

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 IAȘI

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

# Cuprins

<b>1 Descrierea problemei</b>	<b>2</b>
1.1 Metode existente . . . . .	2
1.2 Setul de date . . . . .	3
<b>2 Extragerea trăsăturilor</b>	<b>4</b>
2.1 Bag-of-Words . . . . .	4
2.2 Tf-Idf . . . . .	5
2.3 Word-2-Vec . . . . .	5
<b>3 Algoritmii de clasificare</b>	<b>8</b>
3.1 Linear SVC . . . . .	8
3.2 Regresia Logistică . . . . .	8
3.3 Random Forest . . . . .	11
<b>4 Descrierea metodei proprii</b>	<b>13</b>
4.1 Setul de date . . . . .	13
4.2 Procesarea datelor . . . . .	14
4.2.1 Tf-Idf . . . . .	15
4.2.2 Bag-Of-Words . . . . .	15
4.2.3 Word-2-Vec . . . . .	16
4.3 Utilizarea algoritmilor . . . . .	16
4.3.1 Linear SVC . . . . .	17
4.3.2 Regresie Logistică . . . . .	17
4.3.3 Random Forest . . . . .	17
4.4 Reprezentări grafice . . . . .	19
4.4.1 UMAP . . . . .	19
4.4.2 T-SNE . . . . .	21
4.5 Tehnologii folosite . . . . .	24
4.5.1 Python . . . . .	24
4.5.2 SciKit-Learn . . . . .	25
4.5.3 BeautifulSoup . . . . .	25
4.5.4 Gensim . . . . .	26
4.5.5 Pandas . . . . .	26
4.5.6 Plotlib . . . . .	27
4.6 Rezultatele . . . . .	27
<b>5 Concluzii</b>	<b>28</b>
<b>6 Bibliografie</b>	<b>29</b>

# 1 Descrierea problemei

Mâncarea stă la baza civilizației umane, face parte din fundația piramidei nevoilor a lui *Maslow* fiind una dintre nevoile de bază a omului. Încă din faza incipientă a omului, hrana a fost una dintre ţintele 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 secolului *XX* umanitatea încă era separată de ziduri și cortine, odată cu intrarea în mileniul *III*, a început și procesul de globalizare, unde mai fiecare cultură mai bogată a reușit să expordeze niște influențe culinare celorlalte.

Având un număr generos de bucătării disponibile și pentru fiecare dintre acestea cel puțin 200 de rețete, m-am decis să realizez o clasificare pe text a acestor rețete și în funcție de modul de preparare a fiecareia 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, extrăgând 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 și 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 ingrediente care le conțineau iar altele propuneau o clasificare a rețetelor în funcție de imagini, cu ajutorul rețelelor neuronale convolutionale. 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 ingrediente rețetelor, m-au ajutat să vizualizez și să înțeleg pașii necesari în realizarea acestei lucrări. Dar 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 să folosesc niciun set de date 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 să-mi extrag de pe fiecare rețetă în parte ce aveam nevoie.

O rețetă e era reprezentată de o pagină nouă, de unde îmi extrageam 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 Extragerea trăsăturilor

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 propoziții, text, document și o sută de documente care se concretizează într-un astăzi cunoscut *corpus*.

Având în vedere faptul că acești algoritmi au același scop, aceștia au și 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 semnele 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 transformă aceasta colecție de texte într-un vector de trăsături a acestora. [1]

### 2.1 Bag-of-Words

În primul rând, ar trebui explicitată denumirea acestui algoritm; Ea provine de la faptul că fiecare text, cum ar fi o propoziție sau frază, este reprezentată de un multiset (*bag*) de cuvinte, indiferent 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. [2]

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

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

În cadrul bag of words, este introdus și termenul de *n – gram*, care încearcă să grupeze cuvintele din text în perechi de cuvinte succesive de câte *n* și să numere frecvența acestora din *n – grame*. Pentru a înțelege mai ușor 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 încât 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 numărul 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 în 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 finală la tf-idf, aceea fiind:  $tfidf(t, d, D) = tf(t, d) \times idf(t, D)$ . [3]

## 2.3 Word-2-Vec

Word2Vec este un algoritm relativ recent apărut în domeniul Învățării Automate și presupune transformarea unui corpus de texte(*Word*) într-un spațiu vectorial(*2Vec*). Scopul său principal este acela de a grupa cuvintele care sunt similare în acel spațiu vectorial.

Acest algoritm de extragere de trăsături este la bază o rețea neuronală cu două straturi și se împarte în cele două categorii: *Continous Bag of Words(CBOG)* și *Skip Gram*.

Prima dintre ele, Continous Bag of Words este asemănător cu Bag of Words simplu, folosindu-se de cuvintele alăturate pentru a putea face predicția cuvântului întâi, de exemplu, dacă avem 5 cuvinte și a fi nevoie să-ș prezicem pe cel din mijloc(al treilea), modelul CBOG se va folosi de primele 2 și ultimele 2 cuvinte pentru a realiza această predicție.

Dacă presupunem că pe poziția  $i$  avem cuvântul pe care vrem să-l prezicem, atunci vom lua în considerare  $i - 2$ ,  $i - 1$ ,  $i + 1$  și  $i + 2$ .

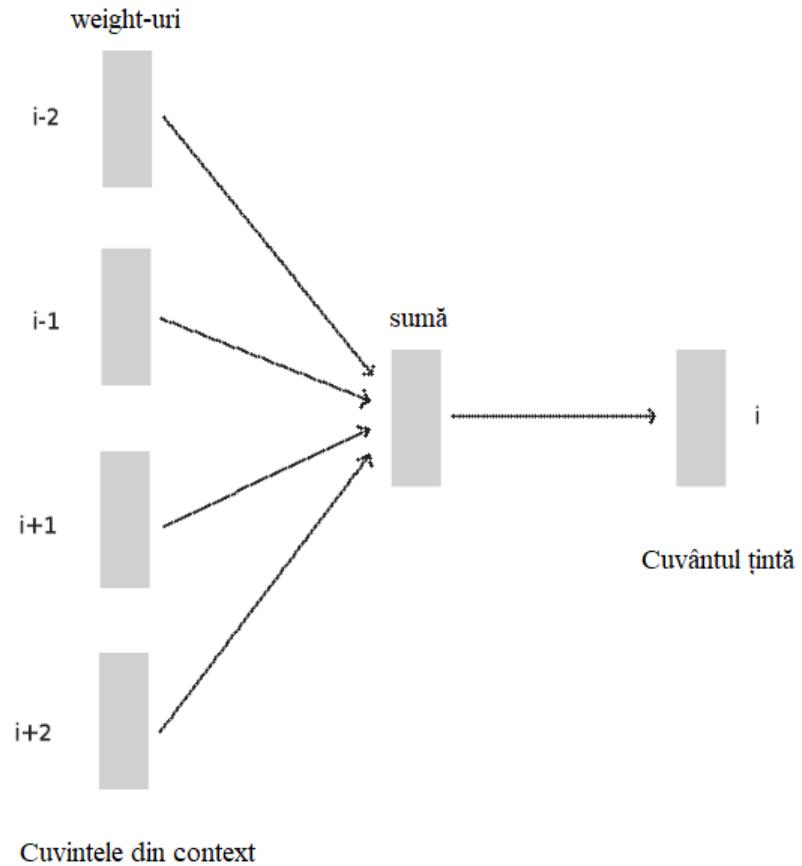


Figure 1: Exemplu de Continous Bag of Words

În cazul în care folosim modelul Skip Gram, parcursul este fix invers față de cel de la CBOG, adică va fi trimis un cuvânt și din acest input, vor rezulta cuvintele care fac parte din contextual cuvântului dat ca input.

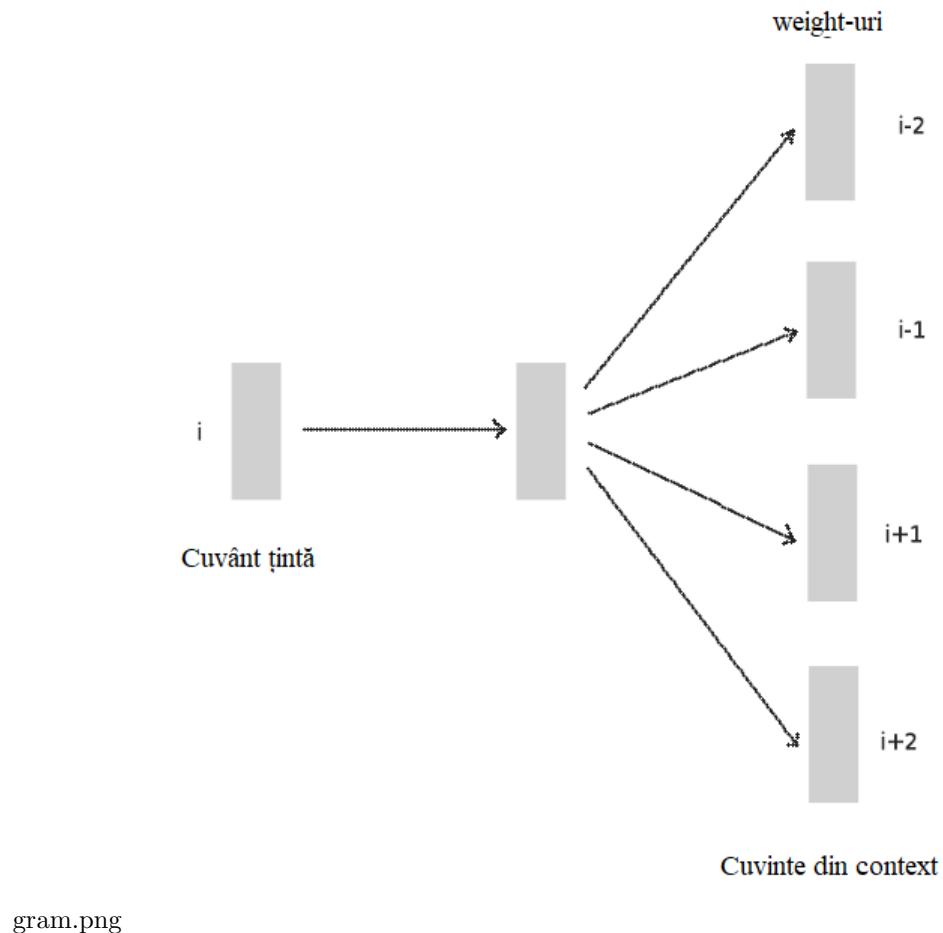


Figure 2: Exemplu de Skip Gram

În urma studiilor făcute după utilizarea acestor două modele, s-a observat ca Bag of Words Continuu are o acuratețe mult mai bună decât Skip Gram singurele dezavantaje fiind că pentru CBOG este de nevoie de un set de date mai mare și de mai mult timp de execuție, față de Skip Gram, care se descurcă pe seturile de date de dimensiune mai mică. [4]

### 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 anterioară, unde aplicăm și 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 detaliu cu algoritmul care îi stă la bază, SVM(*Support Vector Machines*). Acesta este un algoritm de învățare supervizată folosit în separa instanțele aparținând unor clase diferite într-un spațiu multidimensional.

SVM este unul dintre cei mai folosiți algoritmi de clasificare din domeniul Învățării Automate, deoarece sarcina lui este ca într-un spațiu n-dimensional, unde n este numărul de trăsături extrase, să găsească un hiperplan care să clasifice cel puțin două puncte din acel spațiu.

Majoritatea SVM-urilor folosesc să zisa margine maximă în alegerea hiperplanului, deoarece aceasta maximizează distanța dintre cele două puncte. Utilizarea maximului distanței dintre două puncte ne va asigura că punctele care se vor afla pe de o parte sau alta a graniței de decizie, vor fi corect clasificate.

Pentru a crea această margine maximă, SVM se utilizează de niște vectori suport care se află în proximitatea hiperplanului și pot influența poziția acestuia.

[5]

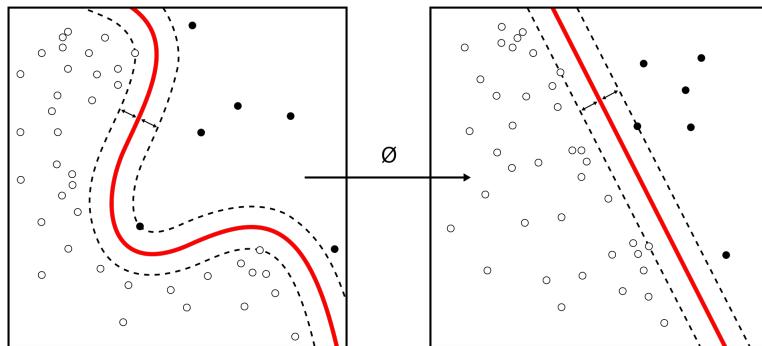


Figure 3: Exemplu de hiperplan cu margine maximă

#### 3.2 Regresia Logistică

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

Acest algoritm măsoară relația dintre variabilele categoric dependente și una sau mai multe variabile independente estimând probabilitățile folosind o funcție

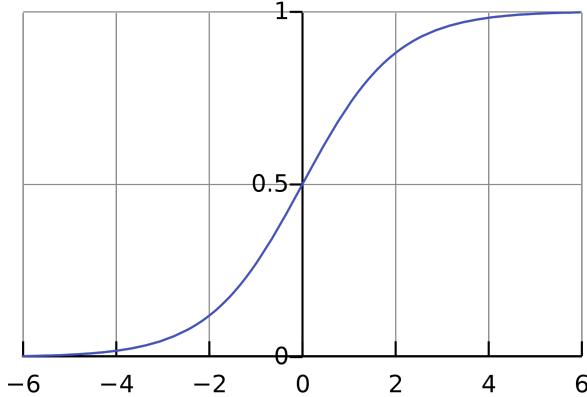


Figure 4: Graficul funcției sigmoid  $\sigma(x) = \frac{1}{1+e^{-x}}$

logistică. Variabilele dependente sunt reprezentate de clasa țintă pe care noi trebuie să o prezicem, aşadar variabilele independente sunt trăsăturile de care ne vom folosi pe a prezice clasa țintă. [6]

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

$$\begin{aligned} \ln P(Y_i = 1) &= \beta_1 \cdot X_i - \ln Z \\ \ln P(Y_i = 2) &= \beta_2 \cdot X_i - \ln Z \\ &\dots\dots \\ \ln P(Y_i = K) &= \beta_k \cdot X_i - \ln Z, \end{aligned}$$

unde  $\beta_i$  este factorul de regresie, atribuit fiecărei valori de input,  $X_i$  este valoarea input iar  $-\ln Z$  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:

$$\begin{aligned} P(Y_i = 1) &= \frac{1}{Z} e^{\beta_1 \cdot X_i} \\ P(Y_i = 2) &= \frac{1}{Z} e^{\beta_2 \cdot X_i} \\ \dots \\ P(Y_i = K) &= \frac{1}{Z} e^{\beta_K \cdot X_i}(1), \end{aligned}$$

Putem extrage valoarea lui  $Z$  din suma probabilităților, astfel:

$$\begin{aligned} \sum_{k=1}^K P(Y_i = k) &= 1 \Rightarrow \sum_{k=1}^K \frac{1}{Z} e^{\beta_k \cdot X_i} \Rightarrow \frac{1}{Z} \sum_{k=1}^K e^{\beta_k \cdot X_i} \\ &\Rightarrow Z = \sum_{k=1}^K e^{\beta_k \cdot X_i}(2) \end{aligned}$$

Din (1) și (2) putem calcula forma finală 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 funcției softmax:

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

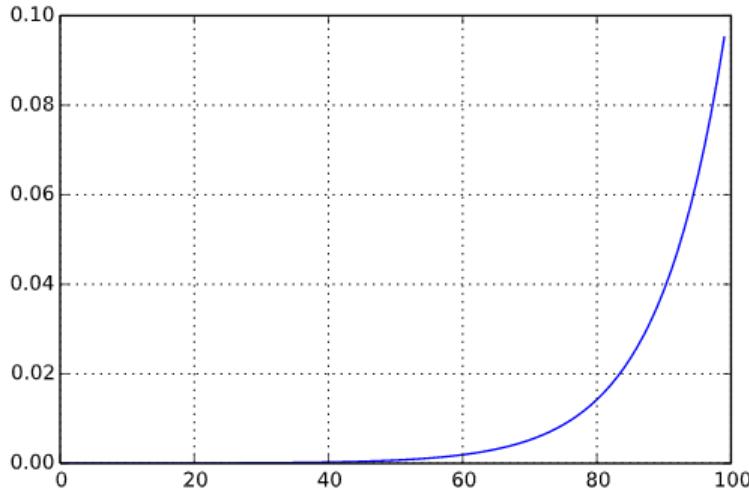


Figure 5: Graficul funcției softmax

Softmax are rolul de a "exagera" diferențele dintre valorile de input, astfel încât  $\text{softmax}(k, x_i)$  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 aproape de maximul respectiv. Așadar, la final se va obține o aproximare a funcției indicator:

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

[7]

Concluzionând, Regresia Logistică se poate dovedi un clasificator bun pentru setul nostru de date, având în vedere că poate fi folosit și pentru probleme de clasificare unde sunt mai mult de două clase. Regresia logistică vine cu propunere nouă de reprezentare a rezultatului, adică prin probabilitățile care le oferă când vine vorba de a clasifica date.

### 3.3 Random Forest

Random Forest(sau pădure aleatorie, dacă se dorește varianta în română) este un algoritm de învățare automată, de ansamblu. Acest concept de ansamblu face referire la faptul că această "pădure" este formată dintr-un ansamblu de arbori de decizie, de unde vine și termenul de *Forest*, singura diferență este că aceștia sunt generați aleator, *Random*, din atributele primite ca input. A fost studiat faptul că în domeniul învățării automate, există unii algoritmi care dau niște rezultate mai slabe dacă sunt folosiți singuri, iar dacă sunt puși la un loc(intr-un ansamblu), pot da rezultate foarte bune.

Pentru a înțelege mai bine cum funcționează Random Forest, trebuie să începem de la arborii de decizie. Acești arbori, sunt formați din atributele care formează datele de intrare; de exemplu, dacă am vrea să vedem dacă va ploua mâine, putem considera umiditatea, dacă a fost înnorat și dacă a fost frig a ultimelor zile, creăm un arbore de decizie în baza acestora, iar la final putem afla vremea de mâine.

Random Forest este alcătuit dintr-o multitudine de astfel de arbori, numai ca de data aceasta, arborii nu vor mai fi creați neapărat din toate atributurile de intrare, ci se va face ceea ce se numește *bagging* sau *bootstraping*. Aceasta metodă presupune utilizarea același număr de atrbute(cu tot cu deciziile aferente), dar sunt luate aleator și pot exista duplicate în acest set nou de atrbute, numit și *bootstrap*. S-a observat ca după *bootstraping*, mai rămân în afară aproximativ o treime din atrbute într-un set de date care se va numi *out – of – bag*.

Algoritmul va folosi mai departe valorile din *bootstrap* pentru a genera un arbore și până se ajunge la numărul dorit de arbori, va repeta acești doi pași: generează *bootstrap* și din acesta creează un arbore. Este preferat să se folosească un număr cât mai mare de arbori, deoarece cu cât este pădurea mai deasă cu atât modelul este mai robust și se poate ajunge la o acuratețe cât mai bună. Acești arbori, vor avea nodurile de decizie cât mai diferite între ei, putându-se respecta caracteristicile de *Random*.

După generarea "pădurii", se începe procedeul de clasificare, unde fiecare moștră, va trece prin toți arborii. Acești arbori vor oferi un output sau vot, acesta se va centraliza și se va alege eticheta cu cele mai multe voturi, procedura de alegere numindu-se vot majoritar.

Principalul avantaj pentru care am considerat utilizarea algoritmului Random Forest a fost că şansele ca să facă overfitting pe setul de date sunt foarte mici și pentru că având parametrii optimi, putem obține o acuratețe ridicată.

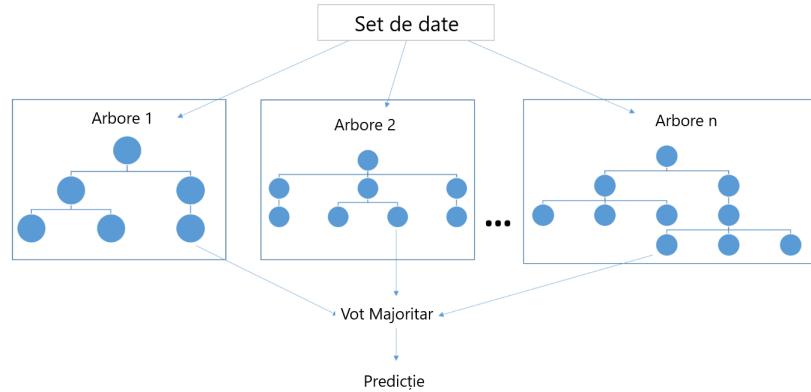


Figure 6: Exemplu de clasificare cu Random Forest

## 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 importantă, fiind explicată în cadrul unui sub punct.

### 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 într-un număr de 15 bucătării și aproximativ vreo 5000 rețete culese pe de site-ul [www.allrecipes.com](http://www.allrecipes.com).

Bucătăriile care fac parte din acest set de date sunt:

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

Acest set de date constă într-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 2 minutes. Drain the corn in a colander set in the sink. Mix together  
the warm corn, black-eyed peas, ham, celery, and onion in a salad bowl.  

```

black-eyed pea mixture until thoroughly mixed. Serve immediately.”

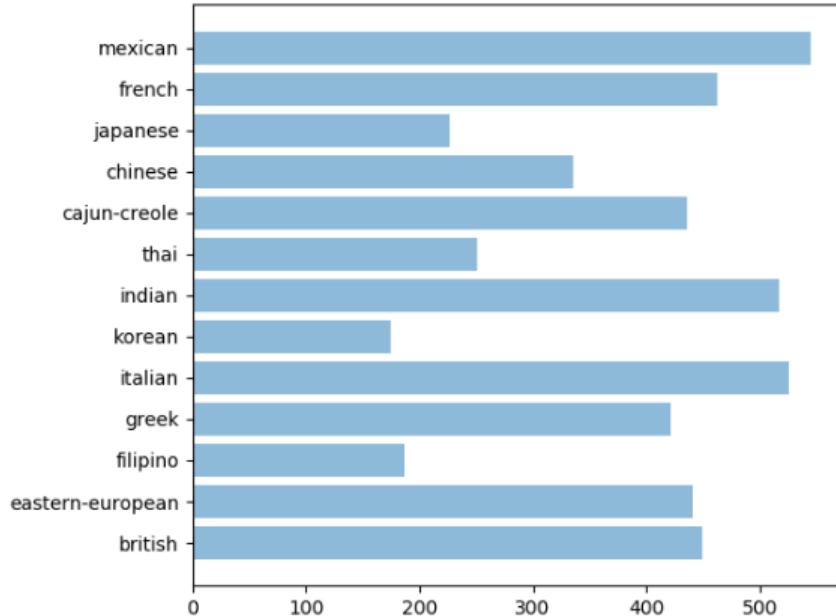


Figure 7: Repartizarea rețetelor pe bucătării

## 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 fiecărui. Înainte de a începe procesarea propriu-zisă, mai întâi voi lua din fișierul JSON datele cu biblioteca *pandas* într-o structură de tip *DataFrame* care îmi va permite să lucrez mai ușor 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 instru-

mentele 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 (acele *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 observat 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 instantiat, urmează etapa de propriu zisă de extragere de trăsături, unde vom crea 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 și 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 Word2Vec

Trebuie explicat cum m-au ajutat funcțiile astea, și trebuie să realizez o ajustare de hiperparametri

```
model = gensim.models.Word2Vec(  
    sentences,  
    size=250,  
    window=5,  
    min_count=1,  
    workers=4)  
%%%%%%%%  
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 proprietatea 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 transmite algoritmului care să fie rația corecției la clasificare, în sensul în care cu cât este mai mare această valoare, cu cât hiperplanul clasifică 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 mai mică, cu atât marginea 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:

Ajustări			
Parametri	Tf-Idf	Bag-of-Words	Word-2-Vec
C	0.5	0.1	25

Figure 8: Ajustări/extractor de trăsături

#### 4.3.2 Regresie Logistică

Parametrul C este mai numit și parametru de regularizare și are același efect asupra algoritmului ca și la Linear SVC.

```
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:

Ajustări			
Parametri	Tf-Idf	Bag-of-Words	Word-2-Vec
C	10	1	1000

Figure 9: Ajustări/extractor de trăsături

#### 4.3.3 Random Forest

La o primă vedere, algoritmul Random Forest este mai bogat în hiperparametri (și în optimizările acestora). Pentru început, am încercat să limităm numărul maxim de arbori din pădure, prin parametrul *n\_estimators*, apoi am setat o valoare parametrului *bootstrap* care e indică dacă vom folosi întreg setul

de date pentru a construi fiecare arbore(acesta este comportamentul în cazul în care am ales *False*). Pentru a păstra o oarecare eficiență timp a execuției clasificării, am limitat și adâncimea maximă a arborilor, cu ajutorul parametru-lui *max\_depth*.

Având o bună parte a parametrilor fixați, am trecut la următoarea etapă de alegere a punctului de împărțire, unde am ales valoare minimă de valori într-un nod intern cu *min\_samples\_split*, apoi am setat numărul minim de valori pentru a fi un nod frunză cu *min\_samples\_leaf*, aceasta mai putând fi utilizat pentru a împărți la orice adâncime a arborului.

Într-un final, am ales și numărul maxim de trăsături pentru a putea face o împărțire a arborelui, *max\_features*

```
classifier_rf = RandomForestClassifier(
    min_samples_split ,
    n_estimators ,
    max_depth ,
    max_features ,
    bootstrap ,
    min_samples_leaf)
classifier_rf.fit(train , classes)
prediction_nb = classifier_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

Figure 10: Ajustări/extractor de trăsături

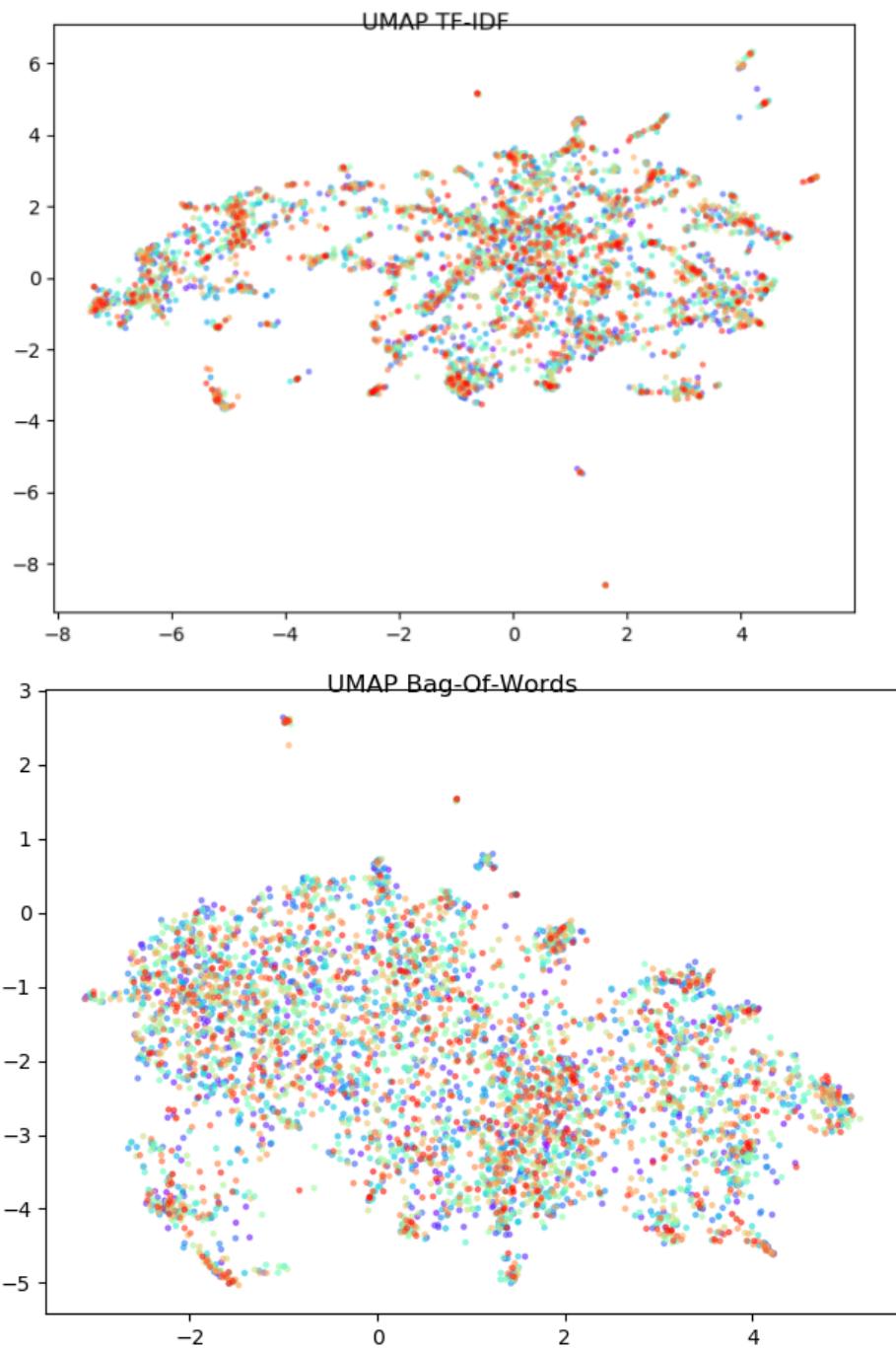
## 4.4 Reprezentări grafice

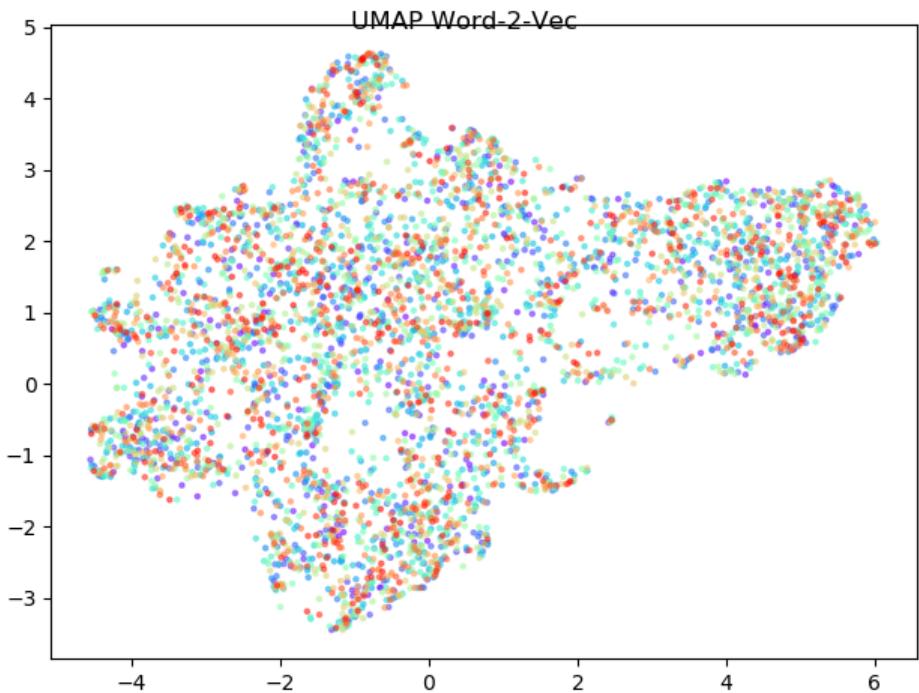
Avem setul de date trecut prin algoritmii de clasificare, astfel încât am putea genera o reprezentare grafică cu ajutorul următorilor algoritmi și a valorilor rezultate în urma procesării datelor.

Vom reduce la 2 dimensiuni, pentru a putea să îi dăm drept input K-Means-ului un set de date cât mai redus din punct de vedere dimensional.

### 4.4.1 UMAP

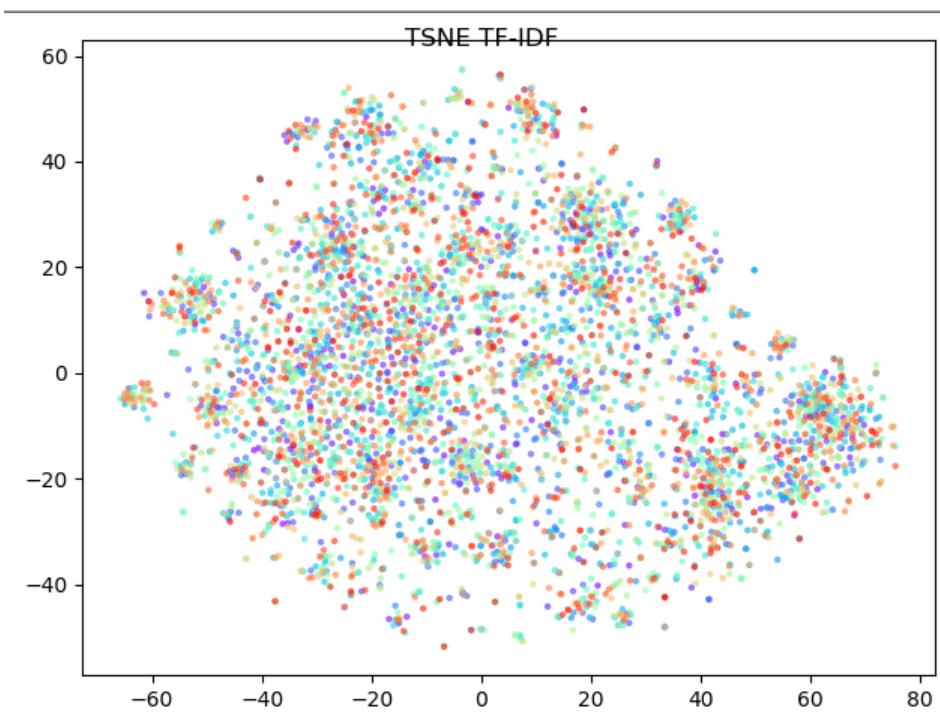
```
umap = UMAP()  
reduced_data = umap.fit_transform(data)
```

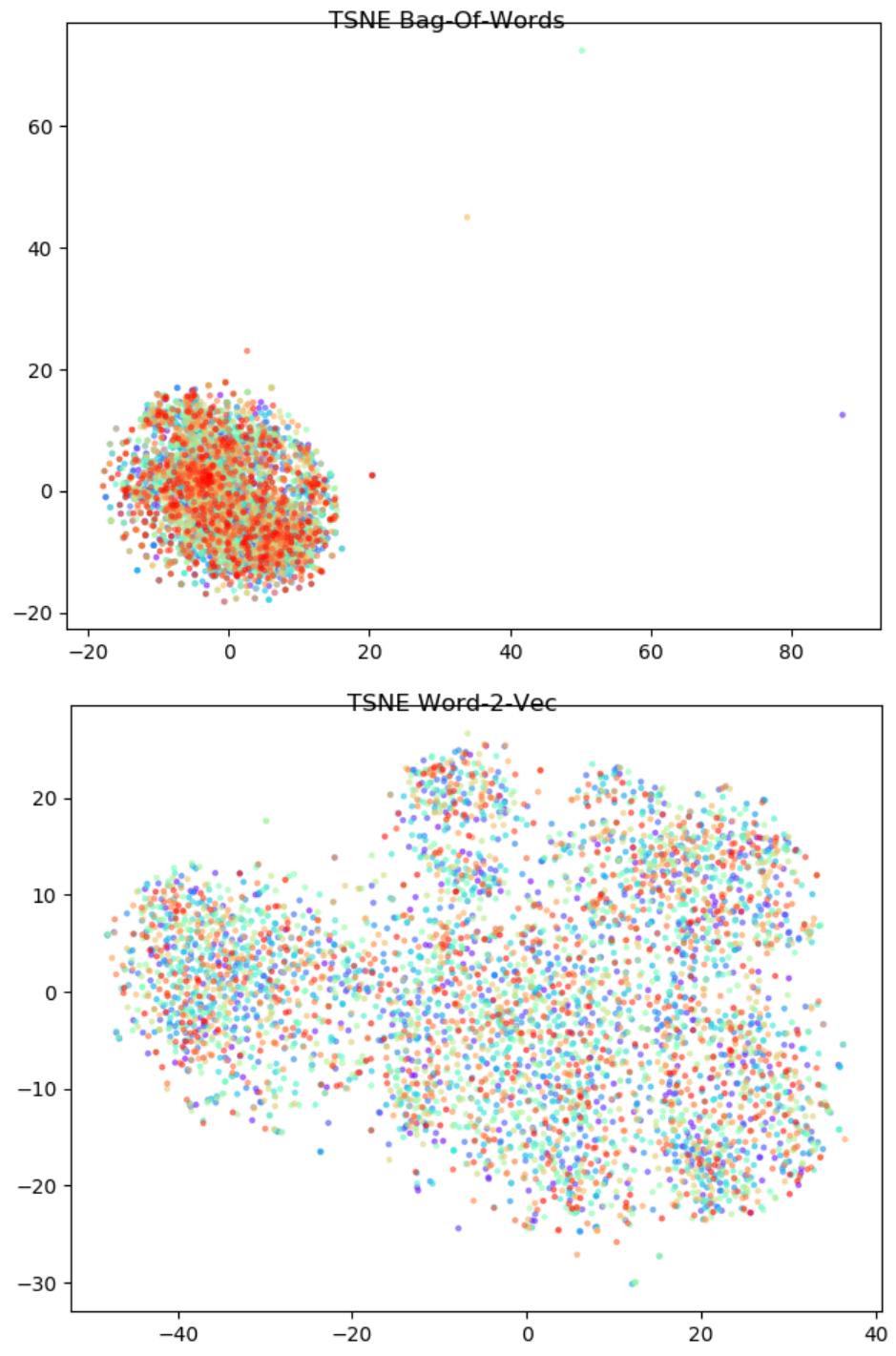




#### 4.4.2 T-SNE

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





## 4.5 Tehnologii folosite

### 4.5.1 Python

Python este un limbaj pe programare interpretat, high-level și utilizat într-o multitudine de domenii, printre care și cel din care face parte această lucrare și anume Învățarea Automată. Popularitatea lui a crescut constant în ultimii ani, datorită faptului că este ușor de scris, de folosit și de citit, arătând destul de mult cu pseudocod.



Figure 11: Logo-ul Python

Este relativ ușor de învățat, este dynamically-typed, adică nu folosește tipuri, este mult mai expresiv deci crește productivitatea, ceea ce foarte multă lume dorește în ziua de astăzi. La baza dezvoltării Python, stau următoarele principii

- Frumos e mai bun decât urât
- Explicit e mai bun decât implicit
- Simplu e mai bun decât complex
- Complex e mai bun decât complicat
- Lizibilitatea contează

Un alt beneficiu important al Python este că acesta conține aproximativ toate bibliotecile de Învățare Automată dezvoltate până acum, cum ar fi *SciKit-Learn*, *TensorFlow + Keras* și pe lângă acestea, conține și biblioteci care să vină în ajutorul procesului, cum ar fi *Pandas* pentru putea umbla cu setul de date mult mai ușor și *matplotlib* pentru a vizualiza datele noastre, fie ele de input sau output.

Toate acestea adunate, fac Python-ul o unealtă, poate cea mai puternică la momentul actual, în domeniul Învățării Automate și este prima alegere pentru mulți entuziaști sau pentru cei care profesează în acest domeniu(Data Scientist). [8, 9]

### 4.5.2 SciKit-Learn

### 4.5.3 BeautifulSoup

BeautifulSoup este biblioteca care a ajutat la extragerea rețetelor de pe internet, aceasta punând la dispoziție un crawler cu care putem parcurge pagini web în căutarea de informații. Site-ul folosit în extragerea rețetelor nu a fost aleator ales, ci printr-o căutare a unui site care oferea o sortare a rețetelor pe bucătării și să prezinte informațiile cât mai succint; acesta a fost [www.allrecipes.com](http://www.allrecipes.com).

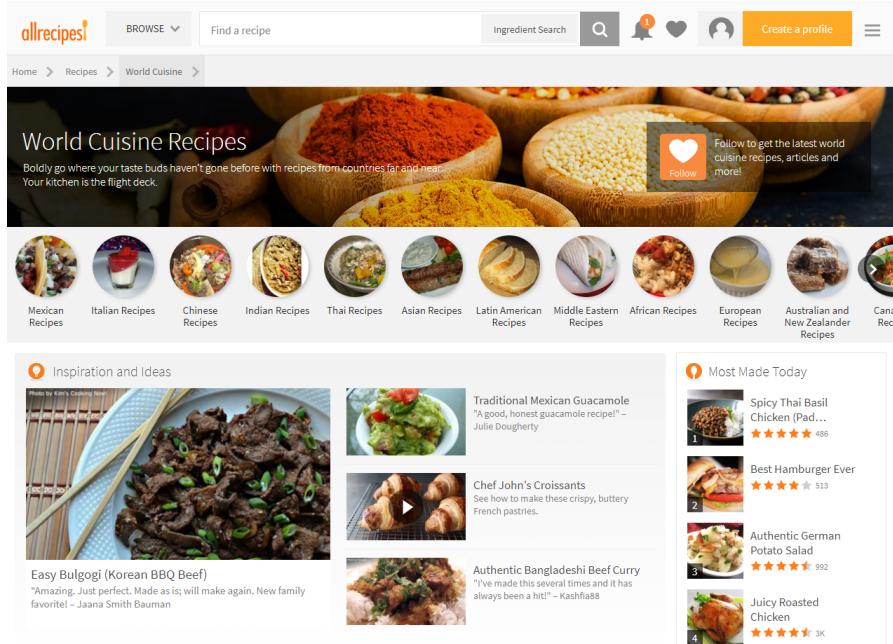


Figure 12: Pagina principală a rețetelor grupate pe bucătării

Primul pas în începerea procesului de "crawl" prin site-uri web, facem un request de încărcare a linkului primit dintr-o listă de bucătării, de exemplu, cea mexicană.

```
webpage = requests.get(link)
```

După ce se face requestul cu success, instanțiem crawler-ul, cu codul html al paginii și specificăm atributului *features* că vom parsa un html.

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

Crawler-ul fiind pregătit, va incepe să parseze fiecare articol, de exemplu casetele de mai sus unde se află o descriere succintă a rețetei, o poză și o notă, în căutarea link-urilor care fac referință la pagina rețetei.

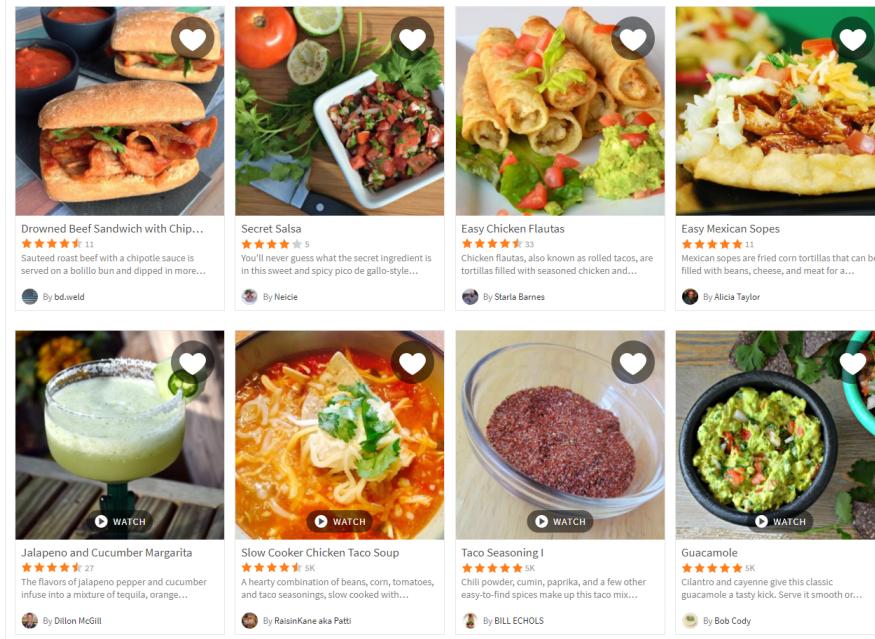


Figure 13: Fragment din prima pagină cu rețete mexicane

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

Pentru fiecare link din listă, va face iar un request de tip GET pentru a-și lua informațiile de pe link-ul respectiv și se va începe căutarea a unor noi informații, cum ar fi titlul, instrucțiunile și ingredientele, pe care le va salva într-o structură de tip cheie valoare, care apoi va fi exportată drept fișier JSON.

```
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"})
```

#### 4.5.4 Gensim

folosit cu w2v

#### 4.5.5 Pandas

folosit pentru jsoane

Ingredients		Directions		
	12 ounces chipotle cooking sauce (such as Knorr®)		3 cloves garlic, minced	
	1 (14 ounce) can reduced-sodium beef broth		1 pound thinly sliced deli roast beef	
	1/4 cup chopped fresh cilantro (optional)		4 bolillo rolls, halved and lightly toasted	
	2 tablespoons vegetable oil		4 sprigs fresh cilantro, or to taste (optional)	
	1 onion, thinly sliced		Add all ingredients to list	
			Prep 15 m	Cook 20 m
				Ready In 35 m
			Combine chipotle cooking sauce, beef broth, and 1/4 cup chopped cilantro in a saucepan; bring to a boil. Reduce heat to medium-low and simmer, stirring occasionally, for 10 minutes.	
			Heat oil in a skillet over medium-high heat; sauté onion until softened, about 5 minutes. Stir garlic into onion and cook for 1 minute. Add roast beef and 1/4 cup chipotle sauce mixture and cook, stirring constantly, until heated through, about 2 minutes.	
			Ladle remaining chipotle sauce mixture into 4 bowls for dipping. Spoon roast beef mixture onto the bottom half of each bun, top with a cilantro sprig, and place the top on each bun. Dip sandwiches into sauce.	

Figure 14: Lista ingredientelor

Figure 15: Lista instrucțiunilor

#### 4.5.6 Plotlib

folosit sa fac reprezentarea grafica cu umap/tsne

### 4.6 Rezultatele

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

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

- SVC: aveam rezultate în 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
- Random Forest: aici au fost notate cele mai bune îmbunătățiri, deoarece, inițial Random Forest avea acuratețe în intervalul 35-50%, după modificările impuse de ajustarea hiperparametrilor, am ajuns la rezultate în zona 70-76

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

## **5 Concluzii**

## 6 Bibliografie

### References

- [1] [https://en.wikipedia.org/wiki/Information\\_retrieval](https://en.wikipedia.org/wiki/Information_retrieval)
- [2] [https://en.wikipedia.org/wiki/Bag-of-words\\_model](https://en.wikipedia.org/wiki/Bag-of-words_model)
- [3] <https://en.wikipedia.org/wiki/Tfidf>
- [4] <https://www.adityathakker.com/introduction-to-word2vec-how-it-works/>
- [5] [https://en.wikipedia.org/wiki/Support-vector\\_machine](https://en.wikipedia.org/wiki/Support-vector_machine)
- [6] [https://en.wikipedia.org/wiki/Logistic\\_regression](https://en.wikipedia.org/wiki/Logistic_regression)
- [7] [https://en.wikipedia.org/wiki/Multinomial\\_logistic\\_regression](https://en.wikipedia.org/wiki/Multinomial_logistic_regression)
- [8] <https://www.quora.com/Why-is-Python-so-popular-in-machine-learning>
- [9] [https://en.wikipedia.org/wiki/Python\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Python_(programming_language))