PROJET 2016 MELS : SENTIMENT ANALYSIS IN TWITTER

Enzo POGGIO

TABLE DES MATIÈRES

1 BASELINE

CE QUE J'AI FAIT :

DÉMARCHE ADOPTÉE :

Afin de répondre au problème de manière exhaustive, je décide d'adopter la démarche scientifique. Ainsi, après une observation minutieuse des données, j'ai formulé un ensemble d'hypothèses que j'ai testé par rapport à une affectation aléatoire de la valeur des tweets (Moyenne pos & neg de 33%). Ainsi, j'écarte premièrement les traits trop peu distinctifs (par exemple :distinguer les tweets contenant des smileys). Ayant obtenu de bons résultats avec l'addition de la valeur sentimentale de chaque mot des tweets (avec premièrement Sentiword 40%), je tente deuxièmement, de corroborer cette hypothèse en utilisant une autre liste de mots positifs et négatifs, en me basant sur les travaux de la NRC-SentimentAnalysis, qui a fini premier dans la "Building the State-of-the-Art in Sentiment Analysis of Tweets". J'utilise alors maintenant la pmilexicon.

GESTION ET PRÉTRAITEMENT DES DONNÉES DE BASE :

SYSTÈMES MODULAIRE & DICTIONNAIRE DE DONNÉES :

Je choisis de lire une fois le fichier testé et de le mettre en mémoire dans un dictionnaire de données. Cette variable sera modifiée, voire rééditée dans la majorité des modules de ce programme. Le système modulaire est plus simple pour essayer une hypothèse portant sur un trait. Je conditionne l'information ainsi :

```
d[i]= (firstNum, secondNum, value, tweet, posScore, negScore)
```

Où `i` est un int (différent pour chaque tweet)
`firstNum`, `secondNum` sont les deux premiers numéros (désignant le numéro utilisateur et le numéro du tweet)
`value` est la valeur du tweet ["positive", "negative", "neutral"]
`tweet` est un tableau contenant toutes les chaînes de caractère du tweet
`posScore`, `negScore` sont des variables de types float qui permettront de définir la valeur du tweet lors de l'affectation.

DESCRIPTION DES MODULES :

`readFile(file)`
- Permet de lire un fichier de tweet
- Crée une variable dictionnaire contenant l'enregistrement
- De chaque tweet (firstNum, secondNum, value, tweet, posScore, negScore)

`writeData(data)`
- Permet d'écrire les données de façon à ce que le script scoredev.py puisse les utiliser

`sentiWordNet(swntxt) & pmilexicon(pmitxt)`:
- Crée et retourne des dictionnaires à partir des ressources entrées.
- Ils sont ainsi constitués (voir le code pour plus de détails)

```
swn[wordkey]=[posScore,negScore, occWord]
pmi[key]=(score, occPos, occNeg)
```

`termeAnalyzerSWN(swn, data)&`
`termeAnalyzerPMI(pmi, data)`:
- Utilise les dictionnaires créés pour pondérer les tweets,
- Change les valeurs de posScore, negScore et retourne le dictionnaire changé

`affectationPNN(data)&`
`affectationPNNCoef(data,coef)` :
- Affecte une valeur ["positive", "negative", "neutral"] au tweet selon posScore, negScore et les coefs calculés dans coefficateur(data, version)
- Retourne le dictionnaire évalué

`randomizer(data)` :
- Permet d'attribuer une valeur aléatoire à chaque tweet et retourne un dictionnaire évalué aléatoirement

`understandData(data)` :
- Affiche des informations qui peuvent aider à trouver un critère d'évaluation à partir des données générées

`Coefficateur(data, version)` :
- Crée des coefficients à partir de données empiriques développées avec des tests.

PROCÉDURE D'ÉVALUATION :

EXPLICATION DE MES CRITÈRES D'AFFECTATION :

Dans `affectationPNNCoef(data,coef)` nous avons la clé de notre affectation !
Procédure :
Tout d'abord on va initialiser la valeur à neutre

```
value="neutral"
```

Ensuite on s'occupe des tweets ambigus qui pourraient être positifs ou négatifs. Ils vont dans un ensemble très tolérant et sont ensuite sélectionnés selon leur plus haut score entre negScore et posScore:

```
if (negScore>negAmb and posScore>posAmb ):
      if negScore>posScore:
            value="negative"
      elif negScore<posScore:
            value="positive"
```

Et pour finir nous traitons les tweets non-ambigus. Si leurs valeurs dépassent les seuils suivants, ils sont déclarés comme positif ou négatif :

```
elif negScore>negNonAmb:
      value="negative"
elif posScore>posNonAmb:
      value="positive"
```

La valeur de sélection a été trouvée en étudiant les données grâce à la fonction `understandData(data)` et `coefficateur(data,version)` :

	Positif	Négatif
Moyenne	0.679180289741	0.3900654977571
Minimum	1.696984865542	3.12605720375
Maximum	12.12688964103	9.59900970943

Ces coefficients ont été calculésgrâce aux valeurs de `affecatation(data)` qui, elles, proviennent de`understandData(data)` puis ont été ajustées empiriquement jusqu'à avoir un score satisfaisant.

	Ambigu	Non-ambigu
Coefficients positifs	0.6791802897	0.39006549775
Coefficients négatifs	1.69698486554	3.126057204
Coefficients moyens	12.1268896410	9.5990097

```
coef = moyenneAjusté / moyenne
```

Ici, j'utilise les moyennes positives et négatives comme critère de sélection. Pour l'améliorer de quelques pourcents, je les ai un peu diminués afin de traiter beaucoup plus de tweets ambigus et un peu plus de tweets non-ambigus!

COMMENT ÇA MARCHE :

COMMENT LANCER MA BASELINE ?

Pour lancer ma Baseline, il vous suffit décrire dans le bash du dossier contenant ma Baseline, le dossier pmilexicon (le fichier sentiWordNet.txt) et le fichier à traiter :
python3 baseline.py -*le nom du fichier à traiter*- > -*le nom du fichier que vous voulez créer*-
Remplacer les -*balises*- par les noms de fichiers appropriés.

RÉSULTATS & CONCLUSION

Avec les données de développement j'arrive à :

```
Confusion table :
gs \ pred| positive| negative| neutral
positive|      259|      102|       91
negative|       46|      179|       36
neutral|      206|      223|      154

Scores:
class      precisio          recall     fscore
positive   (259/511) 0.5068 (259/452) 0.5730 0.5379
negative   (179/504) 0.3552 (179/261) 0.6858 0.4680
neutral    (154/281) 0.5480 (154/583) 0.2642 0.3565
average(pos&neg)                                0.5029
```

Et avec les données de test j'arrive à :

```
Scoring T13:
      positive: P=48.62, R=51.50, F1=50.02
      negative: P=74.04, R=26.89, F1=39.45
      neutral: P=27.89, R=58.22, F1=37.71
      OVERALL SCORE : 44.73

Scoring T14:
      positive: P=58.04, R=59.92, F1=58.97
      negative: P=63.33, R=21.99, F1=32.65
      neutral: P=25.22, R=46.13, F1=32.61
      OVERALL SCORE : 45.81

Scoring TS1 :
      positive : P=42.42, R=33.33, F1=37.33
      negative : P=35.00, R=50.00, F1=41.18
      neutral : P=23.08, R=18.75, F1=20.69
      OVERALL SCORE : 39.25
```

On remarque une baisse significative entre les résultats d'entraînement et les résultats du test. Une adaptation plus générique aux données est certainement nécessaire. De plus, on s'aperçoit que les tweets neutres sont largement sous représentés. Avec mon système, une augmentation de leurs rappels serait bénéfique. Il me faudrait de meilleurs facteurs de discernement pour ne pas surestimer les tweets neutres.

2 PERCEPTRON

CE QUE J'AI FAIT :

DÉMARCHE ADOPTÉE :

CONCEPTION DU PERCEPTRON :

Premièrement, je me suis attardé sur la conception d'un perceptron pour un aussi grand nombre de données. Je me basais sur mon neurone réalisant la porte & qui donnait des résultats justes, mais qui avait un biais de confirmation à l'intérieur. En effet, je testais ma condition de sortie à chaque entrée de l'ensemble d'entraînement ! L'ensemble était de quatre éléments, c'est pour cela que la différence n'était pas majeure et que la convergence arrivait vite. Le perceptron ainsi créé arrivait à convergence mais les résultats avaient complètement perdu leur puissance significative. Les poids descendaient parfois jusqu'à 1*10^-232. De plus, j'utilisais comme condition de sortie la proximité (np.allclose()) entre les anciens et les nouveaux poids. Ceci rendait la convergence très lente et une mise à jour continuelle des poids. Le seul avantage de cette solution était de ne pas utiliser de fonction pour déterminer si l'apprentissage avait été assimilé.
Sur ce premier échec, je décide de repartir sur de nouvelles bases. J'ai donc réalisé un perceptron qui utilise le taux d'erreur commis lors de l'apprentissage pour juger la pondération des poids. J'ai ajouté un compteur d'erreurs et une fonction pour les retourner lors de la mise à jour des poids.

RECHERCHE D'UN MODÈLE :

Ma première hypothèse était que si le vecteur traité était de taille minimale, on arriverait plus vite à convergence ! J'ai donc créé, premièrement, des vecteurs où les mots étaient rentrés en minuscule sans les hashtags, les linkats et les liens. Je n'arrivais seulement qu'à une convergence avec un taux de 6% d'erreurs, soit environ 42% de réussite avec les données de développement.
Mon erreur était dans mon choix de modèle. En effet, plus un ensemble est varié, plus les traits spécifiques ressortent. Avec mon ensemble réduit, je retardais la convergence en plus de devoir accepter un fort taux d'erreur : beaucoup de variables du vecteur du tweet se recoupaient et donc la discrimination devenait plus difficile.
Deuxièmement, j'ai adapté mon modèle pour que celui-ci accepte la capitalisation et prenne en compte les hastags et les linkats (car même un utilisateur tweeter peut être connoté sentimentalement). Seuls les liens et la ponctuation ne sont pas pris en compte. Avec ce nouvel ensemble vectoriel, j'arrive à faire descendre mon taux d'erreurs à moins de 0,25% soit à peine une cinquantaine d'erreurs pour l'ensemble actuel. J'obtiens donc 48% de réussite avec les données de développement.

DESCRIPTION DES MODULES :

`tools(train)`
- crée les outils nécessaires à l'analyse à partir des données d'entraînement :
  - dictonary: dictionnaire<mot,indice du tableau vectoriel> créé sur la base de train
  - vectors: set d'entraînement créé à partir des tweets de train et de dictionary

`checkWord(word)` :
- la fonction checkWord(word) sert à trier les mots à évaluer : elle exclut les liens https://... ; elle sépare le croisillon ou l'arobase des hashtags et des linkats ; elle prend en compte la capitalisation.

`readFile(file, dictonary)` :
-Permet de lire un fichier de tweets
-Crée une variable dictionnaire contenant l'enregistrementde chaque tweet (firstNum , secondNum, value, tweet, vector)

`weightsValue(vectors)` :
- weightsValue calcule les différents vecteurs de poids avec une tolérance d'erreurs selon leurs valeurs.

`neurone(vectors,weight, SelecValue, pourcentErreur)` :
- neurone calcule les poids à partir des poids aléatoires, des vecteurs d'apprentissage de la valeur sélectionnée et dupourcentage d'erreur.

`weightCalculation(entry,weight,t)` :
- weightCalculation est la suite d'opérations pour mettre à jour un poidsentry est le vecteur traité et t est sa valeur: 1 si est la valeur sentimentale désirée sinon 0

`binarisation(x)` :
- binarisation transforme en 1 le paramètre s'il est plus grand de 0 sinon en 0

`affectationPNN(dictTw,weights)` :
- affectationPNN affecte une valeur à chaque tweet selon les poids calculés. Selon le produit interne du vecteur de chaque tweet à traiter avec lesdifférentes valeurs de poids possibles,on sélectionne la valeur max desproduits internes. On affecte cette valeur au tweet ensuite.

`writeData(dictTw,nomfichier)` :
- Permet d'écrire les données dans un fichier de façon à ce que le script scoredev.py puisse les utiliser

PROCÉDURE D'APPRENTISSAGE & D'ÉVALUATION :

EXPLICATION DU FONCTIONNEMENT DE MON PERCEPTRON :

```
def neurone(vectors,weight, SelecValue, pourcentErreur):
          while True :
                    erreur=0
                    for entry in vectors:
                              value=entry[1][SelecValue]
                              weight,e=weightCalculation(entry[0],weight,
value)
                              erreur+=e
                    if  erreur<pourcentErreur*len(weight):
                              break
          return weight

def weightCalculation(entry,weight,t):
          theta=0.17
          o=binarisation(np.dot(entry,weight))
          prediction=np.subtract(t,o)
          deltaWeight= entry*prediction*theta
          newWeight=np.sum([weight, deltaWeight], axis=0)
          erreur=0
          if not prediction==0:
                    erreur=1
          return newWeight, erreur
```

Mon neurone marche grâce à ces deux modules. Tant que le taux d'erreur n'est pas satisfaisant, on recommence à lire le set d'entraînement et on met à jour les poids. Le taux d'erreurs influe directement sur le temps d'exécution et la précision des réponses.

EXPLICATION DE MON AFFECTATION :

On ne peut pas négliger le fait que les tweets neutres ont un trait distinctif particulier - des mots peuvent être sentimentalement neutres - au même titre que les positifs et les négatifs. Je ne considère donc pas deux ensembles de tweets positifs, négatifs et les tweets restants comme neutres, mais je décide de calculer le potentiel neutre de chaque tweet. Le produit interne le plus grand entre le vecteur du tweet et les poids sentimentaux détermine la valeur du tweet.

COMMENT ÇA MARCHE :

COMMENT LANCER MON PERCEPTRON2 ?

Pour lancer mon perceptron2 il suffit de lancer la commande suivante dans un bash UNIX :

```
python3 perceptron2.py
```

Ensuite, le script vous demandera quel fichier vous voulez traiter.Puis, le nom de fichier de sortie que vous voulez donner.Après 1 à 3 minutes selon les tolérances d'erreurs choisies, un fichier sera créédans le dossier contenant le script.

Exemple :

```
enzo@EnzoP300:~/.../poggioe0perceptron$ python3
perceptron2.py
```

Nom du fichier à traiter : development.input.txt
Nommer votre fichier de sortie : development.output (pas besoin de mettre le ".txt")

RÉSULTATS & CONCLUSION

Avec les données de développement j'arrive à :

```
Confusion table:
gs \ pred | positive| negative| neutral
positive | 326|  23|  103
negative | 88|       69|  104
neutral   |  193|  46|     344

Scores:
class           precisionrecall     fscore
positive (326/607) 0.5371 (326/452) 0.7212 0.6157
negative (69/138) 0.5000 (69/261) 0.26440.3459
neutral (344/551) 0.6243 (344/583) 0.5901    0.6067
average(pos&neg)                                0.4808
```

Et avec les données de test j'arrive à:

```
Scoring T13:
       positive: P=63.47, R=61.77, F1=62.61
       negative: P=24.60, R=43.60, F1=31.46
       neutral: P=66.45, R=59.49, F1=62.77
       OVERALL SCORE : 47.03

Scoring T14:
       positive: P=62.81, R=70.03, F1=66.23
       negative: P=20.00, R=31.91, F1=24.59
       neutral: P=65.08, R=52.34, F1=58.02
       OVERALL SCORE : 45.41

Scoring TS1:
       positive: P=84.85, R=45.90, F1=59.57
       negative: P=10.00, R=50.00, F1=16.67
       neutral: P=84.62, R=64.71, F1=73.33
       OVERALL SCORE : 38.12
```

CONCLUSION ET REMARQUE :

Aussi bien pour les valeurs de développement que pour celles de test, on remarque que la valeur négative est largement sous-représentée et mal discriminée ! Il faudrait peut-être ajouter un biais pour obtenir plus de résultats négatifs (comme augmenter légèrement les poids négatifs).
Pour conclure, on peut dire que ce système est consistant. En effet, entre les résultats du test et du développement, il n'y a pas tant de différence, les résultats restent similaires. En comparaison avec ma Baseline, les résultats du test sont assez similaires. En revanche, les résultats des données de développement sont inférieurs de 3% à ceux trouvés ici. Mais cela est dû au fait que ma Baseline était trop spécifique.
Maintenant, les seuls facteurs pour avoir de meilleurs résultats sont : le modèle adopté et la tolérance d'erreur acceptée.

3AMÉLIORATION

CE QUE J'AI FAIT :

DÉMARCHE ADOPTÉE :

Pour répondre aux objectifs de cette étape, j'ai procédé méthodiquement :

- Formulation d'hypothèses sur la base d'intuition et d'expérience
- Codage d'une solution
- Test du système une dizaine de fois (Annexe 1)
- Analyse des résultats, des erreurs et des biais possibles.
- Décision de garder ou non la solution selon les résultats !0.8464

Ainsi on adopte une démarche scientifique.
Pour répondre aux aspects que l'on pouvait modifier pour le 1) le type de Learning, je passe d'un simple neurone d'apprentissage à 2, permettant un Actif Learning.
Pour 2), j'ai intégré dans mon second modèle créé, justement, la gestion des bi-grams, grâce à la PMI-Lexicon.
Pour 3), j'ai tenté une hypothèse simple qui est de dire que le sentiment est l'inverse de celui que l'on classifie !

On verra plus en détails ce que j'ai fait dans la section des versions suivantes.

VERSIONS & RECHERCHES

BASIC PERCEPTRON

3.0

Ici, on reprend juste notre perceptron que l'on a produit dans l'étape précédente pour l'améliorer.

HYPOTHÈSES PRINCIPALES :

Notre première hypothèse est que calculer des poids en partant de l'équilibre donnera des meilleurs résultats, du fait que seuls les poids représentatifs vont augmenter ou bien diminuer de manière drastique.

DESCRIPTION DES MODULES :

Ici et dans toutes les versions suivantes, on n'évoquera que les changements ou les apparitions de modules dans le fichier.
Dans cette version, on retrouve tous les modules que l'on avait créés dans le Perceptron2 précédent.
Le seul changement apporté est donc dans l'initialisation des poids dans `weightsValue(vectors)`. L'initialisation n'est plus aléatoire mais à zéro pour tout le vecteur.

RÉSULTATS :

Chaque pourcent dans la sous-section résultats exprimera la moyenne positifs/négatifs des données de développement. On indiquera à chaque fois ce que sont les autres chiffres.On obtient 0.4675 dans tous les cas où on génère avec un vecteur de poids équilibrés au début, et on obtient 0.4711pour le maximum et 0.4508pour le minimum quand on génère avec un vecteur de poids aléatoires au début, soit en moyenne 0.4609. On se donne une autre mesure pour juger de l'efficacité du système créé, qui est le testTraining. Le principe est d'évaluer notre fichier d'entraînement avec notre système, puis de voir les résultats obtenus indiquant si c'est un bon système d'apprentissage par rapport aux autres versions. Ici le testTraining est de 0.8464.
Pour consulter les autres valeurs

AMÉLIORATION RETENUE :

Cette amélioration n'est pas retenue. En effet, elle nous permet certes, d'améliorer la précision des poids et l'aléatoire est à double tranchant : parfois il peut être très bon comme très mauvais. Cependant, il nous permet surtout de donner plus d'indications lors de l'analyse des données dans les différentes matrices de confusion généré. Donc nous ne gardons pas cette amélioration minime qui nous fait perdre des données qui pourraient nous aiguiller vers des solutions intéressantes.

AVERAGE PERCEPTRON

3.1

L'average perceptron est une technique conseillée pour avoir de meilleurs résultats et pour remonter le rappel dans nos classifications.

HYPOTHÈSES PRINCIPALES :

L'average perceptron améliore la classification. Une intuition que vous nous proposez.

DESCRIPTION DES MODULES :

La seule modification apportée se trouve dans `neurone(vectors,weight, SelecValue, pourcentErreur)` Maintenant, à chaque fois que l'on produit un vecteur de poids, on l'ajoute à un autre vecteur qui marche comme un compteur. On va ainsi obtenir la somme de tous les poids calculés jusqu'à la convergence. On retourne ensuite le vecteur divisé par le nombre de poids que l'on a calculé pour arriver à convergence.

RÉSULTATS :

Meilleur résultat :0.4721. Pire résultat :0.4485. Moyenne :0.4587.
Amélioration des positifs et des neutres non négligeable 1 à 4%, mais chute des neutres de 5 à 9%.
Le test training est de 0.8504.

AMÉLIORATION RETENUE :

Maintenant nous n'utiliserons plus que l'Average Perceptron !

AVERAGE PERCEPTRON AVEC BIAIS

HYPOTHÈSES PRINCIPALES :

Un biais est l'idée de donner plus d'importance à un type de vecteur d'apprentissage afin d'altérer la pondération de la classification pour augmenter les résultats du type voulu.
Le biais relatif, ou l'équilibrage des données d'entraînement, se base sur le principe précédent mais calcule de manière proportionnelle la valeur du biais en fonction du nombre d'objets à classifier et du nombre de vecteurs de chaque type.
3.2 est un biais négatif et 3.3 est un équilibrage des données d'entrée.

3.2
DESCRIPTION DES MODULES :

Le seul module modifié est `tools(train)`: lors de la création des vecteurs de type « negative » d'entraînement ici, au lieu de mettre des 1 si le mot est présent, on met un 3.

RÉSULTATS :

Meilleur résultat :0.5276.Pire résultat :0.502. Moyenne :0.5201.
Nette amélioration des négatifs avec parfois des améliorations jusqu'à 12%.
Le testTraining est au plus haut dans cette version : 0.8754
Le testTraining de 3.10 n'a que 1.20% de différence !

AMÉLIORATION RETENUE :

Cette solution, bien que bonne, ne va pas être utilisée par la suite. En effet, elle est trop spécifique. Par contre, elle nous indique quelle hypothèse privilégier pour 3.3

3.3
DESCRIPTION DES MODULES :

Ici, on remodifie`tools(train)` : lors de la lecture du fichier d'entraînement, on compte le nombre d'éléments positifs, négatifs et neutres, et on pondère les vecteurs selon cette formule :

```
1/ (nb de tweets d'une valeur/nb de tweets total)
```

RÉSULTATS :

Meilleur résultat :0.5292.Pire résultat :0.511. Moyenne :0.5183.
Il y a seulement une Chute de trois pourcents pour les neutres par rapport à 3.1.
Le testTraining est assez proche de 3.2 et juste en dessous de 3.10 :0.8626

AMÉLIORATION RETENUE :

Cettesolution est plus générique, le biais est fait sur toutes les valeurs et pas uniquement sur les négatifs. Cette solution est adoptée car plus universelle.

TENTATIVE DE GESTION DES BIGRAMS SUR AVEC LES DONNÉES ENTRAÎNEMENTS

3.4
HYPOTHÈSES PRINCIPALES :

Plus on a de grands vecteurs, mieux on arrive à classifier les tweets car plus il peut y avoir une singularité qui fait ressortir les traits positifs, négatifs ou neutres.
Utiliser les bi-grams serait une bonne manière d'augmente la taille des vecteurs car ils peuvent donner des indications non négligeables sur la valeur sentimentale.

DESCRIPTION DES MODULES :

Ici, on remodifie encore`tools(train)` : lors de la lecture du fichier d'entraînement pour chaque tweet, on ajoute tous ses bi-grams au dictionnaire qui nous permettra de construire les vecteurs ! On ajoute seulement les bigrams que l'on croise plus d'une fois dans le dictionnaire.

ERREURS& PROBLEMES :

Même en montant la tolérance d'erreur pour lecas où le calcul puisse aboutir, on ne peut faire d'apprentissage !
Mon erreurest dans la non significativité des traits ! En effet, il y a trop de bruit dans mes vecteurs pour que l'entraînement puisse extraire des différences significatives dans les vecteurs.

RÉSULTATS :

Quelque soit l'essai, le résultat est toujours 0.2586.
En même temps, les tolérances d'erreurs sont beaucoup trop élevées :

- Tolérance neutre 15.0%
- Tolérance positif 11.0%
- Tolérance négatif 5.0%

AMÉLIORATION RETENUE :

La descente de gradient étant impossible, on ne peut considérer cette solution comme consistante. Son taux d'erreur est beaucoup trop tolérant pour donner des résultats significatifs.
J'arrête ici mes recherches sur les bi-grams des données d'entraînement car je ne vois pas comment les gérer.

À LA RECHERCHE D'UN SECOND MODÈLE

3.5
HYPOTHÈSES PRINCIPALES :

Si on peut comparer nos résultats entre deux modèles différents traitant le même fichier, alors on corroborera nos résultats. Puis, ceci nous permettra de faire un Actif Learning.
Il nous faut faire un second modèle qui n'a pas les mêmes traits que le premier.
Pour ce faire, je vais tenter de faire un modèle linguistique qui se basera sur les traits suivants :

- Le dernier mot : si le dernier mot du tweet est positif ou négatif
- Les smileys : si le tweet contient des smileys :
- HashTag : le nombre majoritaire de hashtags positifs ou négatifs du tweet, par rapport au nombre total.
- Unigrams : le nombre majoritaire d'Unigrams positifs ou négatifs du tweet, par rapport au nombre total.
- Bigrams: le nombre majoritaire de Bigrams positifs ou négatifs du tweet, par rapport au nombre total.
- La Neutralité : pénalité accordée si les traits précédents ne sont pas retrouvés.

DESCRIPTION DES MODULES :

Les modules qui apparaissent pendant cette version sont nombreux mais la plupart ne seront pas réutilisés par la suite :

`readPmiFile(file)`
Permet de lire tous les fichiers du dossier PMIlexicon et les transforme en dictionnaire.

`dictPmi(dictpmi)` :
Permet de transformer un dictionnaire key : string et value : float(str), en deux dictionnaires. Un positif si d[key]<0 et un autre négatif pour les autres cas.*Disparaîtra dans la version 3.6.*

`fonctionRessources(files)` :
Crée une variable de ressources qui sera utilisée par la suite dans le programme à partir des fonctions ci-dessus
`linguisticVector(tweet, ressources)` :
Permet la construction d'un vecteur pour chaque tweet selon les traits déterminés plus haut.*Disparaîtra dans la version 3.6.*

`pmiGrams(score, setPos, setNeg, dictGrams, setTweet)` :
Permet de ne pas répéter le code dans `linguisticVector(tweet, ressources)` pour la construction du vecteur.*Disparaîtra dans la version 3.6.*

Quelques modules ont dû être adaptés pour marcher avec ce nouveau modèle :

`tools(train,ressources)` :
Tools prend maintenant ressources comme paramètres afin de pouvoir créer les vecteurs d'entraînement du second modèle.

`readFile(file, dictonary, ressources)` :
Readfile prend maintenant aussi ressources comme paramètres afin de pouvoir créer les vecteurs de classification du second modèle.

`neurone(vectors,weight, SelecValue,traitement,pourcentErreur)` :
Le paramètre traitement apparaît dans neurone pour savoir quel type de vecteur on utilise pour l'entraînement.

ERREURS :

Mon erreur ici est de n'avoir pas assez de traits distinctifs pour que cela soit possible de classifier les tweets. On n'a pas assez d'éléments pour obtenir une descente de gradient satisfaisante !

RÉSULTATS :

Quelque soit l'essai, on obtient : 0.1676
Tolérance d'erreur nécessaire pour exécuter le programme :

- Tolérance neutre 50.0%
- Tolérance positif 49.0%
- Tolérance négatif 25.0%

Et c'est le pire testTraining de toutes les versions : 0.1225

AMÉLIORATION RETENUE :

Ce modèle n'est évidemment pas retenu car les résultats sont trop mauvais.

3.6

La version 3.6 ne m'a pas servi à produire des données, mais à créer le second modèle. À apprendre comment gérer de grandes masses d'informations pour créer des vecteurs intelligemment, et aussi apprendre à utiliser pickle.

DESCRIPTION DES MODULES :

Modules outils :
`fileVar(var,strName)` :
Permet de sauver une variable var dans un fichier strName.p

`unFileVar(file)` :
Permet d'ouvrir un fichier file.p

`inter(a,b)` :
Retourne true si l'intersection de a et de b est non vide

`valeurAbsolue(x)`
Retourne la valeur absolue de x

Modules de traitement des données :
`sentimentalBindingDictionnary(files)` :
Permet de créer le dictionnaire de sentiments liés aux uni-grams et aux bi-grams de PMIlexicon.
`fullThisDict(dico,indice,dictSentimental,dictSentIndice)` :
Fonction utile à sentimentalBindingDictionnary(files) pour ne pas répéter du code

`tools(train,bindingDictionnary)`
Prend maintenant bindingDictionnary pour créer les vecteurs du second modèle

`linguisticVector(tweet, bindingDictionnary, value)` :
Cette fonction est maintenant bien différente. Elle produit des vecteurs sur la longueur de bindingDictionnary. Elle marche selon le même principe que tools pour le remplissage des vecteurs.

3.7

3.7 est une modification de 3.6 qui enfin marche après avoir nettoyé le vecteur des données inutiles parmi les bi-grams de PMIlexicon.

HYPOTHÈSES PRINCIPALES :

Ici, nous sommes toujours à la recherche d'un second modèle qui donne de bons résultats comme dans 3.5.

DESCRIPTION DES MODULES :

`deleteNotUseValue(tweets,bindingDictionnary)` :
Crée un dictionnaire lié sentimentalement de manière intelligente afin d'enlever du vecteur les bi-grams ou les uni-grams non présents dans l'ensemble des tweets d'entraînement.

`readFile(file, dictonary, bindingDictionnary)` :
Prend maintenant bindingDictionnary pour créer les vecteurs de classifications

RÉSULTATS :

On obtient toujours 0.4944 car l'initialisation du vecteur des poids se fait à zéro pour ce second modèle !
En revanche, le testTraining à 0.6939 n'est pas très bon. En effet, le second modèle étant très entraîné à reconnaître les positifs et les négatif, les neutres ne sont pas bien classifiés.

AMÉLIORATION RETENUE :

On garde le modèle de 3.7 pour passer à l'Actif Learning.

ACTIF LEARNING

3.8

À vrai dire, 3.8 n'est pas un algorithme d'Actif Learning, mais il est une recherche pour y arriver, c'est pour cela qu'il est dans cette section.

HYPOTHÈSES PRINCIPALES :

Si on évalue avec les deux modèles les tweets à classer, puis que l'on sélectionne au hasard une des deux valeurs entre les deux classifications, on devrait obtenir une moyenne entre celle de 3.7 et celle de 3.3.

DESCRIPTION DES MODULES :

`affectationPNN(dictTw,weights,x)` :
x nous permet maintenant de faire la sélection pour chaque tweet de quel vecteur on veut affecter !

`actifLearnig(dictTw)` :
Permet de trouver tous les tweets qui ont été affectés de la même manière par les deux modèles, et affecte une valeur aléatoire entre les deuxpossibles aux tweets classés différemment.

RÉSULTATS :

Meilleur résultat :0.5273.Pire résultat :0.4996. Moyenne :0.5112.
L'hypothèse est vérifiée :0.4944< 0.5112 <0.5183
Le testTraining :0.7801

AMÉLIORATION RETENUE :

3.8 Nous ouvre de nouvelles perspectives pour l'Actif Learning.

3.9

HYPOTHÈSES PRINCIPALES :

Utiliser les tweets qu'on est sûr d'avoir bien classés pour classer les non-classés améliorera les résultats.

DESCRIPTION DES MODULES :

`actifLearnig(dictTw, vectors, dictonary, tweets, classify, noClassify, recurSBD)` :
ActifLearning est une fonction récursive. Si l'ensemble des tweets classés de manière similaire par les deux modèles ne dépasse pas 60% des tweets totaux, alors on recalcule les poids en ajoutant les tweets classés à l'ensemble d'entraînement. Une fois les soixante pourcents dépassés,les tweets non-classés ont une des deux valeurs possibles attribuées aléatoirement.

RÉSULTATS :

Meilleur résultat :0.5445.Pire résultat :0.503. Moyenne :0.5231*.
*Moyenne faite sur 7 éléments car lors du lancement, au bout d'un certain temps (et parce que je testais un projet Java en même temps, mon CPU a fait une surchauffe et mon ordinateur s'est arrêté pour ne pas abîmer les composants.
Testtraining : 0.7822

AMÉLIORATION RETENUE :

3.10

HYPOTHÈSES PRINCIPALES :

Si on recalcule les vecteurs d'entraînement de manière relative à l'ensemble des tweets classés et ceux d'entraînement, on aura de meilleurs résultats.
Le premier modèle étant meilleur que le deuxième, les tweets non-classés seront classés selon le premier modèle et plus de manière aléatoire.

DESCRIPTION DES MODULES :

`recalculVectors(tweets,bindingDictionnary)`
Permet de recréer les vecteurs d'apprentissage en ajoutant les tweets déjà classés à l'ensemble d'entraînement.

`reRead(dictTw, recdictonary, bindingDictionnary)`
Relit les tweets non-classés et recrée des vecteurs de classification à partir de recdictonary et bindingDictionnary redéfinis par `recalculVectors` et `actifLearning`.

`sarcastique(dictTw)` :
Permet de changer la valeur des tweets sarcastiques entre les positifs et les négatifs (seulement utile pour les tweets de test.input.txt)

RÉSULTATS :

Meilleur résultat :0.546.Pire résultat :0.5267. Moyenne :0.5348.
L'utilisation du premier modèle au lieu de l'aléatoire fait monter de quelques dixièmes de pourcent en général !
57% de similarité donne à ma surprise de meilleurs résultats que 75%.
Overfeeder son système n'est pas rentable ni en temps ni pour les résultats.

PERCEPTRON4 : UNE SIMPLIFICATION DU CODE ET POTIENTIELLE AMÉLIORATION

Vous trouverez ci-joint à ce rapport toutes les versions des codes que nous avons présentés plus haut. Et vous trouverez aussi un version de mon perceptron4. C'est la version bien commenté, épuré et avec les quelques bug set incohérences enlevés. Perceptron4 est plus une vitrine pour comprendre comment mon code marche que réellement l'algorithme le plus puissant de ce projet. Les descriptions de chaque module y est la plus détaillé, c'est pour ça que ici je ne vais pas reprendre une description des modules. Et pour la même raison je n'ai pas commenté les versions précédentes au fur et à mesure !
Je vous conseille la lecture de ce code la uniquement !

COMPARAISONS DES RÉSULTATS DES TESTS ET DU DÉVELOPMENT

Si on se base uniquement sur AL10 sur l'annexe on voit largement que cette version supplante toutes les autres ! (tableau 2) Elle dépasse même les résultats de développement !
La gestion des sarcastique en inversant les positifs et les négatifs est un fiasco par contre !
Les versions 3.2 et 3.3 sont pas mauvaise du moins elle reste constante avec leur version de développement.
Pour la version 4 AL, elle est descendu par rapport à mes attentes je pensais qu'en corrigeant les différent problèmes soulevés dans le code cela rendrait de meilleur résultat. Mais apparemment cette version est overfeed par les bi-grams du second modèle.

CONCLUSION

Nous avons fini par dépassé notre Baseline avec ces améliorations, alors que notre perceptron avait des résultat modeste nous voilà avec une belle envolé des résultat ! 10% de différence avec le premier des systèmes ! Cela nous classerait aux38éme rang de la tache ! Avec la complexité de notre algorithme nous sommes arrivés à de très bons résultats ! Mais à quels prix ?

CONCLUSION DU PROJET

Pour conclure ce projet, revenons sur ce que celui-ci nous a appris et montré :
Premièrement, ce projet était pour moi mon premier projet de recherche scientifique : ce type de projet ouvert, où nous sommes libres de traiter le sujet comme nous le souhaitons est d'autant plus exigent, mais pas moins passionnant ! En effet, il demande de la rigueur et beaucoup d'organisation. Mais c'est seulement ainsi que l'on peut avoir une approche globale du sujet à traiter, en se plongeant dedans à avoir l'impression d'être dépassé par le flux et la gestion de données.
Deuxièmement, ce projet nous a montré comment la linguistique est une discipline complexe pour le machine Learning. Avec une simple couche de neurones on n'arrive même pas à classifier à 100% des données dont on connaît le résultat et sur lesquels on s'est entraîné ! (aucun testTraining ne dépasse les 90%, c'est pour dire). Par contre, les résultats nous montrent très bien que même pour classer des données linguistiques, voire psychologiques, les statistiques sont déjà une aide précieuse !
Troisièmement, ce projet souligne que la complexité de algorithme ne le rend pas forcément meilleur ! Il faut 1min30 à 3.2 pour classifier les tweets contre 5min pour 3.10 et chacun n'obtient qu'une différence minime de 2%. De plus, la complexité de 3.9 et de 3.10 est dangereuse pour l'intégrité de la machine. L'actif Learning n'est certainement pas la meilleure des solutions pour ce type de problème ! L'apprentissage est minime et ajoute seulement de la complexité en temps. C'est malheureux pour ce cas-là, mais c'est ainsi que la science avance aussi, en faisant des erreurs et en explorant des hypothèses inutiles. On sait maintenant que ce n'est pas la meilleure technique d'approche avec ces modèles.
En somme, le machine Learning en est encore à son balbutiement et la gestion informatique de données linguistiques est un champ énorme à explorer, surtout à l'heure des réseaux sociaux et de la production d'articles sur le net ! Personnellement, ce projet aura suscité beaucoup d'intérêt pour ces deux disciplines !

SOURCES:

pmiLexicon tiré de cette page :
http://saifmohammad.com/WebPages/Abstracts/NRC-SentimentAnalysis.htm

stopwords tirés de cette page :
http://www.ranks.nl/stopwords

Inspiration pour le second modèle :
http://www.aclweb.org/anthology/S/S14/S14-2.pdf#page=93
&
http://www.aclweb.org/anthology/S14-2#page=463

Annexes n°1 :

1 Tableau des résultats des tests de développement :

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10	Moyenne	TestTrain
P3.0	0.4711	0.4556	0.4594	0.4559	0.4584	0.4508	0.4676	0.4696	0.4555	0.4651	0.4609	0.8464
AP3.1	0.4654	0.4577	0.45	0.464	0.4721	0.4485	0.4606	0.4573	0.4602	0.452	0.4587	0.8504
AP3.2	0.5267	0.5201	0.5196	0.5244	0.5276	0.517	0.5209	0.5246	0.502	0.5184	0.5201	0.8754
AP3.3	0.5193	0.5202	0.5195	0.5265	0.5292	0.511	0.5152	0.5153	0.515	0.5125	0.5183	0.8626
AP3.4	0.2586	0.2586	0.2586	0.2586	0.2586	0.2586	0.2586	0.2586	0.2586	0.2586	0.2586	0.2721
AP3.5	0.1676	0.1676	0.1676	0.1676	0.1676	0.1676	0.1676	0.1676	0.1676	0.1676	0.1676	0.1225
AP3.6				PAS	DE	DONNÉES	POUR	CETTE	VERSION			
AP3.7	0.4944	0.4944	0.4944	0.4944	0.4944	0.4944	0.4944	0.4944	0.4944	0.4944	0.4944	0.6939
AP3.8	0.4996	0.5245	0.5087	0.5106	0.5081	0.5273	0.5102	0.5048	0.5156	0.5027	0.5112	0.7801
AP3.9	0.5056	0.5426	0.503	0.5235	0.5147	0.5445	0.5279	SURCHAUFFE	DU	CPU	0.5231*	0.7822
AL3.10	0.5328	0.5322	0.5272	0.5406	0.5321	0.5267	0.546	0.5457	0.5292	0.5357	0.5348	0.8634

P : perceptron ; AP : average perceptron ; AL : actif learning ; max d'une ligne ; min d'une ligne ; max d'une colonne ; min d'une colonne ; * : moyenne sur 7 éléments

2 Résultats des tests finaux :

	Développement	T13	T14	TS
AP3.2	0.5201	0.5261	0.5264	0.3786
AP3.3	0.5183	0.5116	0.5369	0.4212
AL3.10	0.5348	0.5324	0.5531	0.4616
AL3.10sar	//	0.5295	0.5561	0.3937
4 Mod1	0.5353	0.5037	0.5154	0.3815
4 Mod2	0.4876	//	//	//
4 AL	0.5304	0.4933	0.5248	0.3989

// : pareil qu'au-dessus.

Une malheureuse manipulation m'a fait certainement perdre les éléments de test 4 Mod2 et remplacé par 4 Mod1 car il me parait très improbable qu'ils aient les même résultats ! De plus le second modèle n'est pas vraiment bon et n'excède pas les 50% d'habitude !

3 Comparaison des résultats des étapes :

	Dev	T13	T14	TS
Baseline	0.5029	0.4473	0.4581	0.3925
Perceptron	0.4808	0.4703	0.4541	0.3812
Amélioration	0.5348	0.5324	0.5531	0.4616

Ici amélioration réfère bien sur au meilleur résultat que l'on a obtenu, c'est-à-dire bien sur AL3.10 !

La tache d'amélioration est donc un succès !