In [1]:

```
import os
import time
import string
import math
from collections import defaultdict, Counter
import random
import json

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import plotly.express as px

import nltk
from nltk.corpus import brown, gutenberg, reuters
from nltk.tokenize import word_tokenize
from nltk.util import ngrams


try:
    import Levenshtein
    lev_distance = lambda a,b: Levenshtein.distance(a,b)
except Exception:
    # fallback pure-python
    def lev_distance(a,b):
        if a==b: return 0
        la, lb = len(a), len(b)
        dp = list(range(lb+1))
        for i in range(1, la+1):
            prev = dp[0]
            dp[0] = i
            for j in range(1, lb+1):
                tmp = dp[j]
                cost = 0 if a[i-1]==b[j-1] else 1
                dp[j] = min(dp[j]+1, dp[j-1]+1, prev+cost)
                prev = tmp
        return dp[lb]

sns.set_style('whitegrid')
%matplotlib inline
```

```
C:\Users\patil\anaconda3\lib\site-packages\pandas\core\computation\expressions.py:21: UserWarning: Pandas requires vers
ion '2.8.4' or newer of 'numexpr' (version '2.8.3' currently installed).
  from pandas.core.computation.check import NUMEXPR_INSTALLED
C:\Users\patil\anaconda3\lib\site-packages\pandas\core\arrays\masked.py:61: UserWarning: Pandas requires version '1.3.
6' or newer of 'bottleneck' (version '1.3.5' currently installed).
  from pandas.core import (
```

In [2]:
```python
from nltk.corpus import brown, gutenberg
import random

print("Using NLTK corpora: brown + gutenberg (sample)")
texts = [' '.join(sent) for sent in brown.sents()[:50000]]  # sample many sentences
texts += [' '.join(sent) for sent in gutenberg.sents()[:20000]]
random.shuffle(texts)

len(texts), texts[:5]
```

Out[2]:
```
Using NLTK corpora: brown + gutenberg (sample)
(70000,
 ['This , he claims , would reasonably account for the expansion of the universe .',
  "And what of that poor tarred and feathered wretch he had seen on the road driving down from Schuyler's ? ?",
  "A brisk , amusing man , apparently constructed on an ingenious system of spring-joints attuned to the same peppery r
hythm as his mind , Smith began his academic career teaching speech to Barnard girls -- a project considerably enlivene
d by his devotion to a recording about `` a young rat named Arthur , who never could make up his mind '' .",
  'The very circumstance , in its unpleasantest form , which they would each have been most anxious to avoid , had fall
en on them .-- They were not only all three together , but were together without the relief of any other person .',
  '" Oh !'])
```

In [3]:
```python
# Cell 2 - load a text corpus. Option: use local corpus file if present, else use NLTK corpora.
local_corpus_path = "corpus.txt"  # put your text data here if you have it

texts = []
if os.path.exists(local_corpus_path):
    print("Using local corpus:", local_corpus_path)
    with open(local_corpus_path, 'r', encoding='utf-8', errors='ignore') as f:
        texts = [f.read()]
else:
    # ensure nltk corpora are available
    try:
        brown.sents()[:1]
    except:
        nltk.download('brown', quiet=True)
        nltk.download('punkt', quiet=True)
        nltk.download('gutenberg', quiet=True)
        nltk.download('reuters', quiet=True)
```

```
    print("Using NLTK corpora: brown + gutenberg (sample)")
    texts = [' '.join(sent) for sent in brown.sents()[:50000]]  # sample many sentences
    texts += [' '.join(sent) for sent in gutenberg.sents()[:20000]]
    random.shuffle(texts)

print("Loaded text blocks:", len(texts))
```

```
Using NLTK corpora: brown + gutenberg (sample)
Loaded text blocks: 70000
```

In [4]:
```
num_chars = sum(len(t) for t in texts)
num_tokens = sum(len(word_tokenize(t)) for t in texts)
print(f"Corpus blocks: {len(texts)}, total chars ~ {num_chars:,}, total tokens ~ {num_tokens:,}")
```

```
Corpus blocks: 70000, total chars ~ 8,149,768, total tokens ~ 1,624,973
```

In [5]:
```
import re
_word_re = re.compile(r"[a-z']+")

def normalize_text(s):
    s = s.lower()
    s = s.replace('\n',' ')
    # keep apostrophes for contractions
    s = re.sub(r"[^a-z0-9'\s]", ' ', s)
    s = re.sub(r'\s+', ' ', s).strip()
    return s

def tokenize_text(s):
    # simple tokenizer returning words only
    return _word_re.findall(s.lower())

# small test
print(tokenize_text(normalize_text("Hello, I'm testing this text — it's great!")))
```

```
['hello', "i'm", 'testing', 'this', 'text', "it's", 'great']
```

In [6]:
```
MIN_CORPUS_BLOCKS = 100  # require >100 text blocks to build robust model; adjust as needed

tokens_all = []
for t in texts:
    norm = normalize_text(t)
    toks = tokenize_text(norm)
    tokens_all.extend(toks)

print("Total tokens after preprocessing:", len(tokens_all))
```

```
# unigram counts
unigram_counts = Counter(tokens_all)
V = len(unigram_counts)
print("Vocabulary size:", V)

# bigrams and trigrams
bigram_counts = Counter()
trigram_counts = Counter()

window_text = ' '.join(texts)[:2000000]  # build from a large string to get contiguous ngrams
window_toks = tokenize_text(normalize_text(window_text))
bigram_counts.update(ngrams(window_toks, 2))
trigram_counts.update(ngrams(window_toks, 3))

print("Bigram types:", len(bigram_counts), "Trigram types:", len(trigram_counts))
```

```
Total tokens after preprocessing: 1394818
Vocabulary size: 44243
Bigram types: 174310 Trigram types: 296435
```

In [7]:

```
def top_k_next_unigram(prev_word, k=5):
    # from bigram_counts find most common next words given prev_word
    choices = []
    prefix = (prev_word,)
    for (w1,w2),c in bigram_counts.items():
        if w1 == prev_word:
            choices.append((w2,c))
    choices.sort(key=lambda x: -x[1])
    return choices[:k]

def top_k_next_bigram(prev_words, k=5):
    # prev_words is tuple of length 2 (w1,w2) for trigram lookup
    choices = []
    for (w1,w2,w3),c in trigram_counts.items():
        if (w1,w2) == tuple(prev_words):
            choices.append((w3,c))
    choices.sort(key=lambda x: -x[1])
    return choices[:k]

# quick demos
print("Top next for 'the':", top_k_next_unigram('the', k=8)[:8])
print("Top continuation for ('in','the'):", top_k_next_bigram(('in','the'), k=8))
```

```
Top next for 'the': [('first', 243), ('same', 235), ('lord', 231), ('most', 134), ('other', 130), ('house', 125), ('wor
ld', 116), ('new', 103)]
Top continuation for ('in','the'): [('world', 45), ('same', 44), ('first', 32), ('land', 28), ('morning', 21), ('fiel
d', 18), ('way', 16), ('midst', 15)]
```

In [8]:

```python
def autocomplete(prefix_tokens, k=5):
    """
    prefix_tokens: list of prior tokens (can be 0,1,2)
    returns: list of (token, score) top k
    """
    if not prefix_tokens:
        # return top unigrams
        return unigram_counts.most_common(k)
    if len(prefix_tokens) == 1:
        prev = prefix_tokens[-1]
        res = top_k_next_unigram(prev, k)
        # fall back to unigram if no bigram
        if not res:
            return unigram_counts.most_common(k)
        return res
    if len(prefix_tokens) >= 2:
        prev2 = tuple(prefix_tokens[-2:])
        res = top_k_next_bigram(prev2, k)
        if not res:
            return autocomplete(prefix_tokens[-1:], k)
        return res

# example
print("Autocomplete for ['I'] ->", autocomplete(['i'], k=6))
print("Autocomplete for ['in','the'] ->", autocomplete(['in','the'], k=6))
```

```
Autocomplete for ['I'] -> [('am', 256), ('have', 227), ('will', 115), ('was', 114), ('had', 95), ('do', 91)]
Autocomplete for ['in','the'] -> [('world', 45), ('same', 44), ('first', 32), ('land', 28), ('morning', 21), ('field',
18)]
```

In [9]:

```python
WORDS = dict(unigram_counts)  # mapping word -> freq

alphabet = 'abcdefghijklmnopqrstuvwxyz\''
def P(word, N=sum(WORDS.values())):
    "Probability of `word`."
    return WORDS.get(word,0) / N

# Norvig edits1 & candidates
def edits1(word):
    splits = [(word[:i], word[i:]) for i in range(len(word)+1)]
```

```
    deletes = [L + R[1:] for L,R in splits if R]
    transposes = [L + R[1] + R[0] + R[2:] for L,R in splits if len(R)>1]
    replaces = [L + c + (R[1:] if len(R)>1 else '') for L,R in splits if R for c in alphabet]
    inserts = [L + c + R for L,R in splits for c in alphabet]
    return set(deletes + transposes + replaces + inserts)

def known(words):
    return set(w for w in words if w in WORDS)

def candidates(word):
    c = known([word]) or known(edits1(word)) or known(e2 for e2 in (e2 for e1 in edits1(word) for e2 in edits1(e1))) or
    return c

def norvig_correction(word):
    cands = candidates(word)
    return max(cands, key=P)
```

In [10]:
```
test_words = ['teh','recieve','definately','autocmplete','langauge','accomodate','occured']
for w in test_words:
    t0 = time.time()
    corr = norvig_correction(w)
    t1 = time.time()
    print(f"{w} -> {corr}  (time {1000*(t1-t0):.1f} ms, cand_count ~ {len(candidates(w))})")
```

```
teh -> the  (time 0.0 ms, cand_count ~ 14)
recieve -> receive  (time 0.0 ms, cand_count ~ 2)
definately -> definitely  (time 0.0 ms, cand_count ~ 1)
autocmplete -> autocmplete  (time 119.4 ms, cand_count ~ 1)
langauge -> language  (time 0.0 ms, cand_count ~ 1)
accomodate -> accommodate  (time 0.0 ms, cand_count ~ 1)
occured -> occurred  (time 0.0 ms, cand_count ~ 1)
```

In [11]:
```
SAMPLE_N = 1000
sentences = []
for t in texts[:SAMPLE_N]:
    s = normalize_text(t)
    toks = tokenize_text(s)
    if len(toks) >= 3:
        sentences.append(toks)

len(sentences), sentences[:3]
```

Out[11]:
```
(946,
 [['now',
   'if',
   'i',
   'can',
   'just',
   'figure',
   'out',
   'what',
   "he's",
   'talking',
   'about',
   "i'll",
   'use',
   'it',
   "'''"],
  ['as',
   'an',
   'aid',
   'in',
   'reducing',
   'losses',
   'due',
   'to',
   'enterotoxemia',
   'overeating',
   'disease',
   'feed',
   'a',
   'complete',
   'ration',
   'containing',
   'not',
   'less',
   'than',
   'and',
   'not',
   'more',
   'than',
   'grams',
   'of',
   'aureomycin',
   'per',
   'ton'],
  ['you',
```

```
    'are',
    'very',
    'good',
    'i',
    'hope',
    'it',
    'won',
    "'",
    't',
    'hurt',
    'your',
    'eyes',
    'will',
    'you',
    'ring',
    'the',
    'bell',
    'for',
    'some',
    'working',
    'candles']])
```

In [12]:

```
def evaluate_autocomplete(sentences, k_values=(1,3,5), context_len=1):
    results = {k:0 for k in k_values}
    total = 0
    for toks in sentences:
        for i in range(context_len, len(toks)-1):
            context = toks[i-context_len:i]
            true_next = toks[i]
            preds = [p for p,_ in autocomplete(context, k=max(k_values))]
            total += 1
            for k in k_values:
                if true_next in preds[:k]:
                    results[k] += 1
    return {k: results[k]/total for k in k_values}

# Evaluate with 1-word and 2-word contexts
ac1 = evaluate_autocomplete(sentences, k_values=(1,3,5), context_len=1)
ac2 = evaluate_autocomplete(sentences, k_values=(1,3,5), context_len=2)
print("Autocomplete accuracy (context 1):", ac1)
print("Autocomplete accuracy (context 2):", ac2)
```

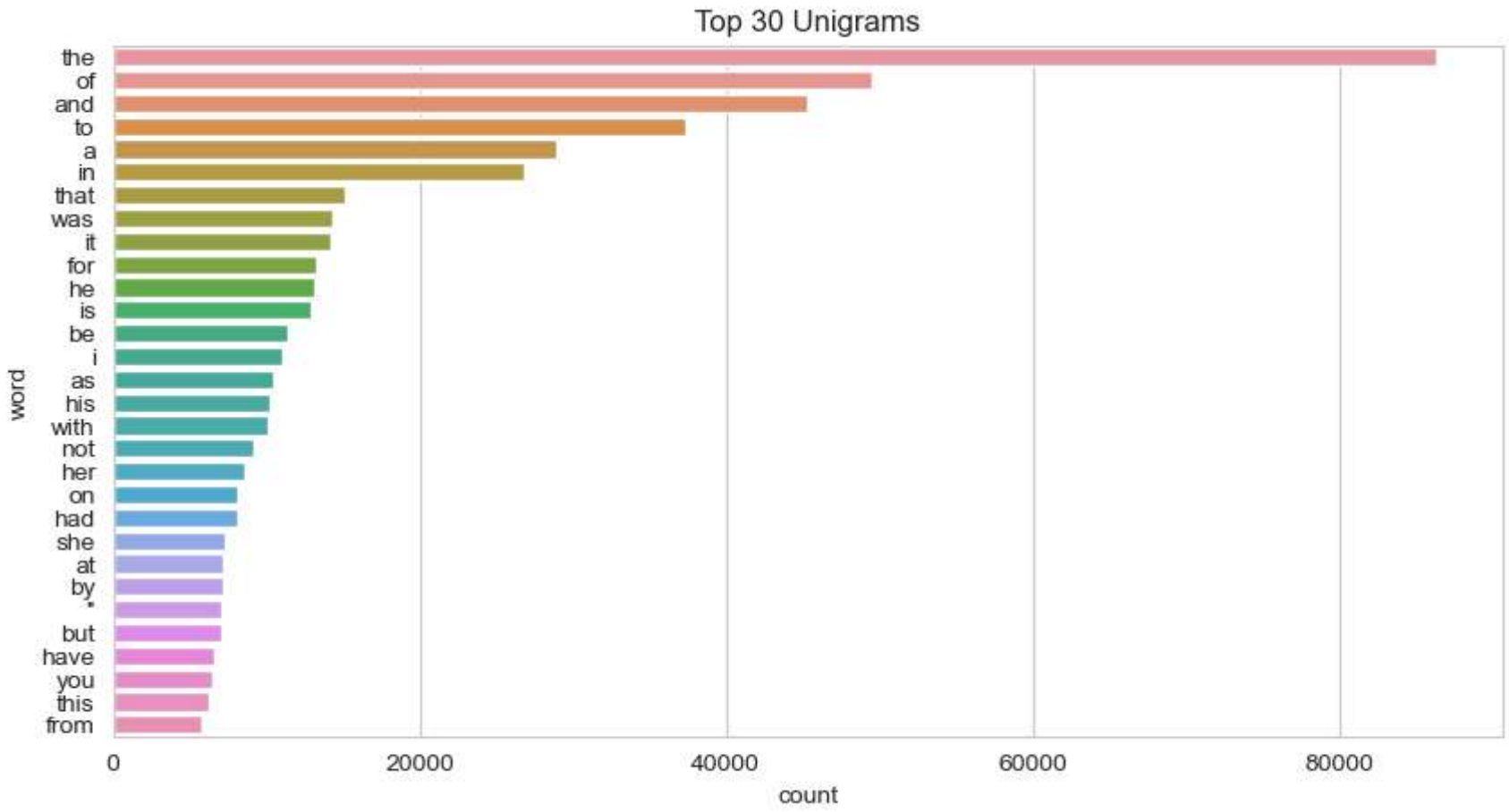
```
Autocomplete accuracy (context 1): {1: 0.26542491268917345, 3: 0.4197017573036199, 5: 0.48921780586507013}
Autocomplete accuracy (context 2): {1: 0.7058444977476159, 3: 0.8325045340197742, 5: 0.8750950681565554}
```

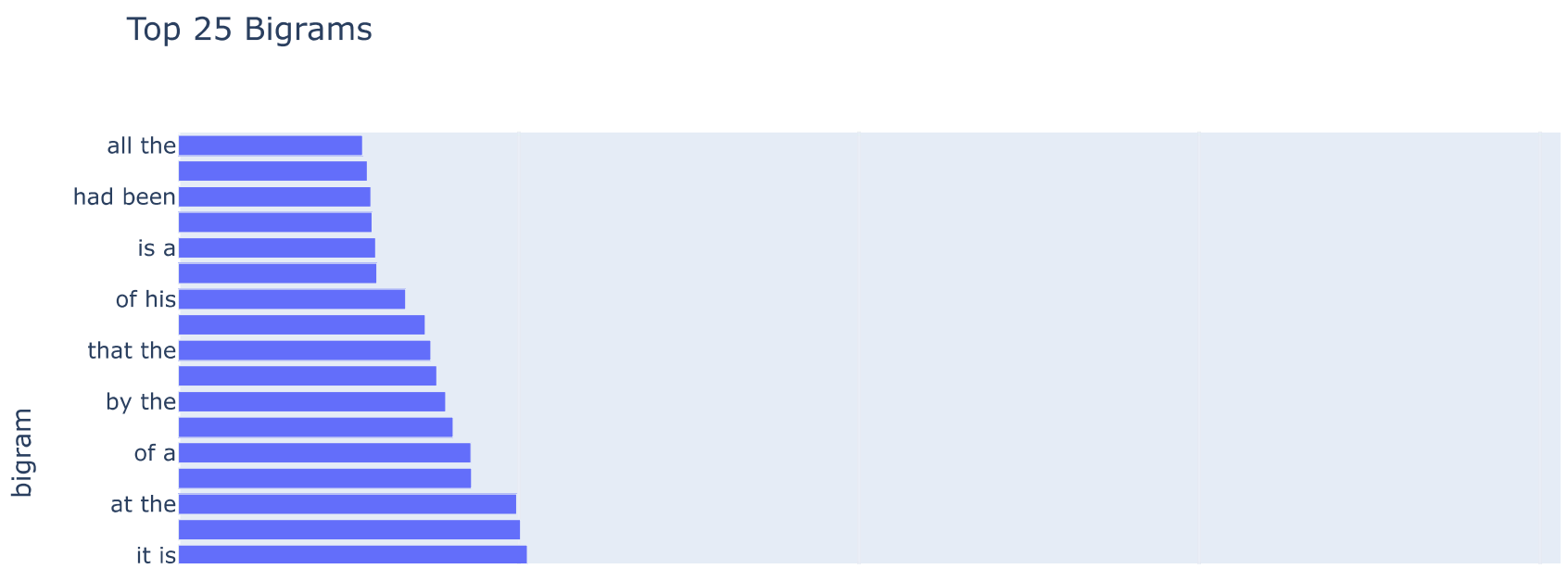
In [13]:
```
def random_typo(word, n_edits=1):
    if len(word)==0: return word
    w = word
    for _ in range(n_edits):
        op = random.choice(['del','ins','sub','trans'])
        i = random.randrange(len(w)) if len(w)>0 else 0
        if op=='del' and len(w)>1:
            w = w[:i] + w[i+1:]
        elif op=='ins':
            w = w[:i] + random.choice(alphabet) + w[i:]
        elif op=='sub':
            w = w[:i] + random.choice(alphabet) + w[i+1:]
        elif op=='trans' and len(w)>1 and i < len(w)-1:
            w = w[:i] + w[i+1] + w[i] + w[i+2:]
    return w

def evaluate_autocorrect(words, n_trials=2000):
    correct = 0
    total_red = 0.0
    total = 0
    for _ in range(n_trials):
        w = random.choice(words)
        if w not in WORDS or len(w)<3: continue
        typo = random_typo(w, n_edits=random.choice([1,1,2]))
        pred = norvig_correction(typo)
        total += 1
        if pred == w: correct += 1
        total_red += (lev_distance(typo, w) - lev_distance(pred, w))
    return {'accuracy': correct/total if total else 0.0, 'avg_levenshtein_reduction': total_red/total if total else 0.0

words_sample = [w for w in unigram_counts if unigram_counts[w] > 5]
ac_autocorr = evaluate_autocorrect(words_sample, n_trials=2000)
ac_autocorr
```

Out[13]:
```
{'accuracy': 0.759090909090909,
 'avg_levenshtein_reduction': 0.9656565656565657,
 'total': 1980}
```

In [14]:
```
top_unigrams = unigram_counts.most_common(30)
df_uni = pd.DataFrame(top_unigrams, columns=['word','count'])
plt.figure(figsize=(10,5))
sns.barplot(data=df_uni, x='count', y='word')
plt.title("Top 30 Unigrams")
plt.show()
```

```
top_bigrams = bigram_counts.most_common(25)
df_bi = pd.DataFrame([(' '.join(k),c) for k,c in top_bigrams], columns=['bigram','count'])
fig = px.bar(df_bi, x='count', y='bigram', orientation='h', title='Top 25 Bigrams')
fig.show()
```





In [15]:
```
algorithms = [
    {'name':'N-Gram Autocomplete (MLE)', 'top1': ac1[1], 'top3': ac1[3]},
    {'name':'N-Gram (context-2)', 'top1': ac2[1], 'top3': ac2[3]},
    {'name':'Norvig Autocorrect', 'accuracy': ac_autocorr['accuracy'], 'lev_red': ac_autocorr['avg_levenshtein_reductio
]
pd.DataFrame(algorithms).fillna('-')
```

Out[15]:

	name	top1	top3	accuracy	lev_red
0	N-Gram Autocomplete (MLE)	0.265425	0.419702	-	-
1	N-Gram (context-2)	0.705844	0.832505	-	-
2	Norvig Autocorrect	-	-	0.759091	0.965657

In [16]:
```
def time_function(func, args_list, repeat=200):
    times = []
    for _ in range(repeat):
        a = random.choice(args_list)
        t0 = time.time()
        func(a)
        t1 = time.time()
        times.append(t1-t0)
    return np.mean(times), np.std(times)

# prepare inputs
autocomplete_inputs = [['the'],['in','the'], ['i'], ['this','is']]
autocorrect_inputs = ['teh','recieve','langauge','autocmplete']

t_auto = time_function(lambda x: autocomplete(x, k=5), autocomplete_inputs, repeat=500)
t_corr = time_function(lambda w: norvig_correction(w), autocorrect_inputs, repeat=500)
print(f"Autocomplete avg time: {1000*t_auto[0]:.3f} ms ± {1000*t_auto[1]:.3f} ms")
print(f"Autocorrect avg time: {1000*t_corr[0]:.3f} ms ± {1000*t_corr[1]:.3f} ms")
```

```
Autocomplete avg time: 21.116 ms ± 9.822 ms
Autocorrect avg time: 33.274 ms ± 56.501 ms
```

In [17]:
```
print("UX plan (summary):")
print("""
1) Recruit n users, split into Group A (N-Gram autocomplete) and Group B (Model X).
2) Measure: typing speed (WPM), acceptance rate of suggestions, manual corrections, user satisfaction (Likert).
3) A/B test metrics: difference-in-means, t-tests, and conversion uplift.
4) Collect qualitative feedback.
""")

# Simulate small A/B sample (toy)
groupA_accept = np.random.binomial(1, 0.35, size=100)  # 35% accept rate
groupB_accept = np.random.binomial(1, 0.42, size=100)  # 42% for improved model
print("Simulated accept rate A mean:", groupA_accept.mean(), "B mean:", groupB_accept.mean())
```

```
UX plan (summary):

1) Recruit n users, split into Group A (N-Gram autocomplete) and Group B (Model X).
2) Measure: typing speed (WPM), acceptance rate of suggestions, manual corrections, user satisfaction (Likert).
3) A/B test metrics: difference-in-means, t-tests, and conversion uplift.
4) Collect qualitative feedback.

Simulated accept rate A mean: 0.37 B mean: 0.41
```

In [18]:
```python
import pickle
os.makedirs('artifacts', exist_ok=True)
with open('artifacts/words_freq.pkl','wb') as f:
    pickle.dump(WORDS, f)
with open('artifacts/bigram_counts.pkl','wb') as f:
    pickle.dump(bigram_counts, f)
with open('artifacts/trigram_counts.pkl','wb') as f:
    pickle.dump(trigram_counts, f)
print("Saved artifacts to ./artifacts")
```

```
Saved artifacts to ./artifacts
```

In [19]:
```python
from IPython.display import display, HTML, clear_output
import ipywidgets as widgets

txt = widgets.Text(placeholder='Type prefix (space separated), e.g. "in the"')
k_slider = widgets.IntSlider(value=5, min=1, max=10, description='Top k')
out = widgets.Output()

def on_change(change):
    with out:
        clear_output()
        prefix = txt.value.strip().lower()
        toks = tokenize_text(prefix)
        res = autocomplete(toks, k=k_slider.value)
        print("Autocomplete suggestions (token, count):", res)
        # autocorrect
        if prefix and len(toks)==1:
            print("Autocorrect suggestion:", norvig_correction(toks[0]))

txt.on_submit(on_change)
display(txt, k_slider, out)
```

```
Text(value='', placeholder='Type prefix (space separated), e.g. "in the"')
IntSlider(value=5, description='Top k', max=10, min=1)
Output()
```

In [20]:
```
summary = {
    'vocab_size': V,
    'autocomplete_top1_context1': ac1[1],
    'autocomplete_top3_context1': ac1[3],
    'autocomplete_top1_context2': ac2[1],
    'autocomplete_top3_context2': ac2[3],
    'autocorrect_accuracy': ac_autocorr['accuracy'],
    'autocorrect_avg_lev_red': ac_autocorr['avg_levenshtein_reduction']
}
print("Project summary:", summary)

pd.Series(summary).to_frame('value').to_csv('artifacts/project_summary.csv')
print("Saved project summary to artifacts/project_summary.csv")
```

```
Project summary: {'vocab_size': 44243, 'autocomplete_top1_context1': 0.26542491268917345, 'autocomplete_top3_context1':
0.4197017573036199, 'autocomplete_top1_context2': 0.7058444977476159, 'autocomplete_top3_context2': 0.8325045340197742,
'autocorrect_accuracy': 0.759090909090909, 'autocorrect_avg_lev_red': 0.9656565656565657}
Saved project summary to artifacts/project_summary.csv
```

In [ ]: