Lab 1 Report

Liam Duignan
Machine Learning for NLP 2

The most fundamental step in implementing our k-NN classifier is computing the squared norms of all the "training" examples. As the definition of the squared norm of a vector is the sum of all its elements squared, this was calculated in the `read_examples` function which is called when the user inputs the trainining therefore each individual squared norm is obtained right away and this calculation is only performed once.

```
# at the end of read_examples function
for ex in examples:
        for feat, val in ex.vector.f.items():
                ex.vector.norm_square += val**2
```

Next comes filling in the `dot_product` function which for two vectors, takes the intersection of their features and sums the values of the first vector times the corresponding values of the `other_vector`.

```
def dot_product(self, other_vector):
    """ Returns dot product between self and other_vector """
    # take intersection of keys
    res = 0
    for feat, val in self.f.items():
        if (feat in other_vector.f):
            res += val * other_vector.f[fe
    return res
```

We now have the building blocks for the `distance_to_vector` function which computes the euclidean distance between two vectors.

```
def distance_to_vector(self, other_vecto
    """ Euclidian distance between self and other_vect
    Requires: that the .norm_square values be already computed """
    # compute squared of norm of vector; norm_squ
    # NB: use the calculation tr
    #   sigma [ (ai - bi)^2 ] = sigma (ai^2) + sigma (bi^2) -2 sigma (ai*
    #                           = norm_square(A) + norm_square(B) - 2
```

```
    return math.sqrt(self.norm_square + other_vector.norm_square \
         - (2 * (self.dot_product(other_vector))))
```

And also `cosine` which computes cosine distance.

```
def cosine(self, other_vecto
    """ Returns cosine of self and other_vector """
    return 1 - self.dot_product(other_vector) / (math.sqrt(self.norm_square
         * math.sqrt(other_vector.norm_square))
```

Finally, we can implement the k-NN predict function, adapted to either distance metric and with optional weighting.

```
    def classify(self, ovector):

        """
        K-NN prediction for this ovector,
        for k values from 1 to self.K
        Returns: a K-long list of predicted classes,
        the class at position i is the K-NN prediction when using K=i
        """
        all_distances_ovector = list() # store tuples containing (dist, gold)
        for ex in self.examples:
            # compute either cosine or euclidean distance
            if self.use_cosine:
                all_distances_ovector.append((ovector.cosine(ex.vector),
ex.gold_class))
            else:
                all_distances_ovector.append(( \
                    ovector.distance_to_vector(ex.vector), ex.gold_class))

        all_distances_ovector.sort() # sort list in ascending order
        counts = defaultdict(int) # dict(class: count)
        k_predicted_classes = list()
        for k in range(self.K):
            if self.weight_neighbors:
                # get freq of each class of first k values of all_distances_ovector
                first_k_distances = all_distances_ovector[:k+1]
                class_frequencies = defaultdict(int) # dict(class: sum of inverse
dist)

                for d in first_k_distances:
                    # get sum of inverse distances for each k nearest class
                    class_frequencies[d[1]] += 1 / d[0]
```

```
                k_predicted_classes.append(max(class_frequencies,
key=class_frequencies.get))

            else:
                counts[all_distances_ovector[k][1]] += 1
                # get list of all classes with same max count
                max_ties = sorted([class_ for class_, val in counts.items() if val
== max(counts.values())])
                # as list is sorted alphabetically, choose first element to append

                k_predicted_classes.append(max_ties[0])

        return k_predicted_classes
```

Results:

```
$ python knn_dict_implementation_TOFILL.py medium.train.examples
medium.dev.examples -k 5
```

```
ACCURACY FOR k = 1 = 61.5% (123/200)
ACCURACY FOR k = 2 = 60.0% (120/200)
ACCURACY FOR k = 3 = 61.5% (123/200)
ACCURACY FOR k = 4 = 59.5% (119/200)
ACCURACY FOR k = 5 = 60.0% (120/200)
```

```
$ python knn_dict_implementation_TOFILL.py medium.train.examples
medium.dev.examples -k 5 -c
```

```
ACCURACY FOR k = 1 = 78.5% (157/200)
ACCURACY FOR k = 2 = 76.0% (152/200)
ACCURACY FOR k = 3 = 77.5% (155/200)
ACCURACY FOR k = 4 = 81.0% (162/200)
ACCURACY FOR k = 5 = 79.5% (159/200)
```

```
$ python knn_dict_implementation_TOFILL.py medium.train.examples
medium.dev.examples -k 5 -w
```

```
ACCURACY FOR k = 1 = 61.5% (123/200)
ACCURACY FOR k = 2 = 61.5% (123/200)
ACCURACY FOR k = 3 = 61.0% (122/200)
ACCURACY FOR k = 4 = 61.5% (123/200)
ACCURACY FOR k = 5 = 62.5% (125/200)
```

```
$ python knn_dict_implementation_TOFILL.py medium.train.examples
medium.dev.examples -k 5 -w -c
```

```
ACCURACY FOR k = 1 = 78.5% (157/200)
ACCURACY FOR k = 2 = 78.5% (157/200)
ACCURACY FOR k = 3 = 80.5% (161/200)
ACCURACY FOR k = 4 = 81.0% (162/200)
ACCURACY FOR k = 5 = 84.0% (168/200)
```