Codierungstheorie – Praktikum 1

Ilja Tscherkassow,
Marko Funke

Inhalt

1. Erzeugung von Q – Allgemein
2. Berechnung der Entropie
3. Huffman Codierung, Erzeugung des Binärbaumes
4. Erzeugung des Codebooks+
5. Encode
6. Decode
7. Main

1. Erzeugung von Q - Allgemein

- Nachdem der Text eingelesen wurde muss zuerst Q=(X,p) berechnet werden
- X ist dabei die Menge der Nachrichtenzeichen
- p die Wahrscheinlichkeit des Auftretens jedes Zeichens

1. Erzeugung von Q - Algorithmus

- Der input Text wird als String übergeben
- Dieser wird dann Zeichenweise durchlaufen
- Jedes Zeichen wird mit der Häufigkeit des Auftretens in einer map gespeichert
- Anschließend berechnet man die Wahrscheinlichkeit des Auftretens jedes Zeichen mit (Anzahl des Zeichen / Gesamtanzahl aller Zeichen)

1. Erzeugung von Q - Codeauszug

```
void ComputeQFromString(const string& rStr, map<char, double>& rQ)
{
    map<char, int> charCount;

    // count chars in string
    int iStrLen = rStr.length();

    for (int iI = 0; iI < iStrLen; iI++)
    {
        pair<map<char, int>::iterator, bool> InsertionResult;

        char cChar = rStr.at(iI);

        InsertionResult = charCount.insert(pair<char, int>(cChar, 0));

        // increment char count
        (InsertionResult.first->second)++;
    }

    // calculate Q
    for (auto It : charCount)
    {
        // calculate p
        double dP = static_cast<double>(It.second) /
            static_cast<double>(iStrLen);

        rQ.insert(pair<char, double>(It.first, dP));
    }
}
```

2. Berechnung der Entropie – Allgemein

- Die Entropie H(Q) einer Informationsquelle Q=(X,p) ist folgendermaßen definiert:

$$H(Q) \coloneqq \sum_{x \in X} p(x) I(x) = -\sum_{x \in X} p(x) log_2(p(x))$$

- Entropie gibt den durchschnittlichen Informationsgehalt an

2. Berechnung der Entropie – Algorithmus

- Das im ersten Schritt erzeugte Q wird übergeben
- Für jedes Zeichen wird dann (-log(p) / log2) berechnet
- Das Ergebnis daraus wird mit der Wahrscheinlichkeit des Auftretens des jeweiligen Zeichens Multipliziert und zur Entropie Addiert

2. Berechnung der Entropie – Codeauszug

```
double ComputeEntropyFromQ(map<char, double>& rQ)
{
    double dLog2 = log(2);

    double dE = 0.0;

    for (auto It : rQ)
    {
        double dP = It.second;

        double dI = -(log(dP) / dLog2);

        dE += dP * dI;
    }

    return (dE);
}
```

3. Huffman Codierung, Erzeugung des Binärbaumes - Allgemein

- Eingabe ist die Informationsquelle Q
- 1. Gestartet wird mit Bäumen welche nur eine Konten besitzen, beschriftet werden diese mit der jeweiligen Wahrscheinlichkeit p(x)
- 2. Solange noch zwei Bäume vorhanden sind werden die mit der niedrigsten Wahrscheinlichkeit miteinander verbunden

3. Huffman Codierung, Erzeugung des Binärbaumes - Allgemein

- Zu 2.:

a) Ein neuer Knoten wird mit dem Wurzelknoten der beiden Bäume verbunden

b) Beschriftet wird der Knoten mit der Summe der Wahrscheinlichkeiten der Wurzelknoten der beiden Bäume

c) Die neue Kannte wird mit 0 beschriftet die ander mit 1

3. Huffman Codierung, Erzeugung des Binärbaumes - Algorithmus

- Der Algorithmus arbeitet wie im Allgemeinen schon beschrieben
- Als Eingebe erhält dieser Q
- Dann werden Bäume mit einem Knoten erzeugt diese bestehen aus dem Strukturtyp *STreeNode*
- Nach der Erzeugung der Bäume müssen dies noch nach ihr Wahrscheinlichkeit sortiert werden
- Anschließend könne die Bäume wie bereits beschrieben verbunden werden

Zu 3. – Strukturtyp: STreeNode

```
struct STreeNode
{
    STreeNode* m_pChildRhs;
    STreeNode* m_pChildLhs;

    double m_dP;
    char m_cChar;

    STreeNode(double dP, char cChar,
        STreeNode* pChildLhs = nullptr,
        STreeNode* pChildRhs = nullptr)
    {
        m_dP = dP;
        m_cChar = cChar;
        m_pChildRhs = pChildLhs;
        m_pChildLhs = pChildRhs;
    }
};
```

Zu 3. – Sortierfunktion der Bäume

```
int QSortCmpFunc(const void* pRhs, const void* pLhs)
{
    const STreeNode& rNodeLhs =
        **((const STreeNode**)(pLhs));
    const STreeNode& rNodeRhs =
        **((const STreeNode**)(pRhs));

    if (rNodeLhs.m_dP > rNodeRhs.m_dP)
        { return (-1); }
    else if (rNodeLhs.m_dP < rNodeRhs.m_dP)
        { return ( 1); }
    else
        { return ( 0); }

    return (0);
}
```

3. Huffman Codierung, Erzeugung des Binärbaumes - Codeauszug

```
STreeNode* ConstructHTreeFromQ(map<char, double>& rQ)
{
    size_t stNumLeafs = rQ.size();

    if (stNumLeafs == 1)
    {
        map<char, double>::iterator It = rQ.begin();
        return (new STreeNode(It->second, It->first));
    }

    // for each x in rQ, create a tree entry
    vector<STreeNode*> NodeVector(stNumLeafs);

    // fill node vector
    size_t stVecIt = 0;
    for (auto It : rQ)
    {
        NodeVector[stVecIt++] = new STreeNode(It.second,
          It.first);
    }

    // sort vector
    size_t stNumNodes = NodeVector.size();
```

3. Huffman Codierung, Erzeugung des Binärbaumes - Codeauszug

```
while (stNumNodes > 1)
    {
        STreeNode** ppVectorData = NodeVector.data();
        qsort(ppVectorData, stNumNodes, sizeof(STreeNode*),
          QSortCmpFunc);

        // connect nodes to trees
        size_t stNumTrees = (stNumNodes / 2);
        size_t stNodeRest = (stNumNodes % 2);
        vector<STreeNode*> TempNodeVector(stNumTrees +
          stNodeRest);

        for (size_t stIt = 0; stIt < stNumTrees; stIt++)
        {
            STreeNode* pChildLhs = NodeVector[stIt * 2 + 0];
            STreeNode* pChildRhs = NodeVector[stIt * 2 + 1];
            double dPSum = (pChildLhs->m_dP + pChildRhs->m_dP);

            STreeNode* pTreeRoot = new STreeNode(dPSum, '\0');

            // append children to parent node
            pTreeRoot->m_pChildLhs = pChildLhs;
            pTreeRoot->m_pChildRhs = pChildRhs;

            TempNodeVector[stIt] = pTreeRoot;
        }

        if (stNodeRest)
        {
            TempNodeVector[stNumTrees] =
               NodeVector[stNumTrees * 2+ 0];
        }

        NodeVector = move(TempNodeVector);

        // get new vector size
        stNumNodes = NodeVector.size();
    }

    return (NodeVector[0]);
}
```

4. Erzeugung des Codebooks - Allgemein

- Das Codebook wird später für die encode-Funktion verwendet
- Mit dessen Hilfe kann man zu jedem Nachrichtenzeichen den dazugehörigen Code „nachschlagen“

4. Erzeugung des Codebooks - Algorithmus

- Für jedes Nachrichtenwort wird der vorher erzeugte Binärbaum Durchlaufen und das Dazugehörige Codewort erzeugt
- Die Funktion arbeitet rekursiv bis ein Codewort erzeugt wurde, diese wird dann in einer map gespeichert

4. Erzeugung des Codebooks - Codeauszug

```
void CreateCodebookFromHTree(
    const STreeNode* pHTreeRoot,
    map<const char, const string>& rC)
{
    string sCodeWord;
    PreOrderTraversal(pHTreeRoot, sCodeWord, rC);
}
```

4. Erzeugung des Codebooks - Codeauszug

```
void PreOrderTraversal(const STreeNode* pRootNode,
     string& rCodeWord, map<const char, const string>& rC)
{
    if (pRootNode->m_pChildLhs != nullptr)
    {
        string sCodeWord = rCodeWord;
        sCodeWord += "0"; // FIXME: placed here because vc12 has
                                  some issues with the + operation
        PreOrderTraversal(pRootNode->m_pChildLhs, sCodeWord, rC);
    }

    if (pRootNode->m_pChildRhs != nullptr)
    {
        string sCodeWord = rCodeWord;
        sCodeWord += "1"; // FIXME: placed here because vc12 has
                                  some issues with the + operation
        PreOrderTraversal(pRootNode->m_pChildRhs, sCodeWord, rC);
    }

    if (pRootNode->m_cChar != '\0')
    {
        rC.insert(pair<const char, const string>(
           pRootNode->m_cChar, rCodeWord.c_str()));
    }
}
```

5. Encode – Allgemein

- Der eingegeben Text wird Zeichenweise durchlaufen
- Für jedes Zeichen wir dann das Dazugehörige Codewort aus dem Codebook herausgesucht

5. Encode – Algorithmus

- Übergeben wird das Codebook sowie der input Text
- Mit Hilfe eines Cursers wird der Text zeichenweise durchlaufen
- Anschließend wird jedes Zeichen im Codebook gesucht
- Wird ein Zeichen gefunden wird das dazugehörige Codewort zum Ausgabestring hinzugefügt

5. Encode – Codeauszug

```
string EncodeStringStream(const map<const char, const string>& rC
     const string& rStrStream)
{
    string sBitStream;

    const char* pCursor = rStrStream.c_str();

    char cChar = *pCursor;

    while (cChar != '\0')
    {
        map<const char, const string>::const_iterator
          FoundEntryIt = rC.find(cChar);

        sBitStream += FoundEntryIt->second;

        cChar = *(++pCursor);

    }

    return (sBitStream);
}
```

6. Decode – Allgemein

- Die Decodierung erfolgt mit Hilfe das Binärbaumes
- Dieser muss für jedes Codewort durchlaufen werden um das dazugehörige Zeichen zu erhalten

6. Decode – Algorithmus

- Übergeben wird der Binärbaum sowie der BitStream
- Mit Hilfe eines Cursers wird der der Stream bitweise durchlaufen
- Für jedes Bit wird der Baum solange durchlaufen bis ein gültiges Zeichen gefunden wurde
- Das gefunden Zeichen wird dann zum Ausgabestring hinzugefügt

6. Decode – Codeauszug

```
string DecodeBitStream(const STreeNode* pHTreeRoot,
     const string& rBitStream)
{
    string sStrStream;
    const char* pCursor = rBitStream.c_str();

    const STreeNode* pCurrentNode = pHTreeRoot;

    char cBit = *pCursor;

    while (cBit != '\0')
    {
        if (pCurrentNode->m_cChar == '\0')
        {
            if ('0' == cBit)
            {
                pCurrentNode = pCurrentNode->m_pChildLhs;
            }
            else
            {
                pCurrentNode = pCurrentNode->m_pChildRhs;
            }

            cBit = *(++pCursor);
        }

        if (pCurrentNode->m_cChar != '\0')
        {
            sStrStream += pCurrentNode->m_cChar;
            pCurrentNode = pHTreeRoot;
        }
    }

    return (move(sStrStream));
}
```

7. Main – Codeauszug

```
int main(int iArgc, const char** ppcArgv)
{
    if (iArgc < 2)
    {
        fprintf_s(stderr, "Invalid number of arguments!\n");
        return (1);
    }

    string sStrStream;

    // read file
    ifstream IFStream(ppcArgv[1]);

    for (string sLine; getline(IFStream, sLine);)
    {
        sStrStream += sLine;
    }

    IFStream.close();

    map<char, double> Q;

    // compute Q from string
    ComputeQFromString(sStrStream, Q);

    double dEntropy = ComputeEntropyFromQ(Q);

    fprintf_s(stdout, "Entropy: %f\n", dEntropy);

    // create huffman tree
    STreeNode* pHTreeRoot = ConstructHTreeFromQ(Q);
```

7. Main – Codeauszug

```
    // create codebook from huffman tree
    map<const char, const string> C;
    CreateCodebookFromHTree(pHTreeRoot, C);

    for (auto It : C)
    {
        fprintf_s(stdout, "%c: %s\n", It.first, It.second.c_str());
    }

    double dPSum = 0;
    for (auto It : Q)
    {
        dPSum += It.second;
        fprintf_s(stdout, "%c: %f\n", It.first, It.second);
    }
    fprintf_s(stdout, "Probability sum: %f\n\n", dPSum);

    string sBitStream;
    string sStrStreamDecoded;

    // encode the string using the codebook
    sBitStream = EncodeStringStream(C, sStrStream);

    // decode the string using the huffman tree
    sStrStreamDecoded = DecodeBitStream(pHTreeRoot, sBitStream);

    // print all data
    fprintf_s(stdout, "Input:\n%s\n\n", sStrStream.c_str());
    fprintf_s(stdout, "Encoded:\n%s\n\n", sBitStream.c_str());
    fprintf_s(stdout, "Decoded:\n%s\n", sStrStreamDecoded.c_str());

    DestroyHTree(pHTreeRoot);

    return (0);
}
```