Machine Learning – Task 2

משה ביניאלי (Moshe Binieli), 311800668

Data preparation

The way I split the data is by taking 80% of the whole dataset and declare it as a training set and the other 20% of the data is test set.

```
def splitDataset(train_x, train_y, splitSize=0.8):
    splitSize = round(len(train_x) * splitSize)

    train_x_new = np.array(train_x[:splitSize, :])
    test_x_new = np.array(train_x[splitSize:, :])
    train_y_new = np.array(train_y[:splitSize])
    test_y_new = np.array(train_y[splitSize:])

    return train_x_new, test_x_new, train_y_new, test_y_new
```

So, we got 2673 entities for the train set and 668 entities for the test set.

Measurement of algorithms

The way I calculate the measurement of how the model trained well is by taking all the correct predictions and divide them by the sample size.

```
print(f"Perceptron accuracy: {(testSize - perceptronMissCounter) / testSize}")
print(f"SVM accuracy: {(testSize - svmMissCounter) / testSize}")
print(f"PA accuracy: {(testSize - paMissCounter) / testSize}")
```

Shuffle

I shuffle the training set and the test set always, so the predictions will be different every time but small changes.

Dealing with categorical data

The categorical data has been converted like this:

F is 0, I is 1, M is 2

```
def prepareData(train_x, train_y, test_x):
    train_x[train_x == 'F'] = '0'
    train_x[train_x == 'I'] = '1'
    train_x[train_x == 'M'] = '2'
    train_x = train_x.astype(np.float)

    test_x[test_x == 'F'] = '0'
    test_x[test_x == 'I'] = '1'
    test_x[test_x == 'M'] = '2'
    test_x = test_x.astype(np.float)

    train_x = np.append(train_x, np.ones((len(train_x), 1)), axis=1)
    test_x = np.append(test_x, np.ones((len(test_x), 1)), axis=1)

    train_y = train_y.astype(np.int)

    return train_x, train_y, test_x
```

Perceptron

I've defined a class for Perceptron algorithm which receives in the constructor a learning rate and iteration amount, and a fit method that receives the training set and test set and computes the weights of the perceptron algorithm.

```python
class Perceptron(object):
    def __init__(self, learningRate=0.01, iterationsAmount=10):
        self.learningRate = learningRate
        self.iterationsAmount = iterationsAmount

    def fit(self, train_x, train_y):
        numberOfClasses = np.unique(train_y).size
        weights = np.zeros((numberOfClasses, train_x[0].size))

        for _ in range(self.iterationsAmount):
            for xi, yi in zip(train_x, train_y):
                y_hat = np.argmax(np.dot(weights, xi))

                if(yi != y_hat):
                    weights[yi, :] += self.learningRate * xi
                    weights[y_hat, :] -= self.learningRate * xi

        return weights
```

In the beginning, I set the learning rate to 0.1 and the iteration amount to 10, this method was kind of bad because it gave me prediction results such as 0.45, 0.55, 0.62, it was very noisy and I couldn't work well with that, I want something more stable.

So I decided to give the algorithm more time to work with more epochs and I set the iteration amount to 100 and the learning rate to 0.1 and still, it was very bad, so I raised the learning rate to 0.001 and this choice of hyperparameters gave me the stability of getting the same prediction of value 0.57.

I took the learning rate to 0.0001 and iteration amount to 100 and the predictions was stable and it was mostly around 0.59, this value was pretty good for me.

Support Vector Machine

I've defined a class for Support Vector Machine algorithm which receives learning rate, iteration amount and lamda as regularization parameter.

```python
class SupportVectorMachine(object):
    def __init__(self, learningRate=0.01, iterationsAmount=10, lamda=0.1):
        self.lamda = lamda
        self.learningRate = learningRate
        self.iterationsAmount = iterationsAmount

    def fit(self, train_x, train_y):
        numberOfClasses = np.unique(train_y).size
        weights = np.zeros((numberOfClasses, train_x[0].size))

        for _ in range(self.iterationsAmount):
            for xi, yi in zip(train_x, train_y):
                y_hat = np.argmax(np.dot(weights, xi))

                if(yi != y_hat):
                    beta = 1 - self.learningRate * self.lamda
                    weights[yi, :] = beta * weights[yi, :] + self.learningRate * xi
                    weights[y_hat, :] = beta * weights[y_hat, :] - self.learningRate * xi

                    otherWeights = np.arange(0, len(weights)).tolist()
                    otherWeights.remove(yi)
                    otherWeights.remove(y_hat)
                    weights[otherWeights,:] = (1 - self.learningRate * self.lamda) *  weights[otherWeights,:]
                else:
                    weights[:, :] = (1 - self.learningRate * self.lamda) *  weights[:, :]


        return weights
```

I started with naive hyperparameters of learning rate 0.1, lamda 0.1 and iteration amount of 10.

those parameters were too noisy and I couldn't stable it, so I decided to take the iteration amount to 100 and the learning rate to 0.01, those changes changed the predictions to around 0.5, so I changed the learning rate to 0.001 and it changed the prediction value and it moves around 0.58 – 0.6.

Changing the hyperparameters of the learning rate or iteration number affected badly on the prediction, so I decided to remain it the same and work with the lamda, but also changing the lamda to 0.001 or 0.0001 reduced the predictions to 0.3 - 0.4, so I gave up on this and remain it on 0.1.

Passive Aggressive

I've defined a class for Passive-Aggressive algorithm which receives iteration amount.

```
class PassiveAggressive(object):
    def __init__(self, iterationsAmount=10):
        self.iterationsAmount = iterationsAmount

    def fit(self, train_x, train_y):
        numberOfClasses = np.unique(train_y).size
        weights = np.zeros((numberOfClasses, train_x[0].size))

        for _ in range(self.iterationsAmount):
            for xi, yi in zip(train_x, train_y):
                y_hat = np.argmax(np.dot(weights, xi))

                if(yi != y_hat):
                    loss = max(0, 1 - np.dot(weights[yi, :], xi) + np.dot(weights[y_hat, :], xi))
                    gama = loss / (2 * (np.linalg.norm(xi) ** 2))

                    weights[yi, :] += gama * xi
                    weights[y_hat, :] -= gama * xi

        return weights
```

Setting the iteration amount to 10 made the predictions unstable, it was running around 0.4 to 0.6, so I decided to change the iteration amount to 100, this change made the algorithm give bad prediction numbers.

I decided to change the iteration numbers around 70 and then lower to 50 and then lower to 20, those changes made the algorithm increase to 0.58 - 0.6 prediction accuracy.