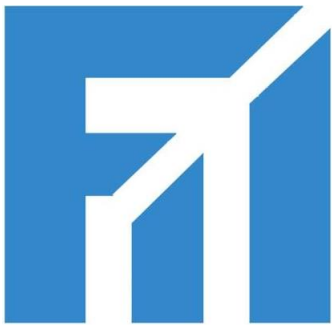
Universitatea "Alexandru Ioan Cuza" din Iaşi

FACULTATEA DE INFORMATICĂ



LUCRARE DE LICENȚĂ

Tehnici de clasificare text în domeniul culinar

propusă de

Mircea Rareș - Gabriel

Sesiunea: *iulie, 2019*

Coordonator științific

Conf. Dr. Răschip Mădălina

Universitatea "Alexandru Ioan Cuza" din Iaşi

FACULTATEA DE INFORMATICĂ

LUCRARE DE LICENȚĂ

Tehnici de clasificare text în domeniul culinar

propusă de

Mircea Rareș - Gabriel

Sesiunea: *iulie, 2019*

Coordonator științific

Conf. Dr. Răschip Mădălina

Contents

1 Descrierea problemei **2**
- 1.1 Metode existente . . . 2
- 1.2 Setul de date . . . 3

2 Reprezentarea **4**
- 2.1 Bag-of-Words . . . 4
- 2.2 Tf-Idf . . . 5
- 2.3 Word-2-Vec . . . 5

3 Algoritmii de clasificare **6**
- 3.1 Linear SVC . . . 6
- 3.2 Regresia Logistică . . . 6
- 3.3 Random Forrest . . . 9

4 Descrierea metodei proprii **10**
- 4.1 Setul de date . . . 10
- 4.2 Procesarea datelor . . . 11
  - 4.2.1 Tf-Idf . . . 12
  - 4.2.2 Bag-Of-Words . . . 12
  - 4.2.3 T . . . 12
- 4.3 Utilizarea algoritmilor . . . 14
  - 4.3.1 Linear SVC . . . 14
  - 4.3.2 Regresie Logistică . . . 14
  - 4.3.3 Random Forrest . . . 14
- 4.4 Reprezentări grafice . . . 15
  - 4.4.1 UMAP . . . 15
  - 4.4.2 T-SNE . . . 17
- 4.5 Limbajul folosit . . . 20
- 4.6 Biblioteci folosite . . . 20
  - 4.6.1 Sklearn . . . 20
  - 4.6.2 BeautifulSoup . . . 20
  - 4.6.3 Gensim . . . 20
  - 4.6.4 Pandas . . . 20
  - 4.6.5 Plotlib . . . 21
- 4.7 Rezultatele . . . 21

5 Concluzii **22**

6 Bibliografie **23**

1 Descrierea problemei

Mâncarea stă la baza civilizației umane, face parte din fundația piramidei nevoilor a lui $Maslow$ fiind una dintre nevoie de bază a omului. Încă din faza incipientă a omului, hrana a fost una dintre țintele de bază a societății, împreună cu adăpostul și reproducerea speciei. De-a lungul timpului, noi am evoluat, am migrat pe toate cele 7 continente, unde au apărut civilizații care mai de care diferite, acestea venind fiecare cu o anumită perspectivă asupra mâncării.

Perspectiva asta a rămas neschimbata odată cu trecerea timpului, fiecare civilizație rămânând fidelă ei, anii aducând mici îmbunătățiri la nivel culinar, sau chiar rețete noi. Dacă până la sfârșitul secoluilui XX umanitatea încă era separată de ziduri și cortine, odată cu intrarea in mileniul III, a început și procesul de globalizare, unde mai fiecare cultură mai bogată a reușit să exporteze niște influențe culinare celorlalte.

Având un număr generos de bucătării disponibile și pentru fiecare dintre acestea cel putin 200 de rețete, m-am decis sa realizez o clasificare pe text a acestor rețete și în funcție de modul de preparare a fiecăreia să o clasific cât mai corect.

Pentru a face asta, mă voi folosi de limbajul Python, care pune la dispoziție o suită de biblioteci pentru Machine Learning, cum ar fi *scikit*, de unde am folosit *sklearn* pentru algoritmii de extragere de trăsături și cei de clasificare, *matplotlib* pentru graficele care o să-mi vina în ajutor la descrierea problemei, *pandas* pentru a lucra mai ușor cu fișierele de tip $json$ și în ultimul rând, $BeautifulSoup$, pe care l-am folosit la crearea setului de date, extragând date de pe internet.

Scopul lucrării este de a folosi acești algoritmi și biblioteci în a calcula acuratețea în domeniul clasificării text pe rețete, astfel încât nu ar mai trebui implementați de la 0, unde s-ar fi pus si problema corectitudinii și a eficienței.

1.1 Metode existente

După o căutare riguroasă în domeniul clasificării rețetelor, am găsit unele lucrări de specialitate dar care propuneau fie clasificarea rețetelor în funcție de ingredientele care le conțineau iar altele propuneau o clasificare a rețetelor în funcție de imagini, cu ajutorul rețelelor neuronale convoluționale. Au fost unele lucrări, care deși lucrau cu rețete, propuneau cu totul altă soluție, fapt ce nu mă ajuta prea mult.

Trebuie precizat că cele care făceau clasificare text pe ingredientele rețetelor, m-au ajutat să vizualizez și să înțeleg pașii necesari în realizarea acestei lucrări. Dat fiind faptul că aceste lucrări foloseau seturi de date cât mai diferite unul de celalalt, am fost nevoit să consider crearea propriului meu set de date, unde voi putea face clasificare text pe pașii rețetei, sau instrucțiunile sale.

1.2 Setul de date

Având în vedere că nu puteam sa folosesc niciun dataset din lucrările enumerate mai sus, niciunul nefiind conform specificațiilor impuse de mine, am decis să utilizez un crawler pe un site de rețete, unde puteam parcurge paginile acestui site și sa-mi extrag de pe fiecare rețetă în parte ce aveam nevoie.

O rețetă e era reprezentată de o pagină nouă, de unde îmi extrăgeam instrucțiunile de preparare, ingredientele și numele acesteia, le puneam într-o listă de dicționare, pe care la final o exportam într-un fișier JSON. Spre finalul acestui procedeu am ajuns la o sumă de aproximativ 5000 rețete de la 15 bucătării, având cel puțin vreo 150 de rețete pe fiecare bucătărie în parte

2 Reprezentarea

Pentru reprezentare, am folosit algoritmi din categoria de extragere de trăsături (*Information Retrieval*) cum ar fi Tf-Idf, Bag-of-Words și Word-2-Vec. Această extragere de trăsături constă într-o căutare de informații la nivelul unei propo-ziții, text, document și o suită de documente care se concretizează într-un așa zis *corpus*.

Având în vedere faptul că acești algoritmi au același scop, aceștia au si cam același parcurs în execuția lor. La început, după ce se face inițializarea lor, se trece la etapa de tokenizare, în care fiecare cuvânt este considerat un token și îi este atribuit un id(spațiile libere și semenele de punctuație sunt considerate separatori).

După etapa de tokenizare, se trece la etapa de numărare unde fiecare algoritm își aplică propria metodă de calcul, de exemplu, Bag-of-Words numără frecvența cuvintelor într-un text, după care se trece la etapa finală de vectorizare, unde se trasnformă aceasta colecție de texte intr-un vector de trăsături a acestora.

2.1 Bag-of-Words

În primul rând, ar trebui explicată denumirea acestui algoritm; Ea provine de la faptul că fiecare text, cum ar o propoziție sau frază, este reprezentată de un multiset(*bag*) de cuvinte, indifierent de gramatică sau ordine, în care se păstrează valorile într-o structură de tip dicționar(cheie - valoare), unde cheia este cuvântul în sine, iar valoarea este frecvența acestui cuvânt în textul respectiv.

Exemplu: *Ana are mere și George are pere.*, output-ul $bag - of - words$ va fi:

{Ana: 1, are: 2, mere: 1, si: 1, George: 1, pere: 1}

În cadrul bag of words, este introdus si termenul de $n - gram$, care incearcă grupeze cuvintele din text în perechi de cuvinte succesive de câte n și va număra frecvența acestora din $n - grame$. Pentru a înțelege mai usor conceptul, voi exemplifica mai jos, folosind exemplul anterior:

Exemplu: *Ana are mere și George are pere.*, bi-gramele(2-gramele) vor fi:

{Ana are,
are mere,
mere și,
și George,
George are,
are pere}

2.2 Tf-Idf

Algoritmul tf-idf provine în engleză de la *"term frequency-inverse document frequency"*, adică acesta vrea să arate cât de important este un cuvânt într-un corpus, astfel încat valoarea tf-idf, este proporțională cu frecvența cuvântului în text și invers proporțională cu numărul de texte din corpus care conțin acel cuvânt.

Pentru a putea înțelege mai bine acest algoritm, ne uităm la formulele matematice a celor două componente care îl alcătuiesc.

Începem cu *"term-frequency"* :

$$tf(t, d) = f_{t,d}$$

unde $f_{t,d}$ se traduce în frecvența cuvântului respectiv, t, în textul d.

Apoi avem *"inverse document frequency"*:

$$idf(t, D) = log \frac{N}{1+|\{d \in D : t \in d\}|}$$

, unde N este numarul de texte în corpus și $|\{d \in D : t \in d\}|$ este numărul de texte în care apare tokenul t, în cazul în care tokenul nu se va regăsi in niciun text, acea formula va da 0, astfel fiind nevoie de a adăuga 1+ pentru a nu avea nicio împărțire la 0.

Punând cap la cap formulele de mai jos, va rezulta formula finala la tf-idf, aceea fiind: $tfidf(t, d, D) = tf(t, d) \times idf(t, D)$.

2.3 Word-2-Vec

După cum spune și numele, acest algoritm preia un corpus de texte(*Word*) pe care îl transformă într-un spațiu vectorial (*2-Vec*), reprezentat de vectorii de trăsături a acelor cuvinte din corpus. Vectorii de trăsături care împart trăsături comune, sunt plasate de obicei cat mai aproape unul de celălalt.

În implementarea sa, sunt folosite rețelele neuronale cu două straturi

3 Algoritmii de clasificare

Următoarea etapă, de după cea de extragere de trăsături, este cea de clasificare. Aceasta clasificare se face în baza vectorilor de trăsături rezultați la etapa anterioara, unde aplicăm si setului de antrenare și setului de testare transformarea specifică fiecărui algoritm de extragere de trăsături.

3.1 Linear SVC

Pentru a putea discuta despre Linear SVC(*Support Vector Classifier*), va trebui să intrăm puțin în detalii cu SVM(*Support Vector Machines*). Acesta este un algoritm de învățare supervizată folosit în separa instanțele aparținând unor clase diferite într-un spațiu multidimensional.

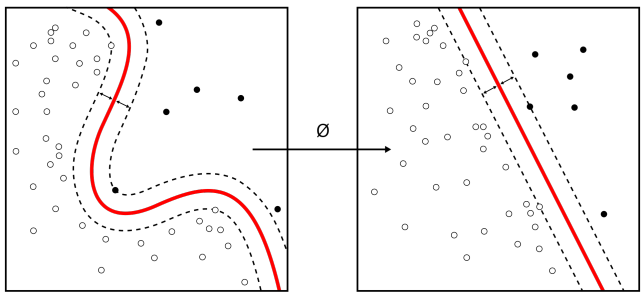
Mai sus am utilizat termenul separare, deoarece output-ul SVM-ului este reprezentat de un hiperplan, sau mai multe, cu ajutor cărora putem separa valori care ar părea imposibil de separat într-un spațiu bi-dimensional.

Find vorba de linearitate, SVM se folosește de hiperplan cu margine-maximă, adică acesta va căuta cele mai apropiate valori din două clase diferite pe care va construi hiperplanul, maximizându-se în așa fel distanța dintre cele două clase. Valorile folosite pentru a stabili acest hiperplan, sunt numite *vectori suport*.

Am ales acest algoritm datorită faptului că este eficient în spații cu multe dimensiuni, deoarece în timpul extragerii de trăsături, se poate ajunge si la valori mari (undeva pe la 3500).

Linear SVC este diferit de SVM prin două caracteristici, prima fiind cea că acest algoritm foloseste o funcție kernel lineară și că poate primi și vectori de trăsături denși sau rari, fapt care ne va fi de ajutor mai încolo când vom trece la etapa de implementare.

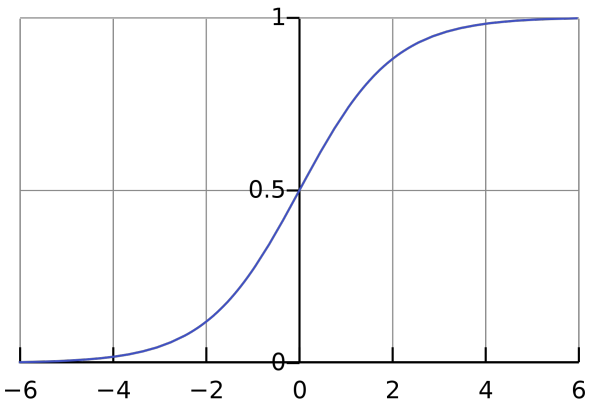
Mai jos este un exemplu de hiperplan în SVM.



3.2 Regresia Logistică

Regresia logistică este un model care se bazează pe funcția logistică, sau sigmoid:

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Acest algoritm măsoară relația dintre variabilele categoric dependente și una sau mai multe variabile independente estimând probabilitățile folosind o funcție logisitică. Variabilele depedente sunt reprezentate de clasa țintă pe care noi trebuie să o prezicem, așadar varibilele independente sunt trăsăturile de care ne vom folosi pe a prezice clasa țintă.

Având în vedere faptul că în setul nostru de date este împărțit in 15 clase, putem discuta despre o problemă de *multi-clasificare*, astfel încât vom folosi o particularitate a acestui algoritm, numit și regresie logistică multinominală(cunoscută și sub numele de *MaxEnt*.

Așadar, nu mai poate fi vorba de variabile dependente binare, ci mai degrabă categorice, deoarece se va calcula posibilitatea a n valori în loc de doar două.

Calculele pentru aceasta sunt următoarele:

$$lnP(Y_i = 1) = \beta_1 \cdot X_i - lnZ$$
$$lnP(Y_i = 2) = \beta_2 \cdot X_i - lnZ$$
$$\cdots\cdots\cdots$$
$$lnP(Y_i = K) = \beta_k \cdot X_i - lnZ,$$

unde β_i este factorul de regresie, atribuit fiecărei valori de input, X_i este valoarea input iar $-LnZ$ este factorul de normalizare, pentru a ne asigura că suma probabilităților de mai sus este egală cu 1:

$$\sum_{k=1}^{K} P(Y_i = k) = 1$$

Putem transforma ecuația probabilităților prin scoaterea logaritmului:

$$P(Y_i = 1) = \frac{1}{Z} e^{\beta_1 \cdot X_i}$$
$$P(Y_i = 2) = \frac{1}{Z} e^{\beta_2 \cdot X_i}$$
$$\text{.....}$$
$$P(Y_i = K) = \frac{1}{Z} e^{\beta_k \cdot X_i} (1),$$

Putem extrage valoarea lui Z din suma probabilităților, astfel:

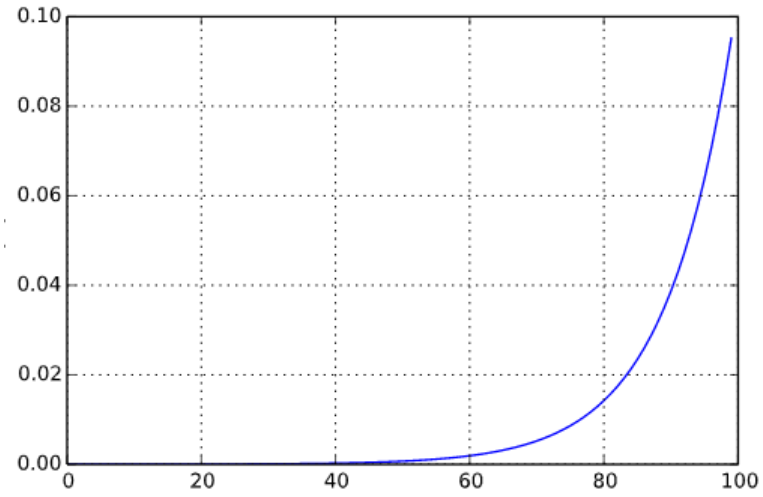
$$\sum_{k=1}^{K} P(Y_i = k) = 1 => \sum_{k=1}^{K} \frac{1}{Z} e^{\beta_k \cdot X_i} => \frac{1}{Z} \sum_{k=1}^{K} e^{\beta_k \cdot X_i}$$
$$=> Z = \sum_{k=1}^{K} e^{\beta_k \cdot X_i} (2)$$

Din (1) și (2) putem calcula forma finala a probabilităților:

$$P(Y_i = c) = \frac{e^{\beta_c \cdot X_i}}{\sum_{k=1}^{K} e^{\beta_k \cdot X_i}}$$

Se observă că forma generală a probabilităților seamănă cu cea a funției softmax:

$$softmax(k, x_i) = \frac{e^{x_k}}{\sum_{i=1}^{n} e^{x_i}}$$



graficul funcției softmax

Softmax are rolul de a "exagera" diferențele dintre valorile de input, astfel încat $softmax(k, x_i)$ va returna o valoare aproape de 0 atunci când x_i este destul de mic față de maximul tuturor valorilor și va returna o valoare apropiată de 1 x_i sunt destul de aprope de maximul respectiv. Așadar, la final se va obține o aproximare a funcției indicator:

$$f(k) = \begin{cases} 1 & \text{k = argmax}(x_1,...,x_k), \\ 0 & \text{altfel} \end{cases}$$

3.3 Random Forrest

Urmeaza sa intru in detalii

4 Descrierea metodei proprii

În următoarele puncte voi discuta despre cum am ajuns la o soluție cu ajutorul algoritmilor explicați înainte, și cei de clasificare și cei de reprezentare. Aceasta a constat în mai multi pași, fiecare etapă importanta, fiind explcată în cadrul unui subpunct.

4.1 Setul de date

Setul de date a fost generat de la 0, de către mine, cu ajutorul unei biblioteci din Python, BeautifulSoup și constă intr-un numar de 15 bucătării și aproximativ vreo 5000 rețete culese pe de site-ul www.allrecipes.com.

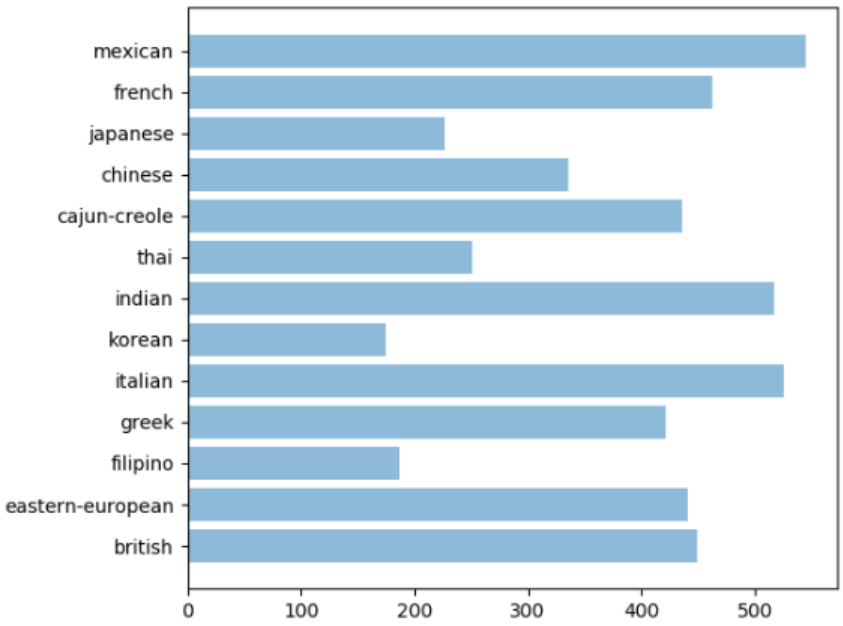
Bucătăriile care fac parte din acest set de date sunt:

- French
- Eastern European
- Italian
- British
- Indian
- Chinese
- Korean
- Mexican
- Cajun-Creole
- Thai
- Greek
- sss
- Filipino
- Japanese
- sss

Acest set de date constă intr-un id, nume, tipul bucătăriei, o listă de ingrediente și metoda de preparare, după cum se poate vedea mai jos.


```
"id": 7456,
"title": "Spicy Creamy Cajun Ham and Black Eyed Peas Salad",
"cuisine": "cajun-creole",
"ingredients": [
        "2 cups fresh corn kernels",
        "2 (15 ounce) cans black-eyed peas, rinsed and drained",
        "1 cup cubed fully cooked ham",
        "3 stalks celery, finely chopped",
        "2 tablespoons chopped red onion",
        "2/3 cup sour cream",
        "1 tablespoon ketchup",
        "1 tablespoon dried cilantro",
        "1 teaspoon Cajun seasoning",
        "2 dashes hot pepper sauce (such as Tabasco\u00ae), or to taste"
],
"directions": "Place the corn into a saucepan, cover with water, and
bring to a boil. Reduce heat and simmer until the corn is fully cooked,
about 2minutes. Drain the corn in a colander set in the sink. Mix together
the warmcorn, black-eyed peas, ham, celery, and onion in a salad bowl.
Whisk together the sour cream, ketchup, cilantro, Cajun seasoning, and hot
pepper sauce in a bowl until smooth. Stir the dressing lightly into the
```

black−eyed pea mixtureuntil thoroughly mixed. Serve immediately."



repartizarea rețetelor pe bucătării

4.2 Procesarea datelor

La această etapă ne vom folosi de setul de date creat anterior și îl vom trimite mai departe algoritmilor care vor procesa acest corpus și ne vor oferi vectorii de trăsături caracteristici fiecăruia. Înainte de a începe procesarea propriu-zisă, mai intâi voi lua din fișierul JSON datele cu biblioteca *pandas* într-o structură de tip *DataFrame* care îmi va permite să lucrez mai usor cu rețetele stocate în acest JSON.

Având în vedere faptul că am aproape 5000 de rețete în setul de date voi folosi în jur de 90% drept set de antrenare, iar restul de 10% pentru testare, deci undeva la 4500 și 500.

După ce am realizat împărțirea setului mare de date în două mai mici, unul de antrenare și unul de testare, le voi trimite mai departe la etapa de extragere de trăsături, reprezentată de algoritmii descriși anterior la capitolul 2.

Pentru fiecare în parte dintre acești algoritmi, am realizat o ajustare a hiperparametrilor pentru a fi siguri că acuratețea clasificării este cât mai mare. Pentru a face o paralelă cu lumea noastră, putem spune că ajustarea hiperparametrilor este ca și cum am fi o trupă de cântăreți care își ajustează instrumentele înainte de concert, cu cât sunt mai bine acordate, cu atât ne putem asigura de o calitate cât mai mare a concertului.

4.2.1 Tf-Idf

```
vectorizer = TfidfVectorizer(stop_words='english',
                             ngram_range=(1, 1),
                             min_df=4)
```

În cazul Tf-Idf, inițial ne-am asigurat că am scos cuvintele de legătură(acel $stop - words$), deoarece, acestea nu sunt relevante în a determina o rețetă, ele făcând pur și simplu legătura dintre două propoziții dintr-o frază sau a unor părți de vorbire.

După aceea, nu am folosit modelul de n-grame($(1,1)$ semnifică faptul că un cuvânt, reprezintă o 1-gramă), deoarece am observa că acest algoritm tinde să dea rezultate mult mai bune în felul acesta și l-am grupat cu acel min_df care face referire la frecvența minimă a unui token într-un document.

Având algoritmul instanțiat, urmează etapa de propriu zisă de extragere de trăsături, unde vom creaa pe rând vectorii de trăsături pentru instanțele de antrenare și de testare.

```
corpus_train = train['directions']
tfidf_vect = vectorizer.fit(corpus_train)
tfidf_train = tfidf_vect.transform(corpus_train)

corpus_test = test['directions']
tfidf_test = tfidf_vect.transform(corpus_test)
```

4.2.2 Bag-Of-Words

```
vectorizer = CountVectorizer(stop_words='english',
                             ngram_range=(1, 2),
                             min_df=5)
```

Urmărind tiparul de la $Tf - Idf$, am eliminat cuvintele de legătură și după prin încercări succesive, am reușit să ajung la hiperparametrii ideali, din punctul de vedere al setului nostru de date.

De data aceasta, algoritmul nostru de extragere de trăsături va considera unigramele si bigramele și pe lângă acestea, numărul minim de apariții în document al tokenurilor este de 5.

```
corpus_train = train['directions']
bow_vect = vectorizer.fit(corpus_train)
bow_train = bow_vect.transform(corpus_train)

corpus_test = test['directions']
bow_test = bow_vect.transform(corpus_test)
```

4.2.3 T

rebuie explicat cum m-au ajutat functiile astea, si trebuie să realizez o ajustare de hiperparametri

```
def get_word2vec(sentences, location):
    if os.path.exists(location):
        print('Found_{}'.format(location))
        model = gensim.models.Word2Vec.load(location)
        return model

    model = gensim.models.Word2Vec(
                    sentences,
                    size=250,
                    window=5,
                    min_count=1,
                    workers=4)
    model.save(location)
    return model
```

%%%%%%%%%%%%%%%%%%% %%%%%%%%%%%%%%%%%%%

```
def fit(self, X, y=None):
return self

def transform(self, X):
    X = MyTokenizer().fit_transform(X)

    return np.array([
        np.mean([
            self.word2vec.wv[w] for w in words
                    if w in self.word2vec.wv
        ]
            or [np.zeros(self.dim)], axis=0)
        for words in X
    ])

def fit_transform(self, X, y=None):
    return self.transform(X)
```

%%%%%%%%%%%%%%%%%%% %%%%%%%%%%%%%%%%%%%

```
class MySentences(object):
    def __init__(self, *arrays):
        self.arrays = arrays

    def __iter__(self):
        for array in self.arrays:
            for document in array:
                for sent in nltk.sent_tokenize(document):
                    yield nltk.word_tokenize(sent)
```

4.3 Utilizarea algoritmilor

După procesarea de la etapa anterioară, având vectorii de trăsături creați, putem trece mai departe la etapa de propriu-zisă de clasificare. Aici, pentru fiecare algoritm de clasificare în parte, vom ajusta hiperparametrii pentru a obține o clasificare cat mai bună din punct de vedere al acurateței.

Mai jos voi explica modul de utilizare al algoritmilor pentru fiecare în parte împreună cu hiperparametrii folosiți de către aceștia.

Hiperparametrii au fost reglați pentru fiecare tip de reprezentare în parte, pentru a maximiza acuratețea acestora.

4.3.1 Linear SVC

Parametrul C trasmite algoritmului care să fie rația corecției la clasificare, în sensul în care cu cât este mai mare această valoare, cu cât hiperplanul calsifica mai bine punctele de antrenare, cu atât mai mult va fi aleasă o margine mai mică a hiperplanului. Invers, cu cât valoarea lui C este mia mică, cu atât marignea hiperplanului va fi mai mare, deși valorile de antrenare sunt linear separabile, chit că acuratețea va scădea.

```
classifier_svc = LinearSVC(C)
classifier_svc.fit(train, classes)
prediction_svc = classifier_svc.predict(test)
```

Pentru fiecare reprezentare, am folosit următoarele valori ale lui C:

- Tf-Idf : 0.5
- Bag-of-Words : 0.1
- Word-2-Vec : 25

4.3.2 Regresie Logistică

```
classifier_regr = LogisticRegression(C)
classifier_regr.fit(train, classes)
prediction_regr = classifier_regr.predict(test)
```

Pentru fiecare reprezentare, am folosit următoarele valori ale lui C:

- Tf-Idf : 10
- Bag-of-Words : 1
- Word-2-Vec : 1000

4.3.3 Random Forrest

```
classfier_rf = RandomForestClassifier(
                    min_samples_split,
                    n_estimators,
                    max_depth,
                    max_features,
                    bootstrap,
                    min_samples_leaf)
classfier_rf.fit(train, classes)
prediction_nb = classfier_rf.predict(test)
```

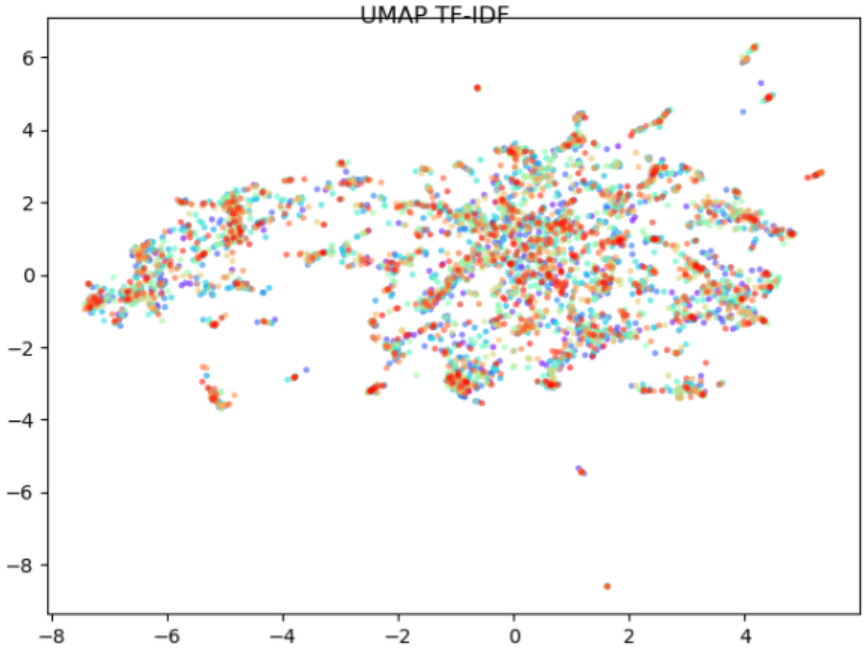
Ajustări			
Parametri	Tf-Idf	Bag-of-Words	Word-2-Vec
min_samples_split	5	2	2
n_estimators	1400	1400	1000
max_depth	80	40	50
max_features	'sqrt'	'auto'	'auto'
bootstrap	False	False	False
min_samples_leaf	1	1	1

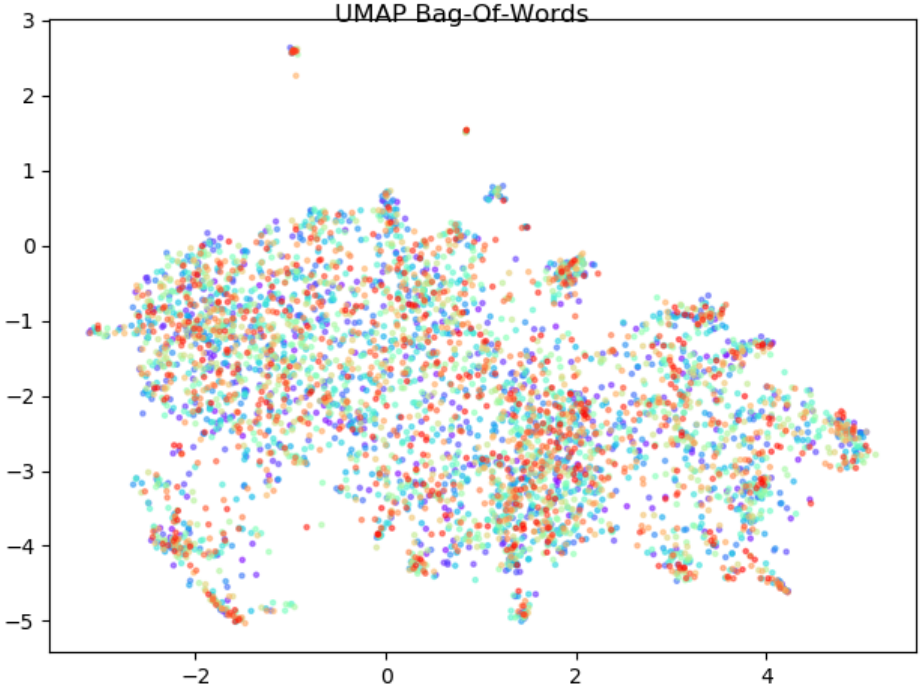
4.4 Reprezentări grafice

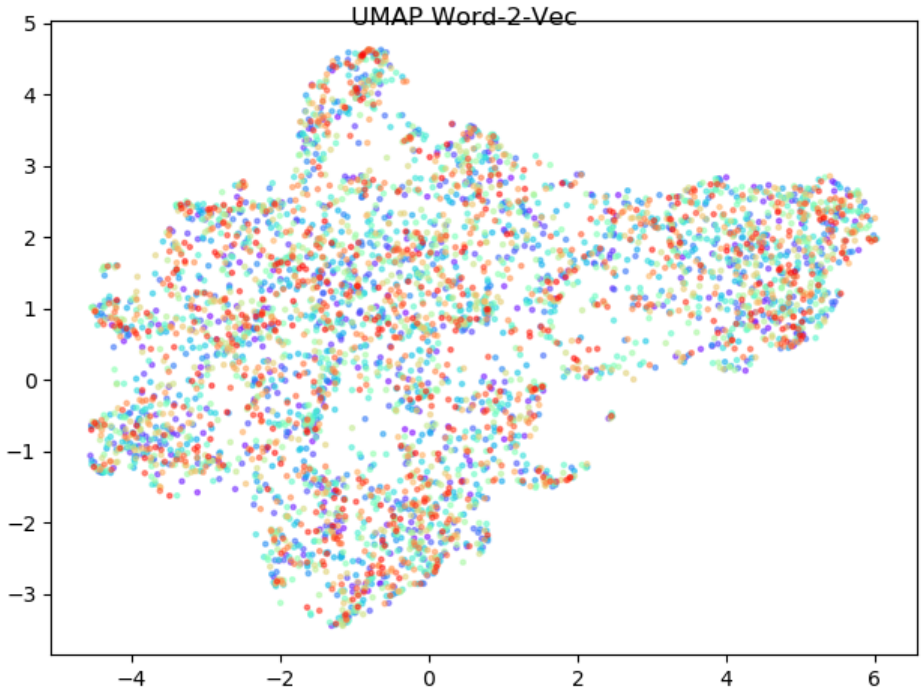
Avem setul de date trecut prin algoritmii de clasificare, astfel încat am putea genera o reprezentare grafică cu ajutorul urmatorilor algoritmi și a valorilor rezultate in urma procesării datelor.
Vom reduce la 2 dimensiuni, pentru a putea sa îi dăm drept input K-Means-ului un set de date cât mai redus din punct de vedere dimensional.

4.4.1 UMAP

```
umap = UMAP()
reduced_data = umap.fit_transform(data)
```

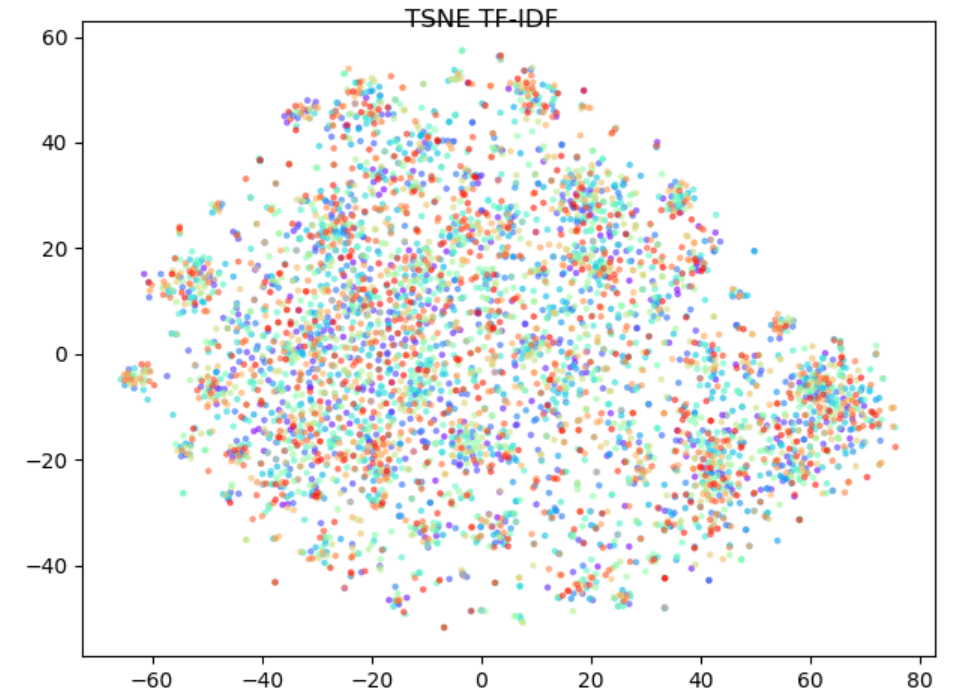

UMAP TF-IDF
6
4
2
0
−2
−4
−6
−8
−8
−6
−4
−2
0
2
4

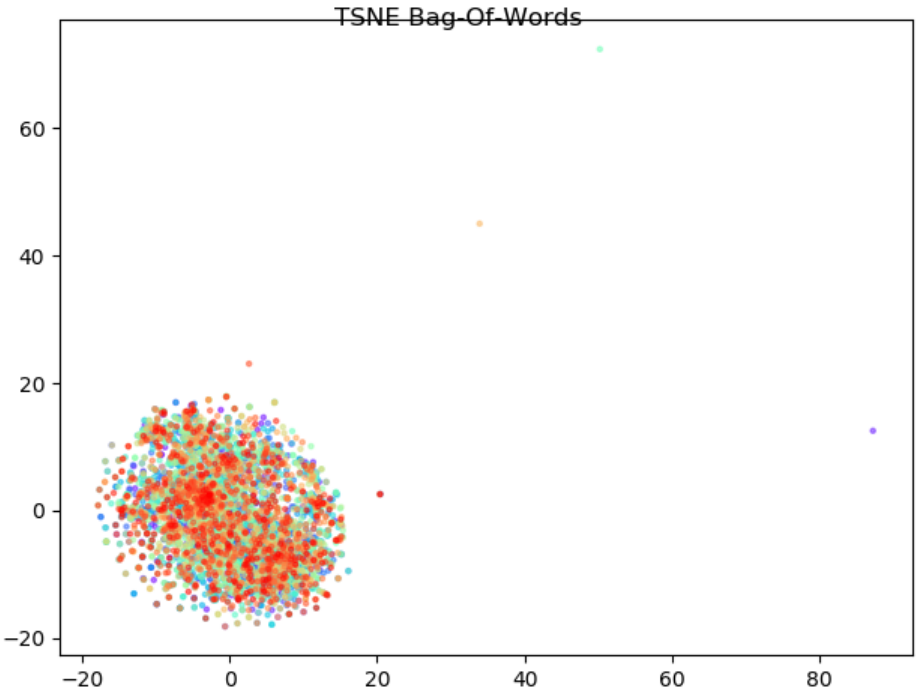

UMAP Bag-Of-Words
3
2
1
0
−1
−2
−3
−4
−5
−2
0
2
4

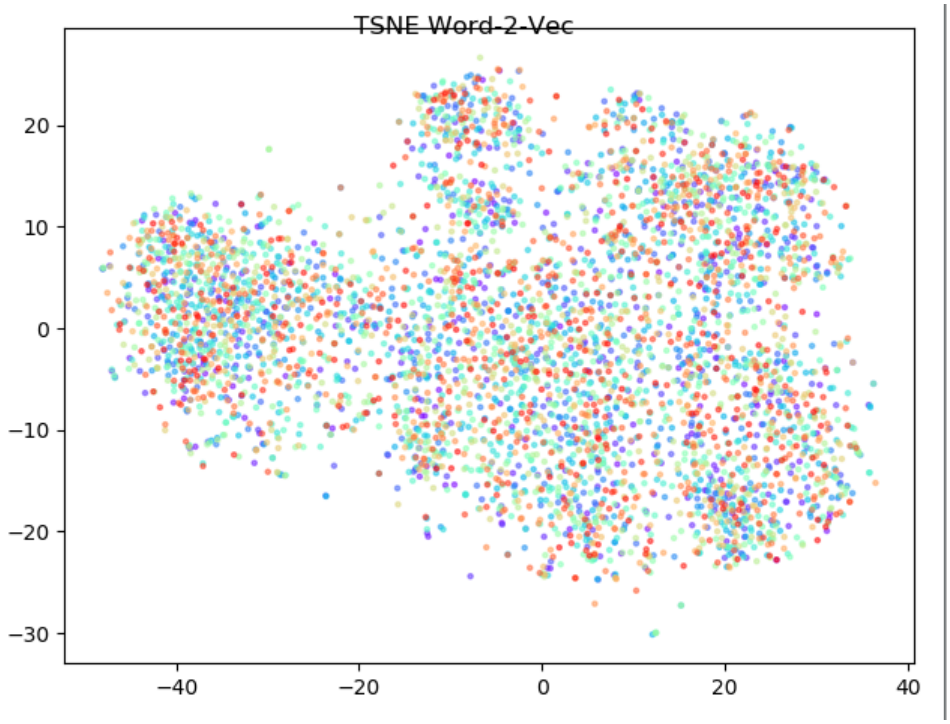


4.4.2 T-SNE

```
tsne = TSNE()
reduced_data = tsne.fit_transform(data.toarray())
```


TSNE TF-IDF
60
40
20
0
−20
−40
−60
−40
−20
0
20
40
60
80


TSNE Bag-Of-Words
60
40
20
0
−20
−20
0
20
40
60
80


TSNE Word-2-Vec
20
10
0
−10
−20
−30
−40
−20
0
20
40

4.5 Limbajul folosit

Limbajul folost este Python, un limbaj interpretat, care este destul de prietenos și care în ultima perioadă a căpătat destul de multă popularitate, tocmai datorită faptului ca este destul de simplu de învațat și că are o multitudine de aplicații, cum ar fi domeniul de Data Science si Machine Learning cu vasta suită de biblioteci pe care pe lune la dispoziție(printre care sunt folosite și în realizarea acestei lucrări) .

4.6 Biblioteci folosite

4.6.1 Sklearn

Algoritmii, tfidf, countvectorizer, svd, tsne

4.6.2 BeautifulSoup

```
webpage = requests.get(link)
soup = BeautifulSoup(webpage.content, features="html.parser")

links = soup.find_all("article", {"class": "fixed-recipe-card"})

 new_soup = BeautifulSoup(new_webpage.content, features="html.parser")

title = re.split(" Recipe - ", new_soup.title.text)[0]

directions = new_soup.find_all("span", {"class":
                "recipe-directions__list--item"})
ingredients = new_soup.find_all("span", {"class":
                "recipe-ingred_txt added"})

for direction in directions:
    direction = direction.text.replace("\n", "")

    if direction != "":
        directions_list.append(direction)

for ingredient in ingredients:
    ingredients_list.append(ingredient.next)
```

crawer, explicat cum am luat

4.6.3 Gensim

folosit cu w2v

4.6.4 Pandas

folosit pentru jsoane

4.6.5 Plotlib

folosit sa fac reprezentarea grafica cu kmeans + svd/tsne

4.7 Rezultatele

Cu ajutorul ajustării hiperparametrilor, am reușit să cresc sa aduc rezultatele la o anumită consistență, în sensul în care pentru toate cazurile de vectori de trăsături si algoritmi de clasificare, acuratețea oscilează între 70-77%.

Inițial, am uitilizat toți algoritmii cu hiperparametrii default, ceea ce nu este recomandat, deoarece am avut rezultate destul de proaste, cum ar fi:

- SVC: aveam rezultate in intervalul 71%-79%, dar majoritatea rezultatelor se învârteau în zona de 73%-74%
- Regresia Logistică: rezultatele se aflau în intervalul 65-75, cu o majoritate în zona 69-70
- RandomForrest: aici au fost notate cele mai bune îmbunătățiri, deoarece, inițial RandomForrest avea acuratețe în intervalul 35-50%, după modificările impuse de ajustarea hiperparametrilor, am ajuns la rezultate în zona 70-76

Urmează sa pun niște grafice cu media la 10 iterații(să spunem) pentru fiecare algoritm în parte + pentru fiecare reprezentare, deci undeva pe la 9 grafice

5 Concluzii

6 Bibliografie

References

[1] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1993.

[2] Albert Einstein. *Zur Elektrodynamik bewegter Körper*. (German) [*On the electrodynamics of moving bodies*]. Annalen der Physik, 322(10):891–921, 1905.

[3] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`