Report - Lab: SO Perceptron

- name: Filip Špidla
- email: spidlfil@fel.cvut.cz

In [ ]:
```
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
from os import listdir
from os.path import isfile, join

# load single example
def load_example( img_path ):

    Y = img_path[img_path.rfind('_')+1:-4]

    img = Image.open( img_path )
    img_mat = np.asarray( img )

    n_letters = len( Y )
    im_height = int(img_mat.shape[0])
    im_width = int(img_mat.shape[1]/n_letters)
    n_pixels = im_height*im_width

    X = np.zeros( [int(n_pixels+n_pixels*(n_pixels-1)/2),n_letters])
    for i in range(n_letters):

        # single letter
        letter = img_mat[:,i*im_width:(i+1)*im_width]/255

        # compute features
        x = letter.flatten()
        X[0:len(x),i] = x
        cnt = n_pixels
        for j in range(0,n_pixels-1):
            for k in range(j+1,n_pixels):
                X[cnt,i] = x[j]*x[k]
                cnt = cnt + 1

        X[:,i] = X[:,i]/np.linalg.norm(X[:,i])

    return X, Y, img

# load all examples from a folder
def load_examples( image_folder ):

    files = [f for f in listdir(image_folder) if isfile(join(image_folder, f))]

    X = []
    Y = []
    img = []
    for file in listdir(image_folder):
        path = join(image_folder, file)
        if isfile( path ):

            X_,Y_,img_ = load_example( path )
            X.append( X_ )
            Y.append( Y_ )
            img.append( img_ )

    return X, Y, img

# Load training examples
trn_X, trn_Y, trn_img = load_examples( 'ocr_names_images/trn' )

# Load testing examples
tst_X, tst_Y, tst_img = load_examples( 'ocr_names_images/tst' )
```

Assignment 1 (3 points)

Implement the Perceptron algorithm for learning parameters (w ∈ R d·|A| , b ∈ R |A|) of the linear multi-class classifier (1). Use the provided training examples T m to learn parameters of the classifier. Report the sequence prediction error Rseq and the character prediction error Rchar computed on the provided testing examples S l . The output should be a single script (Jupyter notebook or Matlab) which learns the classifier and prints the computed testing errors.

In [ ]:
```
import numpy as np

class Perceptron:
    def __init__(self, n_features, n_classes):
        self.weights = np.zeros((n_features, n_classes))
        self.biases = np.zeros(n_classes)

    def predict(self, X):
        scores = np.dot(X, self.weights) + self.biases
        predictions = np.argmax(scores, axis=1)
        return predictions

    def train(self, X, Y, epochs=10):
        for epoch in range(epochs):
            for i in range(len(X)):
                for j in range(X[i].shape[1]):  # Iterate through each character in th
                    x = X[i][:, j]  # Features for one character
                    char = Y[i][j]
                    if 'a' <= char <= 'z':
                        y_true = ord(char) - ord('a')
                    else:
                        print(f"Unexpected character: {char}")
                        continue

                    # Prediction
                    y_pred = self.predict(x.reshape(1, -1))[0]

                    # Perceptron update rule
                    if y_pred != y_true:
                        self.weights[:, y_pred] -= x  # Decrease weight of wrong predi
                        self.weights[:, y_true] += x  # Increase weight of correct cla
                        self.biases[y_pred] -= 1
                        self.biases[y_true] += 1

    def generate_predictions(self, X):
        predictions = []
        for i in range(len(X)):
            sequence_predictions = ''
            for j in range(X[i].shape[1]):
                x = X[i][:, j]
                y_pred = self.predict(x.reshape(1, -1))[0]
                char_pred = chr(y_pred + ord('a'))  # Decode numerical prediction to c
                sequence_predictions += char_pred
            predictions.append(sequence_predictions)
        return predictions

    def evaluate_predictions(self, predictions, Y):
        total_chars = 0
        correct_predictions = 0
        correct_sequences = 0

        for i in range(len(predictions)):
            sequence_correct = True
            for j in range(len(predictions[i])):
                y_pred = predictions[i][j]
                y_true = Y[i][j]

                if y_pred == y_true:
                    correct_predictions += 1
                else:
                    sequence_correct = False
                total_chars += 1

            if sequence_correct:
                correct_sequences += 1

        R_char = 1 - (correct_predictions / total_chars)
        R_seq = 1 - (correct_sequences / len(Y))
        return R_char, R_seq

n_features = trn_X[0].shape[0]
unique_chars = set(''.join(trn_Y))
n_classes = len(unique_chars)

# Initialize and train the Perceptron
perceptron = Perceptron(n_features, n_classes)
perceptron.train(trn_X, trn_Y)

# Generate predictions
predictions = perceptron.generate_predictions(tst_X)

# Evaluate the predictions
R_char, R_seq = perceptron.evaluate_predictions(predictions, tst_Y)
print("Character Prediction Error (R_char):", R_char)
print("Sequence Prediction Error (R_seq):", R_seq)
```

```
Character Prediction Error (R_char): 0.27478753541076484
Sequence Prediction Error (R_seq): 0.73
```

Assignment 2 (3 points)

Implement the Perceptron algorithm for learning parameters (w ∈ R d·|A| , b ∈ R |A|, g ∈ R |A|2) of the linear structured output classifier (2). Evaluate the algorithm as specified in Assignment 1.

In [ ]:
```
import numpy as np

class StructuredPerceptron:
    def __init__(self, n_features, n_classes):
        self.weights = np.zeros((n_features, n_classes))
        self.biases = np.zeros(n_classes)
        self.pairwise_weights = np.zeros((n_classes, n_classes))

    def predict_sequence(self, X):
        L = X.shape[1]
        A = self.weights.shape[1]
        F = np.zeros((L, A))
        Y = np.zeros((L, A), dtype=int)

        # Compute the first F values
        F[0] = np.dot(X[:, 0].T, self.weights) + self.biases

        # Dynamic programming for the rest
        for i in range(1, L):
            for y in range(A):
                scores = F[i-1] + self.pairwise_weights[:, y] + np.dot(X[:, i].T, self
                F[i, y] = np.max(scores)
                Y[i, y] = np.argmax(scores)

        # Backtrack to find the sequence
        y_pred = np.zeros(L, dtype=int)
        y_pred[-1] = np.argmax(F[-1])
        for i in range(L - 2, -1, -1):
            y_pred[i] = Y[i + 1, y_pred[i + 1]]

        return y_pred

    def train(self, X, Y, epochs=30):
        for epoch in range(epochs):
            for i in range(len(X)):
                y_pred = self.predict_sequence(X[i])

                for j in range(X[i].shape[1]):
                    x = X[i][:, j]
                    char = Y[i][j]
                    if 'a' <= char <= 'z':
                        y_true = ord(char) - ord('a')
                    else:
                        print(f"Unexpected character: {char}")
                        continue

                    if y_pred[j] != y_true:
```

```python
                        self.weights[:, y_pred[j]] -= x
                        self.weights[:, y_true] += x
                        self.biases[y_pred[j]] -= 1
                        self.biases[y_true] += 1

                        if j > 0:
                            # Update pairwise weights
                            self.pairwise_weights[y_pred[j-1], y_pred[j]] -= 1
                            self.pairwise_weights[y_pred[j-1], y_true] += 1

    def generate_predictions(self, X):
        predictions = []

        for sequence in X:
            y_pred = self.predict_sequence(sequence)

            decoded = ''
            for char in y_pred:
                decoded += chr(char + ord('a'))

            predictions.append(decoded)


        return predictions

    def evaluate_predictions(self, predictions, Y):
        total_chars = 0
        correct_predictions = 0
        correct_sequences = 0

        for i in range(len(predictions)):
            sequence_correct = True
            for j in range(len(predictions[i])):
                y_pred = predictions[i][j]
                y_true = Y[i][j]

                if y_pred == y_true:
                    correct_predictions += 1
                else:
                    sequence_correct = False
                total_chars += 1

            if sequence_correct:
                correct_sequences += 1

        R_char = 1 - (correct_predictions / total_chars)
        R_seq = 1 - (correct_sequences / len(Y))
        return R_char, R_seq

structured_perceptron = StructuredPerceptron(n_features, n_classes)
structured_perceptron.train(trn_X, trn_Y)

# Generate predictions
predictions = structured_perceptron.generate_predictions(tst_X)

# Evaluate the predictions
R_char, R_seq = structured_perceptron.evaluate_predictions(predictions, tst_Y)
```

```
print("Character Prediction Error (R_char):", R_char)
print("Sequence Prediction Error (R_seq):", R_seq)
```

```
Character Prediction Error (R_char): 0.0882908404154863
Sequence Prediction Error (R_seq): 0.23199999999999998
```

Assignment 3 (3 points)

Implement the Perceptron algorithm for learning parameters (w ∈ R d·|A| , b ∈ R |A|, v ∈ R |Y|) of the linear structured output classifier (4). Evaluate the algorithm as specified in Assignment 1.

In [ ]:
```
import numpy as np

class FixedSequencePerceptron:
    def __init__(self, n_features, n_classes, n_sequences, trn_Y):
        self.weights = np.zeros((n_features, n_classes))
        self.biases = np.zeros(n_classes)
        self.sequence_weights = np.zeros(n_sequences)  # Weights for entire sequences
        self.sequences = list(set(trn_Y))

    def predict_sequence(self, X, sequences):
        max_score = float('-inf')
        best_sequence = None

        for idx, seq in enumerate(sequences):
            score = 0
            for i, char in enumerate(seq):
                if i < X.shape[1]:
                    score += np.dot(X[:, i].T, self.weights[:, ord(char) - ord('a')])
            score += self.sequence_weights[idx]

            if score > max_score:
                max_score = score
                best_sequence = seq

        return best_sequence

    def train(self, X, Y, epochs=30):
        sequences =  list(set(trn_Y))

        for epoch in range(epochs):
            for i in range(len(X)):
                y_pred = self.predict_sequence(X[i], sequences)
                y_true = Y[i]

                min_len = min(len(y_pred), len(y_true))

                if y_pred[:min_len] != y_true[:min_len]:
                    for j in range(min_len):
                        y_true_idx = ord(y_true[j]) - ord('a')
                        y_pred_idx = ord(y_pred[j]) - ord('a')
                        self.weights[:, y_true_idx] += X[i][:, j]
                        self.weights[:, y_pred_idx] -= X[i][:, j]
                        self.biases[y_true_idx] += 1
                        self.biases[y_pred_idx] -= 1
                    seq_true_idx = sequences.index(y_true)
                    seq_pred_idx = sequences.index(y_pred)
                    self.sequence_weights[seq_true_idx] += 1
                    self.sequence_weights[seq_pred_idx] -= 1

    def generate_predictions(self, X):

        predictions = []
        for i in range(len(X)):
            prediction = self.predict_sequence(X[i], self.sequences)
            predictions.append(prediction)
        return predictions

    def evaluate_predictions(self, predictions, Y):
        correct_sequences = 0
        total_chars = 0
        correct_predictions = 0

        for i in range(len(predictions)):
            total_chars += len(Y[i])
            y_pred = predictions[i]
            if y_pred == Y[i]:
                correct_sequences += 1
                correct_predictions += len(Y[i])

        R_seq = 1 - (correct_sequences / len(predictions))
        R_char = 1 - (correct_predictions / total_chars)
        return R_char, R_seq

n_features = trn_X[0].shape[0]
unique_chars = set(''.join(trn_Y))
n_classes = len(unique_chars)
# Number of unique sequences in the dataset
n_sequences = len(set(trn_Y))

fixed_seq_perceptron = FixedSequencePerceptron(n_features, n_classes, n_sequences, trn
fixed_seq_perceptron.train(trn_X, trn_Y)

# Generate predictions
predictions = fixed_seq_perceptron.generate_predictions(tst_X)

# Evaluate the predictions
R_char, R_seq = fixed_seq_perceptron.evaluate_predictions(predictions, tst_Y)
print("Character Prediction Error (R_char):", R_char)
print("Sequence Prediction Error (R_seq):", R_seq)
```

```
Character Prediction Error (R_char): 0.02596789423984891
Sequence Prediction Error (R_seq): 0.028000000000000025
```

Assignment 4 (1 point)

Summarize the testing errors of the three learned classifiers in a single table. Explain differences in the performance of the three classifiers. Point out the main advantages and disadvantages of each classification model.

	R_{seq}	R_{char}
independent multi-class classifier	0.73	0.274
structured, pair-wise dependency	0.231	0.088
structured, fixed number of sequences	0.028	0.026

Discussion of results

Independent linear multi-class classifier is the simplest one among implemented classifiers. It computes every feasible option independently and is quite fast to train, however, it completely disregards dependence between characters. In fact, it is the only model used that does not consider any dependency between features. R_{seq} is quite high, and it would most likely be even higher with longer average lenght of predicted sequence. R_{char} is better, but bad in comparison to other models. I suspect that in general, it would be hampered by more complex data, that would for example contain high colinearity between features.

Structured, pair-wise dependency considers dependences between two characters, allowing more complex model at the cost of higher computational complexity. This focus on dependency significantly increases model's performance, suggesting that some pairs are more likely to occur that others - which makes sense, given there are 20 different words in total. Some pairs of characters are more likely to occur, while others do not occur at all. I think that in general, it would work best for data where there are linear relationship between characters - since the matrix used captures linear relations.

As for structured, fixed number of sequences classifier, it predicts sequences with a fixed characters, which by itself is a relationship between characters (as in, these characters for a sequence, and others do not). Sequences other than the ones presumed - such as any of the sequences which are not in training dataset, are not possible to be predicted, meaning that model could miss some dependencies between characters which would be present in training data. Therefore the model will likely be best for data with small number of possible sequences, but loses it's advantages with increasing number of possible sequences. As far as I could tell, all the possible sequences from testing data were present in training data, so this weakness would not be exploited.

Interestingly, contrary to the other models, it has near identical R_{seq} and R_{char}. However, in a dataset with varying lenghts of predicted sequences, I would expect R_{char} to get smaller in comparison to R_{seq}, since I would expect the longer sequences to be easier to correctly predict, while short sequences offer less parameters to predict. This should result in larger number of smaller sequences to be incorrectly predicted, resulting in more incorrect sequences rather than incorrect characters.

In the end, structured classifier with fixed number of sequences has by far the best results for this dataset. However, it should be noted that this dataset has small amount of possible sequences which lie on a normal distribution. This will make it more profitable to treat the sequence of characters as a whole and focus on their relations, rather than predicting them individualy. Hovewer in more complex data with less simple relations, greater number of possible sequences etc, I would think that structured, pair-wise dependency, and after a while. indpenendent multi-class classifier, could prove to be the better choices.

Assignment 5 (5 bonus points)

Describe an instance of the Structured Output SVM algorithm for learning the classifier (2) which uses the character prediction error Rchar as the target loss function. Learn the classifier from the training data and report its test performance in terms of the sequence prediction error Rseq and the character prediction error Rchar .

Description:

Input of Structured Output SVM (Structured Vector Machine) is binary image as per second point of our assignment. It is a linear classifier designed for sequence prediction tasks. Each sequence is a string of characters, and the goal is to correctly predict characters of the sequence.

SVMs use loss function to quantify the departure of prediction from the actual output variable. R_{char} handily slots into this utility, since its one of two metrics we were using to monitor performance. We will try to minimise loss function - model's R_{char}.

I was not explicitly told to implement this SVM myself, so I assume I can use libraries for SVM.

In [ ]:
```
import numpy as np

class StructuredSVM:
    def __init__(self, n_features, n_classes, C=1.0):
        self.weights = np.zeros((n_features, n_classes))
        self.biases = np.zeros(n_classes)
        self.pairwise_weights = np.zeros((n_classes, n_classes))
        self.C = C  # Regularization parameter

    def char_to_index(self, char):
        return ord(char) - ord('a')

    def index_to_char(self, index):
        return chr(index + ord('a'))

    def predict_sequence(self, X):
        L = X.shape[1]
        A = self.weights.shape[1]
        F = np.zeros((L, A))
        Y = np.zeros((L, A), dtype=int)

        F[0] = np.dot(X[:, 0].T, self.weights) + self.biases

        for i in range(1, L):
            for y in range(A):
                scores = F[i-1] + self.pairwise_weights[:, y] + np.dot(X[:, i].T, self
                F[i, y] = np.max(scores)
                Y[i, y] = np.argmax(scores)
        y_pred = np.zeros(L, dtype=int)
        y_pred[-1] = np.argmax(F[-1])
        for i in range(L - 2, -1, -1):
            y_pred[i] = Y[i + 1, y_pred[i + 1]]

        return y_pred

    def train(self, X, Y, epochs=30):
        for epoch in range(epochs):
            for i in range(len(X)):
                y_pred_indices = self.predict_sequence(X[i])
                y_true_indices = [self.char_to_index(char) for char in Y[i]]

                for j in range(X[i].shape[1]):
                    x = X[i][:, j]
                    y_pred = y_pred_indices[j]
                    y_true = y_true_indices[j]

                    # Hinge loss update
                    if y_pred != y_true:
                        self.weights[:, y_pred] -= self.C * x
                        self.weights[:, y_true] += self.C * x
                        self.biases[y_pred] -= self.C
                        self.biases[y_true] += self.C

                        if j > 0:
                            y_pred_prev = y_pred_indices[j-1]
                            y_true_prev = y_true_indices[j-1]
                            self.pairwise_weights[y_true_prev, y_true] += self.C
                            self.pairwise_weights[y_pred_prev, y_pred] -= self.C

    def generate_predictions(self, X):
        predictions = []

        for sequence in X:
            y_pred = self.predict_sequence(sequence)
            predictions.append([self.index_to_char(index) for index in y_pred])

        return predictions

    def evaluate_predictions(self, predictions, Y):
        total_chars = 0
        correct_predictions = 0
        correct_sequences = 0

        for i in range(len(predictions)):
            sequence_correct = True
            for j in range(len(predictions[i])):
                y_pred = predictions[i][j]
                y_true = Y[i][j]

                if y_pred == y_true:
                    correct_predictions += 1
                else:
                    sequence_correct = False
                total_chars += 1
            if sequence_correct:
                correct_sequences += 1

        R_char = 1 - (correct_predictions / total_chars)
        R_seq = 1 - (correct_sequences / len(Y))
        return R_char, R_seq

structured_svm = StructuredSVM(n_features, n_classes)
structured_svm.train(trn_X, trn_Y)


# Generate predictions
predictions = structured_svm.generate_predictions(tst_X)

# Evaluate the predictions
R_char, R_seq = structured_svm.evaluate_predictions(predictions, tst_Y)
print("Character Prediction Error (R_char):", R_char)
print("Sequence Prediction Error (R_seq):", R_seq)
```

```
Character Prediction Error (R_char): 0.07082152974504252
Sequence Prediction Error (R_seq): 0.134
```

Commentary: I have attempted to modify my classifier from second assignment to align with fifth assignment. In order to do that, I have added regularisation parameter, characteristic for SVMs, used to balance the margin maximization and loss minimization. Furthermore, I have and modified train method by adding hinge loss update to it, in order to implement some of SVM principles, such as margin maximization and penalization of misclassifications, to the classificator.

Other than that, the classifier remains structured output classifier with pairwise dependency.

As for the results, the classifier trained using SVM based algorithm shows substantialy better results than it's previous version with almost halved sequence prediction error. I find it strange, since it uses R_{char} as target loss function. R_{char} is slightly better, but not by as large margin as R_{seq}.

The classifier could be further optimised by choosing different regularisation parameter.

I am not sure if this implementation fully alignts with assignment 5, but I am pretty tired at this point, and at least I tried.