lab2

September 15, 2024

Zachary Proom

EN.605.646.81: Natural Language Processing

1 Lab #2

1.1 a

[1]:
```
from charlm import *
```

[2]:
```
mylm = train_char_lm('subtitles.txt', 4)
```

Below are the continuations for 'atio':

[3]:
```
print_probs(mylm, 'atio')
```

```
[('n', 0.9940436161014506),
 (' ', 0.0022096262849457 2),
 ('.', 0.0013930252665962147),
 (',', 0.0009607070804111826),
 ('?', 0.0003362474781439139),
 ("'", 0.00024017677010279565),
 ('u', 0.00019214141608223654),
 ('"', 0.0001441060620616774),
 ('s', 0.0001441060620616774),
 ('-', 9.607070804111827e-05),
 ('!', 4.8035354020559135e-05),
 (':', 4.8035354020559135e-05),
 ('m', 4.8035354020559135e-05),
 ('p', 4.8035354020559135e-05),
 ('r', 4.8035354020559135e-05)]
```

Next, here are the continuations for 'nivi':

[4]:
```
print_probs(mylm, 'nivi')
```

```
[('n', 0.8), ('e', 0.1), ('s', 0.1)]
```

Finally, here are the continuations for 'supe':

[5]:
```
print_probs(mylm, 'supe')
```

```
[('r', 0.9992144540455616), ('s', 0.0007855459544383347)]
```

Next, I generate some random strings (up to 80 characters) from the model, using generate_text():

[6]:
```
import random
random.seed(1) # Set a random seed so the results are reproducible.

num_sentences = 20
sentence_list = [] # Store sentences in this list.
for i in range(0, num_sentences): # Generate 10 random sentences.
    sentence_list.append(generate_text(mylm, 4, 80))
sentence_list
```

[6]:
```
["You're company infect descene.\nWhat?\nMormous.\nPay mom.\nSince a this, I
won.\nI'm ",
 "Nothing you recept well you doing for Nicole.\nNo, it's to cool.\n- Shave the
cant",
 'Are you.\nOverpopulational Cell you have an in Russion even anyone go
intribute d',
 'About it.\nCopy that was Dr. X.\nLesbian.\n- l make land. Fabrief.\nl does no
long w',
 'Where we door.\n-You… at 4:15 PM, MMMMM. HEAR TO THE BEGINNING WHAT HAVE A
LOT ',
 "Cheer to understantibility family.\n'And girls.\nLucius if you the proposed to
an ",
 "Get the worry.\nGo back to do.\nUnderstand-- - # Be can didn't talk togethere
hund",
 "Do you can was we first statement.\nAt lease!\nOnly only man, I'm almost of
oil.\nA",
 "Get in the people my. Yeah, I'll fight, Agnew, you.\nThe odd the
drank.\nEleveryth",
 "You the freedom. - Well, that.\nIt's missie: Hero.\nYou're have and take do. ~
Oh?",
 "You!\nThey're resence in he true.\nThat's times by. Thursday who did you
teless yo",
 'Ah, my cause!\nMay 1.0 LOL.\nNow I know.\nWHAT?\nI know who do repeat!\nHer
Mothere.\n',
 'Joseph had a bit ghost breats. here?\nWhat?\nMomowaka too.\nSorry to blade a
stoppe',
 "You this won't know.\n- What?\nYou see time soldiers all the looking in
him.\nOkay,",
 "Everybody sistandau defeature.\nI've being, almost had backpackage.\nThis the
turn",
 "I willing with you've been work?\nRight-blad… ok, I am…\nThe greaten,
broke Da",
 'One of you had to the been real is so heard to eart!\nDo your rentired.\nAh!
```

```
No! G',
 '- We left to fight. Clyde and I wenty is in the fools!\nDean totalking
passportan',
 "Stop!\nIgor, broke a human invited syster Brand.\nYou're going? Have to resume
it ",
 'Good anythink we´re ship with you beautiful. - You ruin it?\nJust give means
in.\n']
```

Below are three of my favorite sentences produced by the model:

[7]:
```
print("First sentence:")
print(sentence_list[4])
print("")
print("Second sentence:")
print(sentence_list[11])
print("")
print("Third sentence:")
print(sentence_list[18])
```

```
First sentence:
Where we door.
-You… at 4:15 PM, MMMMM. HEAR TO THE BEGINNING WHAT HAVE A LOT

Second sentence:
Ah, my cause!
May 1.0 LOL.
Now I know.
WHAT?
I know who do repeat!
Her Mothere.


Third sentence:
Stop!
Igor, broke a human invited syster Brand.
You're going? Have to resume it
```

1.2 b

Below I demonstrate that my perplexity() function works on the test sentences provided in the prompt:

[8]:
```
perplexity('The boy loves his mother', mylm, 4)
```

[8]: 3.9091903673746224

[9]:
```
perplexity('The student loves homework', mylm, 4)
```

[9]: 4.606972940490915

[10]:
```
perplexity('The yob loves homework', mylm, 4)
```

[10]: inf

[11]:
```
perplexity('It is raining in London', mylm, 4)
```

[11]: 3.711236000904451

[12]:
```
perplexity('asdfjkl; qwerty', mylm, 4)
```

[12]: inf

1.3 c

Below I demonstrate that my smoothed_perplexity() function works on the same test sentences above:

[13]:
```
smoothed_perplexity('The boy loves his mother', mylm, 4)
```

[13]: 3.9091903673746224

[14]:
```
smoothed_perplexity('The student loves homework', mylm, 4)
```

[14]: 4.606972940490915

[15]:
```
smoothed_perplexity('The yob loves homework', mylm, 4)
```

[15]: 3.8414414343307257

[16]:
```
smoothed_perplexity('It is raining in London', mylm, 4)
```

[16]: 3.711236000904451

[17]:
```
smoothed_perplexity('asdfjkl; qwerty', mylm, 4)
```

[17]: 2.01098439084096

1.4 d

1.4.1 Unigrams

First, I train the six unigram models, one per language.

[18]:
```
da_unigram = train_char_lm('da.train.txt', 0)
de_unigram = train_char_lm('de.train.txt', 0)
en_unigram = train_char_lm('en.train.txt', 0)
```

```
fr_unigram = train_char_lm('fr.train.txt', 0)
it_unigram = train_char_lm('it.train.txt', 0)
nl_unigram = train_char_lm('nl.train.txt', 0)
```

Below I loop through each line of the test file. For each line, I calculate the smoothed perplexity for each of the six unigram models and return the language code for the model with the lowest smoothed perplexity. For the first line in the test file, I show all the smoothed perplexity scores.

[19]:
```
predicted_languages = []
actual_languages = []
with open("test.txt") as file:
    i = 0
    for line in file:
        split_tab = line.split("\t")
        actual_language = split_tab[0]
        actual_languages.append(actual_language)
        text = split_tab[1] # Only use the text after the tab. Ignore the␣
 ↪correct language code.
        da_score = smoothed_perplexity(text, da_unigram, 0)
        de_score = smoothed_perplexity(text, de_unigram, 0)
        en_score = smoothed_perplexity(text, en_unigram, 0)
        fr_score = smoothed_perplexity(text, fr_unigram, 0)
        it_score = smoothed_perplexity(text, it_unigram, 0)
        nl_score = smoothed_perplexity(text, nl_unigram, 0)
        min_score = min(da_score, de_score, en_score, fr_score, it_score,␣
 ↪nl_score)
        if min_score == da_score:
            predicted_language = "da"
        elif min_score == de_score:
            predicted_language = "de"
        elif min_score == en_score:
            predicted_language = "en"
        elif min_score == fr_score:
            predicted_language = "fr"
        elif min_score == it_score:
            predicted_language = "it"
        elif min_score == nl_score:
            predicted_language = "nl"
        predicted_languages.append(predicted_language)
        # Print all the smoothed perplexity scores.
        if i == 0:
            print("Smoothed perplexity scores for the six unigram models:")
            print("da_unigram: " + str(da_score))
            print("de_unigram: " + str(de_score))
            print("en_unigram: " + str(en_score))
            print("fr_unigram: " + str(fr_score))
            print("it_unigram: " + str(it_score))
```

```
            print("nl_unigram: " + str(nl_score))
        i += 1
```

```
Smoothed perplexity scores for the six unigram models:
da_unigram: 29.315540257386687
de_unigram: 29.519916658268038
en_unigram: 20.720193646415265
fr_unigram: 21.573215271232797
it_unigram: 23.4811101760081
nl_unigram: 26.631669733860637
```

I calculate and report accuracy for the six languages below:

[20]:
```
actual_language_counts = {
    "da": 0,
    "de": 0,
    "en": 0,
    "fr": 0,
    "it": 0,
    "nl": 0
}
correct_prediction_counts = {
    "da": 0,
    "de": 0,
    "en": 0,
    "fr": 0,
    "it": 0,
    "nl": 0
}
for i in range(0, len(actual_languages)):
    actual_language_counts[actual_languages[i]] += 1
    if predicted_languages[i] == actual_languages[i]:
        correct_prediction_counts[predicted_languages[i]] += 1
```

[21]:
```
print(correct_prediction_counts)
print(actual_language_counts)
print(correct_prediction_counts["da"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["de"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["en"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["fr"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["it"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["nl"]/actual_language_counts["da"]*100)
```

```
{'da': 37, 'de': 103, 'en': 183, 'fr': 41, 'it': 160, 'nl': 172}
{'da': 200, 'de': 200, 'en': 200, 'fr': 200, 'it': 200, 'nl': 200}
18.5
51.5
```

```
91.5
20.5
80.0
86.0
```

The unigram models produce the following results, which are displayed above:

- da: 37 correct out of 200 lines - 18.5%
- de: 103 correct out of 200 lines - 51.5%
- en: 183 correct out of 200 lines - 91.5%
- fr: 41 correct out of 200 lines - 20.5%
- it: 160 correct out of 200 lines - 80.0%
- nl: 172 correct out of 200 lines - 86.0%

1.4.2 Bigrams

Next, I repeat the same steps with bigrams.

[22]:
```
# Train bigram models.
da_bigram = train_char_lm('da.train.txt', 1)
de_bigram = train_char_lm('de.train.txt', 1)
en_bigram = train_char_lm('en.train.txt', 1)
fr_bigram = train_char_lm('fr.train.txt', 1)
it_bigram = train_char_lm('it.train.txt', 1)
nl_bigram = train_char_lm('nl.train.txt', 1)

# Create predictions.
predicted_languages = []
actual_languages = []
with open("test.txt") as file:
    i = 0
    for line in file:
        split_tab = line.split("\t")
        actual_language = split_tab[0]
        actual_languages.append(actual_language)
        text = split_tab[1] # Only use the text after the tab. Ignore the␣
↪correct language code.
        da_score = smoothed_perplexity(text, da_bigram, 1)
        de_score = smoothed_perplexity(text, de_bigram, 1)
        en_score = smoothed_perplexity(text, en_bigram, 1)
        fr_score = smoothed_perplexity(text, fr_bigram, 1)
        it_score = smoothed_perplexity(text, it_bigram, 1)
        nl_score = smoothed_perplexity(text, nl_bigram, 1)
        min_score = min(da_score, de_score, en_score, fr_score, it_score,␣
↪nl_score)
        if min_score == da_score:
            predicted_language = "da"
        elif min_score == de_score:
            predicted_language = "de"
```

```
        elif min_score == en_score:
            predicted_language = "en"
        elif min_score == fr_score:
            predicted_language = "fr"
        elif min_score == it_score:
            predicted_language = "it"
        elif min_score == nl_score:
            predicted_language = "nl"
        predicted_languages.append(predicted_language)
        # Print all the smoothed perplexity scores.
        if i == 0:
            print("Smoothed perplexity scores for the six bigram models:")
            print("da_bigram: " + str(da_score))
            print("de_bigram: " + str(de_score))
            print("en_bigram: " + str(en_score))
            print("fr_bigram: " + str(fr_score))
            print("it_bigram: " + str(it_score))
            print("nl_bigram: " + str(nl_score))
        i += 1

# Calculate and report accuracies.
actual_language_counts = {
    "da": 0,
    "de": 0,
    "en": 0,
    "fr": 0,
    "it": 0,
    "nl": 0
}
correct_prediction_counts = {
    "da": 0,
    "de": 0,
    "en": 0,
    "fr": 0,
    "it": 0,
    "nl": 0
}
for i in range(0, len(actual_languages)):
    actual_language_counts[actual_languages[i]] += 1
    if predicted_languages[i] == actual_languages[i]:
        correct_prediction_counts[predicted_languages[i]] += 1
print(correct_prediction_counts)
print(actual_language_counts)
print(correct_prediction_counts["da"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["de"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["en"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["fr"]/actual_language_counts["da"]*100)
```

```
print(correct_prediction_counts["it"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["nl"]/actual_language_counts["da"]*100)
```

```
Smoothed perplexity scores for the six bigram models:
da_bigram: 23.5642679039043
de_bigram: 23.426090754988575
en_bigram: 13.615689587540372
fr_bigram: 10.366607233458831
it_bigram: 20.40655862278738
nl_bigram: 21.898388667057226
{'da': 164, 'de': 192, 'en': 198, 'fr': 170, 'it': 198, 'nl': 197}
{'da': 200, 'de': 200, 'en': 200, 'fr': 200, 'it': 200, 'nl': 200}
82.0
96.0
99.0
85.0
99.0
98.5
```

The bigram models produce the following results, which are displayed above:

- da: 164 correct out of 200 lines - 82.0%
- de: 192 correct out of 200 lines - 96.0%
- en: 198 correct out of 200 lines - 99.0%
- fr: 170 correct out of 200 lines - 85.0%
- it: 198 correct out of 200 lines - 99.0%
- nl: 197 correct out of 200 lines - 98.5%

1.4.3 4-grams

Finally, I repeat the experiment with 4-grams.

[23]:
```
# Train bigram models.
da_4gram = train_char_lm('da.train.txt', 3)
de_4gram = train_char_lm('de.train.txt', 3)
en_4gram = train_char_lm('en.train.txt', 3)
fr_4gram = train_char_lm('fr.train.txt', 3)
it_4gram = train_char_lm('it.train.txt', 3)
nl_4gram = train_char_lm('nl.train.txt', 3)

# Create predictions.
predicted_languages = []
actual_languages = []
with open("test.txt") as file:
    i = 0
    for line in file:
        split_tab = line.split("\t")
        actual_language = split_tab[0]
        actual_languages.append(actual_language)
```

```
        text = split_tab[1] # Only use the text after the tab. Ignore the␣
↪correct language code.
        da_score = smoothed_perplexity(text, da_4gram, 3)
        de_score = smoothed_perplexity(text, de_4gram, 3)
        en_score = smoothed_perplexity(text, en_4gram, 3)
        fr_score = smoothed_perplexity(text, fr_4gram, 3)
        it_score = smoothed_perplexity(text, it_4gram, 3)
        nl_score = smoothed_perplexity(text, nl_4gram, 3)
        min_score = min(da_score, de_score, en_score, fr_score, it_score,␣
↪nl_score)
        if min_score == da_score:
            predicted_language = "da"
        elif min_score == de_score:
            predicted_language = "de"
        elif min_score == en_score:
            predicted_language = "en"
        elif min_score == fr_score:
            predicted_language = "fr"
        elif min_score == it_score:
            predicted_language = "it"
        elif min_score == nl_score:
            predicted_language = "nl"
        predicted_languages.append(predicted_language)
        # Print all the smoothed perplexity scores.
        if i == 0:
            print("Smoothed perplexity scores for the six 4-gram models:")
            print("da_4gram: " + str(da_score))
            print("de_4gram: " + str(de_score))
            print("en_4gram: " + str(en_score))
            print("fr_4gram: " + str(fr_score))
            print("it_4gram: " + str(it_score))
            print("nl_4gram: " + str(nl_score))
        i += 1

# Calculate and report accuracies.
actual_language_counts = {
    "da": 0,
    "de": 0,
    "en": 0,
    "fr": 0,
    "it": 0,
    "nl": 0
}
correct_prediction_counts = {
    "da": 0,
    "de": 0,
    "en": 0,
```

```
    "fr": 0,
    "it": 0,
    "nl": 0
}
for i in range(0, len(actual_languages)):
    actual_language_counts[actual_languages[i]] += 1
    if predicted_languages[i] == actual_languages[i]:
        correct_prediction_counts[predicted_languages[i]] += 1
print(correct_prediction_counts)
print(actual_language_counts)
print(correct_prediction_counts["da"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["de"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["en"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["fr"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["it"]/actual_language_counts["da"]*100)
print(correct_prediction_counts["nl"]/actual_language_counts["da"]*100)
```

```
Smoothed perplexity scores for the six 4-gram models:
da_4gram: 6.72610373823322
de_4gram: 5.998307660876636
en_4gram: 7.246468761709272
fr_4gram: 4.7966715688365
it_4gram: 5.491558296054048
nl_4gram: 7.643868612856132
{'da': 44, 'de': 82, 'en': 103, 'fr': 105, 'it': 136, 'nl': 46}
{'da': 200, 'de': 200, 'en': 200, 'fr': 200, 'it': 200, 'nl': 200}
22.0
41.0
51.5
52.5
68.0
23.0
```

The 4-gram models produce the following results, which are displayed above:

- da: 44 correct out of 200 lines - 22.0%
- de: 82 correct out of 200 lines - 41.0%
- en: 103 correct out of 200 lines - 51.5%
- fr: 105 correct out of 200 lines - 52.5%
- it: 136 correct out of 200 lines - 68.0%
- nl: 46 correct out of 200 lines - 23.0%

1.4.4 Summary

Below is a summary of the accuracies from the three different experiments.

Language	Unigram	Bigram	4-gram
da	18.5	82.0	22.0

Language	Unigram	Bigram	4-gram
de	51.5	96.0	41.0
en	91.5	99.0	51.5
fr	20.5	85.0	52.5
it	80.0	99.0	68.0
nl	86.0	98.5	23.0

The bigrams are significantly more accurate at predicting language than the unigrams across all languages. The bigrams are also better at predicting every language than the 4-grams, and the unigrams perform better on four languages than the 4-grams (de, en, it, nl).

1.5 e

To perform classification, I divide the training file into two sets based on the value in the gender field (M vs. F). I do this below.

[24]:
```
training_data_male = []
training_data_female = []
with open("tennis.train.txt") as file:
    for line in file:
        split_tab = line.split("\t")
        actual_gender = split_tab[0]
        text = split_tab[1]
        if actual_gender == "M":
            training_data_male.append(text)
        else:
            training_data_female.append(text)

# Convert the training sets into strings and save in text files. This is the␣
↪expected data type in train_char_lm().
training_data_male = "".join(str(question) for question in training_data_male)
training_data_female = "".join(str(question) for question in␣
↪training_data_female)
with open("tennis.train.male.txt", "w") as file:
    file.write(training_data_male)
with open("tennis.train.female.txt", "w") as file:
    file.write(training_data_female)
```

1.5.1 Unigrams

Next, I train two unigram models, one model per training set.

[25]:
```
# Train bigram models.
male_unigram = train_char_lm("tennis.train.male.txt", 0)
female_unigram = train_char_lm("tennis.train.female.txt", 0)

# Create predictions.
```

```
predicted_genders = []
actual_genders = []
with open("tennis.test.txt") as file:
    for line in file:
        split_tab = line.split("\t")
        actual_gender = split_tab[0]
        actual_genders.append(actual_gender)
        text = split_tab[1] # Only use the text after the tab. Ignore the␣
 ↪correct language code.
        male_score = smoothed_perplexity(text, male_unigram, 0)
        female_score = smoothed_perplexity(text, female_unigram, 0)
        min_score = min(male_score, female_score)
        if min_score == male_score:
            predicted_gender = "M"
        elif min_score == female_score:
            predicted_gender = "F"
        predicted_genders.append(predicted_gender)

# Calculate and report accuracies.
actual_gender_counts = {
    "M": 0,
    "F": 0
}
correct_prediction_counts = {
    "M": 0,
    "F": 0
}
for i in range(0, len(actual_genders)):
    actual_gender_counts[actual_genders[i]] += 1
    if predicted_genders[i] == actual_genders[i]:
        correct_prediction_counts[predicted_genders[i]] += 1
print(correct_prediction_counts)
print(actual_gender_counts)
print(correct_prediction_counts["M"]/actual_gender_counts["M"]*100)
print(correct_prediction_counts["F"]/actual_gender_counts["F"]*100)
```

```
{'M': 2453, 'F': 2123}
{'M': 4518, 'F': 3696}
54.293935369632585
57.44047619047619
```

Next, I repeat this experiment with bigrams and 4-grams.

1.5.2 Bigrams

[26]:
```
# Train bigram models.
male_bigram = train_char_lm("tennis.train.male.txt", 1)
female_bigram = train_char_lm("tennis.train.female.txt", 1)

# Create predictions.
predicted_genders = []
actual_genders = []
with open("tennis.test.txt") as file:
    for line in file:
        split_tab = line.split("\t")
        actual_gender = split_tab[0]
        actual_genders.append(actual_gender)
        text = split_tab[1] # Only use the text after the tab. Ignore the␣
↪correct language code.
        male_score = smoothed_perplexity(text, male_bigram, 1)
        female_score = smoothed_perplexity(text, female_bigram, 1)
        min_score = min(male_score, female_score)
        if min_score == male_score:
            predicted_gender = "M"
        elif min_score == female_score:
            predicted_gender = "F"
        predicted_genders.append(predicted_gender)

# Calculate and report accuracies.
actual_gender_counts = {
    "M": 0,
    "F": 0
}
correct_prediction_counts = {
    "M": 0,
    "F": 0
}
for i in range(0, len(actual_genders)):
    actual_gender_counts[actual_genders[i]] += 1
    if predicted_genders[i] == actual_genders[i]:
        correct_prediction_counts[predicted_genders[i]] += 1
print(correct_prediction_counts)
print(actual_gender_counts)
print(correct_prediction_counts["M"]/actual_gender_counts["M"]*100)
print(correct_prediction_counts["F"]/actual_gender_counts["F"]*100)
```

```
{'M': 2505, 'F': 2459}
{'M': 4518, 'F': 3696}
55.44488711819389
66.53138528138528
```

1.5.3 4-grams

[27]:
```
# Train bigram models.
male_bigram = train_char_lm("tennis.train.male.txt", 3)
female_bigram = train_char_lm("tennis.train.female.txt", 3)

# Create predictions.
predicted_genders = []
actual_genders = []
with open("tennis.test.txt") as file:
    for line in file:
        split_tab = line.split("\t")
        actual_gender = split_tab[0]
        actual_genders.append(actual_gender)
        text = split_tab[1] # Only use the text after the tab. Ignore the␣
 ↪correct language code.
        male_score = smoothed_perplexity(text, male_bigram, 3)
        female_score = smoothed_perplexity(text, female_bigram, 3)
        min_score = min(male_score, female_score)
        if min_score == male_score:
            predicted_gender = "M"
        elif min_score == female_score:
            predicted_gender = "F"
        predicted_genders.append(predicted_gender)

# Calculate and report accuracies.
actual_gender_counts = {
    "M": 0,
    "F": 0
}
correct_prediction_counts = {
    "M": 0,
    "F": 0
}
for i in range(0, len(actual_genders)):
    actual_gender_counts[actual_genders[i]] += 1
    if predicted_genders[i] == actual_genders[i]:
        correct_prediction_counts[predicted_genders[i]] += 1
print(correct_prediction_counts)
print(actual_gender_counts)
print(correct_prediction_counts["M"]/actual_gender_counts["M"]*100)
print(correct_prediction_counts["F"]/actual_gender_counts["F"]*100)
```

```
{'M': 2699, 'F': 2503}
{'M': 4518, 'F': 3696}
59.73882248782647
67.72186147186147
```

1.5.4 Summary

Below is a table summarizing the prediction accuracies across all the genders and n-gram models tested. Note that accuracies are rounded to the nearest tenth.

Gender	Unigram	Bigram	4-gram
M	54.3	55.4	59.7
F	57.4	66.5	67.7

As the order of the n-gram model increases, the prediction accuracy across both genders increases steadily. The 4-gram model has the highest prediction accuracies.