RCP 216 – Ingénierie de la fouille et visualisation de données massives

Projet : Analyse de sentiment (discrimination à deux classes - avis positif ou négatif) de données issues de la base IMDB.

Daniel PONT
5 février 2019

Remerciements

Je tiens ici à exprimer ma vive gratitude à l'équipe pédagogique : MM. Michel Crucianu, Pierre Cubaud, Raphaël Fournier-S'niehotta, Marin Ferecatu pour la qualité de l'enseignement et de l'organisation de l'UE ainsi que pour leur disponibilité via le site moodle et lors des séances de regroupement.

Un grand merci également à toutes les personnes du CNAM qui ont permis de rendre RCP 216 accessible en enseignement à distance.

Je souhaite également témoigner ma reconnaissance à pôle emploi (particulièrement à Mmes Chantal CHALVIDAN et Marie-Noëlle BADOL) pour le financement de ma formation,

« *Data! Data! Data! I can't make bricks without clay!* »

Sherlock Holmes

Sommaire

1. Introduction

L'objectif de ce projet est de réaliser la discrimination à deux classes (avis positif ou négatif), à partir du texte de la revue d'un film, pour les données issues de la base IMDB (accessible sur http://ai.stanford.edu/~amaas/data/sentiment/).

Ces données sont organisées de la manière suivante :

- 75 000 revues d'apprentissage
  - 12 500 avec un avis positif
  - 12 500 avec un avis négatif
  - 50 000 non étiquetées
- 25 0000 revues de test
  - 12 500 avec un avis positif
  - 12 500 avec un avis négatif

Chaque revue est stockée dans un fichier texte relativement court.

Les traitements seront réalisés avec le framework open source Spark 2.3.2 et le code utilisera le langage Scala 2.11.8.

2. Analyse exploratoire – nuages de mots

Afin d'avoir un premier aperçu qualitatif des revues IMDB nous allons construire deux nuages de mots se basant sur les données d'apprentissage. Le premier sera constitué à partir de 12 500 revues positives (répertoire « train/pos »), le second à partir des 12 500 revues négatives (répertoire « train/neg »). Une revue étant stockée dans un fichier individuel, nous commençons par fusionner les fichiers de chaque répertoire. Le code Spark / Scala est le suivant :

```
// répertoire des données initiales (ENTREE) :
val dataDirInit="data/aclImdb/"
// répertoire des données pré-traitées (SORTIE):
val dataDirPreproc="data/input/"

// le prétraitement consiste à merger les fichiers textes de manière à obtenir
// un seul fichier texte par sous-répertoire (train/pos, train/neg)
sc.textFile(dataDirInit+"train/pos").coalesce(1).saveAsTextFile(dataDirPreproc+"train/pos")
sc.textFile(dataDirInit+"train/neg").coalesce(1).saveAsTextFile(dataDirPreproc+"train/neg")
```

En examinant les deux fichiers de synthèse résultant, on constate qu'ils contiennent un tag HTML («
 »). Ce tag n'ayant aucune valeur sémantique on le supprime (par exemple avec la fonction « Remplacer tout... » d'un éditeur de texte).

Pour construire les nuages de mots nous utilisons le freeware wordle. Nous obtenons les images présentées dans les paragraphes suivants.

2.1 Nuage de mots pour les revues positives

Sans surprise, les termes « film » et « movie » sont ceux qui apparaissent le plus fréquemment dans les revues. Nous notons également l'importance des qualificatifs ayant une connotation positive :

- « good »
- « great »

- « well »
- « best »

D'autres termes laudatifs sont également présents bien qu'avec un poids nettement moindre :

- « wonderful »
- « beautiful »
- « perfect »

Nous remarquons enfin la quasi-absence de termes péjoratifs parmi les termes prédominants (seul « bad » apparaît mais avec un poids très faible).

2.2 Nuage de mots pour les revues négatives

Comme pour les revues positives, les termes « movie » et « film » sont les plus employés.

Ce qui est plus étonnant, c'est que contrairement aux revues positives où les qualificatifs positifs sont clairement prépondérants, dans les revues négatives les qualificatifs négatifs n'apparaissent pas de manière aussi évidente.

L'adjectif « good » est même affiché avec un poids similaire à celui de l'adjectif « bad ». Ceci peut s'expliquer de plusieurs façons :

- dans les revues négatives, les qualificatifs positifs peuvent être accompagnés d'une négation (ex. : « this is **not** a good movie »)
- les revues globalement négatives peuvent comporter des avis positifs concernant certains aspects du film (« Despite good performances, the result is boring. The story is not credible and the scenario poorly written »)

Nous en tirons les conclusions suivantes :

- il peut s'avérer pertinent lors du traitement des données textuelles qui va suivre de prendre en compte les négations (« no », « nor », « not »,etc. …) et donc de ne pas les inclure dans la liste des « stop words »
- compte-tenu de l'ambiguïté inhérente aux revues, les classes « avis positifs » et « avis négatifs » ne seront sans doute pas tout à fait linéairement séparables. L'expression de l'erreur sera pondérée par un coefficient dont la valeur sera déterminée par une procédure de validation croisée.

3. Représentation vectorielle du texte des revues

3.1 TF-IDF vs Word2Vec

Afin de pouvoir développer puis appliquer le modèle décisionnel, nous allons construire une représentation vectorielle du texte des revues IMDB. Pour ce faire nous avons considéré deux options :

La première option est de créer une matrice documents-termes avec pondération TF-IDF. Cette approche est naturelle, simple à mettre en œuvre mais présente un certain nombre d'inconvénients :

- Les termes rares ont un poids IDF et donc TF-IDF élevé même s'ils ne sont pas pertinents pour la discrimination en avis positif / avis négatif. Ceci risque de fausser le modèle décisionnel.
- Deux synonymes sont associés à deux dimensions différentes. Ainsi deux revues employant des synonymes pour décrire le même avis seront représentés par des vecteurs dont une comparaison directe ne mettra en évidence aucune similarité.
- Les vecteurs représentant chaque texte ont une taille élevée, la matrice documents-termes est très creuse

Pour répondre aux limitations évoqués ci-dessus, nous avons étudié une autre possibilité - représenter les mots de chaque revue par des vecteurs Word2Vec. Avec cette approche :

- les termes rares non pertinents ne sont pas sur-pondérés
- les synonymes son représentés par des vecteurs similaires
- les vecteurs sont compacts et de dimension peu élevée (typiquement entre 100 et 1000)

Cette solution pose aussi un certain nombre de problèmes. Word2Vec a été développé pour modéliser des mots et non des textes entiers. Afin de représenter une revue IMDB, nous avons choisi une approche très basique : calculer la moyenne (sans pondération) des vecteurs Word2Vec associés à chaque terme du texte de la revue.

D'après Le and Mikolov [1], cette technique "perd l'ordre des mots (de la même façon qu'un modèle bag-of-words standard)" et "ne permet pas de reconnaître des phénomènes linguistiques avancés comme le sarcasme".

En principe, pour représenter des textes, il vaudrait donc mieux utiliser une extension de Word2Vec tel que Doc2Vec (qui malheureusement n'est pas encore implémenté dans Spark).

En pratique cependant, les revues IMDB sont relativement courtes (quelques phrases). Or comme le notent Kenter et al. en 2016 [2], lorsque les textes sont courts "une simple moyenne arithmétique [des vecteurs Word2Vec] de l'ensemble des mots d'un texte s'est révélée être une base solide ou une fonctionnalité importante pour une multitude de tâches".

Pour finir, soulignons que quelle que soit la méthode de représentation choisie (TF-IDF ou Word2Vec) nous produirons des dataframes avec une structure commune - texte de la revue, label (avis positif ou négatif), vecteur – afin de pouvoir leur appliquer le même traitement de classification.

3.2 Représentation vectorielle avec TF-IDF

Pour construire la représentation vectorielle TF-IDF, nous employons la procédure suivante :

1. Les « stops words » sont retirés du texte original qui est ensuite lemmatisé (ce traitement est également réalisé avec la représentation Word2Vec, cf. paragraphe suivant pour plus de détails)
2. On applique le transformer Spark HashingTF pour calculer la fréquence de chaque terme dans chaque revue
3. On construit ensuite le modèle IDF (lui aussi avec les classes natives de Spark) à partir des données d'apprentissage. Ce même modèle est appliqué sur les données d'apprentissage et de test.

Le code Spark / Scala complet est indiqué en annexe.

Notons que lorsqu'on applique le transformer HashingTF, il est possible de réduire la dimension des données en limitant le nombre de termes à retenir (paramètre *numFeatures*). Nous avons procédé, pour différentes valeurs de *numFeatures*, à la validation croisée présentée dans la suite de ce document (avec les paramètres listés §4.2) et obtenu à l'issue de la phase d'apprentissage les scores AUC (Area Under ROC) ci dessous :

numFeatures	1024	2048	4096	8192
Constante de Régularisation	0.3	0.5	0.5	1.2
Score AUC	0.9054	0.9357	0.9579	0.9696

Remarques :

- L'implémentation Spark de HashingTF recommande d'utiliser des puissances de 2 pour le paramètres *numFeatures* (d'où le choix de valeurs ci-dessus)
- Dans un premier temps nous retenons la valeur *numFeatures* = 4096, car à partir de 8192 *features* :
  - le ratio nbre de variables / nbre de données est élevé (1/3)
  - la valeur de la constante de régularisation augmente fortement

  Ceci nous amène à suspecter un risque de sur-apprentissage. De plus le temps de traitement augmente fortement avec la dimension sans garantie d'obtenir une meilleure discrimination.

3.3 Représentation vectorielle avec Word2Vec

3.3.1 Construction du modèle Word2Vec

Produire des représentation Word2Vec de bonne qualité requiert un modèle construit à partir de textes de même nature et de même vocabulaire que les textes à analyser. Nous allons donc construire le modèle Word2Vec à partir des 75 000 revues des données d'apprentissage (25 000 données étiquetées, 50 000 non étiquetées). Cela constitue un volume de texte assez important mais qui est loin d'être exhaustif.

Pour compenser cette limitation, nous créerons et emploierons le modèle Word2Vec non pas avec des textes bruts mais avec des textes lemmatisés.

La lemmatisation sera réalisée avec des bibliothèques du projet Stanford NLP, packagées dans le jar suivant : http://cedric.cnam.fr/~ferecatu/RCP216/tp/tptexte/lsa.jar

Nous retirerons également des textes initiaux les « stops words » ne présentant pas d'intérêt pour la classification de revues. La liste des « stops words » à filtrer est basée sur celle contenue dans le zip suivant : http://cedric.cnam.fr/~ferecatu/RCP216/tp/tptexte/deps.zip

Nous avons essayé de supprimer de cette liste les négations (« no », « nor », « not ») car comme mentionnée dans le §2.2 nous pensions que celles-ci pouvaient être intéressantes pour la classification.

En fait lorsqu'on applique la classification présentée dans la suite de ce document avec les paramètres listés §4.2) on obtient à l'issue de la phase d'apprentissage un score AUC de 0.9514 en enlevant les négations et un score de 0.9503 en les maintenant. Le gain réalisé est donc marginal.

Le code Spark / Scala pour construire le modèle Word2Vec est reproduit ci-dessous :

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.ml.feature.Word2Vec

val dataDir = "data/input/"

// lecture de l'ensemble des données d'apprentissage
// (avis positifs, avis négatifs et avis non étiquetés)
val trainPos = sc.textFile(dataDir + "train/pos")
val trainNeg = sc.textFile(dataDir + "train/neg")
val trainUnsup = sc.textFile(dataDir + "train/unsup")

// fusion de l'ensemble des données d'apprentissage en  1 seul RDD : trainAll
val trainAll = trainPos.union(trainNeg).union(trainUnsup)

// lecture de la liste des « stop words »
val stopWords = sc.broadcast(
               ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")
               ).value

// génération du RDD corpus par lemmatisation du texte contenu dans le RDD trainAll
val corpus = trainAll.mapPartitions(iter => {
           val pipeline = ParseWikipedia.createNLPPipeline();
           iter.map{ review  =>  ParseWikipedia.plainTextToLemmas(review, stopWords,
                                    pipeline)};
      })

// creation d'un dataframe à partir du RDD corpus
val corpusDF = corpus.toDF("text")

// création du modèle word2vec à partir de corpusDF
val word2Vec = new
Word2Vec().setInputCol("text").setOutputCol("features").setVectorSize(400).setMinCount(0)
val w2vModel = word2Vec.fit(corpusDF)

// enregistrement du modèle word2vec
w2vModel.save("data/output/w2vModel")
```

En examinant le code, plus spécifiquement la ligne suivante :

```
Word2Vec().setInputCol("text").setOutputCol("features").setVectorSize(400).setMinCount(0)
```

on note la présence de deux paramètres numériques :

- min count : le nombre d'occurrences minimal pour qu'un terme soit pris en compte dans le modèle Word2Vec. Nous avons fixé ce paramètre à 0 de façon à prendre en compte tous les termes
- vector size : intuitivement le fait d'augmenter la taille des vecteurs (au moins jusqu'à un certain point) doit améliorer la qualité de la représentation Word2Vec. Pour vérifier cette hypothèse, nous avons appliqué la même procédure de classification sur les données d'apprentissage en faisant varier uniquement la taille des vecteurs Word2Vec. Nous avons obtenu (avec les paramètres listés §4.2) les résultats présentés dans le tableau ci-dessous .

Taille des vecteurs Word2Vec	100	200	300	400
Score AUC	0.9476	0.9505	0.9512	0.9514
Constante de régularisation	0.02	0.02	0.02	0.05

Au vu de ce tableau, nous n'avons pas juger utile d'augmenter la taille des vecteurs Word2Vec au-delà de 400, car l'amélioration obtenue aurait été extrêmement minime.

3.3.2 Application du modèle Word2Vec

Le but du traitement détaillé dans de ce paragraphe est de construire un dataframe Spark, dans lequel chaque ligne correspond à une revue, et qui comporte les colonnes suivantes :

- text : le texte de la revue
- label : 1 si la revue présente un avis positif, 0 en cas d'avis négatif
- features : la représentation du texte sous forme d'un vecteur Word2Vec

Schématiquement, l'algorithme est composé de trois étapes :

1. Lecture du texte de chaque revue, auquel on joint le label 1 si elle est positive, 0 si elle est négative
2. Suppression des « stop words » et lemmatisation
3. Application du modèle Word2Vec

Si les deux premières étapes peuvent être réalisées avec des RDD, le fait d'utiliser des dataframes pour appliquer le modèle Word2Vec offre un avantage appréciable. En effet cela permet d'employer le package ml de Spark, capable de calculer directement la représentation vectorielle d'un texte entier (Spark transforme chaque mot du texte en vecteur Word2Vec puis calcule le barycentre de l'ensemble des vecteurs).

En comparaison, le recours à la version précédente du package - mllib (forcé si on choisit d'employer des RDD au lieu de dataframes) - ne permet de calculer la représentation vectorielle que d'un seul mot à la fois.

Il convient aussi de remarquer que la transformation de textes en données vectorielles doit être réalisée pour les données d'apprentissage puis pour les données de test. Aussi nous avons créé une fonction pour éviter de dupliquer le code. Cette fonction ainsi que son application sur les données d'apprentissage est présentée ci-dessous :

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.ml.feature.Word2Vec
import org.apache.spark.ml.feature.Word2VecModel

// pathDataPos : emplacement des avis positifs
// pathDataNeg : emplacement des avis négatifs
// stopWords   : collection de stop words
// w2vModel    : modèle Word2Vec
// outputFile  : fichier parquet contenant le dataframe résultat
def buildDataframe(pathDataPos: String, pathDataNeg : String, stopWords:Set[String],
w2vModel: Word2VecModel, outputFile: String) = {

    // lecture des avis positifs et ajout de l'étiquette "1.0"
    val dataRawPos = sc.textFile(pathDataPos).map(t => (1.0,t));

    // lecture des avis negatifs et ajout de l'étiquette "0.0"
    val dataRawNeg = sc.textFile(pathDataNeg).map(t => (0.0,t));

    // fusion des RDD contenant les avis positifs et les avis négatifs
    val dataRaw = dataRawPos.union(dataRawNeg);

    // suppression des stop words et lemmatisation du texte brut
    val data = dataRaw.mapPartitions(iter => {
                val pipeline = ParseWikipedia.createNLPPipeline();
                iter.map{ case(label,review) =>
                        (label,ParseWikipedia.plainTextToLemmas(review, stopWords,pipeline))};
            }).cache()

    // création d'un dataframe avec une colonne "label" et une colonne "text" à partir du RDD
    val dataDF = spark.createDataFrame(data).toDF("label", "text")

    // application du modèle word2vec au texte :
    // ceci se traduit par la création d'un nouveau dataframe
    // comportant une colonne "features"
    // représentant le texte sous forme vectorielle
    val dataFeaturesDF = w2vModel.transform(dataDF)

    // export du dataframe résultat sous forme d'un fichier parquet
    dataFeaturesDF.write.parquet(outputFile)

}


val stopWords =
sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value
val w2vModel = Word2VecModel.load("data/output/w2vModel")

// traitement des données d'apprentissage
val pathDataPosTrain = "data/input/train/pos"
val pathDataNegTrain = "data/input/train/neg"
val outputFileTrain  = "data/output/trainDF"

buildDataframe(pathDataPosTrain, pathDataNegTrain, stopWords, w2vModel, outputFileTrain)
```

3.3.3 Visualisation des vecteurs Word2Vec

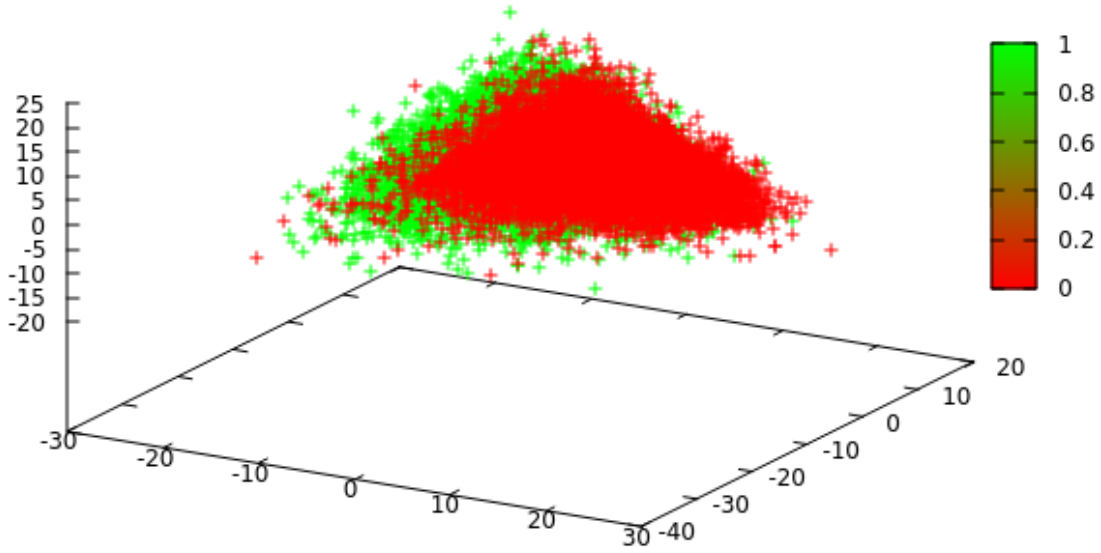
Afin de mieux comprendre comment sont regroupées les revues nous souhaitons visualiser les vecteurs Word2Vec avec un graphique 3D. Sachant que chaque vecteur a 400 composantes, le fait de choisir 3 dimensions prises au hasard ne permettrait pas d'observer une séparation significative entre les avis positifs et négatifs. Pour améliorer la valeur informative du graphique, nous réaliserons donc une analyse en composante principale (centrée et normée) sur les données et retiendrons les 3 composantes principales.

Le code Spark / Scala est le suivant :

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.feature.PCA
import org.apache.spark.ml.feature.PCAModel

val sqlContext = new SQLContext(sc)
val trainDF = sqlContext.read.parquet("data/output/trainDF")


// Définition du scaler pour centrer et normer les variables initiales
val scaler = new StandardScaler().setInputCol("features").setOutputCol("scaledFeatures")
                                 .setWithStd(true).setWithMean(true)

// Construction et application d'une nouvelle instance d'estimateur PCA
val pca = new PCA().setInputCol("scaledFeatures").setOutputCol("pcaFeatures").setK(3)

// Définition du pipeline (scaler puis PCA)
val pipeline = new Pipeline().setStages(Array(scaler,pca))

// Construction du modèle
val pipelineModel = pipeline.fit(trainDF)

// Application du « transformateur » pipeline
val resultat = pipelineModel.transform(trainDF).select("pcaFeatures")

// Extraction du modèle PCA à partir du pipeline
val pcaModel = pipelineModel.stages(1).asInstanceOf[PCAModel]

// Affichage de la variance expliquée
pcaModel.explainedVariance

// Sauvegarde des 3 premières composantes principales dans un fichier texte
resultat.repartition(1).map(v => v.toString.filter(c => c != '[' & c !=
']')).write.text("data/output/pcaFeatures")

// Sauvegarde des labels associés dans un autre fichier texte
trainDF.select("label").repartition(1).map(v => v.toString.filter(c => c != '[' & c !=
']')).write.text("data/output/pcaLabel")
```

La part de variance expliquée par les 3 premiers composantes principales est respectivement : 17 % , 7 % et 6 % soit un cumul de 30 %.

Bien qu'améliorée par l'ACP, la visualisation n'offre donc qu'une représentation très partielle de l'information totale :



Les revues positives (en vert) et celles négatives (en rouge) semblent former deux clusters. Ceci dit, une quantité d'information trop importante a été perdue lors de la projection des 400 dimensions initiales sur les 3 premiers composantes principales pour distinguer une séparation nette entre les classes.

4. Construction du modèle SVM

Plusieurs types d'algorithme sont envisageables pour discriminer les revues positives des revues négatives à partir des données vectorielles construites dans le paragraphe précédent : classification naïve bayésienne, régression logistique et SVM (Support Vector Machine) entre autres.

Dans la littérature consacrée à l'analyse de sentiment sur des textes courts, SVM (avec un noyau linéaire) est considérée comme l'une des méthodes les plus performantes - cf. Pang et alt. (2002) [3], Kharde and Sonawane (2016) [4]. C'est donc la méthode que nous avons retenue ici.

4.1 Description du traitement

Tout l'enjeu est de construire un modèle qui réalise une discrimination précise des données d'apprentissage tout en ayant une erreur de généralisation la plus faible possible. Un solution à ce problème est d'employer une procédure de validation croisée afin de trouver le facteur de régularisation qui offre le meilleur compromis biais / variance. La procédure est la suivante :

1. Centrage des données. Par défaut l'implémentation Spark de SVM « standardise » les données (i.e. les divise par la déviation standard) mais ne les centre pas. Le centrage offre une légère amélioration du modèle. Ex: avec les paramètres listés §4.2 , sans centrage le modèle basé sur Word2Vec a un score AUC de 0.9441, avec centrage : 0.9514
2. Définition d'une grille d'hyper-paramètres correspondant à différentes valeurs de la constante de régularisation de la SVM
3. Chaque modèle SVM correspondant à une valeur du paramètre de régularisation est évaluée avec la méthode de validation croisée k-fold
4. L'évaluation du score est calculée avec une instance de BinaryClassificationEvaluator qui utilise par défaut la métrique AUC
5. Le modèle retenu est celui pour lequel la valeur AUC moyenne sur les k jeux de données k-folds est la plus élevée. Ce modèle est ensuite appliqué sur l'ensemble des données d'apprentissage.

4.2 Paramètres initiaux

Afin d'obtenir un ordre de grandeur du facteur de régularisation, nous exécutons le code présenté dans le paragraphe suivant avec les paramètres ci-dessous :

- k (partitionnement de la validation croisée) : 5
- nombre maximal d'itérations :10
- valeurs de la constante de régularisation : 0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 1,1.2

Nous obtenons :

	Représentation TF-IDF	**Représentation Word2Vec**
Taille des vecteurs	4096	400
Constante de régularisation	0.5	0.05
Score AUC	0.9579	0.9514
Précision	89.79 %	88.61 %

4.3 Code Spark / Scala

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.classification.LinearSVC
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.param.ParamMap

// dataPath       : emplacement du fichier contenant les données
// linSvc         : estimateur linéaire
// paramGrid      : grille de (hyper)paramètres utilisée pour grid search
// cvSVCmodelPath : emplacement du fichier où sera sauvegardé le modèle SVM
def svmTrain(dataPath: String, linSvc: LinearSVC, paramGrid: Array[ParamMap], cvSVCmodelPath:
String) = {
    // Chargement des données d'apprentissage
    val sqlContext = new SQLContext(sc)
    val trainDF = sqlContext.read.parquet(dataPath)

    // Définition du scaler pour centrer les variables initiales
    val scaler = new StandardScaler().setInputCol("features").setOutputCol("scaledFeatures").
                      setWithStd(false).setWithMean(true)

    // Définition du pipeline (scaler puis SVM linéaire)
    val pipeline = new Pipeline().setStages(Array(scaler,linSvc))

    // Définition de l'instance de CrossValidator : à quel estimateur l'appliquer,
    //  avec quels (hyper)paramètres, combien de folds, comment évaluer les résultats
    val cv = new CrossValidator().setEstimator(pipeline).setEstimatorParamMaps(paramGrid).
                      setNumFolds(5).setEvaluator(new BinaryClassificationEvaluator())

    // Construction et évaluation par validation croisée des modèles correspondant
    // à toutes les combinaisons de valeurs de (hyper)paramètres de paramGrid
    val cvSVCmodel = cv.fit(trainDF)

   // Afficher les meilleures valeurs pour les (hyper)paramètres
   println("PARAMS : "+
        cvSVCmodel.getEstimatorParamMaps.zip(cvSVCmodel.avgMetrics).maxBy(_._2)._1)

   // Enregistrement du meilleur modèle
   cvSVCmodel.write.save(cvSVCmodelPath)

   // Calculer les prédictions sur les données d'apprentissage
   val resApp = cvSVCmodel.transform(trainDF)

   // Calculer et afficher AUC sur données d'apprentissage
   println("AUC : " + cvSVCmodel.getEvaluator.evaluate(resApp))

   // Calculer et afficher la précision
   resApp.createOrReplaceTempView("res")
   spark.sql("SELECT count(*)/25000  FROM res  WHERE label == prediction").show()
}

// Définition de l'estimateur SVC linéaire
val linSvc = new LinearSVC().setMaxIter(10).
                      setFeaturesCol("scaledFeatures").setLabelCol("label")

// Construction de la grille de (hyper)paramètres utilisée pour grid search
// Une composante est ajoutée avec .addGrid() pour chaque (hyper)paramètre à explorer
val paramGrid = new ParamGridBuilder().addGrid(linSvc.regParam,
    Array(0.02, 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, 0,7, 1.0, 1.2)).build()

// Apprentissage du modèle SVM
svmTrain("data/output/trainDF", linSvc, paramGrid, "data/output/cvSVCmodel")
```

4.4 Amélioration des résultats

Nous pouvons tenter d'améliorer les résultats obtenus au §4.2 en affinant la grille du facteur de régularisation et en augmentant le nombre d'itérations :

- représentation TF-IDF

```
val paramGrid = new ParamGridBuilder().
       addGrid(linSvc.regParam, Array(0.3,0.35,0.4,0.45,0.5,0.55,0.6,0.65,0.7)).
       addGrid(linSvc.maxIter, Array(10,50,100,150,200)).build()
```

- représentation Word2Vec

```
val paramGrid = new ParamGridBuilder().
       addGrid(linSvc.regParam, Array(0.01,0.02,0.03,0.04,0.05,0.06,0.07,0.08,0.09)).
       addGrid(linSvc.maxIter, Array(10,50,100,150,200)).build()
```

Nous obtenons :

	Représentation TF-IDF	**Représentation Word2Vec**
Taille des vecteurs	4096	400
Constante de régularisation	0.5	0.02
Nombre max. d'itérations	10	200
Score AUC	0.9579	0.9564
Précision	89.79 %	89.17 %

Le fait d'affiner la recherche de la constant de régularisation et d'augmenter le nombre d'itérations (de 10 à 200) permet d'améliorer légèrement le modèle basé sur Word2Vec : le score AUC passe de 0.9514 à 0.9564 et la précision de 88.61 % à 89.17 %. En revanche cela n'apporte aucune amélioration pour le modèle TF-IDF.

Nous essayons alors, à nombre maximal d'itération constant (10), d'augmenter le paramètre *numFeatures* de TF-IDF en balayant, pour la constante de régularisation, les valeurs : 1.1, 1.2, 1.3, 1.4, 1.5. Nous observons les résultats suivants :

	Représentations TF-IDF			**Représentation Word2Vec**
Taille des vecteurs	4096	8192	16384	400
Constante de régularisation	0.5	1.3	1.4	0.02
Nombre max. d'itérations	10	10	10	200
Score AUC	0.9579	0.9688	0.9834	0.9564
Précision	89.79 %	91.23 %	93.89 %	89.17 %

A première vue augmenter le nombre de features pour les représentations TF-IDF semble donc améliorer la qualité de classification. Cependant, en diminuant le biais du modèle, n'avons nous pas trop augmenté sa variance, réalisant ainsi un sur-apprentissage comme évoqué au §3.2 ?

5. Application du modèle SVM aux données de test

5.1 Code Spark / Scala

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.tuning.CrossValidatorModel

// dataPath       : emplacement du fichier contenant les données
// cvSVCmodelPath : emplacement du fichier contenant le modèle SVM
def svmTest(dataPath: String, cvSVCmodelPath: String) = {
    // Chargement des données de test
    val sqlContext = new SQLContext(sc)
    val testDF = sqlContext.read.parquet(dataPath)

    // Chargement du modèle SVM
    val cvSVCmodel =  CrossValidatorModel.load(cvSVCmodelPath)

    // Calculer les prédictions sur les données de test
    val resultats = cvSVCmodel.transform(testDF)

    // Afficher 10 lignes complètes de résultats (sans la colonne features)
    resultats.select("label", "rawPrediction", "prediction").show(10, false)

    // Calculer et afficher AUC sur données de test
    println("AUC : " + cvSVCmodel.getEvaluator.evaluate(resultats))

    // Calculer et afficher la précision
    resApp.createOrReplaceTempView("res")
    spark.sql("SELECT count(*)/25000  FROM res  WHERE label == prediction").show()
}

// Evaluation du mdèdle SVM avec les données de test
svmTest("data/output/testDF", "data/output/cvSVCmodel")
```

5.2 Résultats

	Représentations TF-IDF			Représentation Word2Vec
Taille des vecteurs	4096	8192	16384	400
Constante de régularisation	0.5	1.3	1.4	0.02
Score AUC	0.9189	0.9188	0.9159	0.9384
Précision	84.65 %	84.88 %	84.94 %	86.81%

Nous remarquons que :

- le fait d'augmenter le nombre de *features* au-delà de 4096 pour le modèle TF-IDF n'améliore pas sensiblement la qualité des résultats. Au contraire d'après l'évolution du score AUC cela paraît ajouter plus de « bruit » que d'informations pertinentes.
- les modèles TF-IDF et Word2Vec offrent des scores proches avec un léger avantage pour le modèle Word2Vec.
- Il semble y avoir une nette corrélation entre l'augmentation de la constante de régularisation et l'augmentation de la dimension des vecteurs

6. Conclusion

Représenter les textes des revues IMDB sous forme :

- de matrice documents-termes avec pondération TF-IDF ou
- de vecteurs Word2Vec

nous a permis, avec SVM, de discriminer les avis positifs et négatifs avec des précisions comparables.

Ceci dit, la solution Word2Vec, au prix d'une mise en œuvre plus complexe (car nécessitant de construire un Word2VecModel à partir d'un large corpus de revues) offre une précision (87 %) légèrement meilleure que la solution TF-IDF (85 %) avec une erreur de généralisation un peu plus faible et une dimension de données nettement plus petite.

Il est bien sûr envisageable d'améliorer les résultats obtenus. Voici quelques pistes :

- Pour la solution TF-IDF : nous pouvons au lieu de limiter le nombre de *features* prises en compte par HashingTF, garder toutes les *features* et procéder ensuite à une réduction de dimension pour améliorer le rapport signal/bruit et diminuer le volume de données
- Pour la solution Word2Vec : plutôt que de découper les revues en mots et calculer une moyenne arithmétique simple des vecteurs représentant chaque mot :
  - découper les revues en phrases
  - calculer un vecteur Word2Vec pour chaque phrase (moyenne simple des vecteurs représentant chaque mot)
  - calculer un score reflétant le reflétant le sentiment de chaque phrase sur une échelle de 0 (très négatif) à 5 (très positif) avec la fonction sentiment de core-nlp
  - utiliser le score de chaque phrase pour calculer une moyenne pondérée des vecteurs Word2Vec associés

Finalement, rappelons que les revues étant des textes, l'emploi de Doc2Wec (non implémenté dans le version actuelle de Spark) devrait s'avérer plus performant que celui de Word2Vec.

7. Références

1. Q. Le, T. Mikolov . Distributed Representations of Sentences and Documents
2. T. Kenter, A. Borisov, M. de Rijke. Siamese CBOW: Optimizing Word Embeddingsfor Sentence Representations
3. B. Pang, L. Lee, and S. Vaithyanathan. Thumbs up?: sentiment classification using machine learning techniques. In Proceedings of the ACL- 02 conference on Empirical methods in natural language processing-Volume 10, pages 79–86 , 2002.
4. V. Kharde, P. Sonawane, et al. Sentiment analysis of twitter data: a survey of techniques. arXiv preprint arXiv:1601.06971 , 2016.

8. Annexe : Code Spark / Scala pour TF-IDF

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.sql.{Dataset,Row}
import org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer}

// pathDataPos : emplacement des avis positifs
// pathDataNeg : emplacement des avis négatifs
// stopWords   : collection de stop words
def buildDataframe(pathDataPos: String, pathDataNeg : String, stopWords:Set[String]) :
Dataset[Row] = {
    // lecture des avis positifs et ajout de l'étiquette "1.0"
    val dataRawPos = sc.textFile(pathDataPos).map(t => (1.0,t));

    // lecture des avis negatifs et ajout de l'étiquette "0.0"
    val dataRawNeg = sc.textFile(pathDataNeg).map(t => (0.0,t));

    // fusion des RDD contenant les avis positifs et les avis négatifs
    val dataRaw = dataRawPos.union(dataRawNeg);

    // suppression des stop words et lemmatisation du texte brut
    val data = dataRaw.mapPartitions(iter => {
                val pipeline = ParseWikipedia.createNLPPipeline();
                iter.map{ case(label,review) =>
                        (label,ParseWikipedia.plainTextToLemmas(review, stopWords,pipeline))};
            }).cache()

    // création d'un dataframe avec une colonne "label" et une colonne "text" à partir du RDD
    val dataDF = spark.createDataFrame(data).toDF("label", "text")

    dataDF
}

val stopWords =
sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value

// lecture des données d'apprentissage
val pathDataPosTrain = "data/input/train/pos"
val pathDataNegTrain = "data/input/train/neg"
val train = buildDataframe(pathDataPosTrain, pathDataNegTrain, stopWords)

// lecture des données de test
val pathDataPosTest = "data/input/test/pos"
val pathDataNegTest = "data/input/test/neg"
val test = buildDataframe(pathDataPosTest, pathDataNegTest, stopWords)

// création des transformers HashingTF et IDF
val hashingTF = new HashingTF().setInputCol("text").setOutputCol("rawFeatures").
                                                    setNumFeatures(4096)
val idf = new IDF().setInputCol("rawFeatures").setOutputCol("features")

// vectorization et sauvegarde du dataframe d'apprentissage
val trainTF = hashingTF.transform(train)
val idfModel = idf.fit(trainTF)
val trainTFIDF = idfModel.transform(trainTF)
trainTFIDF.write.parquet("data/output/trainDF_TFIDF")

// vectorization et sauvegarde du dataframe de test
val testTF = hashingTF.transform(test)
val testTFIDF = idfModel.transform(testTF)
testTFIDF.write.parquet("data/output/testDF_TFIDF")
```