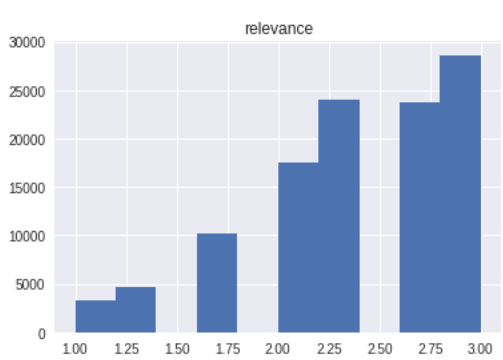
Deep Learning Workshop – Assignment 3 – Report

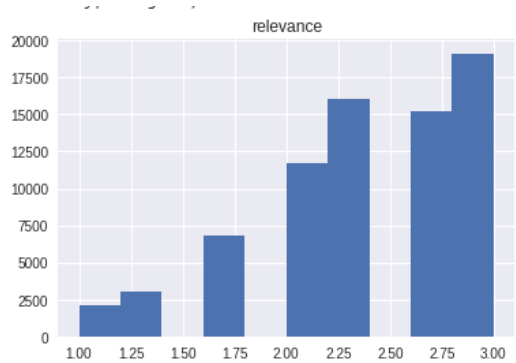
Character-level LSTM

בחלק זה ניצור מודל מבוסס רשת סיאמית (כמו במאמר, רק עם LSTM במקום CNN), שיקבל כקלט 2 רצפים של תווים, אחד של search_term שייצג את מילות החיפוש, והשני של product_title שייצג את כותרת המוצר, ושניהם תואמים אחד לשני ברמת רלוונטיות relevance בין 1-3, אותה אנחנו רוצים לחזות.

טענו את הטבלה השלישית בנתונים, product_descriptions, עשינו איתה join לטבלת train ו-test, ושרשרנו את product_title לסוף המחרוזות בעמודה הזו, ולבסוף לקחנו את 1000 התווים הראשונים. שיטה נתנה לנו תוצאות טובות, אך שימוש רק ב-product_title נתן לנו תוצאות מעט יותר טובות, לכן בחרנו להתעלם מהטבלה של product_descriptions. אולי בחלק השני של word-level זה יעבוד יותר טוב, נבדוק את זה שם.

בדקנו את התפלגות הדגימות בסטים השונים, כדי לבדוק גם שהם מתפלגים אחיד וגם מתפלגים אותו הדבר בערך (ימין – train, שמאל – test).





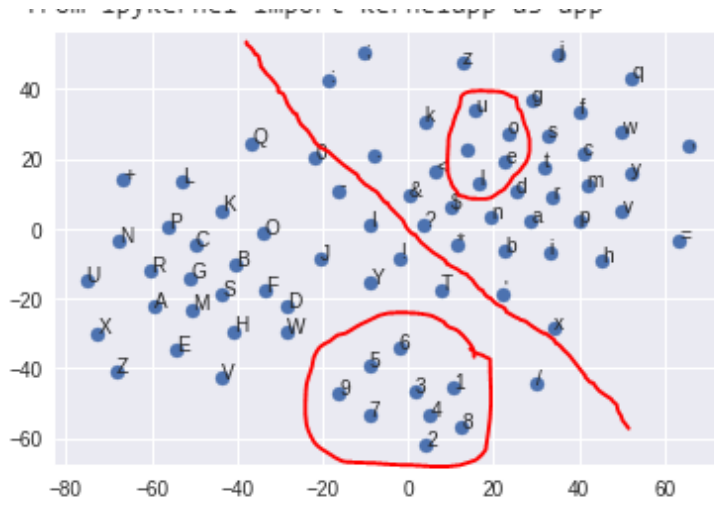
תחילה אימנו מודל Word2Vec שייצר לנו מטריצת embeddings לאותיות השונות בנתונים שלנו. עשינו מספר ניסויים בשלב הזה, שילבנו כל פעם עיבוד מקדים שונה על הנתונים (עם\בלי תווים כמו +=)(, Stemmer למיניהם וכו').

לאחר אימון מודל ה-W2V חקרנו אותו כדי לראות מה התוצאות שלו, לדוגמא עשינו TSNE כדי להוריד מימדים ולראות את הקרבה פחות או יותר בין התווים השונים. ניתן לראות כי אכן המספרים התקבצו בסוג של cluster אחד. אפשר לראות גם שאותיות אהו"י באנגלית יחסית קרובות אחת לשניה, ושאפשר

לשים קו בין אותיות גדולות לאותיות קטנות.

```
word2vec.most_similar('1')
```

```
/usr/local/lib/python3.6/dis
  if np.issubdtype(vec.dtype
[('3', 0.8836517333984375),
 ('4', 0.8594918251037598),
 ('2', 0.7550736665725708),
 ('8', 0.7248693108558655),
 ('7', 0.6337199211120605),
 ('5', 0.5918388366699219),
 ('9', 0.5060371160507202),
 ('6', 0.4579069912433624),
```



בתור עיבוד מקדים ל-embedding, השתמשנו ב-regex הבא, לאחר הרבה ניסויים עם מספר regex שונים ושילובים שלהם:

```
(r"[^A-Za-z0-9^,!.\/'+-=]", " ",
```

בחרנו בסופו של דבר לייצג כל אות בתור וקטור של 128 מאפיינים, ועם window_size=10.

לאחר מכן, הכנו את הנתונים שלנו כך שיתאימו ל-input של שכבת ה-embedding (לדעתנו החלק הכי קשה ומסובך בעבודה, ציפינו שעם כל הקסמים שקורים בשורה אחת בפייתון, יהיה קסם גם לזה). כלומר, הפכנו סטרינג של 'my item' לרשימה של מספרים שתואמים לאינדקסים במטריצת ה-embeddings שיצר ה-word2vec (כמו [1,94,126,6,31]...). הוספנו padding לכל entry כך שיהיה בגודל המקסימלי מבין כל הקלטים (כל עמודה בנפרד – מקסימום לsearch_term ומקסימום ל-product_title). כמו כן, בשכבת ה-Embedding שמנו input=None מכיוון שיש הבדל באורך של הקלטים האחרונים.

כעת, לאחר שיש לנו מטריצת embeddings מוכנה ונתונים מעובדים ומותאמים למטריצה, נבנה את ארכיטקטורת המודל שלנו. עשינו זאת בהתאם למה שדיברנו בשיעור ובהתאם למאמר המדובר:

```
__________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==========================================================================================
input_11 (InputLayer)           (None, 60)           0
__________________________________________________________________________________________
input_12 (InputLayer)           (None, 147)          0
__________________________________________________________________________________________
embedding_6 (Embedding)         multiple             10240       input_11[0][0]
                                                                 input_12[0][0]
__________________________________________________________________________________________
lstm_6 (LSTM)                   (None, 100)          91600       embedding_6[0][0]
                                                                 embedding_6[1][0]
__________________________________________________________________________________________
dot_3 (Dot)                     (None, 1)            0           lstm_6[0][0]
                                                                 lstm_6[1][0]
__________________________________________________________________________________________
lambda_8 (Lambda)               (None, 1)            0           dot_3[0][0]
==========================================================================================
Total params: 101,840
Trainable params: 91,600
Non-trainable params: 10,240
__________________________________________________________________________________________
```

יש לנו 2 "שכבות" קלט למודל, שמקבלות כל אחת מחרוזת (search_term, product_title), ואז שניהם עוברים לשכבת ה-embedding לקבלת מטריצות הייצוג שלהם. כל אחת מהמטריצות נכנסת לשכבת LSTM, ולבסוף 2 הפלטים מה-LSTM נכנסים לשכבת Dot שמחשבת דמיון ביניהם.

החלטנו על נוסחת דמיון של Cosine Similarity כי זה הדבר שעבד הכי טוב (ניסינו גם Exponential Negative Manhattan Distance ועוד וריאציות של הנוסחה). לבסוף, עשינו טרנספורמציה לנוסחת הקוסינוס כדי לעבור לתחום [1,3]. השתמשנו בפונקציית הפסד של Mean Squared Error ו-optimizer של Adadelta עם פרמטר gradient_clipping_norm=1.25, לאחר הרבה ניסויים עם כל מיני optimizers שונים כמו adam ודומיו.

ההרצה של Character-Level LSTM כללה את ה-hyper parameters הבאים:

- LSTM nodes: 50
- Gradient clipping norm: 1.25
- Batch-Size: 128
- Epochs: 3

```
Train on 59253 samples, validate on 14814 samples
Epoch 1/3
59253/59253 [==============================] - 155s 3ms/step - loss: 0.2986 - mean_absolute_error: 0.4416 - val_loss: 0.2700 - val_mean_absolute_error: 0.4241
Epoch 2/3
59253/59253 [==============================] - 155s 3ms/step - loss: 0.2663 - mean_absolute_error: 0.4214 - val_loss: 0.2653 - val_mean_absolute_error: 0.4172
Epoch 3/3
59253/59253 [==============================] - 154s 3ms/step - loss: 0.2594 - mean_absolute_error: 0.4155 - val_loss: 0.2595 - val_mean_absolute_error: 0.4182
Training time finished.
3 epochs in 0:07:45.521943
```

Train RMSE: 0.509
Val RMSE: 0.509
Test RMSE: 0.5195

Naïve Baseline Model

המודל הנאיבי שלנו יתבסס על נתונים סטטיסטים של כמות מופעי המילים בשאילתת החיפוש שזוהו בתיאור המוצר שאוחזר. כלומר לאחר שנבצע ניקוי מקדים ועיבוד למילים בשאילתה וגם בתיאור המוצר (התואם) נספור כמה מילים הופיעו גם בשאילתה וגם בתיאור המוצר. האינטואיציה לשימוש זה היא שכאשר מספר מילים גדול מופיע בשאילתה וגם בתיאור המוצר אזי המוצר יהיה בעל רלוונטיות גבוהה.
כדי לנצל את הידע ולשפר את יכולת המודל הנאיבי נאמן מודל רגרסיה Lasso על נתוני המופעים אל מול מדד הרלוונטיות. המודל הנאיבי הסופי מגיע ל RMSE 0.5348 תוצאה מכובדת שננסה להיעזר כהשוואה למודלים היותר מתחכמים בהמשך.

```
test:                      validation:                train:
RMSE:0.5339850961042517    RMSE:0.5336004531677611    RMSE:0.5341132511672262
MAE:0.437719038904579      MAE:0.43764936268212334    MAE:0.43774226473016337
```

Character-level Feature Extraction

בחלק זה לקחנו את הפלט של שכבת ה-LSTM שעובדת על שני הקלטים ומוציאה שני פלטים. את שני הפלטים העברנו למודל למידת מכונה קלאסי, כמו למשל XGBoost. להעביר את 2 הוקטורים (לאחר concat) שיוצאים מה-LSTM נראה לנו לא הגיוני כל כך, כי המודל לא מייחס חשיבות לדמיון בין שני החלקים בוקטור. חשבנו על מספר פתרונות, כמו למשל להגדיר loss function משלנו שתתאים לבעיה שלנו, כלומר לקחת את החלקים הספציפיים בוקטור ולחשב דמיון ביניהם, ולבסוף MSE, אך לאחר חיפוש בגוגל הבנו שזה מאוד מסובך לזמננו הקצר.

חשבנו בנוסף להוסיף למודל LSTM הרגיל שכבת Dense ולאמן את המודל מחדש כשרק השכבה הזו לומדת, כך שתדע להוציא לנו features מתאימים שנוכל להשתמש בהם ב-LR/XGB, אך גם זה לא הניב תוצאות טובות במיוחד.

לבסוף, חזרנו לרעיון המקורי. ראינו שהחיזוי של ה-XGB לא יוצא בדיוק בטווח שלנו, כלומר יש כמה דגימות שיצאו בין 0.8-1, אך עשינו תיקון לדגימות אלה שיהיו 1.

XGBoost:

השתמשנו בפרמטרים הבאים במודל:

```
xgb_model = XGBRegressor(n_estimators=16, learning_rate=0.1,
                         gamma=0, subsample=0.8, colsample_bytree=1, max_depth=16)
```

```
Train RMSE: 0.4972732533176009
Train MAE: 0.41449894255707215
Test RMSE: 0.6477284014579732
Test MAE: 0.5536709733521079
Validation RMSE: 0.6065050118555447
Validation MAE: 0.5176879072002639
```

ניתן לראות שהמודל הוא overfitted ל-train, אך ה-Train RMSE נמוך יחסית – אפילו יותר נמוך מהמודל המקורי.

Random Forest Regressor:

במודל השני השתמשנו ב-Random Forest Regressor, שעובד אחרת מ-XGBoost. השתמשנו בפרמטרים הבאים:

```
RandomForestRegressor(max_depth=2, random_state=seed, n_estimators=192)
```

```
Train RMSE: 0.5272091030800764
Train MAE: 0.4318662674783223
Validation RMSE: 0.5280024925676449
Validation MAE: 0.43178558457585536
Test RMSE: 0.5306539137179183
Test MAE: 0.434309663244662
```

Word-level LSTM

בחלק זה במקום שהמודל יהיה ברמת האות, נשנה את ה-preprocess שלנו על הטקסט כדי שיהיה ברמת המילה. גם ה-embedding יהיה ברמת המילה, כמובן. השתמשנו ב-NLTK בשביל לקבל tokenizer למשפטים בטקסט, שיפריד לנו אותם למילים. לפני ההפרדה למילים, השתמשנו בכל מיני regular expressions שיעזרו לנו לעשות סוג של stemming על המילים. ניסינו גם להשתמש ב-Stemmer מוכן כדוגמת PorterStemmer, אך זה עבד מעט פחות טוב או אותם ביצועים אך בזמן ריצה ארוך יותר. דוגמאות ל-RegExp שהשתמשנו בהם:

```
text = re.sub(r"what's", "what is ", text)
text = re.sub(r"\'s", " ", text)
text = re.sub(r"\'ve", " have ", text)
text = re.sub(r"can't", "cannot ", text)
text = re.sub(r"n't", " not ", text)
text = re.sub(r"i'm", "i am ", text)
text = re.sub(r"\'re", " are ", text)
text = re.sub(r"\'d", " would ", text)
text = re.sub(r"\'ll", " will ", text)
```

למשל שהמילה what's תהפוך ל-what is, המילה I'm תהפוך ל-I am, וכו'. עשינו בדיקת שפיות למודל ה-Word2Vec שנוצר, ולקחנו לדוגמא את המילה 'stool', וראינו איזה מילים במילון הכי דומות לה:

```
wvmodel.most_similar('stool')
```

```
/usr/local/lib/python3.6/dist-packages/
  """Entry point for launching an IPyth
/usr/local/lib/python3.6/dist-packages/
  if np.issubdtype(vec.dtype, np.int):
[('bench', 0.6790950894355774),
 ('homesullivan', 0.6700759530067444),
 ('sofa', 0.6494880318641663),
 ('barstool', 0.626984715461731),
 ('stools', 0.6052266359329224),
 ('upholstered', 0.6011030673980713),
 ('recliner', 0.5902377963066101),
 ('ottoman', 0.5840995907783508),
 ('backless', 0.5779173970222473),
 ('pub', 0.5669509172439575)]
```

הנ"ל הן מילים יחסית דומות במשמעותן ל-stool, כמו homesullivan ו-bench שהם גם מתקנים שיושבים עליהם.

מבחינת ארכיטקטורה של מודל, היא נשארה אותו הדבר בדיוק. רק המשקולות של שכבת ה-embedding כמובן השתנו, כי אימנו מודל Word2Vec חדש מבוסס מילים במקום אותיות. בחלק זה מכיוון שמאגר המילים הייחודיות הוא בגודל עצום, בטוח יותר גדול ממאגר התווים בשלב הקודם, פלט של TSNE לא מראה תוצאות טובות במיוחד כי

```
Train on 59253 samples, validate on 14814 samples
Epoch 1/3
59253/59253 [==============================] - 221s 4ms/step - loss: 0.3173 - mean_absolute_error: 0.4575 - val_loss: 0.2806 - val_mean_absolute_error: 0.4377
Epoch 2/3
59253/59253 [==============================] - 219s 4ms/step - loss: 0.2740 - mean_absolute_error: 0.4297 - val_loss: 0.2720 - val_mean_absolute_error: 0.4288
Epoch 3/3
59253/59253 [==============================] - 219s 4ms/step - loss: 0.2612 - mean_absolute_error: 0.4194 - val_loss: 0.2693 - val_mean_absolute_error: 0.4298
```

```
Epoch 1/2
59253/59253 [==============================] - 218s 4ms/step - loss: 0.2514 - mean_absolute_error: 0.4112 - val_loss: 0.2646 - val_mean_absolute_error: 0.4192
Epoch 2/2
59253/59253 [==============================] - 218s 4ms/step - loss: 0.2447 - mean_absolute_error: 0.4054 - val_loss: 0.2612 - val_mean_absolute_error: 0.4198
```

```
Test RMSE: 0.649103710776161
Test MAE: 0.5388912125566555
```

(אימנו בשני חלקים בזמנים שונים – לאחר טעינה מחדש של המודל)

Word-level Feature Extraction

בחלק זה ניסינו לשנות מעט את הפרמטרים למודלים, וגם את המודלים עצמם, אך בסופו של דבר המסקנה נשארה זהה – השתמשנו באותם מודלים ובאותם פרמטרים כמו בחלק הקודם.

XGBoost:

```
Train RMSE: 0.5162644156477525
Train MAE: 0.43237481842358405
Validation RMSE: 0.6155048821827286
Validation MAE: 0.5260890213680667
Test RMSE: 0.671742581106928
Test MAE: 0.5756632738981748
```

תוצאות אלה הן פחות טובות ממודל ה-LSTM הרגיל, להבדיל מהחלק הקודם שראינו שיפור ב- Train RMSE של המודל המקורי וה-XGBoost.

Random Forest Regressor:

גם פה נשארנו עם אותם פרמטרים, וקיבלנו את התוצאות הבאות:

```
Train RMSE: 0.5291250464023634
Train MAE: 0.43335822211081404
Validation RMSE: 0.5326810155494316
Validation MAE: 0.4379692741003557
Test RMSE: 0.5323707328359302
Test MAE: 0.4361651097691325
```

התוצאות של ה-Random Forest מאוד יציבות יחסית לשאר, כלומר אין overfitting בולט וה-RMSE על כל הנתונים דומה, בניגוד ל-XGBoost מהסעיף הקודם.

טבלאת השוואה בין כל המודלים:

Model type	runtime	Train RMSE	Val-RMSE	Test-RMSE	Train MAE	Val-MAE	Test-MAE
Naïve Model	27 sec.	0.5341	0.5336	0.5339	0.4377	0.4376	0.4377
Character-level LSTM	464 sec.	0.509	0.5090	0.5195	0.4155	0.4182	0.4253
Character-level LSTM Feature Extraction - XGBoost	30 sec.	0.4972	0.6065	0.6477	0.4144	0.5176	0.5536
Character-level LSTM Feature Extraction – Random Forest Regressor	169 sec.	0.5272	0.5288	0.5306	0.4318	0.4317	0.4343
Word-Level LSTM	1095 sec.	0.4946	0.5110	0.6491	0.4054	0.4198	0.5388
Word-Level LSTM Feature Extraction - XGBoost	29 sec.	0.5162	0.6155	0.6717	0.43237	0.5260	0.5756
Word-Level LSTM Feature Extraction – Random Forest Regressor	178 sec.	0.5291	0.5326	0.5323	0.4333	0.4379	0.4361
Minimum Value	27 sec.	0.4946	0.509	0.5195	0.4054	0.4182	0.4253
Minimum Value's Model	Naïve	Word-Level LSTM	Character-Level LSTM	Character-Level LSTM	Word-Level LSTM	Character-Level LSTM	Character-Level LSTM

מסקנות מהטבלה ומכל המחקר:

אם נסתכל על הערכים הכי טובים בכל עמודה, נראה שמודלי ה-LSTM המקוריים שולטים, כלומר השימוש ב-feature extraction לא עזר לנו באף חלק בעבודה, לא ברמת המילה ולא ברמת האות (כמובן, אם נשקלל את כל המדדים יחדיו אולי נקבל תוצאה שונה ואחד מהם כן עולה על המודלים המקוריים – כמו למשל בהסתכלות מהירה רואים שה-Random Forest Regressor נותן תוצאות מאוד יציבות ביחס לשאר המודלים, גם ברמת מילה וגם ברמת אות).

בנוסף, ניתן לראות שברוב המודלים ה-RMSE על ה-Test Set הוא גבוה משמעותית מה-Train וגם מה-Validation. מה שיכול להסביר את התופעה הזו הוא כנראה התפלגות שונה של הנתונים בין ה-train וה-test, כלומר שה-relevance של הדגימות בסטים השונים לא מתפלג בצורה אחידה מספיק – יש imbalance. היינו צריכים לחלק את ה-train כך שההתפלגות בין ה-train/val תהיה זהה, במקום להשתמש ב-validation_split=0.2. לא ייחסנו לתוצאות על ה-test חשיבות במקרה שלנו, גם בגלל חוסר הזמן וגם כי לדעתנו הסתכלות על ה-train/val בלבד במהלך בניית המערכת היא יותר נכונה, כי בתצורת production אין לנו באמת test set. בנוסף, ה- imbalance יכול לנבוע מהעובדה שה-test set קטן מדי

(במקרה שלנו הוא פי 1.5 בערך ביחס ל-train). אולי גם היינו צריכים להגדיל את ה-validation (הוא 20% אצלנו בכל המקרים) או אפילו לעשות KFold.

בעתיד:

היינו רוצים להוסיף יותר שלבים ב-preprocessing של הטקסט בשני השלבים, גם ברמת האות ובמיוחד ברמת המילה, כמו למשל שילוב יותר חכם בין ה-product_title וה-product_description (במקום סתם לשרשר אותם אחד אחרי השני), ואפילו הוספה של טקסט מהטבלאות האחרות שבנתונים שמתארות את החלקים השונים שמרכיבים את המוצר עצמו. עוד דבר שלא עשינו ונרצה לעשות הוא התייחסות ספציפית למילים\רצף מילים מיוחדות, כמו יחידות מידה (בגלל שמדובר בקטגוריית מוצרים של home depot), צבעים, סוגי עץ, מחירים ($200) ועוד.

בנוסף, הוספה של שכבת Dense למודל המקורי שה-output שלו יהווה קלט ל-feature extraction יכול להיות רעיון טוב (כמו שהסברנו בדו"ח למעלה), כי שילוב של שני הוקטורים מה-LSTM והכנסתם למודל ML קלאסי לא הגיוני כל כך, למרות שהגיעו לתוצאות יחסית בסדר.

שיפור הארכיטקטורה המקורית של ה-LSTM גם יכול להיות רעיון נחמד, הוספת שכבות Dense לאחר ה-LSTM ולפני חישוב הדמיון יכול לעזור. גם לפני ה-LSTM או אפילו ביחד – כמה שכבות של Dense ו-LSTM משולבות, אך כמובן זה יצריך זמן ריצה הרבה יותר גדול מה שאין לנו.

בנוגע למודלים, גם של ה-feature extraction וגם ה-LSTM, אולי היינו צריכים גם לבצע נרמול על ה-target כך שיהיה בין 0-1 במקום לבצע טרנספורמציה ברשת לטווח של הבעיה שלנו, ובסוף לעשות de-normalization כדי לבדוק את השגיאה.