# Εργασία 2 – Αναφορά Παράδοσης

## Θέμα

Το θέμα της εργασίας είναι η υλοποίηση διαφόρων αλγορίθμων μηχανικής μάθησης της επιλογής μας (Bernoulli Naive Bayes, Logistic Regression, Random Forest, RNN) και χρησιμοποιώντας τους να κάνουμε sentiment analysis στις κριτικές του IMBD μέσω του IMDb Dataset καθώς επίσης και να συγκρίνουμε τα αποτελέσματα μας με αυτά των έτοιμων αλγορίθμων του πακέτου scikit-learn.

## Προεπεξεργασία δεδομένων

Για να φορτώσουμε τα δεδομένα μας χρησιμοποιούμε τον κώδικα του φροντιστηρίου του μαθήματος όπου χρησιμοποιώντας την εντολή load_data του keras και με τις κατάλληλες τιμές m, n, k* στα ορίσματα num_words και skip_top χωρίσαμε τις μισές κριτικές με τα labels τους στα train δεδομένα και τις υπόλοιπες στα test. Πιο συγκεκριμένα, για τους αλγορίθμους που χρειάζονται validation set για το fine tuning των υπερπαραμέτρων όπως ο Logistic Regression, χωρίσαμε εκ νέου το training set στα δύο δίνοντας ένα μικρό ποσοστό γύρω στο 20% στα dev δεδομένα. Ειδικότερα, μετά από δοκιμές στον κάθε αλγόριθμο καταλήξαμε στις ακόλουθες τιμές των υπερπαραμέτρων :

1) Bernoulli Naive Bayes : m = 2500, n = 200, k = 0
2) Logistic Regression : m = 3000, n = 30, k = 20
3) Random Forest : m = 3000, n = 20
4) RNN : m = 1000, k = 20

Σε αυτό το σημείο τα δεδομένα έχουν την μορφή κειμένου οπότε για να μπορέσουμε να τα χρησιμοποιήσουμε σε έναν binary classifier χρησιμοποιούμε τον CountVectorizer, όπως και στο φροντιστήριο, για να τα μετατρέψουμε σε πίνακες από άσσους και μηδενικά όπου το κάθε στοιχείο τους δηλώνει την ύπαρξη(1) ή όχι(0) του feature, δηλαδή της λέξης της στήλης την οποία κοιτάμε, στην κριτική της γραμμής που βρισκόμαστε.

*η μεταβλητή m είναι ο αριθμός των λέξεων του λεξιλογίου και οι n και η k ο αριθμός των περισσότερο και λιγότερο συχνών λέξεων που θα προσπεράσουμε.

**το feature matrix X είναι ένας δισδιάστατος πίνακας με δυαδικές τιμές όπου η κάθε του γραμμή είναι μια κριτική και η κάθε του στήλη ένα γνώρισμα, δηλαδή μία λέξη του λεξιλογίου. Τιμή 1 σε κάποια θέση του matrix δηλώνει ότι η κριτική της γραμμής στην οποία βρισκόμαστε περιέχει μέσα της την λέξη της στήλης την οποία κοιτάμε. Το target vector y είναι ένα διάνυσμα μήκους όσο και το μήκος του X με τιμές 1 ή 0 ανάλογα με το αν η κριτική της αντίστοιχης γραμμής στον X έχει χαρακτηριστεί ως θετική ή αρνητική.

# Bernoulli Naive Bayes

## 1. Υλοποίηση

Για την υλοποίηση του Bernoulli Naive Bayes έχουμε την κλάση `class BernoulliNaiveBayes():` η οποία έχει τις ακόλουθες μεθόδους :

a) `def __init__(self):` δεν έχει κάποια χρήση, λόγω του γεγονότος ότι η υλοποίηση μας δεν δέχεται ως ορίσματα υπερπαραμέτρους, και υπάρχει απόκλειστικά για λόγους πληρότητας.

b) `def fit(self, X, y):` δέχεται σαν όρισμα ένα feature matrix** X και ένα target vector y** και ουσιαστικά εκπαιδεύει το μοντέλο μας. Ειδικότερα, υπολογίζει αρχικά τις 2 “prior” πιθανότητες, δηλαδή την πιθανότητα μία κριτική να ανήκει στην θετική κατηγορία και την πιθανότητα να ανήκει στην αρνητική. Την πρώτη (θετική) την βρίσκουμε αθροίζοντας όλους τους άσσους του target vector και διαιρώντας με το συνολικό μήκος του και η δεύτερη είναι απλά η διαφορά της πρώτης από το 1. Στη συνέχεια, υπολογίζουμε την πιθανότητα για κάθε feature να έχει την τιμή 1 και την πιθανότητα να έχει τιμή 0 με δεδομένο ότι ανήκει σε μία κατηγορία. Επομένως, έχουμε να υπολογίσουμε την πιθανότητα $P(X_i = x_i \mid C = c)$ για ($x_i$ = 1 και c = 1), ($x_i$ = 0 και c = 1) , ($x_i$ = 0 και c = 0) και ($x_i$ = 1 και c = 0). Η πρώτη και η τρίτη που αφορούν την ύπαρξη του feature $X_i$ με δεδομένη την κάθε κατηγορία υπολογίζονται κρατώντας από το feature matrix X μόνο τις γραμμές που έχουν τιμή 1 και τιμή 0 αντίστοιχα στο target vector και αθροίζοντας κατακόρυφα τις γραμμές του matrix ώστε να πάρουμε ένα νέο vector μήκους όσο και οι στηλες του X το οποίο θα δείχνει σε κάθε index i τις εμφανίσεις του feature $X_i$ για την κάθε κατηγορία .Τέλος, αν διαιρέσουμε αυτό το vector με το πλήθος των παραδειγμάτων κάθε κατηγορίας θα λάβουμε τις ζητούμενες πιθανότητες. Αντίστοιχα για να βρούμε τις άλλες δύο, δεύτερη και τέταρτη, ουσιαστικά παίρνουμε τις ήδη υπολογισμένες και τις αφαιρούμε από την μονάδα. Σε αυτό το σημείο αξίζει να σημειωθεί ότι για τον υπολογισμό των πιθανοτήτων χρησιμοποιούμε την τεχνική του Laplace smoothing για να αποφύγουμε την ύπαρξη μηδενικών πιθανοτήτων και κρατάμε επίσης τους λογαρίθμους των πιθανοτήτων για μεγαλύτερη σταθερότητα στις πράξεις (αποφυγή πολλαπλασιασμών με πολύ μικρούς αριθμούς).

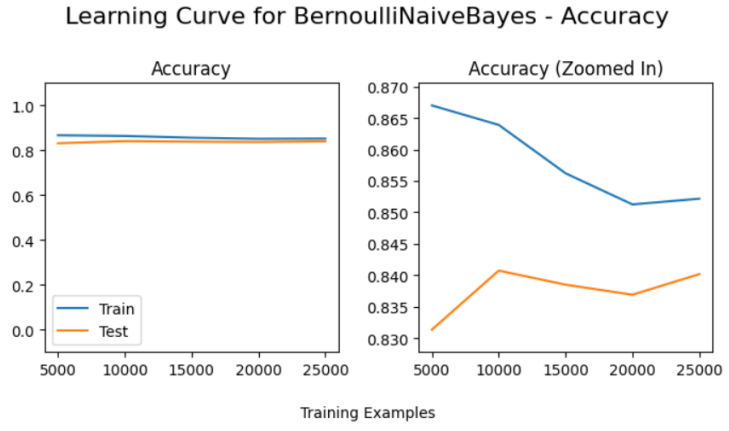
c) `def predict(self, X):` δέχεται σαν όρισμα μόνο ένα feature matrix X και με βάση αυτό και τις πιθανότητες που έχει ήδη υπολογίσει προσπαθεί να κατη-

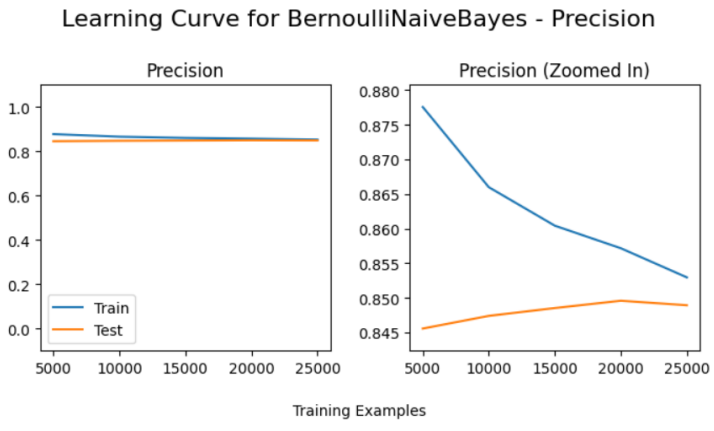
γοριοποιήσει την κάθε γραμμή του X, δηλαδή την κάθε κριτική, σε μία από τις δύο κατηγορίες. Για να το πετύχουμε αυτό υπολογίζουμε την πιθανότητα κάθε κριτικής να ανήκει στην θετική και στην αρνητική κατηγορία με δεδομένα τα features τους. Το πρώτο γίνεται αθροίζοντας το γινόμενο του matrix X με το πρώτο διάνυσμα πιθανοτήτων που υπολογίσαμε παραπάνω (περίπτωση των θέσεων του X που έχουν τιμή 1) και το γινόμενο του matrix 1-X, ο οποίος είναι ο ίδιος με τον X με την διαφορά ότι εκεί που ο X έχει τιμή 1 ο 1-X έχει 0 και το αντίστροφο, με το δεύτερο διάνυσμα πιθανοτήτων που υπολογίσαμε παραπάνω (περίπτωση των θέσεων του X που έχουν τιμή 0). Σε αυτό το σημείο έχουμε ένα διάνυσμα όσο και οι γραμμές του X και του προσθέτουμε σε κάθε γραμμή την prior πιθανότητα της θετικής κατηγορίας ώστε τώρα η κάθε γραμμή να λέει την πιθανότητα η κριτική της γραμμής να ανήκει στην θετική κατηγορία. Κάνουμε την ίδια δουλειά και για την αρνητική κατηγορία και επιστρέφουμε ένα διάνυσμα όπου η κάθε γραμμή του έχει τιμή 1 αν η πιθανότητα της θετικής κατηγορίας είναι μεγαλύτερη από την αρνητική για την συγκεκριμένη κριτική και 0 στην αντίθετη περίπτωση.

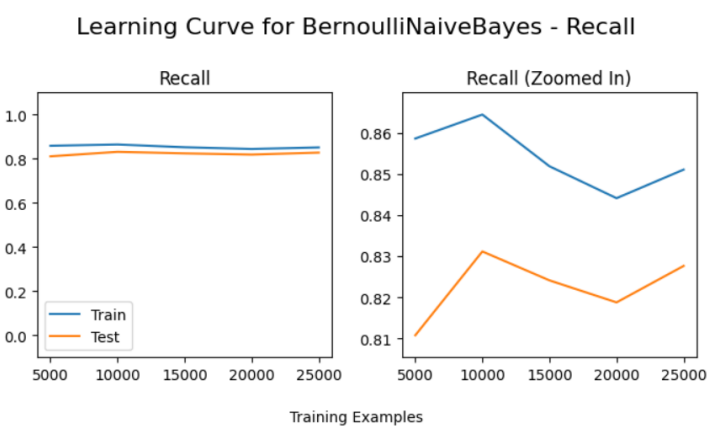
## 2. Αποτελέσματα (Metrics Table, Confusion Matrix, Learning Curves)

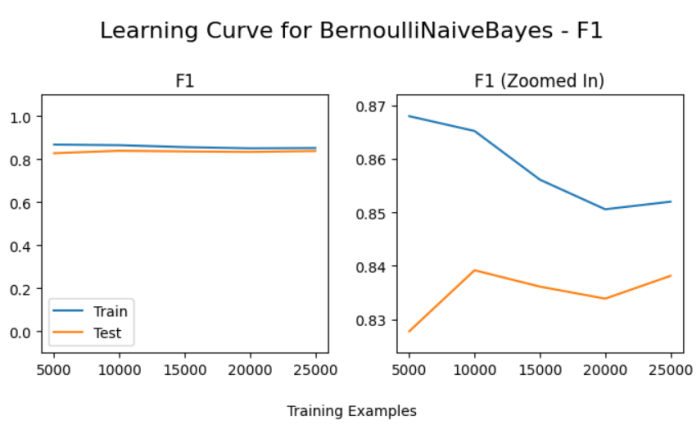
Αφού κάνουμε train τον αλγόριθμο μας σε 5 splits μέσω της βοηθητικής μεθόδου evaluate estimator έχουμε τα ακόλουθα αποτελέσματα, τα οποία φαίνονται αρκετά ικανοποιητικά με υψηλό accuracy και f1 score στα test δεδομένα, αρκετά καλή ισορροπία μεταξύ του recall και του precision και χωρίς ενδείξεις για overfitting στα train δεδομένα.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | 0.87 | 0.83 | 0.88 | 0.85 | 0.86 | 0.81 | 0.87 | 0.83 |
| 10000 | 0.86 | 0.84 | 0.87 | 0.85 | 0.86 | 0.83 | 0.87 | 0.84 |
| 15000 | 0.86 | 0.84 | 0.86 | 0.85 | 0.85 | 0.82 | 0.86 | 0.84 |
| 20000 | 0.85 | 0.84 | 0.86 | 0.85 | 0.84 | 0.82 | 0.85 | 0.83 |
| 25000 | 0.85 | 0.84 | 0.85 | 0.85 | 0.85 | 0.83 | 0.85 | 0.84 |





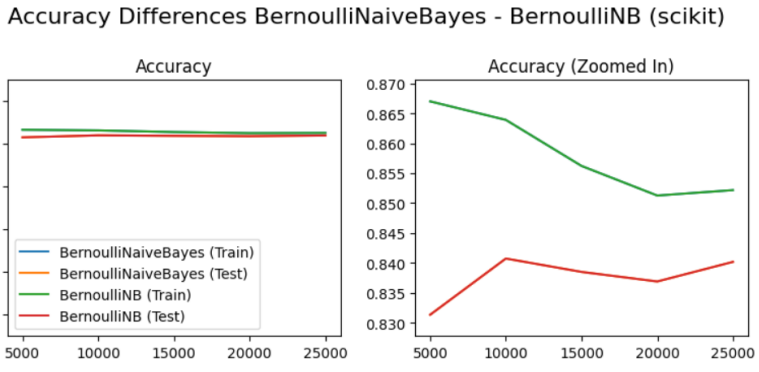

Το confusion matrix μας έδωσε τις ακόλουθες τιμές : TP = 10.345, TN = 10.659, FP = 1.841 και FN = 2.155 οι οποίες είναι αρκετά καλές με υψηλό ποσοστό σωστών και χαμηλό λανθασμένων κατηγοριοποιήσεων.

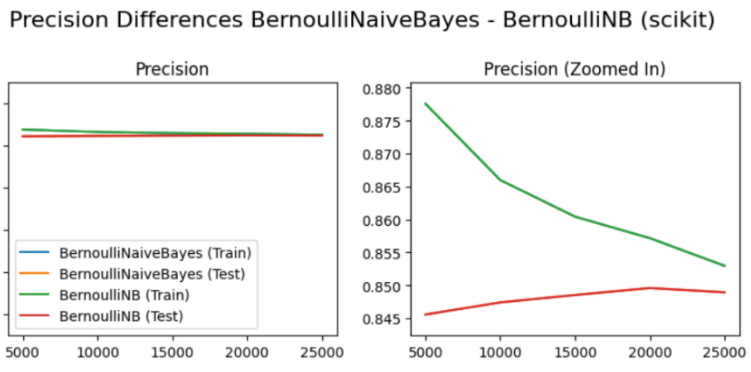
## 3. Συγκρίσεις με Scikit-Learn

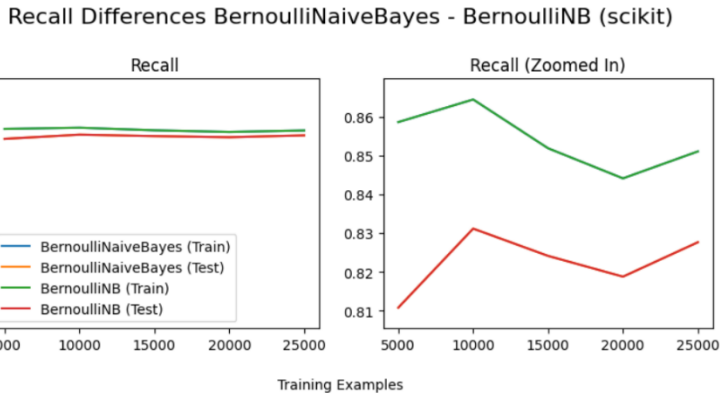
### Bernoulli Naive Bayes VS Bernoulli Naive Bayes (scikit)

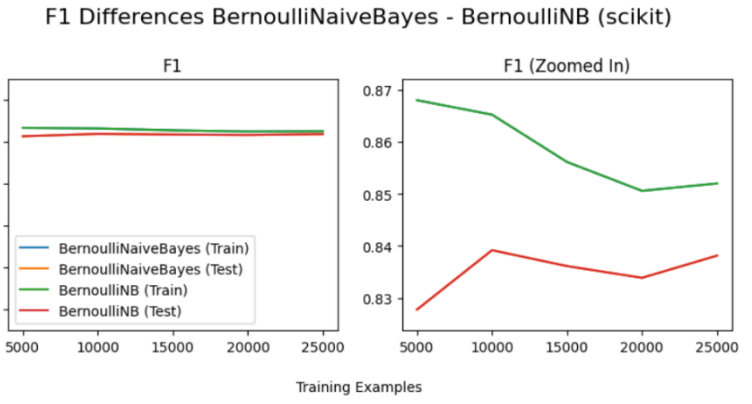
Μπορούμε να παρατηρήσουμε από τις παρακάτω εικόνες ότι τα αποτελεσματα της υλοποίησης μας ταυτίζονται πλήρως με αυτά της έτοιμης υλοποίησης του scikit-learn αφού ο πίνακας διαφοράς των μετρικών έχει την τιμή 0 σε όλα τα πεδία και οι καμπύλες μάθησης για όλες τις μετρικές βρίσκονται η μία πάνω στην άλλη.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 10000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 15000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 20000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 25000 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |



### Bernoulli Naive Bayes VS Logistic Regression (scikit)

Ο αλγόριθμος μας τα πηγαίνει εξίσου καλά και ενάντια στον έτοιμο αλγόριθμο Logistic Regression με στοχαστική ανάβαση κλίσης του scikit-learn, καθώς όπως φαίνεται από τον παρακάτω πίνακα διαφοράς μετρικών πετυχαίνει σχεδόν εξίσου καλό(+/- 0.1) accuracy, precision και f1 score με πολύ μικρότερες τιμές μετρικών στα δεδομένα εκπαίδευσης. Σημαντική διαφορά παρατηρείται στην τιμή της μετρικής recall στα test δεδομένα. (Οι καμπύλες μάθησης παραλείπονται για εξοικονόμηση χώρου)

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | -0.13 | 0.03 | -0.12 | 0.04 | -0.14 | 0.01 | -0.13 | 0.03 |
| 10000 | -0.09 | 0.01 | -0.09 | 0.01 | -0.08 | 0.01 | -0.08 | 0.01 |
| 15000 | -0.06 | 0.00 | -0.05 | 0.02 | -0.09 | -0.04 | -0.06 | -0.01 |
| 20000 | -0.06 | -0.01 | -0.03 | 0.02 | -0.09 | -0.05 | -0.06 | -0.02 |
| 25000 | -0.05 | -0.01 | -0.04 | 0.01 | -0.07 | -0.05 | -0.05 | -0.02 |

### Bernoulli Naive Bayes VS Random Forest (scikit)

Τέλος απέναντι στον Random Forest, χωρίς fine tuned τις υπερπαραμέτρους του, καταφέρνει να πετύχει καλύτερο accuracy, precision και f1 score στα test δεδομένα αλλά συνεχίζει να μειονεκτεί στο recall. (Οι καμπύλες μάθησης παραλείπονται για εξοικονόμηση χώρου)

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | 0.00 | 0.04 | 0.07 | 0.11 | -0.11 | -0.09 | -0.01 | 0.02 |
| 10000 | 0.01 | 0.04 | 0.07 | 0.10 | -0.08 | -0.05 | 0.00 | 0.03 |
| 15000 | 0.02 | 0.05 | 0.07 | 0.10 | -0.09 | -0.07 | 0.00 | 0.03 |
| 20000 | 0.01 | 0.04 | 0.06 | 0.09 | -0.08 | -0.06 | -0.01 | 0.01 |
| 25000 | 0.01 | 0.04 | 0.06 | 0.09 | -0.07 | -0.05 | 0.00 | 0.02 |

## Logistic Regression

### 1. Υλοποίηση

Για τον Logistic Regression με Stohastic Gradient Descent έχουμε την κλάση `class LogisticRegressionSGD:` με τις εξής μεθόδους :

a) `def __init__(self, learning_rate=0.01, num_epochs=1000, lambda_value=0.001, threshold=0.5, tol=1e-4, patience=5):` παίζει τον ρόλο του constructor της κλάσης αρχικοποιώντας τις (υπέρ) παραμέτρους του μοντέλου μας.

b) `def sigmoid(self, x):` επιστρέφει το αποτέλεσμα της σιγμοειδούς συνάρτησης πάνω στο όρισμα x.

c) `def initialize_parameters(self, n_features):` αρχικοποιεί τον πίνακα των βαρών με μηδενικά μαζί και το bias term.

d) `def forward(self, X):` υπολογίζει και επιστρέφει το αποτέλεσμα της σιγμοειδούς συνάρτησης του γραμμικού συνδυασμού του ορίσματος x με τον πίνακα των βαρών.

e) `def stochastic_gradient_descent(self, X, y):` υλοποιεί την στοχαστική κατάβαση κλίσης κατά την οποία για κάθε γραμμή του πίνακα X υπολογίζει την πιθανότητα να ανήκει στην θετική κατηγορίας με την forward και ενημερώνει τα βάρη σύμφωνα με τον $\vec{w}_{t+1} \leftarrow \vec{w}_t - \alpha \nabla L(\vec{w}_t)$ όπου η παράγωγος της συνάρτησης του loss (dw) έχει την μορφή :

$$dw = \frac{1}{m} \sum_{i=1}^{m} (y_{pred}^{(i)} - y^{(i)}) x^{(i)T} + 2\lambda w$$

f) `def compute_loss(self, y_true, y_pred):` υπολογίζει και επιστρέφει το συνολικό error το οποίο περιλαμβάνει το cross entropy loss :

$$-\frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1-y_i) \cdot log(1-p(y_i))$$

καθώς και το regularization term το οποίο χρησιμοποιείται για να τιμωρεί τα μεγάλα βάρη και βοηθάει στην αποφυγή του overfitting.

g) `def fit(self, X, y):` είναι η μέθοδος που εκπαιδεύει το μοντέλο μας. Τρέχει εποχές μέχρι να φτάσει τον μέγιστο αριθμών εποχών (num_epochs) ή μέχρι να διαπιστωθεί ότι το μοντέλο δεν έχει βελτίωση όσο εκπαιδεύεται. Πριν τον συγκεκριμένο έλεγχο και επειδή χρησιμοποιούμε στοχαστική κατάβαση κλίσης ανακατεύουμε τα παραδείγματα με την shuffle ώστε να υπάρχει τυχαίοτητα και καλούμε την μέθοδο stohastic_gradient_descent που περιγράψαμε παραπάνω. Αφού αυτή ολοκληρωθεί υπολογίζουμε το loss της συγκεκριμένης εποχής και με βάση αυτό κάνουμε τον έλεγχο για την σύγκλιση. Αν patience (= 5) φορές βρούμε loss μεγαλύτερο από το ελάχιστο loss κατά μία μικρή τιμή tol (= 1e-4) σταματάμε. Οι τιμές patience και tol είναι υπερπαράμετροι και οι τιμές έχουν προκύψει έπειτα από δοκιμές.

h) `def predict(self, X):` Για κάθε παράδειγμα εκπαίδευσης υπολογίζει την πιθανότητα να ανήκει στην θετική κατηγορία και αν αυτή είναι μεγαλύτερη από την τιμή του threshold, η οποία επίσης είναι υπερπαράμετρος, του δίνει ετικέτα, δηλαδή προβλέπει ‘1’ αλλιώς ‘0’.

## 2. Αποτελέσματα (Metrics Table, Confusion Matric, Learning Curves)

Όμοια με πριν εκπαιδεύσαμε τον αλγόριθμο σε 5 splits και χρησιμοποιήσαμε επίσης το 20% των training δεδομένων για το validation set. Το μοντέλο φαίνεται να τα πηγαινεί αρκετά καλά συμφωνα με τις υψηλές τιμές στα δεδομένα ελέγχου ενώ δεν υπάρχουν ενδείξεις για overfitting. Επιπλέον, για τον ταξινομητή λογιστικής παλινδρόμησης έχουμε κάνει fine-tuning σε δύο από τις υπερπαραμέτους του (lambda, threshold) ξεχωριστά επειδή η τιμή της μίας δεν επηρεάζει την άλλη. Και για τις δύο έχουμε επιλέξει ορισμένες συνηθισμένες τιμές και βλέπουμε ποια από αυτές για το 'λ' μας δίνει καλύτερο accuracy και για το threshold καλύτερο f1 score. Καταλήξαμε στις λ = 0.001 και threshold = 0.465. Αξίζει να σημειωθεί ότι λόγω της στοχαστικότητας του μοντέλου οι προβλέψεις αλλάζουν σε κάθε εκτέλεση και εμείς κρατήσαμε μια από τις πιο ικανοποιητικές.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 4000 | 0.98 | 0.84 | 0.98 | 0.84 | 0.98 | 0.84 | 0.98 | 0.84 |
| 8000 | 0.95 | 0.85 | 0.96 | 0.86 | 0.95 | 0.85 | 0.95 | 0.85 |
| 12000 | 0.92 | 0.86 | 0.89 | 0.83 | 0.96 | 0.91 | 0.93 | 0.87 |
| 16000 | 0.92 | 0.87 | 0.93 | 0.88 | 0.91 | 0.85 | 0.92 | 0.87 |
| 20000 | 0.91 | 0.87 | 0.91 | 0.87 | 0.91 | 0.87 | 0.91 | 0.87 |

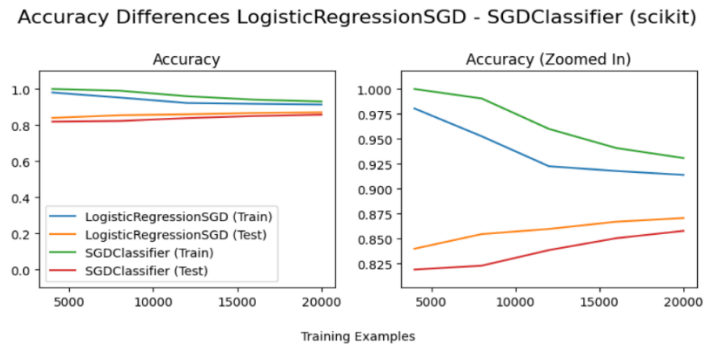
Το confusion matrix μας έδωσε τις ακόλουθες τιμές : TP = 10.872, TN = 10.896, FP = 1.604 και FN = 1.628 οι οποίες είναι επίσης αρκετά καλές με υψηλό ποσοστό σωστών χαμηλό λανθασμένων κατηγοριοποιήσεων.
(Οι καμπύλες μάθησης φαίνονται παρακάτω συγκριτικά με την έτοιμη υλοποίηση του scikit-learn.)

## 3. Συγκρίσεις με Scikit-Learn

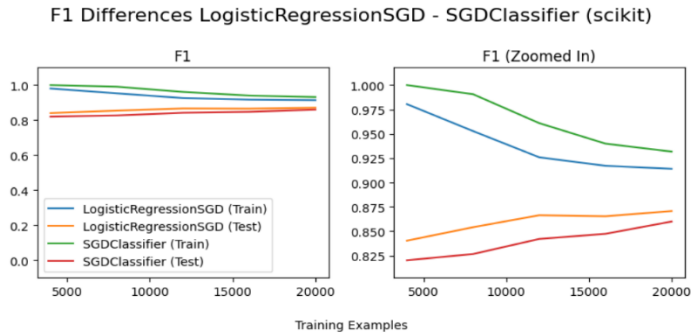
### Logistic Regression VS Logistic Regression (scikit)

Το μοντέλο μας τα πηγαίνει πολύ καλά απέναντι στην έτοιμη υλοποίηση του ίδιου αλγορίθμου του scikit-learn, καθώς όπως φαίνεται από τον πίνακα διαφορών και από τις καμπύλες μάθησης πετυχαίνει μικρότερες τιμές μετρικών στα training δεδομένα και μεγαλύτερες στα testing.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 4000 | -0.02 | 0.02 | -0.02 | 0.02 | -0.02 | 0.02 | -0.02 | 0.02 |
| 8000 | -0.04 | 0.03 | -0.03 | 0.05 | -0.05 | 0.01 | -0.04 | 0.02 |
| 12000 | -0.04 | 0.02 | -0.06 | 0.01 | -0.02 | 0.05 | -0.03 | 0.03 |
| 16000 | -0.02 | 0.02 | -0.03 | 0.01 | -0.01 | 0.02 | -0.02 | 0.02 |
| 20000 | -0.02 | 0.01 | -0.01 | 0.02 | -0.03 | 0.00 | -0.02 | 0.01 |






## Logistic Regression VS Bernoulli Naive Bayes (scikit)

Όπως φαίνεται το μοντέλο μας ξεπερνά τον Bernoulli Naive Bayes σε όλες τις μετρικές, από τις οποίες οι υψηλότερες αλλά φυσιολογικές χωρίς overfitting τιμές στα training δεδομένα μας δείχνουν ότι το μοντέλο μάλλον είναι πιο πιθανό να γενικευτεί καλύτερα, ενώ οι υψηλότερες μετρικές στα testing δεδομένα μας αποδεικνύουν ότι όντως τα πηγαίνει καλύτερα σε νέα δεδομένα.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 4000 | 0.12 | 0.03 | 0.14 | 0.05 | 0.09 | 0.00 | 0.12 | 0.02 |
| 8000 | 0.09 | 0.02 | 0.11 | 0.04 | 0.07 | 0.00 | 0.09 | 0.02 |
| 12000 | 0.07 | 0.03 | 0.05 | 0.01 | 0.09 | 0.06 | 0.07 | 0.03 |
| 16000 | 0.07 | 0.03 | 0.08 | 0.05 | 0.04 | 0.00 | 0.06 | 0.03 |
| 20000 | 0.06 | 0.03 | 0.06 | 0.03 | 0.05 | 0.03 | 0.05 | 0.03 |

## Logistic Regression VS Random Forest (scikit)

Παρατηρούμε και εδώ ότι το μοντέλο μας προσπερνάει στις περισσότερες μετρικές την έτοιμη υλοποίηση του scikit-learn για τον Random Forest, ωστόσο όπως προαναφέραμε μπορεί να οφείλεται στο ότι για τον Random Forest δεν χρησιμοποιούμε τις fine tuned υπερπαραμέτρους του.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 4000 | 0.08 | 0.03 | 0.13 | 0.05 | 0.01 | -0.01 | 0.07 | 0.02 |
| 8000 | 0.07 | 0.03 | 0.12 | 0.08 | 0.00 | -0.02 | 0.06 | 0.02 |
| 12000 | 0.05 | 0.05 | 0.07 | 0.05 | 0.03 | 0.04 | 0.06 | 0.05 |
| 16000 | 0.06 | 0.06 | 0.11 | 0.10 | -0.02 | -0.02 | 0.05 | 0.05 |
| 20000 | 0.06 | 0.06 | 0.10 | 0.09 | -0.01 | 0.00 | 0.05 | 0.05 |

# Random Forest

## 1. Υλοποίηση

Για την υλοποίηση του Random Forest έχουμε την κλάση `class RandomForest:` η οποία έχει τις ακόλουθες μεθόδους:

a) `def __init__(self, n_trees=100, min_samples_split=2, min_samples_leaf=1, max_depth=20, max_features=10):` Ο κατασκευαστής της κλάσης που αρχικοποιεί τις βασικές παραμέτρους του ταξινομητή, όπως τον αριθμό των δέντρων, τον ελάχιστο αριθμό δειγμάτων που απαιτούνται για διακλάδωση, τον ελάχιστο αριθμό δειγμάτων για να θεωρηθεί ένας κόμβος φύλλο, το μέγιστο βάθος κάθε δέντρου, και το μέγιστο αριθμό ιδιοτήτων που χρησιμοποιούνται κατά την εκπαίδευση κάθε δέντρου.

b) `def train_tree(self, X, y):` εκπαιδεύει ένα δέντρο απόφασης σε ένα δείγμα bootstrap των δεδομένων εκπαίδευσης. Χρησιμοποιεί εσωτερικά την συνάρτηση bootstrap_sample.

c) `def bootstrap_sample(X, y):` δέχεται δύο παραμέτρους, X τα δεδομένα εκπαίδευσης και y οι ετικέτες τους και επιστρέφει ένα bootstrap δείγμα αυτών, δηλαδή επιλέγει τυχαία δείγματα από το σύνολο των δεδομένων εκπαίδευσης με αντικατάσταση και τα επιστρέφει μαζί με τις αντίστοιχες ετικέτες τους. Η συγκεκριμένη συνάρτηση είναι global συνάρτηση, δηλαδή δεν ανήκει αποκλειστικά στην κλάση Random Forest. Φτιάχνουμε το κάθε δέντρο ως εξής
```
tree = DecisionTreeClassifier(criterion='entropy',
tree.fit(X_sample, y_sample)
```
χρησιμοποιώντας την κλάση του scikit-learn DecisionTreeClassifier(criterion=’entropy’, …) για να δηλώσουμε ότι θέλουμε ο αλγόριθμος που εφαρμόζεται να κοιτάει το Information Gain κάθε κόμβου, δηλαδή να ασχολείται με την εντροπία, όπως κάνει ο αλγόριθμος ID3. Έπειτα εκπαιδεύουμε το συγκεκριμένο δέντρο στο bootstrapped δείγμα των δεδομένων εκπαίδευσής μας.

d) `def fit(self, X, y):` εκπαιδεύουμε το μοντέλο μας δημιουργώντας παράλληλα τα δέντρα απόφασης με βάση το σύνολο των δεδομένων εκπαίδευσης. Καλεί εσωτερικά την train_tree.

e) `def predict(self, X):` προβλέπει παράλληλα τις κλάσεις (ετικέτες) για κάθε δείγμα στο σύνολο X χρησιμοποιώντας την πλειοψηφία των ψήφων από όλα τα δέντρα. Καλεί εσωτερικά την predict του sklearn και την break_ties.

f) `def break_ties(self, votes):` επιλύει τις ισοπαλίες που ενδέχεται να προκύψουν κατά την ψηφοφορία των δέντρων, επιλέγοντας τυχαία μεταξύ των

κορυφαίων υποψηφίων.

g) `def predict_proba(self, X):` προβλέπει τις πιθανότητες κλάσεων για κάθε δείγμα στο σύνολο X. Καλεί εσωτερικά την predict_proba του scikit-learn και χρησιμοποιείται έμμεσα μόνο για την δημιουργία της καμπύλης ROC.

h) `def get_params(self, deep=True):` και `def set_params(self, **parameters):` χρησιμοποιούνται την ανάκτηση και την ρύθμιση των παραμέτρων του ταξινομητή

## 2. Αποτελέσματα

Για την εύρεση των υπερπαραμέτρων του Random Forest χρησιμοποιούμε την συνάρτηση `randomized_search(` ,η οποία δέχεται ένα λεξικό με πιθανές τιμες για κάθε υπερπαράμετρο, το πλήθος των συνδυασμών που θα δοκιμασει, τι είδους επικύρωση θα κάνει (πόσα folds θα χρησιμοποιήσει) και ένα string που δείχνει αν η συγκρεκριμένη κλήση της συνάρτησης αφορά τη δική μας υλοποίηση του Random Forest ή αυτή του scikit-learn. Με βάση τις τιμές των ορισμάτων εκπαιδεύει και επικυρώνει τόσους συνδυασμούς υπερπαραμέτρων για το συγκεκριμένο μοντέλο και στο τέλος επιστρέφει και εκτυπώνει το καλύτερο μέσο accuracy score από το cross-validation σετ δεδομένων και τους υπερπαραμέτρους με τους οποίους το πέτυχε. Μετά από πολλές κλήσεις και δοκιμές καταλήγουμε στις εξής τιμές για τις υπερπαραμέτρους : max_features = 20, min_samples_leaf = 10, min_samples_split = 20, max_depth = 70, n_trees = 350

Για την αξιολόγηση όμοια με πριν εκπαιδεύσαμε τον αλγόριθμο σε 5 splits, αλλά για κάθε split καλέσαμε την μέθοδο predict για τα test δεδομένα, καθώς όλη η διαδικασία επικύρωσης έχει γίνει στην randomized_search. Το μοντέλο φαίνεται να τα πηγαίνει αρκετά καλά σύμφωνα με τις σχετικά υψηλές τιμές στα δεδομένα ελέγχου ενώ επίσης δεν υπάρχουν ισχυρές ενδείξεις για overfitting.

| | Accuracy Train | Accuracy Test | Error Train | Error Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|---|---|
| 4000 | 0.9452 | 0.8434 | 0.0547 | 0.1566 | 0.9411 | 0.8434 | 0.9482 | 0.8434 | 0.9447 | 0.8434 |
| 8000 | 0.9372 | 0.8467 | 0.0627 | 0.1533 | 0.9278 | 0.8492 | 0.9466 | 0.8449 | 0.9371 | 0.8471 |
| 12000 | 0.9344 | 0.8504 | 0.0656 | 0.1496 | 0.9232 | 0.8596 | 0.9471 | 0.8441 | 0.9350 | 0.8518 |
| 16000 | 0.9338 | 0.8510 | 0.0663 | 0.1490 | 0.9240 | 0.8582 | 0.9442 | 0.8460 | 0.9340 | 0.8521 |
| 20000 | 0.9316 | 0.8513 | 0.0684 | 0.1487 | 0.9228 | 0.8584 | 0.9412 | 0.8463 | 0.9319 | 0.8523 |

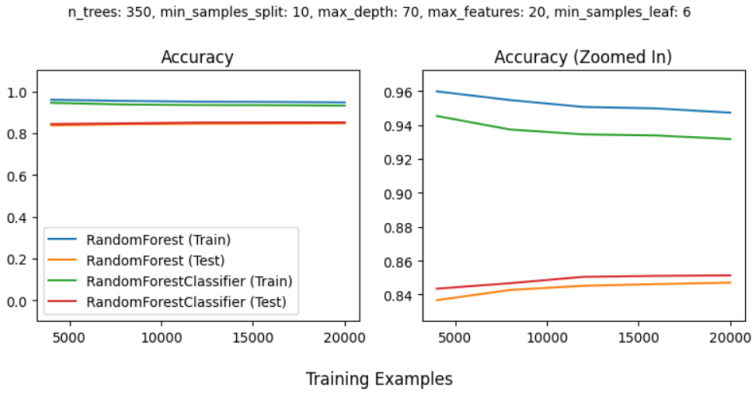
Το confusion matrix μας έδωσε τις ακόλουθες τιμές : TP = 10.730, TN = 10.552, FP = 1.948 και FN = 1.770 οι οποίες είναι επίσης αρκετά καλές με υψηλό ποσοστό σωστών και χαμηλό λανθασμένων κατηγοριοποιήσεων.

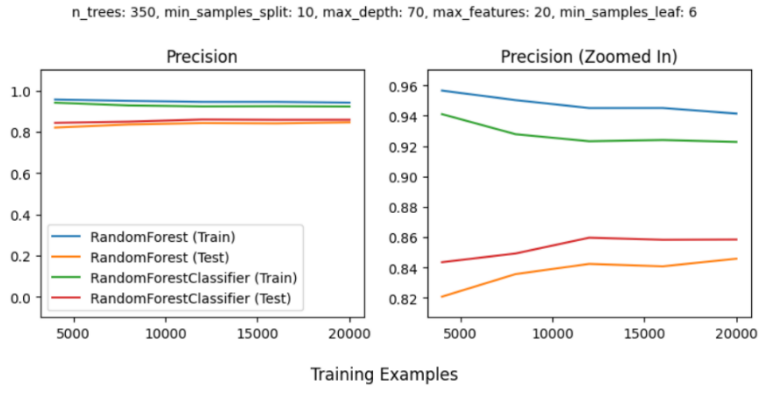
## 3. Συγκρίσεις με Scikit-Learn

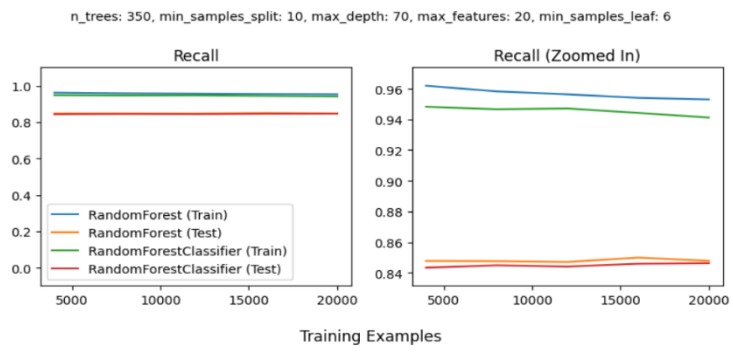
### Random Forest VS Random Forest (scikit)

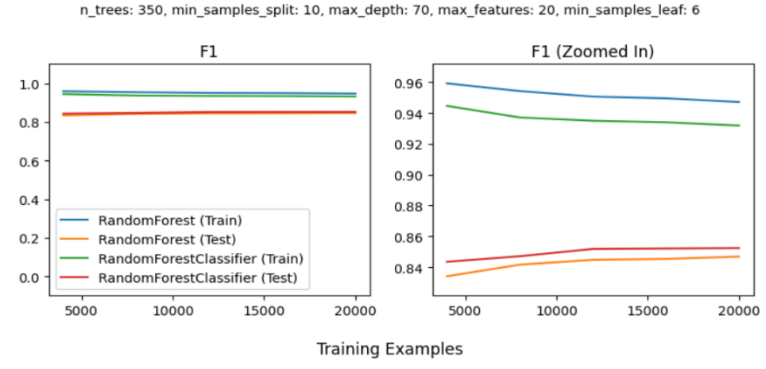
Όπως φαίνεται από τις παρακάτω καμπύλες μάθησης και τον πίνακα σύγκρισης των μετρικών το μοντέλο μας τα πάει σχεδόν εξίσου καλά με την έτοιμη υλοποίηση του scikit-learn για τον Random Forest. Έχουμε πετύχει τις ίδιες τιμές σε όλες τις μετρικές (πλην του precision) στα test δεδομένα, έχοντας μια πολύ μικρή αύξηση στα train δεδομένα της τάξης του $10^{-2}$.

| | Accuracy Train | Accuracy Test | Error Train | Error Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|---|---|
| 4000 | 0.01 | 0.00 | -0.01 | 0.00 | 0.02 | -0.02 | 0.01 | 0.01 | 0.02 | -0.01 |
| 8000 | 0.01 | -0.01 | -0.01 | 0.01 | 0.02 | -0.01 | 0.01 | 0.01 | 0.01 | -0.01 |
| 12000 | 0.02 | 0.00 | -0.02 | 0.00 | 0.03 | -0.02 | 0.01 | 0.01 | 0.01 | -0.01 |
| 16000 | 0.02 | 0.00 | -0.02 | 0.00 | 0.03 | -0.02 | 0.01 | 0.00 | 0.02 | 0.00 |
| 20000 | 0.02 | 0.00 | -0.02 | 0.00 | 0.02 | -0.01 | 0.01 | 0.00 | 0.02 | 0.00 |




### Random Forest VS Bernoulli Naive Bayes (scikit)

Μπορούμε να παρατηρήσουμε ότι το μοντέλο μας τα πηγαίνει πολύ λίγο καλύτερα στα test δεδομένα από την έτοιμη υλοποίηση του Bernoulli Naive Bayes του

scikit-learn, όμως έχει αρκετά υψηλότερες τιμές μετρικών στα train δεδομένα.

| | Accuracy Train | Accuracy Test | Error Train | Error Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|---|---|
| 4000 | 0.08 | 0.01 | -0.08 | -0.01 | 0.07 | 0.01 | 0.09 | 0.00 | 0.08 | 0.00 |
| 8000 | 0.09 | 0.00 | -0.09 | 0.00 | 0.09 | 0.01 | 0.10 | 0.00 | 0.09 | 0.00 |
| 12000 | 0.09 | 0.01 | -0.09 | -0.01 | 0.09 | 0.01 | 0.10 | 0.00 | 0.09 | 0.00 |
| 16000 | 0.09 | 0.01 | -0.09 | -0.01 | 0.10 | 0.00 | 0.09 | 0.01 | 0.09 | 0.01 |
| 20000 | 0.09 | 0.01 | -0.09 | -0.01 | 0.09 | 0.01 | 0.09 | 0.01 | 0.09 | 0.01 |

Random Forest VS Logistic Regression (scikit)

Σύμφωνα με τον παρακάτω πίνακα σύγκρισης μετρικών είναι προφανές ότι η έτοιμη υλοποίηση του Logistic Regression της scikit-learn τα πηγαίνει πολύ καλύτερα από το μοντέλο του Random Forest πετυχαίνοντας μεγαλύτερες μετρικές στα test δεδομένα και μικρότερες στα training.

| | Accuracy Train | Accuracy Test | Error Train | Error Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|---|---|
| 4000 | -0.04 | 0.02 | 0.04 | -0.02 | -0.04 | 0.00 | -0.04 | 0.03 | -0.04 | 0.01 |
| 8000 | -0.04 | 0.01 | 0.04 | -0.01 | -0.03 | -0.01 | -0.04 | 0.03 | -0.04 | 0.01 |
| 12000 | -0.01 | 0.01 | 0.01 | -0.01 | -0.01 | -0.01 | -0.01 | 0.02 | -0.01 | 0.00 |
| 16000 | 0.01 | 0.00 | -0.01 | 0.00 | 0.00 | 0.00 | 0.02 | -0.01 | 0.01 | 0.00 |
| 20000 | 0.02 | -0.01 | -0.02 | 0.01 | 0.02 | -0.03 | 0.00 | 0.00 | 0.02 | -0.01 |

## biGRU-RNN

Η προεπεξεργασία των δεδομένων για το biGRU-RNN διαφέρει ελαφρά από πριν καθώς τώρα παριστάνουμε τις λέξεις με word embeddings. Για να το πετύχουμε αυτό χρησιμοποιούμε τον Text Vectorizer στον οποίο περνώντας σαν όρισμα το συνολικό μήκος του λεξιλογίου και το μήκος της κάθε ακολουθίας (το βρίσκουμε με την χρήση ευρετικής) που επιθυμούμε , συνδέουμε κάθε λέξη με έναν αριθμό φτιάχνοντας έτσι τις ενθέσεις λέξεων.

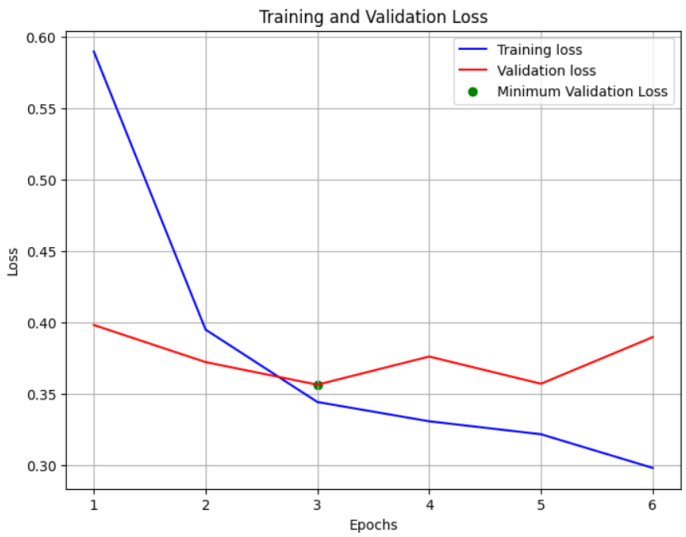
### 1. Υλοποίηση

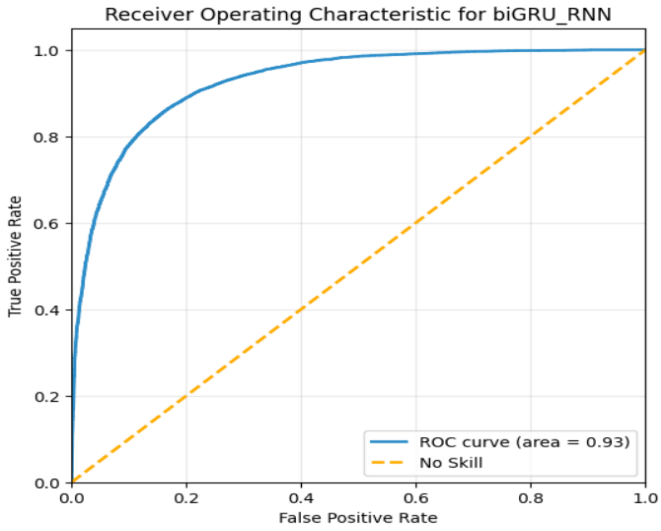
Για την υλοποίηση του biGRU-RNN έχουμε την κλάση `class bigru_rnn():` η οποία δημιουργεί ένα RNN με ένα επίπεδο εισόδου το οποίο δέχεται μία μόνο κριτική σε μορφή συμβολοσειράς. Στην συνέχεια η χρήση του vectorizer μετατρέπει το κείμενο εισόδου σε ακέραιες τιμές και έπειτα υπάρχει το Embedding layer το οποίο μετατρέπει τις ακέραιες τιμές σε πυκνές διανυσματικές αναπαραστάσεις με διαστάσεις που καθορίζονται από το emb_size (=64). Μετά έχουμε τα Bidirectional layers τα οποία μας βοηθούν να επεξεργαζόμαστε τα δεδομένα και από τις δύο κατευθύνσεις. Χρησιμοποιούμε επίσης dropout layer (με πιθανότητα 0.5) για την αποφυγή του overfitting και στο τέλος έχουμε το επίπεδο εξόδου με 1 unit και activation function sigmoid

ώστε να μας βγάλει την πιθανότητα της θετικής κατηγορίας.

## 2. Αποτελέσματα (Loss over epochs, ROC, Metrics Table)

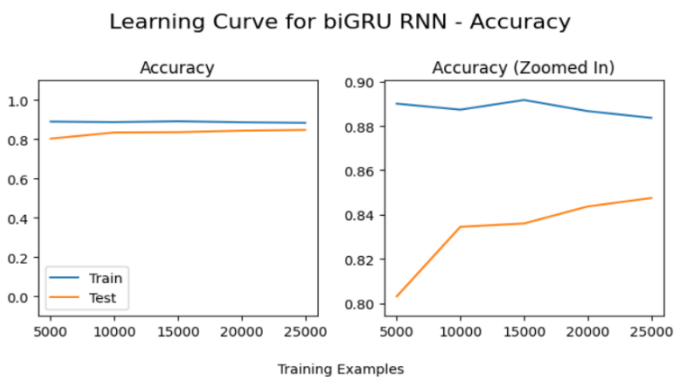
Εκπαιδεύουμε το RNN μας σε 6 εποχές, με batch_size 64 και χρησιμοποιούμε επίσης το 20% των training δεδομένων ως τα validation δεδομένα και προκύπτουν οι ακόλουθες καμπύλες εκ των οποίων η πρώτη είναι η μεταβολή του σφάλματος στα παραδείγματα εκπαίδευσης και επαλήθευσης καθώς προχωράνε τα epochs και η δεύτερη η ROC Curve.

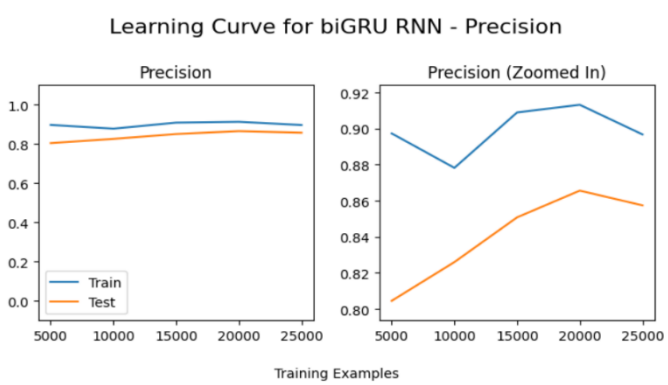


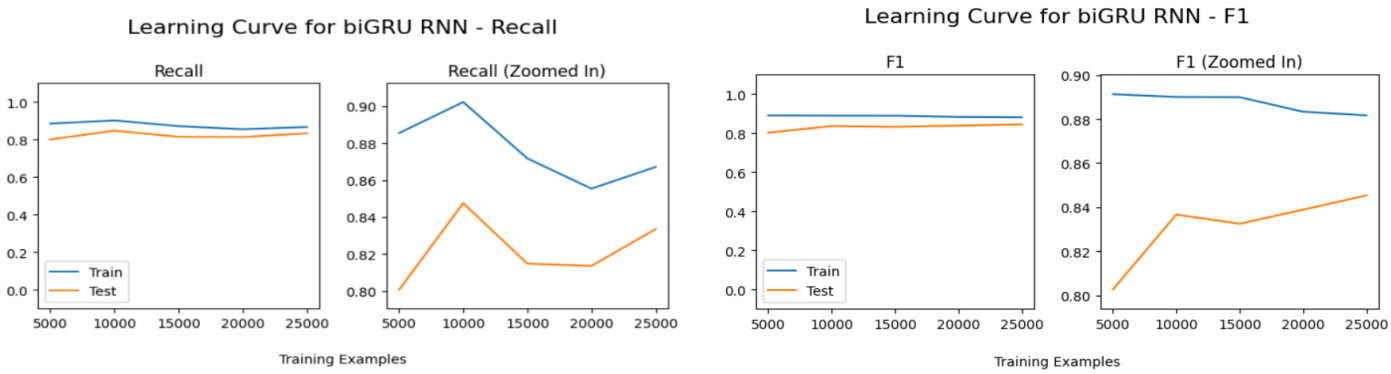




Επίσης για να δούμε πόσο καλά τα πάει το μοντέλο μας και σε νέα δεδομένα το τρέχουμε σε 5 splits εκπαιδεύοντας τον με τις ιδιες υπερπαραμέτρους με πριν και το αξιολογούμε στα testing δεδομένα στα οποία πετυχαίνει τις ακόλουθες μετρικές και καμπύλες μάθησης. Παρατηρούμε ότι τα πηγαίνει αρκετά καλά με υψηλές μετρικές στα δεδομένα αξιολόγησης και μικρή διαφορά μεταξύ των train και test μετρικών.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | 0.89 | 0.80 | 0.90 | 0.80 | 0.89 | 0.80 | 0.89 | 0.80 |
| 10000 | 0.89 | 0.83 | 0.88 | 0.83 | 0.90 | 0.85 | 0.89 | 0.84 |
| 15000 | 0.89 | 0.84 | 0.91 | 0.85 | 0.87 | 0.81 | 0.89 | 0.83 |
| 20000 | 0.89 | 0.84 | 0.91 | 0.87 | 0.86 | 0.81 | 0.88 | 0.84 |
| 25000 | 0.88 | 0.85 | 0.90 | 0.86 | 0.87 | 0.83 | 0.88 | 0.85 |






## 3. Συγκρίσεις με τις υλοποιήσεις μας

### biGRU-RNN VS Bernoulli Naive Bayes (from scratch)

Μπορούμε εύκολα να παρατηρήσουμε ότι το RNN μας πετυχαίνει μεγαλύτερες τιμές μετρικών και στα train και test δεδομένα (πλην του recall στα test) από την υλοποίηση μας για τον Bernoulli Naive Bayes.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | 0.06 | -0.02 | 0.08 | -0.01 | 0.04 | -0.04 | 0.05 | -0.03 |
| 10000 | 0.07 | 0.01 | 0.07 | 0.03 | 0.04 | 0.01 | 0.06 | 0.02 |
| 15000 | 0.07 | 0.02 | 0.10 | 0.05 | 0.02 | -0.03 | 0.06 | 0.01 |
| 20000 | 0.07 | 0.02 | 0.10 | 0.06 | 0.01 | -0.03 | 0.05 | 0.01 |
| 25000 | 0.06 | 0.04 | 0.10 | 0.06 | 0.02 | -0.01 | 0.06 | 0.03 |

### biGRU-RNN VS Logistic Regression (from scratch)

Σύμφωνα με τον παρακάτω πίνακα τον RNN μας πετυχαίνει καλύτερα αποτελέσματα μόνο στην μετρική precision στα test δεδομένα, ενώ έχει επίσης μεγαλύτερες τιμές στα train δεδομένα (πλην recall) από την υλοποίηση μας του Logistic Regression χωρίς τις παραμέτρους του να είναι fine-tuned.

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | 0.00 | -0.03 | -0.04 | -0.07 | 0.05 | 0.04 | 0.00 | -0.01 |
| 10000 | 0.01 | -0.01 | -0.02 | -0.03 | 0.04 | 0.03 | 0.01 | 0.00 |
| 15000 | 0.02 | -0.01 | 0.02 | -0.02 | 0.03 | 0.00 | 0.02 | -0.01 |
| 20000 | 0.03 | -0.01 | 0.07 | 0.05 | -0.04 | -0.08 | 0.01 | -0.01 |
| 25000 | 0.02 | 0.00 | 0.06 | 0.03 | -0.03 | -0.06 | 0.01 | -0.01 |

### biGRU-RNN VS Random Forest (from scratch)

Το RNN μας τα πηγαίνει πολύ καλύτερα στο accuracy και το precision και χειρότερα στο recall και στο f1 score των test δεδομένων από την υλοποίηση μας του Random Forest χωρίς και αυτού τις παραμέτρους του fine-tuned. Επίσης, έχει αρκετά μικρότερες τιμές στα train δεδομένα (πλην precision).

| | Accuracy Train | Accuracy Test | Precision Train | Precision Test | Recall Train | Recall Test | F1 Train | F1 Test |
|---|---|---|---|---|---|---|---|---|
| 5000 | -0.05 | 0.01 | 0.00 | 0.06 | -0.11 | -0.09 | -0.06 | -0.01 |
| 10000 | -0.03 | 0.04 | 0.01 | 0.09 | -0.09 | -0.04 | -0.04 | 0.03 |
| 15000 | -0.03 | 0.05 | 0.04 | 0.09 | -0.11 | -0.06 | -0.03 | 0.02 |
| 20000 | -0.02 | 0.04 | 0.04 | 0.11 | -0.12 | -0.07 | -0.04 | 0.02 |
| 25000 | -0.02 | 0.05 | 0.04 | 0.10 | -0.10 | -0.04 | -0.03 | 0.04 |