdej zmiennej na wynik – napiwek.

Możemy zobaczyć, że:

- Największy wpływ na napiwek ma kwota zamówienia
- Potem ile osób było przy stole
- Czy był to piątek
- Reszta atrybutów ma znikomy efekt

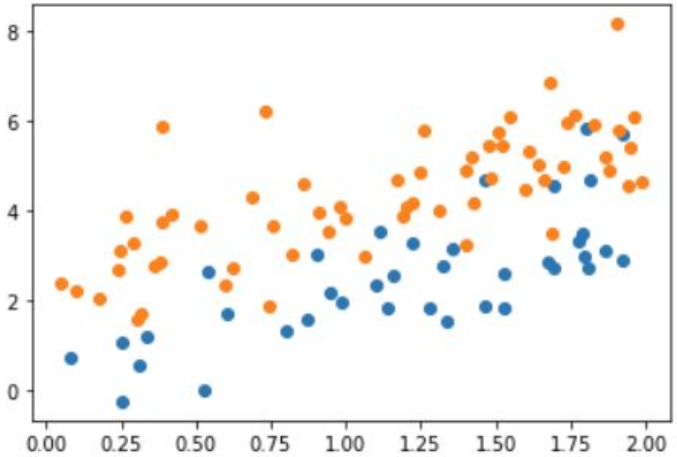
Co zgadza się z naszą intuicją

Regresja logistyczna

W podobny sposób co poprzednio, normalizując wyjścia z modelu liniowego za pomocą funkcji sigmoid $\frac{1}{1+e^{-x}}$ jesteśmy w stanie rozwiązywać problem klasyfikacji binarnej z użyciem modelu liniowego, aka. regresji logistycznej.

Regresja logistyczna

```
y = np.random.randint(0, 2, size=100)
xs = 2 * np.random.rand(100,1)
X = np.concatenate([xs, 1 + 2 * xs + np.random.randn(100,1) + (y.reshape(100
plt.scatter(X[y == 0, 0], X[y == 0, 1])
plt.scatter(X[y == 1, 0], X[y == 1, 1])
plt.show()
```



Regresja logistyczna

Zaimplementujmy algorytm regresji logistycznej, znów za pomocą algorytmu spadku gradientu minimalizując entropię skośną, która jest dobrą miarą odległości między rozkładami:

$$H(p,q) = -\sum_{i=1}^{n} p(x_i) \log q(x_i) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) - \sum_{i=1}^{n} p(x_i) \log \frac{q(x_i)}{p(x_i)}$$

predykcji naszego modelu liniowego z dodaną funkcją sigmoid normalizującą wyjścia naszego modelu do $[0; 1]$, które będziemy traktować jako prawdopodobieństwo klasy pierwszej, warunkowane wejściem, a prawdziwego rozkładu.

Regresja logistyczna

```
def logistic_regression_gd(X, y, learning_rate=0.01, num_iterations=1000, lo
    """
    Performs gradient descent based logistic regression on the input data.

    Parameters:
    X (ndarray): A NumPy array of shape (n_samples, n_features) representing
    y (ndarray): A NumPy array of shape (n_samples,) representing the target
    learning_rate (float): The learning rate for gradient descent. Default i
    num_iterations (int): The number of iterations for gradient descent. Def

    Returns:
    theta (ndarray): A NumPy array of shape (n_features+1,) representing the
    losses (list): values of loss function being optimized after each optimi
    """
```

Regresja logistyczna

```python
# Define the sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Define the cost function
def cost_function(theta, X, y):
    m = len(y)
    h = sigmoid(X.dot(theta))
    J = -(1/m) * (np.log(h).T.dot(y) + np.log(1-h).T.dot(1-y))
    return J

# Define the gradient function
def gradient(theta, X, y):
    m = len(y)
    h = sigmoid(X.dot(theta))
    grad = (1/m) * X.T.dot(h-y)
    return grad
```

Regresja logistyczna

```
# Perform gradient descent
for i in range(num_iterations):
    cost = cost_function(theta, X, y)
    losses.append(cost)
    grad = gradient(theta, X, y)
    theta = ...
    """
    TODO:
    your code goes here
    """
    if i % logging_period == 0:
        print(f"Cost after iteration {i}: {cost}")

return theta, losses
```

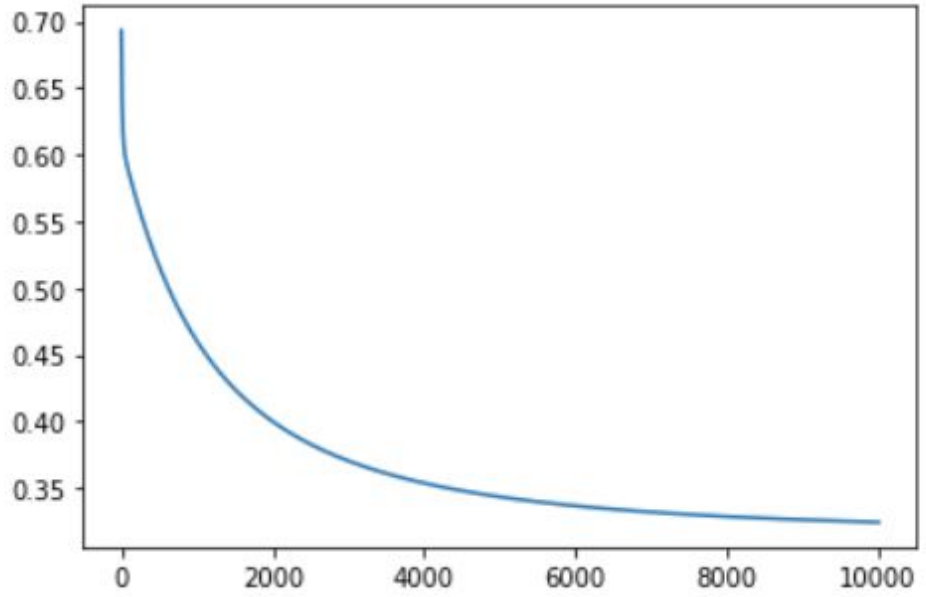
Regresja logistyczna

Uruchommy nasz algorytm znajdowania modelu linowego dla zadania regresji logistycznej:

```
theta, losses = logistic_regression_gd(X, y,
"""
TODO:
your code goes here
"""
)
plt.plot(losses)
plt.show()
```

Regresja logistyczna

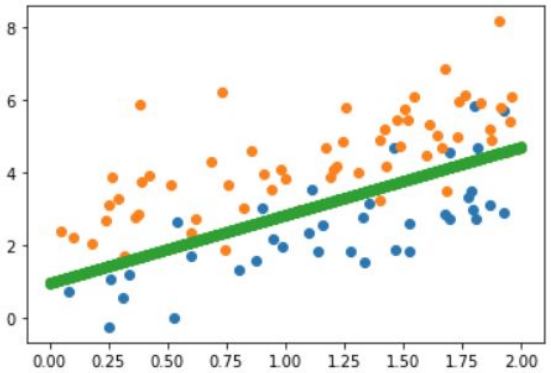
```
Cost after iteration 9600: 0.3250094872132574
Cost after iteration 9700: 0.3248357638911497
Cost after iteration 9800: 0.3246668034049979
Cost after iteration 9900: 0.3245024488605401
```



Regresja logistyczna

Zwizualizujmy predykcje naszego modelu:

```
xs = np.linspace(0, 2, 1000)
plt.scatter(X[y == 0, 0], X[y == 0, 1])
plt.scatter(X[y == 1, 0], X[y == 1, 1])
plt.scatter(xs,
"""
TODO:
your code goes here
"""
)
plt.show()
```



Regresja logistyczna

Obliczmy dokładność naszego modelu – tj. stosunek dobrze sklasyfikowanych przykładów do wszystkich przykładów:

+ Code + Text

```
[ ]  """
     TODO:
     your code goes here
     """
```

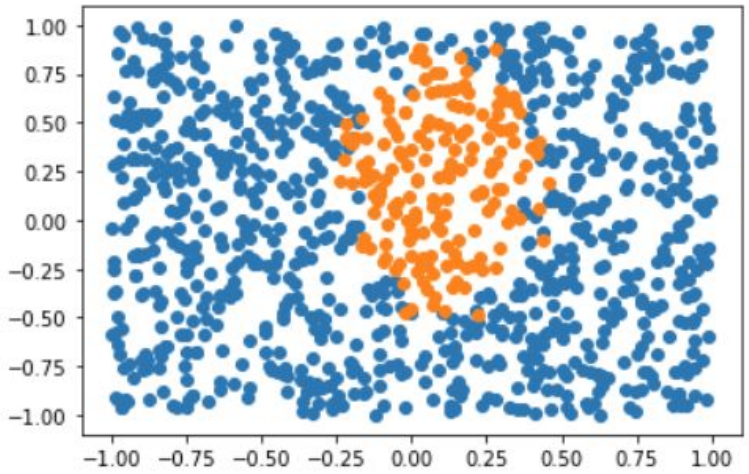
Regresja logistyczna – feature engineering

Podobnie jak w zadaniu regresji liniowej, możemy rozwiązywać nieliniowe problemy klasyfikacji modyfikując wcześniej we właściwy sposób wejściową przestrzeń cech.

Zobaczmy jak to działa na prostym syntetycznym przykładzie zbióru dwóch klas, który da się odseparować elipsą:

Regresja logistyczna – feature engineering

```
X = np.random.uniform(-1, 1, (1000, 2))
y = (np.dot(np.array([2 **2, 1]).reshape(1, 2), ((X - np.array([0.1, 0.2]))

plt.scatter(X[y == 0, 0], X[y == 0, 1])
plt.scatter(X[y == 1, 0], X[y == 1, 1])
plt.show()
```



Regresja logistyczna – feature engineering

We właściwy sposób dobierzmy przestrzeń cech:

+ Code

```
[ ]  X = np.concatenate([
     """
     TODO:
     your code goes here
     """
     ], axis=1)
```

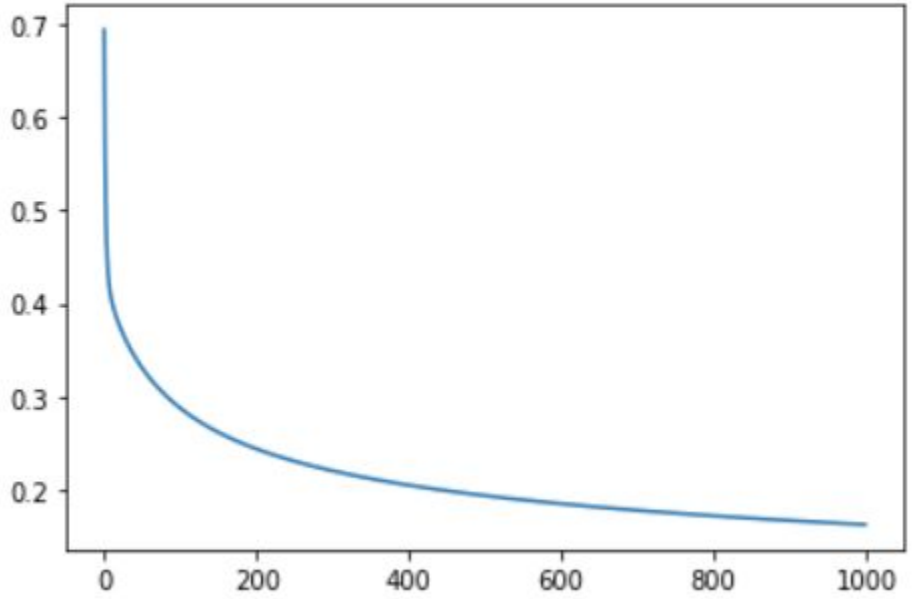
Regresja logistyczna – feature engineering

Uruchommy nasz algorytm regresji logistycznej na przekształconej przestrzeni cech:

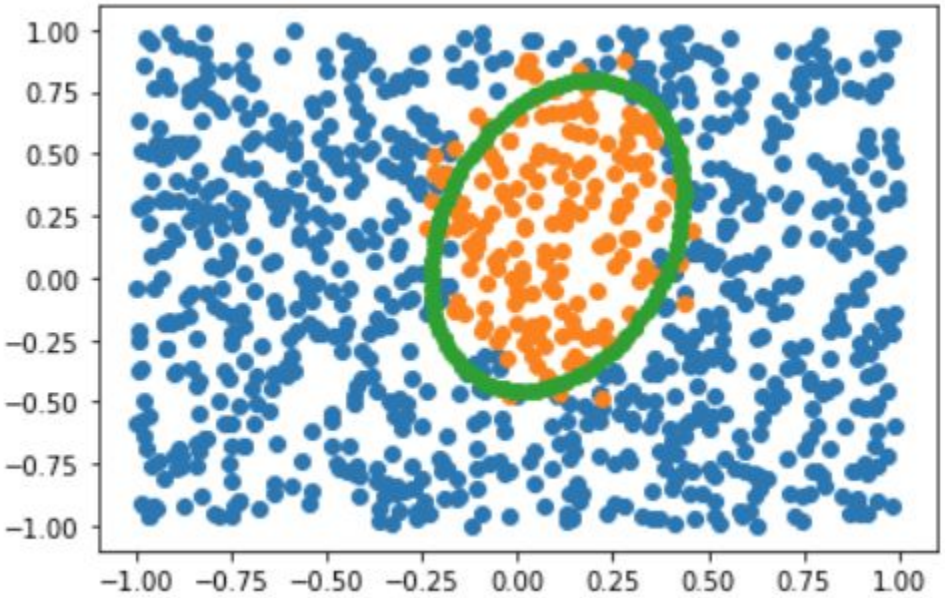
```
[ ] theta, losses = logistic_regression_gd(X, y,
    """
    TODO:
    your code goes here
    """
    )
    plt.plot(losses)
    plt.show()
```

Regresja logistyczna – feature engineering

```
Cost after iteration 700: 0.17821813941188772
Cost after iteration 800: 0.17232426095868067
Cost after iteration 900: 0.16728094643941518
```



Regresja logistyczna – feature engineering



Regresja logistyczna – realny problem

Spróbujmy zastosować wcześniejszy algorytm znajdowania modelu liniowego dla zadania regresji logistycznej do rzeczywistego, poznanego już wcześniej zbioru danych mnist.

Na początek, spróbujmy rozróżnić cyfrę '0' od pozostałych cyfr. Wczytajmy najpierw zbiór danych i podzielmy go na część treningową i testową.

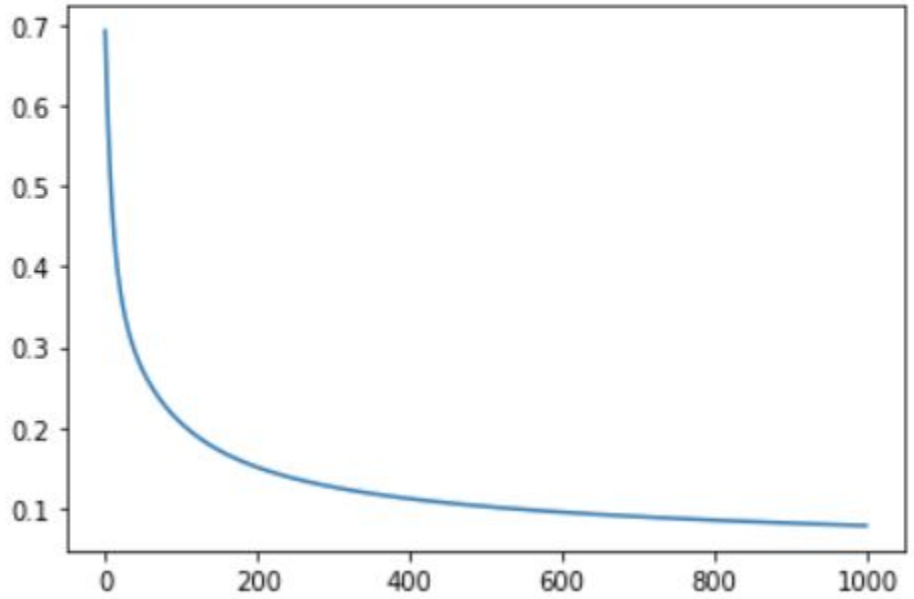
Regresja logistyczna – realny problem

Znormalizujmy nasz zbiór danych, żeby wartości każdego piksela miały oczekiwaną średnią wartość 0 i odchylenie standardowe 1:

```
X_train = X_train / 255.
X_test = X_test / 255.
X_train = ...
X_test = ...
"""
^TODO:
your code goes there
"""
```

Regresja logistyczna – realny problem

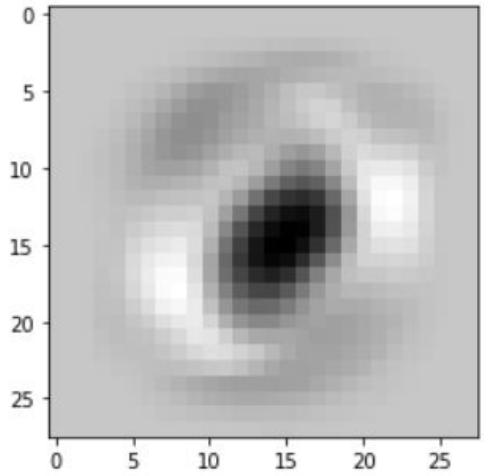
```
Cost after iteration 700: 0.08952273535135805
Cost after iteration 800: 0.08509003504432917
Cost after iteration 900: 0.08140590047860469
```



Regresja logistyczna – realny problem

Spróbujmy zwizualizować wagi naszego klasyfikatora. Jak myślisz, dlaczego dostaliśmy taki kształt?

```
plt.imshow(theta[1:].reshape(28, 28))
plt.show()
```



Klasyfikacja liniowa dla wielu klas – softmax regression, multinomal logistic regression

Nic nie stoi na przeszkodzie, żeby użyć podobnego schematu, do klasyfikacji naszego zbioru danych na więcej niż jedną klasę.

Spróbujemy użyć podobnego schematu jak w przypadku regresji logistycznej, z drobnymi modyfikacjami: nasza funkcja liniowa będzie mapować z rozmiaru wejścia do rozmiaru będącym liczbą możliwych klas, na końcu użyjemy nieco innej funkcji normalizacyjnej, która będzie nam normalizować punkty (score'y) na poszczególnych klasach do dystrybucji nad tymi klasami: $softmax(x_1, \ldots, x_n) = (\frac{exp(x_i)}{exp(x_1)+\ldots+exp(x_n)})_i$.

Nadal będziemy jako funkcji celu używać entropii skośnej, jako miary odległości pomiędzy rozkładami:

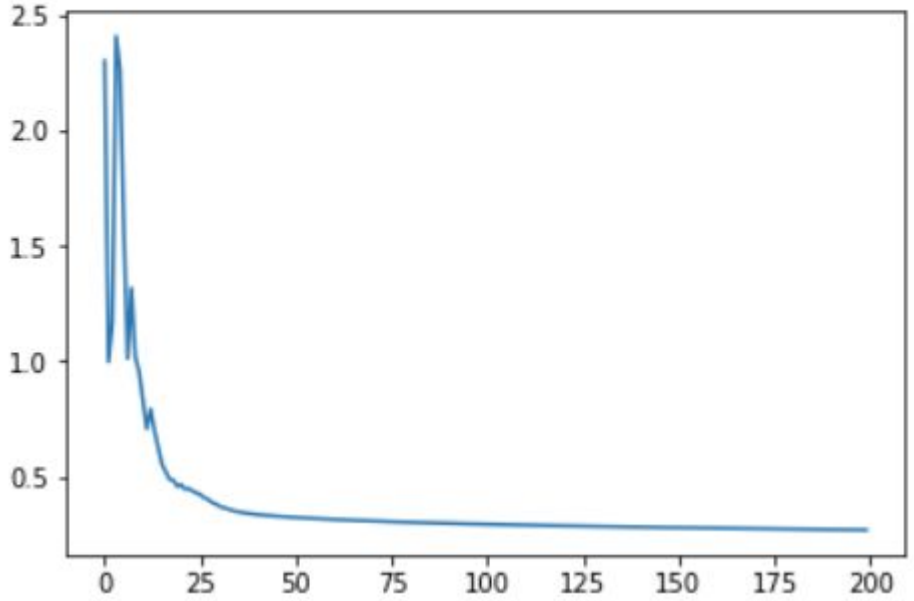
$$H(p,q) = -\sum_{i=1}^{n} p(x_i) \log q(x_i) = -\sum_{i=1}^{n} p(x_i) \log p(x_i) - \sum_{i=1}^{n} p(x_i) \log \frac{q(x_i)}{p(x_i)}$$

Klasyfikacja liniowa dla wielu klas – softmax regression, multinomal logistic regression

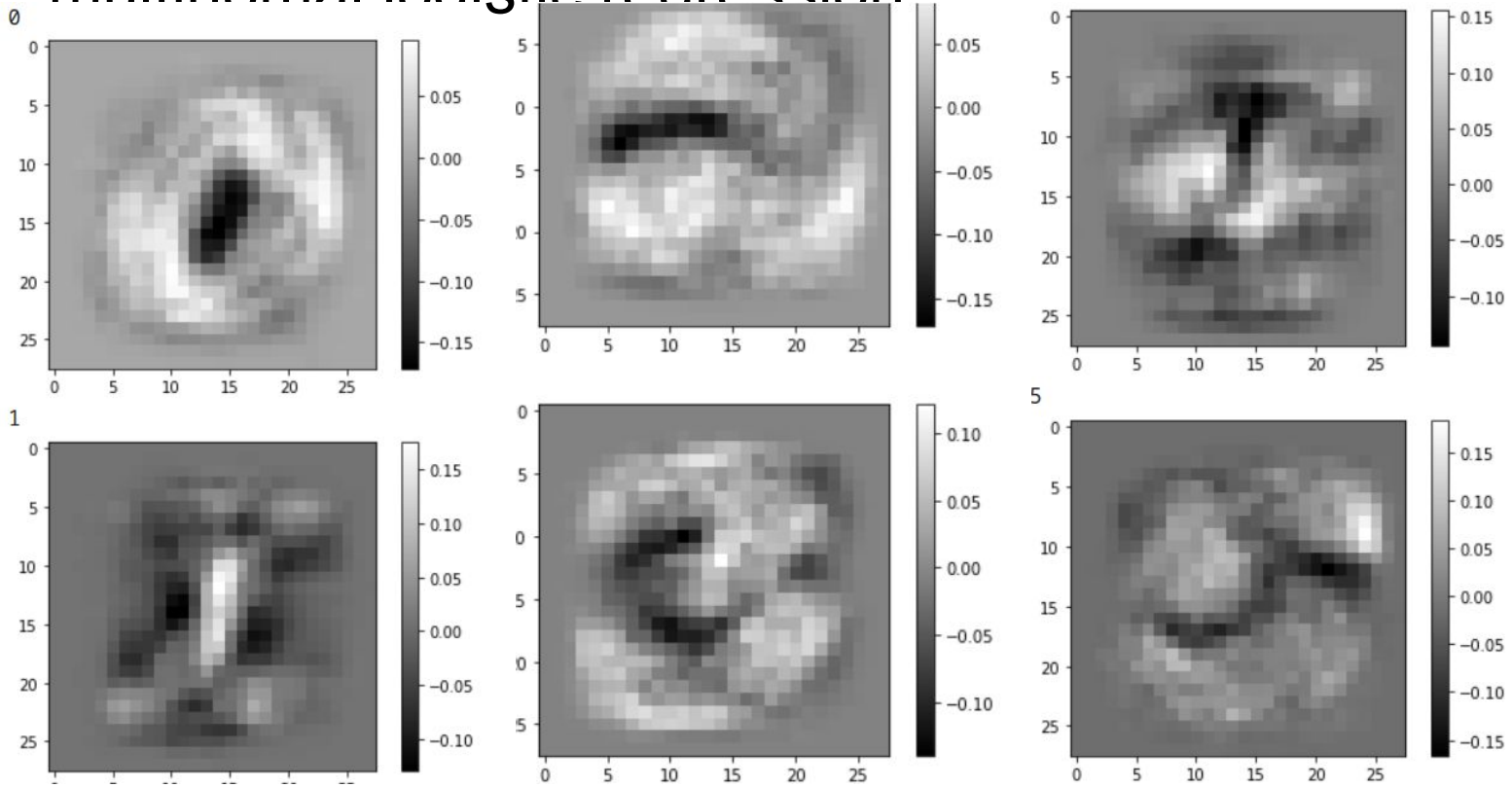
```
# Perform gradient descent
for i in range(num_iterations):
    cost = cost_function(theta, X, y)
    grad = gradient(theta, X, y)
    theta = ...
    """
    TODO:
    your code goes here
    """
    losses.append(cost)
    if i % logging_period == 0:
        print(f"Cost after iteration {i}: {cost}")

return theta, losses
```

Klasyfikacja liniowa dla wielu klas – softmax regression, multinomal logistic regression

```
Cost after iteration 170: 0.27536598206344287
Cost after iteration 180: 0.2734564298227489
Cost after iteration 190: 0.2716709871855081
```



Klasyfikacja liniowa dla wielu klas – softmax regression, multinomal logistic regression



Framework torch

Automatyczną kalkulacją gradientów zajmuje się framework pytorch. Zadanie: proszę przepisać chociażby jeden z przykładów używając tego frameworku

Framework sklearn

Klasyczne algorytmu MLowe, w tym algorytmy regresji poznane dzisiaj są poimplementowane w paczce sklearn.

Proszę z jej pomocą zreimplementować chociażby jeden dzisiejszych przykładów.